Activity 0: UML Activity Diagrams

Why?

Designers must learn various design **processes** (collections of related tasks that transform a set of inputs into a set of outputs) and design software processes. **Activity diagrams** are notations for specifying processes, so they are useful for describing design processes and how design works.

Learning Objectives

- Read activity diagrams, recognizing diagram symbols, understanding activity and action inputs and outputs, and the order of action execution
- Explain the token-based activity diagram execution model
- Write correct and readable activity diagrams describing processes

Success Criteria

- Be able to name the symbols in an activity diagram
- Be able to describe the flow of control and data in an activity diagram
- Be able to make an activity diagram for a process

Resources

What is an Activity Diagram? (Week 1 Module in Canvas)

Plan

1. Do the Exercises as a team and check your answers with the instructor.
2. Do the Problems as a team.
3. Turn in the Problems and Assessment as a team deliverable.

Exercises

True or False

1. Actions and activities are the same.
2. There can be several activity final nodes in a diagram.
3. There can be several initial nodes in a diagram.
4. A decision node always has two branches.
5. In general, there are as many merge nodes as decision nodes in a diagram.

Problems (Deliverable)

1. Draw an activity diagram describing an algorithm for finding the maximum value in a list. The input is the list, and output is the maximum value.
2. Draw an activity diagram describing two people peeling a pot of potatoes. Note that two people do this in parallel and they stop when all the potatoes are peeled.

Activity 1: Software Processes

Why?

Every software development effort uses some process. We consider several of the most important models of software processes that have been proposed.

Learning Objectives

- Understand what a software process model is.
- Understand several important software process models.
- Understand the need for and procedure of evaluating software processes.

Success Criteria

- Be able to describe the waterfall, prototype evolution, spiral, RUP, and agile process models.
- Be able to list the advantages and disadvantages of these models.

Resources

Software Processes (pdf from Canvas)

Exercises

1. What is the difference between a software process and a software lifecycle process?

A software process is a process used in the creation or support of a software product. A software lifecycle process is a process depicting the steps that occur from software product inception through retirement from service—that is, a process that shows all the steps in the life of a software product.

2. What is a prototype?

A prototype is a working model of some or all of a finished product.

3. What is rework?

Rework is discarding or redoing previous work products

Problem

1. Process models can be prescriptive or descriptive. If some organization sets out to use a particular software process in a project, is the process model descriptive or prescriptive?

   Prescriptive

2. Which of the waterfall, prototype evolution, spiral, RUP, and agile processes are iterative? Which are incremental?

Type	Models
Iterative	prototype evolution, spiral, RUP, agile
Incremental	prototype evolution, spiral, RUP, agile

3. Which of the waterfall, prototype evolution, spiral, RUP, and agile processes are lightweight and which are heavyweight?

Type	Models
Lightweight	Prototype evolution, agile
Heavyweight	waterfall, spiral, RUP

4. The waterfall model is present in some sense in all subsequent software process models. Describe the role that the waterfall model plays in the prototype evolution, spiral, RUP, and agile models.

Models	The role of waterfall model
The prototype evolution	exist in each iteration
Spiral	exists in both each iteraction and the whole process
RUP	exists in both each phase and the whole process
Agile	exist in each iteraction

5. List the advantages and disadvantages of each model in the table below.

Models	Advantages	Disadvantages
Waterfall	Refer to "Processes.pdf" in reading	
The prototype evolution		

Spiral		
RUP		
Agile		

Activity 2: Software Product Requirements

Why?

We cannot build a software product until we know what it is we are supposed to build. Product requirements provide such specifications.

Learning Objectives

- Understand what stakeholder needs are, what product requirements are, and how the two are related.
- Understand levels of abstraction in needs and requirements.
- Understand how requirements are handled in traditional and agile processes.
- Understand how requirements are stated in traditional and agile processes.
- Be familiar with standard needs elicitation techniques.
- Understand requirements verification and validation techniques.
- Understand requirements management in traditional and agile processes.
- Understand the relationship between product design and requirements analysis.

Success Criteria

- Be able to distinguish between needs and requirements.
- Be able to provide examples of needs or requirements at different levels of abstraction.
- Be able to explain how requirements are frozen and recorded in software requirements specification documents in traditional processes.
- Be able to explain how in Scrum needs are captured in product backlogs and become requirements in sprint backlogs.
- Be able to write traditional requirements specifications that are clear, testable, atomic, and have a unique identifier.
- Be able to write user stories in user voice form.
- Be able to list and explain several needs elicitation techniques.
- Be able to explain how requirements are validated and verified in traditional and agile processes.
- Be able to explain requirements management in traditional and Scrum processes.
- Explain the relationship between product design and requirements analysis.

Resources

Software Product Requirements (pdf from Canvas)

Exercises

1. What is a stakeholder?

2. What is the difference between functional and non-functional requirements?

3. What is a PBI?

4. What is an example of a needs elicitation technique?

5. What is an SRS?

Problem

1. Name three reasons that stakeholder needs cannot simply be recorded as software product requirements.

2. Name three reasons why freezing requirements as is done in traditional (waterfall) processes does not work well.

3. Name three ways that Scrum makes requirements change as easy and cheap as possible.

4. What is the difference between epics, features, and sprintable stories?

5. How can needs elicitation go wrong in Scrum?

6. How are requirements validated and verified in Scrum?

7. What is the relationship between software product design and software requirements analysis?

Construction Techniques

Depending on the process being used, the inputs to the construction phase can vary considerably. When using a traditional heavyweight process, the inputs include all of the engineering design documents, perhaps including detailed designs. When using an agile process, the inputs are the requirements specification or features list (e.g., in Scrum the input is the product backlog).

The output of the construction phase (which may or may not be incremental or iterative) is a version of the software product. The software product can be implemented in one of two ways (or a hybrid)—it can be "bought" or it can be "built."

Bought and Customized SystemsBought and Customized Systems

In some cases, it is possible to purchase a software product that satisfies all of the stakeholders' needs. Such systems are said to be "bought" as opposed to "built" and are not relevant in a section on software construction. However, there are many ways to purchase subsystems, customize them, and integrate them into a software product that satisfies a specific set of requirements; we call these **bought and customized** systems. Since both the customization and the integration of the subsystems involve construction, they are relevant and require some consideration.

Approaches

Unfortunately, the different ways to buy, customize, and integrate subsystems can't be perfectly categorized—there is always some overlap. Hence, many different schemes and terms exist. The most important distinction, however, seems to be between commercial off-the-shelf systems, component-based systems, and service-oriented systems.

Commercial off-the-shelf (COTS) systems are generic (usually large-scale) products that include a wide variety of functions. The construction of a specific product involves the configuration of a generic product so that it provides specific features in specific ways. Probably the best examples of COTS systems are the enterprise resource planning (ERP) systems PeopleSoft and the SAP Business Suite. These generic products provide all of the front-office (customer or user oriented) and back-office (internally oriented) functionality required to run a business, like accounting, bookkeeping, budgeting, asset management, payroll, benefits, recruiting, training, inventory management, etc. However, this functionality is provided in a generic way. The system must be configured to provide the functionality required by particular stakeholders. For example, an accounting method must be chosen, inventory policies must be described, employee evaluation criteria must be provided, reports must be designed, etc.

Component-based systems are constructed from individual objects that conform to a set of standards and provide a gamut of functions in some domain. Construction of a product involves the integration of different components which, because they use standard interfaces, is relatively simple. Example standards include Java Beans and .NET.

Service-oriented systems are similar to component-based systems in that they are constructed from individual components that provide services in specific ways. Constructing a product involves linking different services together. Unlike component-based systems, the link is often through network communications and the services are often provided by distributed hosts.

Advantages and Disadvantages

The advantages of bought and customized systems are fairly obvious. First, because the components tend to be widely used, they tend to be highly reliable. Not only have they been tested in a development environment, they have also been deployed and used in the field. Second, in order to

make the components customizable and configurable, they must comply with well-documented standards. Third, such systems can, in general, be constructed faster and with greater certainty about costs.

The disadvantages of bought and customized systems are also fairly obvious. First, the integrity and continued availability of the system is highly dependent on the suppliers of the components. This can make some organizations uncomfortable. Second, regardless of how well the components are designed, there is a loss of flexibility. The less obvious disadvantage of bought and customized systems is that many software engineers feel that they have less creative control of such systems. Hence, they may have reduced job satisfaction.

Built Systems

Software products built from scratch always engage three (closely intertwined) activities—the design of algorithms, the design of data structures, and programming.

Design of Algorithms

Perhaps the most fundamental aspect of construction is the design of the algorithms necessary to provide the required capabilities.

> An **algorithm** is a set of computational steps for solving a problem in a finite amount of time using a finite amount of memory.

Loosely, an algorithm can be thought of as a description of a process. As such, algorithms can be recorded in a variety of different ways (for example, using natural language, diagrams, or a formal language). However, in the context of software engineering, an algorithm is much more than that. Since algorithms developed during the construction phase will, ultimately, need to be executed by a computer, their steps must be unambiguous and they must produce results that are appropriate and complete (given any set of appropriate inputs). Also, to provide quality assurance, they must be as simple as possible.

Not all sets of instructions are guaranteed to solve the problem they are intended to solve. In such cases the set of instructions is referred to as a **heuristic** (or, sometimes, a **heuristic algorithm**). For example, a heuristic to find the median of a set of numbers with odd cardinality is to pick one at random. This is fast and sometimes works, but is not guaranteed to work. In contrast, an algorithm to find the median of an odd number of values is to put the numbers into an array, sort the array, and then select the value in the middle. This is guaranteed to work but is slower than the heuristic.

The difference between algorithms and heuristics, and when the two are appropriate, can sometimes be confusing to beginning software engineers. Given a choice between an algorithm and a heuristic that use approximately the same amount of time and memory, the algorithm is always preferred. So, in general, it is inappropriate to defend a set of instructions that "sometimes works" by calling it a heuristic. For example, in the case of finding the median of a set of numbers, there are efficient algorithms (faster than the one mentioned above) and it is hard to imagine a situation in which it would be appropriate to use the heuristic because, though it is fast, it hardly ever finds even a good approximate solution. On the other hand, some problems are notoriously difficult to solve efficiently (for example, large instance of the traveling salesman problem) in which case heuristics, especially those that give good approximate solutions, may be justified.

Obviously, algorithm design is a problem-solving activity. However, as with other design activities, this does not mean that the ability is innate, the process can't be taught, or that you can't improve your problem-solving skills. As with all problem-solving activities, it involves both analysis (understanding the problem) and resolution (generating alternative solutions, evaluating the

solutions, and selecting the best alternative). One gets better at algorithm design by studying algorithms, comparing the properties of algorithms, categorizing algorithms (for example, as greedy, divide-and-conquer, brute force, backtracking, and so forth), and studying algorithms for related problems.

In some cases, the problem to solve is quite simple; the way to solve the problem is known (either by the software engineer or somebody else); and the creation of the algorithm only involves careful coding. In such cases, the algorithm may be written in a programming language and there may not be an apparent "algorithm creation" step. Nonetheless, an algorithm is still being created.

Design of Data Structures

In addition to having an algorithm, to solve a problem on a computer you also need a way of organizing the inputs, intermediate calculations, and outputs in the computer's memory.

A **data structure** is a realization in computer memory of the values used to solve a problem.

Conceptually, computer memory is organized as an ordered sequence of storage locations, where each storage location can hold a value of a particular size. Any algorithm uses some subset of the available memory, and there are two broad ways of organizing this subset.

In a **contiguous data structure**, there are no "gaps" in the storage locations that contain the values. That is, every element in the subset except the first and last is adjacent to two other elements in the subset. In other words, if you think of the storage locations as having integer addresses, if the first element of the subset has address i and the last element has address j then the subset contains all of the addresses i, i+1, i+2, …, j-1, j (i.e., the difference between subsequent element addresses is exactly ine). Letting a white rectangle denote a storage location that contains a single value, a contiguous data structure containing four different values can be represented as follows.

When using a contiguous data structure, the initial value (often called element 0) is stored at a location whose address is recorded; every other location can be calculated using its offset from the initial value and the address of the initial element.

In a **linked data structure**, the storage locations need not be contiguous. Instead, one storage location is explicitly connected to one or more others using their addresses. A storage location containing a value and one or more storage locations containing addresses are grouped together into a **node**. Letting a shaded rectangle denote a storage location that contains an address, called a **link**, a (singly) linked data structure containing three different values can be represented as follows. (In this figure, an arrow is used to visually represent a link. In fact what happens is that the shaded storage locations contain the addresses of other nodes.)

When using a linked data structure, the address of the initial node must be stored somewhere. Then the addresses in that nodes are used to locate linked nodes.

Contiguous data structures are particularly appropriate when the (maximum) number of elements is known in advance or doesn't change frequently. For example, at a university in which the maximum class size is 25, it would make sense to use a contiguous data structure in an attendance program. In most other circumstances, linked data structures are more appropriate. Contiguous data structures have two main advantages over linked data structures. First, every element in a contiguous structure can be accessed with (essentially) the same amount of computation. In particular, every element can be accessed using the location of the initial element and the offset from that element. Second, contiguous structures use less memory than linked structures because they do not require any memory for links. On the other hand, linked data structures can handle arbitrarily large data sets (within the limits of memory). Also, linked structures are often faster to modify than contiguous structures.

Linked data structures can take many forms, and the choice of which form to use depends on the operations that need to be performed and the amount of storage that can be used. The example above is a singly-linked structure in which each node contains a link to one other node. This allows efficient traversal in only one direction. So, for example, if a product needs to access elements in a last-in, first-out order (i.e., like a stack of paper), a singly-linked structure is appropriate.[1] If, instead, the elements need to be traversed both from left to right and right to left, a doubly-linked structure like the following is more appropriate.

In this case, each node contains a link to both the next node in the structure and the previous node in the structure. Note that both this example and the previous example are linear structures. That is, there is no way to get from the last node to the first node or vice versa. One could, instead, have a singly-linked circular structure in which the last node contains the location of the first node, or a doubly-linked circular structure in which the last node contains the location of the first node and the first node contains the location of the last node. This may speed access to structure elements.

Again, data structure design is a problem-solving activity. One gets better at data structure design by studying data structures, categorizing them, and understanding their relative merits. As should be apparent, the choice of data structure can have an impact on both the amount of time needed to provide a particular function and the amount of memory required to provide it. In many cases, the design of the algorithm and the design of the data structure are closely intertwined. Hence, while they are conceptually different, from a practical standpoint the two are often considered together.[2]

Programming

Programming is the process of creating a description of one or more algorithms and data structures that can (ultimately) be executed on a computer. That realization is referred to as software (or a program if it can run on its own) and the term "programming" is synonymous with the phrases "writing a program" and "writing software."

[1] One need only store the location of the top node in the structure. To add an element a new node is created that links to the old top. To remove an element, the location of the top node is changed.

[2] The notion of an **abstract data type** (which is defined as a carrier set of possible values and the set of allowable operations on those values) arose because of the close ties between algorithms and data structures.

Programming (and Other) Languages—Programmers most commonly write software in a language that is human-readable but not directly executable on a computer. Such languages are **high-level programming languages**. Software written in a high-level language must be converted into software in a language that can be executed on a computer (**machine language**). For some high-level languages (for example, C), this conversion happens in its entirety before the software is executed and is called **compilation**. For other high-level languages (for example, PHP), this conversion happens just before a statement is executed and is called **interpretation**. Still other high-level languages (for example, Java) use a combination of compilation and interpretation.

To understand programming languages, one must first understand languages. Intuitively, a language (e.g., English) can be thought of as a collection of "words" and a collection of "rules" for combining words. More formally, a **language** consists of a **lexicon** (loosely, the words that can be used in the language) and a **grammar** or **syntax** (loosely, the way the words can be combined, based on their parts of speech, to form sentences). For example, the words "dog" (which is a noun), "ate" (which is a verb). And "the" (which is a determiner) are all items in the lexicon of English. There are also rules in the grammar of English that say: a sentence can be constructed from a noun phrase and a verb phrase; a noun phrase can be constructed from a determiner followed by a noun; and a verb phrase can be constructed from a verb. Hence, "the dog ate" is a sentence in English.

Semantics are language rules that give syntactically correct constructs their meanings. For example, the semantics of an assignment statement are that the expression on the right-hand-side of the assignment operator is evaluated to yield a value, and then this value is stored in the location designated by the expression on the left-hand-side of the assignment statement.

The **pragmatics** of a language are rules that determine how to use the language to accomplish things. For example, when someone utters a sentence beginning with the words "I promise" he or she is making a statement that describes what they are doing. But that person is also obligating him or herself to perform some action, which goes beyond the simple meaning of the sentence. When writing code, there are rules about how to go about accomplishing things with the language. For example, to iterate over an array, one might use a for loop—this is pragmatic rule about how to accomplish something with the language.

A **programming language** is a formal language used to describe computations. Unlike natural languages (like English), programming languages are designed, and each has a carefully constructed lexicon and grammar. When you read that someone has created a new programming language, it means that that she or he has created a new lexicon and a new grammar (and, ideally, the tools necessary to use them).

Some programming languages (like C, Fortran, and Pascal) are called **imperative** languages. Programs in imperative languages consist of a sequence of specifications or manipulations of values in memory locations. Example specifications include "store this value in the following memory location," "add the values in the following two memory locations," and "compare the values in the following two memory locations." Some imperative languages are control-driven (i.e., first do this, then do this) whereas others are data-driven (i.e., if you encounter this kind of thing do this to it). Programs in control-driven languages look like recipes or instructions. For example, the following fragment in C adds the values stored in the variables named `a` and `b`, stores the result in a variable named `total`, divides the `total` by 2, and returns the result.

```
double mean(double a, double b)
{
    double   total;

    total = a + b;
    return (total / 2.0);
}
```

Programs in data-driven languages look like a collection of rules. For example, the following fragment in XSL-T says that when a volume element is encountered, the characters "Vol. ", followed by the volume, followed by the character "." should be output.

```
<xsl:template match="volume">
  Vol. <xsl:value-of select="." />,
</xsl:template>
```

Other programming languages (like Prolog) are called **declarative** languages. A program in a declarative language is a sequence of statements describing one or more *rules* stating goals and ways to achieve them, and *facts*, which are goals that are already achieved. Execution of the program results in a goal-driven search for a solution. The order that the computations are performed is not controlled by the author of the program. For example, the following fragment in Prolog specifies the rule "if X is a cow then X is domesticated," the fact "bossy is a cow," and the target goal "bossy is domesticated."

```
domesticated(X) :- cow(X).
cow(bossy).
? domesticated(bossy).
```

Running this program in a Prolog interpreter would yield the result "yes" because the computer can find a way to achieve the target goal with the rules and facts available to it.

Obviously, before a program can be written, a programming language (or languages) must be chosen. This is generally not the responsibility of an individual programmer but of an entire team, a project manager, or organizational policy-makers. The choice of a programming language is driven by many factors including the suitability of the language, the experience of the team or organization, and the tools available.

Idioms

In natural languages, idioms are difficult to pin down. Though an idiom is often defined as an expression that is peculiar to a language because its meaning can't be derived from the individual words, the distinction between idioms ("give way" to mean retreat), metaphors or similes ("drowning in money" to mean rich), metonyms ("the suits" to refer to management), and fixed expressions ("salt and pepper" rather than "pepper and salt") can be particularly subtle. What all of them have in common is that they involve a shared culture, and people who are fluent in a language are a part of that shared culture.

Though programming languages are much simpler than natural languages, there are often many grammatical ways to express the same thing. An **idiom** in a programming language is a particularly useful and commonly used way to express something. As in a natural language, one cannot be said to be fluent in a language until one knows its idioms.

Idioms can also be thought of as patterns for solving problems in a particular programming language. Like any pattern, an idiom is a proposed way to solve a common problem that has certain advantages and disadvantages, but is often a appropriate and hence often used.

A good example of an idiom is swapping the values in two variables. In C (and many other languages), the code would look something like the following.

```
temp = a;
a    = b;
b    = temp;
```

That is, first the value in the variable `a` is temporarily stored in the variable `temp`. Then the value in the variable `b` is stored in the variable `a`. Finally, the value that was originally stored in the variable `a` (and is now stored in the variable `temp`) is stored in the variable `b`. In Python the code could look exactly the same (without the semi-colons) but it could also be written as follows (this kind of thing is sometimes referred to as **syntactic sugar,** i.e., an aspect of the grammar that makes the language sweeter for human use).

```
a, b = b, a
```

Python supports *multiple assignment statements* in which several expressions on the right-hand-side of the assignment operator are evaluated and their values assigned to multiple variables on the left-hand-side of the assignment operator. Python thus supports the idiom above for swapping values.

Idioms both make it easier to write programs and to read them. They reduce the cognitive burden of writing programs because they make some parts of the process "automatic." They reduce the cognitive burden of reading programs because they make the programs easier to understand. For example, though infinite loops are uncommon, they are sometimes necessary. In languages like C/C++ they can be written as follows.

```
while (true) {
  // Body of the loop
}
```

However, one could, instead, use the following idiom.

```
for (;;) {
  // Body of the loop
}
```

Though it is unusual to use a `for` statement for an infinite loop, the unusual visual nature of this statement makes it immediately apparent (to those who are fluent) that this is an infinite loop.

While it is important to learn idioms and become fluent, they should not become an obsession, and they should not be used to exclude outsiders. In both natural languages and programming languages, idioms can only be learned over time. Those who are fluent and those who are new to a language must recognize this and behave accordingly.

Programming Style

A style guide (or manual) is, in general, a set of standards. There are style guides for the creation of textual documents (such as *The Associated Press Stylebook* and the *Modern Language Association Handbook*), the creation of street signs, the creation of charts and graphs, and a variety of other

creative pursuits. They tend to focus not on problems with solutions that have clear criteria of right and wrong, but on problems with solutions that can be described as more or less aesthetic.

A programming style guide provides the same kind of information about writing code in a programming language. Hence, a programming style guide is not concerned with what is grammatically correct (or even what is idiomatic) in that language but with how the code itself should appear and be formatted. While one can talk about the aesthetics of an algorithm, programming style guides are concerned about the aesthetics of the manifestation of that algorithm in code. Though most style guides are language-specific, they always address two common issues—naming conventions and layout conventions.

Naming conventions are concerned with which identifiers are associated with program entities, such as variables, functions, methods, classes, files, packages, folders, etc. One issue that often arises is the appropriate length and spelling of names. While there are some obvious guidelines that almost everyone agrees on, issues about the length and spelling can be both complicated and controversial. All but the most obstinate programmers can agree that names should be descriptive. However, it is inconvenient when multiple variables with long names need to be used in the same statement. Hence, some style guides limit the use of long names and specify the appropriate ways to abbreviate (or not to). For example, `cnt` might be preferred to `count` because it saves two letters and `cnt` is unlikely to be an abbreviation for anything else. Some style guides also allow for the use of short variable names (e.g., `x`) used in a very limited scope (like within a small loop).

Another issue that often arises is the way to handle word breaks in names. Some style guides call for the use of an underscore character between words (like `statistics_calculator`) while others call for the use of camel case (like `StatisticsCalculator`).

Naming conventions may also discuss the appropriate use of prefixes and suffixes. Some style guides advocate the use of prefixes (or suffixes) to indicate the type of a variable. For example, `iLength` would make it clear that the variable contains an integer while `fLength` would make it clear that the variable contains a floating point number. Other style guides advocate the use of prefixes or suffixes for semantic information. For example, `totalLength` or `lengthTotal` to indicate that the variable contains a summation of values.

Finally, naming conventions sometimes also specify antonyms that should be used. For example, they might call for the use of `beginValue` and `endValue` rather than `startValue` and `endValue`.

Layout conventions are principally concerned with the visual appearance of code. Particularly common policies involve where and what to indent (for example, always indent the body of a loop), the location of block begin and end markers (for example, the location of the `{` and `}` characters used in C), the length of lines (typically 80 characters or less—a throwback to the days of punch cards) and how to wrap long lines, vertical alignment (for example, aligning the variable names across multiple declaration statements) and the amount of vertical space to leave between statements (for example, the number of blank lines between methods in Java).

Some layout conventions are a matter of personal preference. Others are (almost) unanimously agreed upon. In the first category are conventions about the placement of block markers. For example, some C/C++/Java programmers prefer the `{` character on a line of its own and others do not, as shown in the following example.

```
if (amount > 100.0)
{
   discount = 0.15;
   shipping = 0.00;
}
```

```
if (amount > 100.0) {
   discount = 0.15;
   shipping = 0.00;
}
```

Personal preference also influences conventions about vertical alignment. For example, some C/C++/Java programmers prefer that variable identifiers be vertically aligned and others do not, as shown in the following example.

```
boolean done;
double  amount, discount;
int     size;
```

```
boolean done;
double amount, discount;
int size;
```

On the other hand, there is wide agreement on conventions about indentation. Almost everyone will agree that both of the previous two fragments are preferred to the following.

```
     if (amount
       >
         0) { discount = 0.15;
shipping = 0.00;
   }
```

Regardless of the conventions chosen, it is clear that conventions are important, even though they sometimes mean that individuals will have to subjugate their preferences to the preferences of the organization.

One layout convention that is not about the visual appearance of the code is the choice between tabs and spaces. As discussed above, indentation policies are an important part of many style guides. However, in and of themselves, indentation policies don't specify how to indent; they just specify what and where to indent. Anyone that has used a keyboard of any kind knows that it is possible to indent either using tabs or using spaces. Tabs have two advantages—the tab stops can be changed and a tab is a single character (and, hence, indentation can be undone by deleting a single character). But spaces have the advantage of being portable—the indentation stays the same when the code is moved from machine to machine (i.e., it does not depend on the values of the tab stops). Fortunately, most modern development environments let you easily convert between the two.

Commenting

Programmers are also responsible for the creation of internal documentation using comments. **Comments**, which are part of the grammar of almost every programming language, do not affect the way the software executes but make it easier for humans to understand the code. Most languages distinguish between **block comments**, which contain multiple lines (for example between a `/*` and a `*/` in C/C++/Java), and **inline comments**, which occupy a part of a single line (for example, after a `//` in C/C++/Java).

Ideally, comments would never be necessary because the code itself is so easy to understand that any further embellishment would only be a distraction—this is called **self-documenting code**. Unfortunately, programming languages are not perfectly expressive and programmers are rarely able to understand code without considerable effort. Hence some comments are helpful. However, many beginning programmers (who are often forced to comment in introductory courses) have a tendency

to over-comment, while many experienced programmers (who argue that their code is self-documenting) have a tendency to under-comment.

Over-commenting can make code more difficult to read and can lead to comments that are inconsistent with the code (that is, wrong). Under-commenting can make code more difficult to understand and, hence, to read and modify. This doesn't mean that the number of comments should be counted to determine whether you have over- or under-commented. Instead, it means that it is important to think about the way comments are being used, and try to use them in appropriate ways.

There is one use of comments that is always inappropriate. In other words, if you are using comments in this way you are, by definition, over-commenting. Comments should never repeat the code. If the code is written properly, it will accurately and succinctly provide information about the operations that it performs. So, for example, comments like the following are completely inappropriate.

```
// Increase i by 1
++i;

// Include sales[i] in the total
total = total + sales[i];
```

On the other hand, there are three uses of comments that are always appropriate. That is, you never need worry about over-commenting if you are using comments in these ways.

- It is always good to use comments to describe the intent of a complicated or long piece of code. These kinds of comments both provide an aid to understanding and a means of verifying that the code does what it is supposed to do. However, it is important not to confuse the intent with the process. As discussed above, the code itself documents the process; the comments should explain what the code is supposed to be accomplishing.
- It is always a good idea to use comments to describe the rationale for a particular decision. Remember that programming is a problem-solving activity and that problem solving involves the enumeration of alternative solutions and the selection of a particular alternative. Comments should be used to explain "the road not taken" and the rationale for doing so. For example, one often has to choose between doubly-linked and singly-linked data structures. Comments should be used to explain why one was chosen over the other. As another example, anyone who has worked with collections has been faced with the choice of using a map or a list (for example, in Java, choosing between a `HashMap` and an `ArrayList`). Comments should be used to explain why a particular collection was chosen or not chosen (for example, elements are inserted into the collection and the insertion operation is $O(n)$).
- It is always a good idea to use comments to provide references to external documents. These documents can be published (for example about the performance of a particular algorithm), proprietary (for example, a design document containing UML diagrams), or anything in between. Some tools (such as the `javadoc` utility) can even create links to such documents to make them easily retrievable.

In general there are three uses of comments that are questionable. That is, it is important to think long and hard before using comments in these ways.

- It may not be a good idea to use comments to summarize the code. While describing the intent of complicated or long piece of code is always a good idea, and repeating the code is always a bad idea, summarizing the code is sometimes helpful and sometimes not. In some instances, providing a summary of the steps that will follow can provide guidance. This is

especially true for long pieces of code in which the steps might be performed in multiple orders. In other instances, the summary is nothing more than repetition.

- It may not be a good idea to use comments to describe "to do" items or "future work." In some cases, these kinds of comments can become an excuse for not doing the right thing now. In other cases, they are important reminders. In general, these kinds of comments are appropriate when they are combined with comments that include a rationale, and inappropriate otherwise.
- It is sometimes useful to provide supplementary information to help readers understand the code. For example, suppose an array search algorithm assumes that the array is sorted. This important precondition is not obvious and not expressible in most programming languages, so it is helpful to include a comment stating the precondition. The problem is that it is hard to predict when supplementary information will really be helpful. It is probably good policy to err on the side of over-commenting in these cases.

Before concluding this discussion of comments, it is important to note that some aspects of commenting are directly tied to the tool set being used. For example, you will notice that no mention was made of using comments to maintain a change log (that is, a list of who made what changes and when). This is because of the prevalence of version control systems that (when used properly) keep a precise and detailed record of all this information. Another example is the use of tags in comments. Many tools exist (like `javadoc` and its offspring for other languages) for creating standalone documentation from the comments included in source code. These tools support various special tags and markers in the comments that can be used to control the creation process (for example, including `@see` in a comment will cause to create a link to another document). Hence, organizations will often have guidelines about how to use tags and markers.

Data Organization

As mentioned earlier, software products often include data. Hence, the construction of software products often involves the organization of data. The organization of data should not be confused with the design and implementation of data structures, which is related to the internal operation of the program. Instead, it is concerned with the external representation and description of the data.

Markup (and Related) Languages—One popular way to organize data is using a markup language. Examples of markup languages include the Extensible Markup Language (XML) and the Hypertext Markup Language (HTML). XML is a language for describing any hierarchical data. For example, a book contains chapters, chapters contain sections, and sections contain paragraphs. The following fragment in XML is for a grade book that has three students in it, each of whom has a name, ID, and grade.

```
<gradebook>
  <student>
    <name>Kim Masch</name>
    <id>321-0021</id>
    <grade>A</grade>
  </student>
  <student>
    <name>Doug Sidel</name>
    <id>001-9998</id>
    <grade>C</grade>
  </student>
  <student>
    <name>Wayne Tromerling</name>
    <id>101-4167</id>
    <grade>B-</grade>
  </student>
</gradebook>
```

In XML, the author can create whatever elements are needed to describe the data. HTML is a language for describing one particular kind of hierarchical data—documents (e.g., articles). Hence, HTML predefines the elements that can be used. Examples of elements include the section, header, footer, paragraph (`p`), description (`dl`), term (`dt`), definition (`dd`), and image (`img`). The following HTML fragment is from a textbook on multimedia software.

```
<section>
  <p>
    The sampling of static visual content involves the
    sampling of both the color spectrum and space (usually
    in that order).
  </p>

  <dl>
    <dt>Color Sampling</dt>
    <dd>
      A process for converting from a continuous (infinite)
      set of colors to a discrete (finite) set of colors.
    </dd>
  </dl>

  <img src="raster.png"/>
</section>
```

Note that, in both of these examples, the markup language is being used to describe the function of the data (e.g., the parts of the document), not the form (e.g., what the document should look like when presented to the reader). One can use a presentation language (e.g., CSS) to describe the appearance of a document. The following fragment in CSS (i.e., Cascading StyleSheets) says that all `p` elements should be rendered in a sans-serif font and be indented 0.5in, and all `dt` elements that are contained in dl elements should be italicized.

```
p {
    font-family: sans-serif;
    text-indent: 0.5in;
}

dt dl {
    font-style: italic;
}
```

Databases—In principle, a **database** is any collection of interrelated values. In practice, a database is usually fairly large, contained in multiple files, and interrogated or queried and manipulated using specialized software (a database management system or DBMS). The most common form is the relational database.

In a **relational database**, the data are organized into rectangular tables (called relations), consisting of columns (called **fields**) and rows (called **records**). Operations are then performed to create other tables. For example, in a database recording data about university students, one table might contain a record for each student's ID number and GPA, and another table might contain a record for each student's ID number and major. One could then perform an operation (in this case, a join) to find the average GPA by major.

The organization of tables in a relational database can have a significant impact on the computation required to respond to different queries. Hence, database design is an important part of the construction of any software product that uses a relational database.

Hybrid Systems

As the name implies, **hybrid** systems have some aspects of bought systems and some aspects of built systems.

Libraries

As discussed earlier, a library is a group of related sub-programs for accomplishing a specific collection of tasks. Both proprietary commercial and free and open source libraries exist for accomplishing tasks in a wide variety of domains (for example, numerical optimization or business graphics). Application programmers can buy libraries and incorporate them in the product rather than reinvent the wheel.

Application Frameworks

Application frameworks recognize that applications can be categorized and that the applications in a particular category require support for similar features. For example, all GUI-based applications that deal with documents must provide the user with the ability to create a new document, save a document, save a document with a new name, etc. An application framework would provide this functionality in a generic way so that it could be used to create either a word processor, a spreadsheet program, or a presentation program.

Application frameworks are often object-oriented but provide an additional abstraction layer. That is, they are comprised of numerous classes but the application programmer can ignore the details of the individual objects and focus on the capabilities provided by the framework as a whole.

Quality Assurance in Construction

As discussed earlier, verification and validation occur throughout the software engineering life-cycle and developers can't (and shouldn't) try to isolate these kinds of activities. Nonetheless, from a conceptual standpoint, there are verification and validation activities prominent during the construction phase. We discuss these activities in terms of the role of programmers.

Programmers must accomplish many things while they are programming. First, they must write software that is syntactically correct. That is, they must write software that obeys the grammatical rules of the language they are using. Second, they must write software that is consistent with the standards sanctioned by the project and organization. Third, they must write software that implements algorithms and data structures in accord with design specifications. The verification and validation activities that are conceptually part of the construction phase are concerned with these three objectives.

Some verification and validation activities that occur during construction are automated (and referred to as *code analysis*), and some are not (and referred to as *code review*). Some analysis techniques involve programs while they are running (and are referred to as *dynamic analysis*), while others examine software when it is not running (and are referred to as *static analysis*).

Code Reviews

Code reviews are a specific example of the general category of reviews discussed in the chapter on software quality assurance. As with other kinds of reviews, they differ in their formality. Desk checking software differs from desk checking other work products in that the author of the code executes it by hand when desk checking, while with other work products, the author merely reads the work product carefully. During construction, sub-programs should be desk-checked in isolation. In other words, there is no reason to (and it is very difficult to) desk check a large software system. Code walkthroughs also need not wait for the entire program or system to be completed, but usually involve fairly substantial sub-programs or libraries.

Software work product inspections were originally done for code, and they have been well developed and studied. Code inspections are are regarded as **formal reviews** because each participant has a well-defined role, the activity follows a well-defined process, and the activity is guided by a checklist. We discussed the formal inspection process in some detail in the software quality assurance chapter; here we simply add a few details particular to code inspections to that discussion.

Formal code review preparation and the review meetings themselves are quite taxing. If they are done too quickly, they are usually not done well. Consequently, preparation rates should not exceed about 200 lines of code per hour, and team inspection rates should not exceed about 150 lines of code per hour.

The code inspection checklist should include specific defects that inspectors should look for during preparation. The following are some examples from a Java code inspection checklist.

- All variables and constants are named in accord with naming conventions.
- There no variables or attributes with confusingly similar names.
- Every variable and attribute is correctly typed.
- Every method returns the correct value at every return point.
- All methods and attributes have appropriate access modifiers (private, protected, public).

- No nested if statements be converted into a switch statement.
- All exceptions handled appropriately.

Ideally, the inspector makes a pass through the work product looking for a handful of defects, then another pass looking for another handful, and so forth; looking for all defects at once in a single pass is usually not effective.

Static Analysis

Static analysis involves the automated verification of software before it is executed. Because static analysis techniques do not involve execution, they can be performed on sub-programs.

Syntax Checking

The simplest kind of static analysis is syntax checking. A syntax check ensures that the software obeys the grammatical rules of the language it is written in. Today, syntax checking is most frequently performed by a language-sensitive editor.[1] That is, the editor (i.e., the software product that the author uses to create and edit code) knows the syntax of the language being used and, while the author types, checks to make sure that everything is syntactically correct. The author is informed of syntactic mistakes using some kind of symbology (either color-coding, icons, or underlining). The process is quite similar to the spell checking and grammar checking that is performed by many word processors. However, many editors also can suggest syntactically correct completions of partially entered statements or automatically complete statements.

One important thing to be aware of is that syntax checkers (including compilers) do not always correctly identify the root cause of the problem. For example, consider the following fragment in C/Java/JavaScript.

```
if (hour <= 0) || (hour > 12))
{
   invalid = true;
}
else if (hour === 12)
{
  hour = 0;
}
```

A syntax checker might suggest that the `||` operator in the first line was misplaced, might suggest that there was an extra `)` in the first line, or might suggest that the `else` was misplaced. In fact, the problem is a missing `(` before `hour` in the first line. As another example, consider the following fragment in HTML.

```
<p.>
I never think of the future. It comes soon enough.
</p>
```

A syntax checker might suggest that there is a missing `</p.>`, or might suggest that there is an unexpected `</p>`. In fact, the first line should be `<p>`.

1 In the early days of programming, syntax checking was typically performed by the compiler or interpreter. In the days of punch cards and batch processing, this meant that the author might have to wait hours to learn of a syntax error.

Style Checking

Most projects require programmers to follow a language-specific style guide. Style guides include standards related to typography (such as the use of uppercase and lowercase letters), naming (such as descriptive names for variables), formatting, including indentation and other white space issues (for example, the size of tabs, the location of curly brackets), control structure usage (for example, always having a default case in switch statements), module sizes (for example, the maximum lines in a method or class), language features usage (for example, always declaring variables in languages that do not require this), and so forth.

One of the first style checkers was a tool called *lint*. It was developed at Bell Laboratories in the 1970s to find the "undesirable fiber and fluff" in C programs. (Many current style checkers include "lint" in their name out of respect.) For example, lint checks to see if any variables are declared and not used. While this is not a problem either syntactically or logically, it is bad style.

Idiom and Usage Checking

Though lint (for C) is often described as a style checker, it is actually much more than that. For example, it identifies statements that are unreachable, logical expressions that have constant values, and calls to functions that return values in some places and not in others. Hence, lint is also an idiom and usage checker.

Usage checkers usually look for three different things: suspicious or error-prone constructs (e.g., uninitialized variables, the use of the division operator), non-portable constructs (e.g., constructs that might give rise to range problems), and memory allocation inconsistencies. Idiom checkers, on the other hand, are much more varied.

Idiom and usage checkers tend to be language specific. For example, in C an idiom and usage checker might flag functions that are passed different numbers of arguments or arguments of different types, whereas in Java the compiler would detect this. As another example, in JavaScript, an idiom and usage checker might flag arrays that are constructed using `new Array()` rather than `[]` (for example, `grades = new Array()` rather than `grades = []`, which is purely an idiomatic matter peculiar to JavaScript).

There are exceptions, however. For example, many languages have both for and while loops, and an idiom and usage checker might flag determinate while loops or indeterminate for loops. As another example, some languages do not have block scope, and an idiom and usage checker might flag variables that are declared in a block (that, between curly brackets) as misleading.

Formal Methods

The term **formal methods** refers to the class of static analysis tools that rely on mathematical models. They can be categorized as follows.

Model checking is the process of automatically determining if a program or sub-program satisfies certain requirements. The inputs to a model checker are the program or sub-program and a formal specification of requirements. Most often, requirements are specified using logical expressions.

Data flow analysis is the process of enumerating the set of possible values calculated at various points in a program or sub-program using ideas from graph theory.

Symbolic evaluation is the process of automatically tracing the execution of a program or sub-program using symbolic values rather than numeric values. It is used to identify the values that will cause different statements to be executed.

Despite the rigor of these methods, they are not perfect. It has been proven that no automated process can always determine if an arbitrary program will execute correctly.[2] In addition, these tools tend to be very difficult to use, so they are used only in specialized circumstances.

Unit Testing

As discussed in the chapter on quality assurance, **unit testing** is the testing of individual units or sub-programs in isolation. An individual unit test case consist of one value for each of the sub-program arguments and one corresponding expected value for each of the outputs. The goal of unit testing is to identify all of the faults in a sub-program. An incomplete test suite will give rise to a **false negative** (the conclusion that there are no faults). An incorrect expected value (say, as a result of a mistake made when calculating the value by hand) in a test will give rise to a **false positive** (the conclusion that there is a fault when there isn't).[3]

Developing Tests

There are two broad strategies for testing that differ in what is presumed to be known about the component, sub-system, or product being tested. In **black box** testing the processing details of the component, system, or product are presumed to be unknown. That is, the person developing the tests only knows what can go into the tested unit and what should come out. In **clear box** testing (sometimes also called **white box** or **open box** testing), the person developing the tests knows exactly how the component, sub-system, or product is built and how it should operate. Though it might seem counterintuitive, a combination of the two is generally required for effective testing.

Clear box testing tends to focus on the **coverage** of the tests, which can be loosely defined as the proportion of the tested unit that is exercised. However, it is important to understand that coverage can be measured in a variety of different ways, and many of the measures used are based on a graph-theoretic representation of the component, sub-system, or product. Most commonly, the graph theoretic representation is the *control flow graph (CFG),* but it is also possible to use activity diagrams for this purpose.

In a control-flow graph, an **action node** represents a piece of code with one entry point and one exit point. For example, an assignment statement can be represented by an action. A **decision node**, on the other hand, is used to represent the start of a piece of code with one entry point and multiple exit points. For example, the conditional expression in an if statement is represented by a decision node. Nodes are connected by arrows representing possible flows of execution through the nodes.

The easiest way to understand the difference is with the following example (in which the statement numbers are included as comments at the end of each executable line). For this example, test case input consists of a values for the `calculate()` function parameters `x` and `y`, which we write as the ordered pair (x, y). (Since we are only concerned now with code coverage, we will not worry about test case outputs.)

2 This result is related to the halting problem for Turing machine.

3 The terms false negative and false positive can be confusing but are used in the way they are in medical testing where a positive result indicates the condition is present (often a bad thing from the standpoint of the patient) and a negative result indicates the condition is not present (often a good thing from the standpoint of the patient). The terms type I error and type II error from statistical hypothesis testing are also sometimes used in software testing, bet their interpretation depends on the statement of the hypothesis (e.g., there are no faults).

```
int calculate(int x, int y)
{
   int        a, b;

   a = 1;                  // S1

   if (x > y)              // S2
   {
      a = 2;               // S3
   }

   x++;                    // S4
   b = y * a;              // S5

   if (y <= 0)             // S6
   {
     b++;                  // S7
   }

   return b;               // S8
}
```

This example can be illustrated ina flow-graph, which shows all alternative flows of control through a piece of code. The following activity diagram is the flow graph for the code above.

We can use this flow-graph to calculate various kinds of coverage metrics.

Statement coverage is the percentage of statements exercised when a set of test cases is executed. To measure statement coverage, one identifies the nodes in the activity diagram that have and have not been executed by a set of test cases. For example, consider the test case with input (1, 2). Tracing execution using either the code or the activity diagram, it is easy to see that the statements S1, S2, S4, S5, S6, and S8 are executed while S3 and S7 are not. Hence, the statement coverage for this test is 75% (six statements were covered out of eight total statements).

Branch coverage is the percentage of branch directions taken when a set of test cases is executed. In branch coverage, one identifies the outbound edges from decision nodes that have and have not been traversed. For example, again consider the test case with input (1, 2). Since 1 is not greater than 2, the [else] edge out of S2 will be traversed; since 2 is not less than or equal to 0, the [else] edge out of S6 will be traversed. Hence, the branch coverage for this test is 50% (two outbound edges from decision nodes out of four total).

Path coverage is the percentage of all execution paths taken when a set of test cases executes. In path coverage, one identifies the paths that have and have not been traversed from the initial node to the final node. In this example, there are four possible paths from S1 to S8. The first is S1, S2, S4, S5, S6, S8; the second is S1, S2, S3, S4, S5, S6, S8; the third is S1, S2, S4, S5, S6, S7, S8; and the fourth is S1, S2, S3, S4, S5, S6, S7, S8.[4] Since the test case input (1, 2) causes traversal of only one path, it has a path coverage of 25% (one path out of four).

Of course, one test is never enough, and one must consider the coverage of entire test suites (sets of test cases). Letting $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ denote the set of test case inputs in a test suite (where *n* denotes the number of tests in the suite, x_i denotes the first parameter of `calculate()` and y_i the second, consider coverage for the following three different test suites.

First, consider the test suite input set {(5, -2)}. S1 would be executed, S2 would be executed, S3 would be executed because 5 is greater than -2, S4 would be executed, S5 would be executed, S6 would be executed, S7 would be executed because -2 is less than or equal to 0, and S8 would be executed. Thus, this test suite has statement coverage of 100% (even though it consists of only one test and there are eight statements).

Note, however, that not every branch is executed. In particular, x>y is covered but x<=y is not covered. Similarly, y<=0 is covered, but y>0 is not. It turns out that all branches can be covered with the test suite input set {(5, 2), (-2, -1)}. The first test covers x>y and y>0, and the second test covers x<=y and y<=0. Hence, this test suite has branch coverage of 100% (even though it consists of only two tests and there are four branches).

While this test suite covers all of the branches, it does not cover all four of the paths: it only covers two. However, the test suite input set {(5, 2), (-2, 5), (-1, -2), (-2, -1)} covers all four paths. In particular, the test input (5, 2) covers the path S1, S2, S3, S4, S5, S6, S8; the test input (-2, 5) covers the path S1, S2, S4, S5, S6, S8; the test input (-1, -2) covers the path S1, S2, S3, S4, S5, S6, S7, S8; and the test input (-2, -1) covers the path S1, S2, S4, S5, S6, S7, S8.

Obviously, regardless of the measure being used, a test suite with greater coverage is preferred to a test suite with lesser coverage. However, large test suites take longer to execute than small test suites. Hence, it is important to think about the relationship between the number of tests and coverage. In particular, notice that even though there are eight statements, it is possible to have 100% statement

4 Note that, because each edge joins a unique pair of nodes, paths can be identified either by their edges or by their nodes. Since every node is labeled and every edge is not, it is more convenient to use the nodes in this case.

coverage with only one test. Also, notice that even though there are four branches, it is possible to have 100% branch coverage with only two tests. However, there is no way to have 100% path coverage (i.e., cover all four paths) with fewer than four tests.

A point to note about path coverage is that in most programs it is impossible to achieve 100% path coverage with a reasonable test suite. Consider a sub-program with a loop that may execute as many as a million times. The number of paths through this sub-program is at least one million (bypassing the loop, going once through the loop, going twice through the loop, etc.). It would require at least one million test cases to cover all these paths. The definition of path coverage can be modified to take account of loops in various ways and so generate higher path coverage metrics, but when counting *all* paths through programs, 100% path coverage is generally unachievable.

Furthermore, while 100% path coverage is laudable, it is not a guarantee of correctness. To see why, consider the following specification for a tax calculator.

> The tax due must equal to 1% of the assessed value but must not exceed $1000.

Now, suppose that this functionality is provided (incorrectly) by the following function (in Python).

```
def tax(value):
   result = 0.01 * value
   if (value > 1000):
      result = 1000
   return result
```

Further, suppose that this function is tested with the suite input set {100, 200000} which has anticipated (correct) output values of {1, 1000}. This test suite has 100% statement coverage, 100% branch coverage, and 100% path coverage and the function will produce correct output for both tests. Nonetheless, this function contains a fault. Hence, though unit testing would seem to imply that this implementation is free of faults, in fact, it contains a major fault.

The functionality can be provided correctly by the following function that contains a subtle but important change in the condition of the if statement.

```
def tax(value):
   result = 0.01 * value
   if (result > 1000):
      result = 1000
   return result
```

This implementation produces exactly the same output for the two test cases and also has 100% coverage using all three measures.

Ideally, the fault in the first implementation would be identified during verification. However, because it is subtle, it might be missed. This observation, and others, has led some people to advocate a process known as **test driven development** (TDD) in which tests are developed as the code is and for very small increments in functionality [1]; we discuss test-driven development below.

Some tests should be based on the requirements and, ideally, developed before the code. This helps with the analysis phase of construction (it helps the programmer understand the requirement) and, to some extent, documents the code. Other tests should be developed after the code, and should focus on coverage. Because the increments in functionality are small, the creation of these tests is not overly burdensome.

Tests developed based on requirements (before code is written) are, by their very nature, black box tests, and there are two obvious "brute force" ways to proceed. The first is to use complete enumeration and the second is to use random sampling.

Complete enumeration (or **exhaustive** testing) creates a test suite that contains all possible inputs. For example, suppose you need to create a black box test suite for a sub-program that takes a letter grade as its only input and calculates a corresponding numeric grade. It is relatively easy to create a test suite that contains all possible inputs. At first glance, it even seems like complete enumeration will work in all cases. There are, after all, a finite number of both integer values (i.e., 2^{32} in a 32-bit system) and even floating point values (i.e., 2^{64} in a 64-bit system) in a binary representation. However, remember that a test case consists of both the input and the expected output. Hence, without an **oracle** (i.e., another sub-program that is known to output the correct answers), complete enumeration is very tedious. In addition, most sub-programs have more than one input, and the size of the completely enumerated test suite is the product size of the sets of individual inputs. So, for example, consider a sub-program that receives an array of 10 integers as input. A completely enumerated test suite would have $2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} = 2^{3210} = 10^{320}$ test cases, which is a very large number.

An alternative "brute force" approach is **randomization**, which can be done in either a structured or unstructured fashion. The idea here is to start with (conceptually) the completely enumerated test suite and then select tests from that suite in some probabilistic fashion. In theory, if the distribution of failure trigger conditions is known (or can reasonably be assumed), then one can use probabilistic reasoning to determine the expected number of faults that will remain undetected. In practice, the distribution of triggers is unlikely to be known and the need to generate the expected result for each case remains.

It is important to remember, however, that black-box testing is not "completely uninformed" testing. Specifically, the requirements provide a great deal of information about the kinds of tests that are likely to be failure triggers. One particularly important way to use requirements to generate black-box tests is **boundary value analysis** which is predicated on the idea that failure triggers tend to be found at the "boundaries" of the sets of inputs and outputs. For example, if an input must be in a certain range, test cases should be created using values just below, at, and just above the minimum and maximum limits of the range. Similarly, if an output must be in a certain range, test cases should be created that generate values around the minimum and maximum limits of the output range.

In the tax example above, the requirement itself would naturally lead to inclusion of test cases that would, based on the 1% rule alone, lead to a tax of slightly less than 1000 (for example, an input value of 90000), exactly 1000 (input of 100000), and slightly more than 1000 (for example, input of 110000). The original implementation would fail the first of these tests (producing an output of 1000 when the correct output is 900).

Not surprisingly, there are many other common heuristics. Some popular value-based heuristics for individual parameters include the following.

- The number 0 (because of problems caused by dividing by 0)
- Very large and very small (in both actual value and absolute value) numbers (because of underflow and overflow problems)
- Character or string versions of digits and numbers

Some popular value-based heuristics for groups of parameters include the following.

- Equal values for the parameters
- Different relative values of the parameters (for example, for two parameters x and y, have one test with x greater than y and one with x less than y)

Some popular array and collection heuristics include the following.

- Very small and very large arrays or collections
- Arrays or collections of length 0 and 1
- Arrays or collections that are unsorted, ascending, and descending
- Arrays or collections with duplicated values, and those without duplicates

In some situations it is possible to partition test cases into sets in which each test case in the set is expected to trigger the same failure. Such a set is called an **equivalence class**. The important property of equivalence classes is transitivity[5], which, in the context of three test cases *a*, *b*, and *c*, means that if *a* triggers the same failure as *b*, and *b* triggers the same failure as *c*, then *a* triggers the same failure as *c*. Obviously, if one can identify equivalence classes, then there is no reason to use more than one test case from each equivalence class. To understand how this works in practice, consider a sub-program that is passed an integer representation of the day of the week, with 0 denoting Sunday, 1 denoting Monday, etc. There are three partitions associated with this range—the valid values $\{0, 1, 2, 3, 4, 5, 6\}$, the invalid values that are less than 0, $\{x: x < 0\}$, and the invalid values that are greater than 6, $\{x: x > 6\}$. Similarly, an input that is a single numeric value has three associated partitions (the value itself and both sides of the value), an input that is a Boolean has two associated partitions (true and false), and an input that is an element of a set has two associated partitions (the set and the complement of the set).

Regression Testing

Four things can occur when a developer attempts to correct a fault, as illustrated I the table below.

	No New Fault Introduced	**New Fault Introduced**
Fault Corrected	Good	Bad
Fault Not Corrected	Bad	Very Bad

Project data shows that

- As many as 30% of software changes result in one of the three bad outcomes,
- On average, bad outcomes occur about 10% of the time,
- Faults introduced during bug fixes are more difficult to identify and remove than others.

These facts suggest that programmers should thoroughly test a fix to ensure that it really does correct the defect they intend to correct. Furthermore, because of the likelihood of introducing a new fault, programmers should rerun all tests to try to detect this occurrence. Running *all* tests over again is called **regression testing**. Regression testing should be done after *any* software change, even apparently innocuous changes such as editing comments.

5 An equivalence relation is also reflexive and symmetric.

It used to be that regression testing was an arduous and time-consuming activity. Nowadays unit testing and integration testing tools automate regression testing, so there is no reason whatever not to do frequent regression testing. We discuss such tools next.

Unit Testing Tools

In the early days of software engineering, it was common for testers to to create a custom **test driver** or **test harness** that either contained hard-coded test cases or read test case inputs and corresponding expected outputs from a file, ran the tested unit on the inputs, and compared the results with the expected outputs. Now, however, it is common to use a standard testing framework to scaffold the process.

In most unit testing frameworks, a single test consists of a small block of code that includes an identifier for the test, the sub-program to call, the input values to use, the output values to expect, and an **assertion** that relates actual output to the expected output. Assertions are often stated as equalities (that is, the actual output must equal the expected output). For example, in QUnit (a testing framework for JavaScript), an individual test case for a body mass index (BMI) calculator might look something like the following.

```
test("Underweight Test", function() {
      equal(18.2, calculateBMI(123, 69).toFixed(1));
});
```

The identifier for this test is `"Underweight Test"` and the assertion is that the result of calling `calculateBMI()` with the inputs 123 and 69 (the weight in pounds and height in inches, respectively) must equal 18.2. This example also illustrates an important detail of unit testing that is important to remember—when working with floating point numbers it is important to use either an "approximately equals" assertion or to specify a specific number of significant digits (as is done in this case).

One of the main advantages of using a unit testing framework is robustness. In *ad hoc* testing frameworks it is difficult to isolate different tests. In other words, the test suite may terminate at the first failure or individual tests may change the state of the system and, hence, not be independent. Well-designed unit testing frameworks make it easy to construct isolated tests.

Another big advantage of unit testing frameworks is the way they report results. In general, the reports that are generated are easy to read and highlight the failed tests rather than the successes.

Many unit testing frameworks also include, or can be supplemented with, code coverage tools. These tools provide clear, easy to understand reports about the coverage of individual tests and test suites. In general, these tools require no additional effort on the part of the tester. Continuing with the BMI calculator example from above, after running 13 tests that included inputs for three important partitions, underweight, normal weight, and overweight, the output of a code coverage tool might report the following.

Statements	*Covered*	*% Covered*
9	7	77.78%

```
function calculateBMI(weight, height)
{
      "use strict";
      var bmi;

      bmi = weight / (height * height) * 703;

      if (bmi < 15){
            bmi = 15;
      }else if (bmi > 60){
            bmi = 60;
      }

      return bmi;
}
```

This indicates that the test suite neglected to include important output-related boundary cases and, as a result, did not cover the two highlighted statements (i.e., these two statements were never executed).

Debugging

The discussion of software quality assurance included definitions of the terms defect, failure, and fault. To understand the debugging process fully it is also important to define the terms error and symptom.

A **symptom** is a characteristic of a failure that can be observed.

An **error** (of omission or commission) is something that a person does that gives rise to a fault.

The best way to understand the difference between these concepts is with an example of a very simple software product. The requirements for this product are as follows: it must read an integer from the keyboard, divide 10 by that integer (using integer division), and display either the result or an error message if the user-entered value is 0. The following pseudo-code is (supposed to be) an implementation of that product

```
a ← 10
b ← readInt()
c ← a/b
writeInt(c)
```

If a user executes this fragment and enters the value 20, it will produce the correct output. On the other hand, if the user enters the value 0, the product will "crash" (stop executing). The value 0 is, thus, a trigger condition that causes the product to fail (in particular, to not display an error message). The symptom of this failure is the crash. The fault is the omission of statements that checks the user-supplied value and display an error message.

Now consider the following implementation.

```
a ← 10
b ← readInt()
if b ≠ 0 then
       c ← a/b
       writeInt(c)
endif
```

In this case if a user executes the fragment and enters 0, nothing will be displayed. So, the failure is the same (the error message is not displayed), the trigger condition is the same (the value 0), but the symptom is different (there is no output) and the fault is different (only the statements to display the error message are omitted).

It's important to remember that a failed test case results in one symptom and one corresponding trigger conditions of a failure. Debugging is the process of using trigger conditions to identify and correct faults. It consists of the following steps.

1. **Stabilize**—Understand the symptom and the trigger condition identified by the test so that the failure can be reproduced.
2. **Localize**—Locate the fault.
   - 2.1 Examine the sections of code that are likely to be influenced by the trigger condition.
   - 2.2 Form a hypothesis about the fault.
   - 2.3 Instrument the relevant sections of code.
   - 2.4 Execute the code using the instrumentation to monitor the state of the likely problematic sections of code.
   - 2.5 Prove or disprove the hypothesis. If proven, continue to step 3; if disproven return to step 2.
3. **Correct**—Fix the fault.
4. **Verify**—Test the fix. It is advisable to run all tests to help ensure that the fix did not introduce a new fault with different symptoms.
5. **Globalize**—Look for and fix similar defects in other parts of the system.

This process obviously can be applied to code (that is, the realization of algorithms and data structures). However, it can also be applied to data. Consider, for example, an HTML document. A language-sensitive editor will find all syntax errors, but a structural fault (for example, a `p` element that is incorrectly declared to be an `li` element) may not become apparent until the document is rendered in a browser. The process above would then be used to debug the document. Similarly, consider a CSS document. Again, syntax errors should be found by an editor, but logical mistakes (for example, assuming that all `p` elements are contained in `section` elements) may not be apparent until it is applied to an HTML document, at which point the process above could be used.

Stabilization and verification involve careful use of the testing tools and techniques discussed earlier. Correction and globalization also involve nothing new. Localization, on the other hand, including both the instrumentation and the execution steps, are new and warrant additional discussion. In particular, they involves the use of debug code, a debugger, or both.

Debug Code

Debug code includes both temporary output statements that can be used to monitor state information and temporary input statements that can be used to pause the execution of a component.

When including debug output, it is important to include output identification information along with the output data. One mistake that beginning software engineers make is to include debug code like the following (in Java).

```
for (int i=0; i<n; i++)
{
   System.out.println(i);
   System.out.println(n);
   total = total + sales[i];
   System.out.println(sales[i]);
   System.out.println(total);
}
```

When executed, this kind of debug code generates a screen full of numbers that is virtually impossible to interpret. Instead, one should certainly include information to identify the values, like the following.

```
for (int i=0; i<n; i++)
{
   System.out.println("i: " + i + "(n: " + n + ")");
   total = total + sales[i];
   System.out.println("sales[i]: " + sales[i]);
   System.out.println("total:    " + total);
}
```

In addition, one may want to include debug output statements that indicate that a block, function, or method has been entered, and perhaps even indent the output accordingly, as in the following example.

```
System.out.println("Entering the loop.");
for (int i=0; i<n; i++)
{
    System.out.println("    i: " + i + "(n: " + n + ")");
    total = total + sales[i];
    System.out.println("    sales[i]: " + sales[i]);
    System.out.println("    total:    " + total);
}
System.out.println("Exited the loop.");
```

In some cases, it is convenient to put the debug output in the same stream as the normal output (for example, in a Java console application, use `System.out` for both). This can make it easier to follow the flow of execution. In other cases, this can generate too much output and it is better to put the debug output in a different stream (for example, `System.err` in one or more files).

Experienced developers use a wide variety of techniques to facilitate the use of debug code. Some "comment out" debug code when it is not needed (and some editors provide this capability). Some include debug code in conditional statements and use environment variables or parameters to control which debug statements are executed. Sometimes this even involves multiple "debug levels."

Despite the clunkiness of debug code, it is used quite commonly in practice. Its biggest disadvantages are that it can obscure the code it is meant to debug, it can take a significant amount of time to add, and it can be difficult to know what to output. Hence, its use often involves a trial and error process.

Debuggers

A **debugger** is a tool that can be used to trace the execution of a code without having to modify it in any way (in particular, without having to add debug code). Some debuggers are command-line tools (like gdb) and some have a GUI (for example, front ends for gdb like DDD, and the Java debugger in NetBeans). Regardless, they provide the following functionality.

1. They allow you to set **breakpoints** to pause the execution of a program. That is, they allow you to start executing a program but pause execution when a particular statement is reached.
2. They allow you to execute statements after a breakpoint one at a time. At the lowest level of abstraction, the user can **step into** each statement which, if the statement is a call to a sub-program, enter the sub-program and execute each of its statements one at a time. At a higher level of abstraction, the user can **step over** a sub-program call treating it as if it is a single statement (ignoring its internal details). When executing code in a sub-program, the user can **step out** of the sub-program, continuing execution of its code until it returns, then pausing again after the return.
3. They allow you to use **watches** to monitor state information. For example, as you execute individual statements you can monitor the values contained in different variables to see how they change. In some debuggers, it is even possible to set alerts for certain conditions.

The advantage of debuggers is that they allow you to trace the execution of a piece of code in all of its gory detail. This is, unfortunately, also their downfall. It can take a very long time to localize a fault. Hence, it is quite common for debuggers and debug code to be used in concert.

Performance Optimization/Tuning

For some products, though certainly not all, performance is an important measure of quality. When constructing such products (and some others as well), it sometimes becomes necessary to *optimize* or *tune* the performance of individual sub-programs and the system as a whole.

Some performance tuning amounts to little more than being careful and precise. For example, consider the following fragment in C that determines if an array named `data` contains the value named `target`.

```
found = false;
for (int i=0; i<n; i++) {
   if (data[i] == target) found = true;
}
```

This implementation loops over all `n` arguments whether or not `target` is present in the array. One could, instead, use a **sentinel** as in the following implementation.

```
found = false;
for (int i=0; !found && i<n; i++) {
   if (data[i] == target) found = true;
}
```

Another example of this is the elimination of unnecessary operations in a loop. The following fragment in Python scales x- and y-coordinates to fill a window.

```
for i in range(len(x)):
      x[i] = x[i] * (width / windowWidth)
      y[i] = y[i] * (height / windowHeight)
```

Both the readability and the efficiency of this calculation can be improved by pre-computing the scaling factors as follows.

```
xScale = (width / windowWidth)
yScale = (height / windowHeight)
for i in range(len(x)):
      x[i] = x[i] * xScale
      y[i] = y[i] * yScale
```

On the other hand, some performance tuning is based on empirical evidence, which involves the collection of data. For example, nested conditions (for example, in nested if statements) can be ordered to reduce the number that need to be evaluated based on the frequency of different data values. Other times this involves the collection of data about execution times. In general, a small percentage of the lines of code in a program are responsible for a large percentage of its execution time. Of course, the "offending" lines of code must be identified, and that can be done with a tool called a *profiler*.

Profiling is the process of measuring the amount of time (or space) used by sub-programs and statements during a particular execution of a program. Since it occurs while the program is executing, it is a form of dynamic analysis. A profiler does not provide any information about why the offending lines of code require as much time as they do or how improvements can be made; it just collects and presents the data.

The following is an example of the report generated by profiling the execution of an *n*-dimensional minimization algorithm (the Cyclic Coordinates Algorithm) that makes repeated use of a 1-dimensional minimization algorithm (the Golden Section Algorithm).

Function	Calls	Percent	Own Time	Time	Avg	Min	Max
nD_to_1D/<	7856	39.22%	87.975ms	132.862ms	0.017ms	0.014ms	0.345ms
argminGoldenSection	92	29.1%	65.274ms	220.458ms	2.396ms	1.951ms	9.082ms
anonymous	7857	20.02%	44.894ms	44.894ms	0.006ms	0.004ms	0.097ms
equals	4020	9.7%	21.757ms	21.757ms	0.005ms	0.004ms	0.058ms
CCapp.prototype.solve	1	0.7%	1.569ms	224.164ms	224.164ms	224.164ms	224.164ms
argminCyclicCoordinates	1	0.64%	1.44ms	222.588ms	222.588ms	222.588ms	222.588ms
nD_to_1D	92	0.31%	0.69ms	0.69ms	0.008ms	0.005ms	0.054ms
midpoint	92	0.25%	0.565ms	0.565ms	0.006ms	0.004ms	0.098ms
registerListeners/<	1	0.06%	0.126ms	224.291ms	224.291ms	224.291ms	224.291ms

It shows that 39% of the time is used to evaluate the function being minimized and 29% of the time is used to solve the 1-dimensional problems. Little can be done about the first, but it may be possible to tune the 1-dimensional minimization algorithm or replace it with an alternative.

Refactoring

Refactoring is changing code without altering its behavior. Refactoring is done to improve the structure, presentation, or performance of code. Because refactoring does not change the behavior of code, there must be some mechanisms to help insure this. Hence a thorough test suite in a regression testing framework must be in place before refactoring can be done with confidence.

Refactoring should be done when any of the following conditions obtain.

Duplication—Some portion of the code is duplicated. This may occur with sequences of statements (suggesting a loop, sub-program, or super-class may be needed), expressions (suggesting a function may be needed), and manipulations or declarations of variables (suggesting that collection may be needed), data (suggesting that data structures may be combined or eliminated).

Lack of Clarity—The code is hard to understand. The over-arching aim is to simplify the code as much as possible, and adding more hints for readers. This typically requires renaming variables or sub-programs, changing control structures, reorganizing control structures, adding comments, and so forth.

Code Smells—The term **code smell** is used by the refactoring community for a broad category of code defects. Code smells include comments that duplicate code, classes that do nothing but hold data, failure to hide information, tight coupling, low cohesion, bloated classes, lazy classes (that are too small), long methods, and reliance on switch statements instead of classes.

There are several common refactorings, some of which are supported in IDEs. The following are examples of common refactorings.

Rename—Give a variable or sub-program a new name. The point of this is to make the code more readable. Sometimes choosing a better name can make a comment unnecessary.

Introduce explaining variable—If an expression is long and difficult to understand, introducing a temporary variable with a good name can often help make the code more meaningful. For example, consider the following code fragments.

```
(a) if ( 0 <= (b*b - 4*a*c)) { … }

(b) var quadratic_has_real_roots = (0 <= (b*b - 4*a*c));
    if (quadratic_has_real_roots) { … }
```

The temporary variable explains what is going on in this code without a comment.

Inline temporary variable—This refactoring is the opposite of the previous one. Sometimes introducing a temporary variable can help readability, but usually these variables are not necessary and just add the cognitive load of program readers. Consider the following code fragments.

```
(a) var fraction = x%1;
    if (0 !== fraction) { … }

(b) if (0 !== x%1) { … }
```

In this case the expression is so simple that the extra variable only adds complexity to the code.

Extract sub-program—Decompose a sub-program into two or more sub-programs. There are many occasions to use this refactoring. The most obvious is when a sub-program is very long and complicated—it should be broken into simpler sub-programs. Another is when code is duplicated either in several sub-programs, or in the same sub-program, perhaps with minor differences that can be captured by sub-program parameters.

Inline sub-program—This is the opposite of the last refactoring. This refactoring should be used when a sub-program is simply too short to warrant being a separate entity. Obviously, there is a lot of judgement (and disagreement) over when this is the case.

Replace a conditional with polymorphism—A conditional that takes different actions depending on the type of an object can be replaced with code that processes objects of sub-classes of the original object, each with an overridden method that encapsulates the specialized behavior.

Make template method—Move the essentials of methods in sub-classes to a super-class, and customize the method so it uses features of the sub-classes.

Etcetera—There are many more common refactorings. Martin Fowler's book *Refactoring* [3] lists over 70 of them.

The refactoring process encourages making only small changes and then immediately rerunning all tests. This ensures that the behavior of the code is not changed. Here "small changes" means changing only a few lines of code at a time. The overall refactoring process proceeds a follows.

1. Run tests to make sure they all pass. Fix the code so it passes all tests.
2. Identify a refactoring.
3. Make a small change that gets the code closer to the refactored code.
4. Run tests and fix whatever is broken.
5. If the refactoring is not complete, goto step 3.
6. If the code needs further refactoring, go to step 2.

Test Driven Development

Traditional testing views all testing, even unit testing, as an activity distinctly separate from coding. The paradigm is that developers write code, then write tests to try to get code to fail. Treating testing as a separate activity is probably a good approach for various forms of system testing because system testing focusses on determining whether a product meets requirements. Testing a system against its requirements demands a more-or-less fully functional product to test, and it accommodates a black-box testing approach. Hence it is perfectly appropriate (and probably better) for a separate testing organization to system test a complete product.

But unit and integration testing are different. The test cases should be based on clear-box criteria, not black-box criteria. In unit testing particularly, small pieces of code should be exercised comprehensively to help ensure correctness. None of this can be done properly by anyone but the coders themselves. If the coders treat testing in a traditional manner as an after-the-fact activity, then the following things tend to happen.

- Few tests are written because the coders regard their job as finished already—testing is just a perfunctory activity to confirm what has already been accomplished.

- Poor tests tend to be written because the coders are writing to the code that they have just completed. For example, if a coder has just written code to check a parameter, then a test with a check for a bad parameter value is written. The problem is that the test will assume that the parameter will be bad in just the way that the coder is testing for, so this test is not really likely to fail.

Few and poor unit and integration tests will find few defects.

Traditional approaches to testing also do not assume testing frameworks. People used to write all tests as custom pieces of code. This was a huge amount of work. Also, regression testing was more difficult because the test code could easily become unusable as the product code evolved. Hence the entire process was slow, cumbersome, and expensive. The introduction of standard and easy-to-use testing frameworks, in particular the *xUnit* family of testing frameworks, has changed all this radically.

Test-driven development (**TDD**) is a style of coding in which writing tests is integral to the coding process itself. The TDD process results in many tests that check behavior in ways that do not depend on the code (because they are written before the code). The result is many tests of generally high quality. TDD also assumes a testing framework, so it is easy to write and execute tests, and they can be saved and rerun frequently—in fact constant regression testing is part of TDD.

The following are the essential features of TDD.

A testing framework is assumed. The framework must allow suites of tests to be written separately from product code. The framework must provide facilities to rerun all tests with almost no programmer effort and to report failures with details about expected and actual test results.

Tests are written before code. The coder first writes tests that define the expected behavior of the code that needs to be written. Then the coder writes code to pass the tests. This helps keep coders from writing poor tests that simply check the conditions that the programmer has written into the code.

Tests and code are written incrementally. In implementing a program unit, a few tests are written to check part of the function of the unit. Then code is written to pass the tests. Then a few more tests are written and code to pass them is written. This encourages coders to write extensive and thorough tests.

Code is only written to pass tests. No code is written unless doing so will help pass a test. If a coder wants to add a feature to a program unit, then she must first write a test for it. But if there is no need for the feature, then there will be no motivation to write the test. The idea of this caveat is to keep programmers from adding unnecessary features and complexity to code. Agile methodologists call this "doing the simplest thing that could possibly work."

Refactoring is expected. When code is added incrementally to enable a unit to pass more and more tests, the result may be very poorly structured. TDD encourages refactoring code to improve its structure, readability, and so forth. Note how well this approach supports refactoring: a thorough suite of tests is needed for refactoring, and that is exactly what TDD produces prior to refactoring.

The diagram below illustrates the TDD process.

As this diagram shows and as we have stressed, TDD begins with writing tests, then running them, then writing code to pass them. Once code meets all tests for the unit, then refactoring takes place using the now complete unit test suite.

References

1. Beck, K. *Test Driven Development: By Example*, Addison-Wesley, 2002.
2. Beizer, Boris. *Software Testing Techniques*, Van Nostrand, 1990.
3. Fowler, Martin. *Refactoring: Improving he Design of Existing Code*, Addison-Wesley, 2000.

CSC 131: Computer Software Engineering

California State University, Sacramento (CSUS), Fall 2018 Semester, 3 Credits

Class Times & Locations:

Section **1**: Tu/Th, 12:00PM - 1:15PM, Riverside Hall 1010
Section **2**: Tu/Th, 1:30PM - 2:45PM, Riverside Hall 5029
Section **5**: M/W, 4:00PM - 5:15PM, Mariposa Hall 2030

Your Instructor

Dr. Jingwei Yang

Homepage: http://www.ecs.csus.edu/csc/yang/

Email: yang@csus.edu

Office: Riverside Hall 3006

Phone: (916) 278-7328

Office Hours: M/W 10:30 AM – noon (subject to change)

Email Policy

Please email me directly instead of using Canvas messaging system. Please check your Sac State email at least once a day, in case I reach out to you through email. I do NOT read any emails between Fri 5PM and Mon 8AM.

Course Content

This course is about how to develop high-quality software systems that are delivered on time and within budget using modern development tools. There are two aspects to this task: managing the effort and applying effective tools and techniques. We will survey both aspects and apply them by building a system in teams of 5-6 members during the semester.

Prerequisites:

CSC 130 (may be taken concurrently)

Goals of the Course

The overall objective of this course is survey the field of software engineering. More specifically, by the end of this course you will be able to

1. Define software engineering, name several factors affecting software quality and productivity, and explain common software development processes.
2. Explain phases of a software development effort, including the activities and products of each phase.
3. Understand the differences between traditional and agile software development processes, and be able to use an agile development process (Scrum).
4. Explain and discuss fundamental software project management issues.

5. Explain and use with others in a development project common tools and techniques for planning a project, analyzing risks, estimating effort, and scheduling project work.
6. Explain and use standard techniques and tools for analyzing product requirements, formulating product design, coding, and testing a software product.
7. Recognize and read several common analysis and design notations (such as UML), and write and use these notations properly in specifying software.
8. Explain, recognize, and apply a few common software design patterns.
9. Use basic software development tools, including a modern Interactive Development Environment (IDE), a unit testing tool, a code coverage analysis tool, a version control system, and a debugger.
10. Participate effectively with others in carrying a small software development project from conception through deployment.

Required Texts

The primary textbook for this course is Software Engineering: A Practitioner's Approach, R. Pressman, 8th edition, 2014, McGraw Hill. Other reference books will all be available on either Safari books (which you can access for free through the library), or freely available on the web (in "Text" section in "Modules" in Canvas).

Attendance and Participation

Attendance is not required except at labs. However, attendance is expected in the sense that material missed because of unexcused absences will not be provided by the instructor on other occasions. In other words, there will be no private lectures during office hours for student who don't bother to come to class. All students are expected to participate in their groups during in-class activities and during class discussions. There are no participation grades, however.

Methods of Evaluation

Graded work will include a midterm and a final exam, labs, in-class activities, several homework assignments, and a team project. Assignment specifics will be posted on Canvas. The final grade will be computed as follows (subject to change):

Assignments	Weight
Labs + In-class Activities	10%
Homework Assignments	10%
Team Project	30%
Midterm Exam	20%
Final Exam	30%

At the end of the semester, a final percentage will be calculated according to the above criteria. It will then be rounded to the nearest integer value. Then, a letter grade according to the following scale will be assigned. (**No curving will be further performed**.)

Range	**Letter Grade**
93-100	A
90-92	A-
87-89	B+
83-86	B
80-82	B-
77-79	C+

Range	**Letter Grade**
73-76	C
70-72	C-
67-69	D+
63-66	D
60-62	D-
59 or Less	F

Labs

Attendance is required. Please check Canvas “Modules” for lab dates.

Homework Assignments

There will be several homework assignments. These will be written and graded individually, usually not as part of a team. Late homework submissions are usually accepted within two days after deadline, with 25% off penalty per day, unless otherwise noted.

Team Project

The team project will be done throughout the semester by a team of 5-6 people. You will be assigned to a team. You will use an agile method called **Scrum**, which we will discuss in class. In Scrum, a self-directed team creates working versions of a product about every two or three weeks in efforts called **sprints**. Your team will do at least 5 sprints, each lasting 1-2 weeks. Your product deliverables will include various Scrum artifacts as well as the software product and its documentation. You will also be asked to evaluate the contributions of all team members at the end of every sprint; these evaluations will be used to weight project grades. We will discuss details about this when we discuss the project in more detail.

Important Note: **Quality documentation is critical for the success of the project and counts toward your project grade. All work submitted must be typed.**

Exams

The midterm examinations will cover the material since the beginning of the semester, but the final examination will cover the entire semester.

Missed and Late Assignment Policy

If you are unable to take an exam at the scheduled time because of illness or other problems, you must contact me **beforehand** to arrange to take the exam at a different time. Failure to make prior arrangements for a missed exam will result in a grade of 0 for the exam.

In-class work missed because of absence will only be accepted if arrangements are made **beforehand**. Late project and homework assignments will be accepted within 2 days after due dates, with 20% penalty for each day. Alternate due dates can be arranged in special circumstances provided these arrangements are made **before** the due date.

Tentative Schedule

The following schedule is a plan, not a contract. Modifications will be posted on Canvas as the semester progresses.

Week	Topic
1 (Aug 27)	Introduction; Activity Diagrams
2 (Sep 3)	Processes

3 (Sep 10)	Scrum; Software Quality; Requirements
4 (Sep 17)	User Interface/Interaction Design; Engineering Design Principles;
5 (Sep 24)	Architecture Styles; Design Patterns
6 (Oct 1)	Design Patterns(cont.); Java Swing
7 (Oct 8)	Java Swing(cont.); Version Control
8 (Oct 15)	Verification & Debugging; **Midterm Exam**
9 (Oct 22)	Static Analysis & Refactoring; Scrum Tools
10 (Oct 29)	Deploy & Maintenance; Project Management; Sprint 1 Planning
11 (Nov 5)	Sprint 1 Review; Risk Management; Measurements; Sprint 2 Planning
12 (Nov 12)	Sprint 2 Review; Scheduling; Sprint 3 Planning
13 (Nov 19)	Sprint 3 Review; Thanksgiving break
14 (Nov 26)	Sprint 4 Planning; Software Economics; Sprint 4 Review
15 (Dec 3)	Sprint 5 Planning; Code Inspection; Sprint 5 Review, Final Product Demonstration
16 (Dec 10)	**Final Exam**

Important Dates

August 27, 2018	Instruction Begins
Aug 27 – Sept 7, 2018	Fall 2018 Late Registration and Schedule Adjustment
September 3, 2018	Labor Day (Campus Closed)
September 11 – 24, 2018	Fall 2018 Late Registration and change of schedule done by petition at departments
September 24, 2018	Census Date
November 12, 2018	Veteran's Day (Campus Closed)
November 16, 2018	**Last Day to Withdraw or Drop Courses+**
November 22 – 23, 2018	Thanksgiving (Campus Closed)
December 7, 2018	Last Day of Instruction
December 10 – 14, 2018	Final Examinations
Dec 24, 2018 – Jan 1, 2019	Winter Break (Campus Closed)
January 2, 2019	Fall 2018 Grades Due By Faculty – students will see grades posted the following day at noon

*Please note that not all important deadlines are listed on this calendar. Visit the University Academic Calendar (www.csus.edu/acaf/calendars) for additional deadlines. +See Add/Drop Form for all add/drop deadlines and instructions.

University Policies

Academic Honesty

If you violate the University's Honor Code (https://www.csus.edu/umanual/student/stu-0100.htm), you will receive a reduced or failing grade in the course, other penalties may be imposed, and the violation will be reported to the Student Conduct Officer. Automated tools may be used on any assignment, at any time, to detect inappropriate collaboration and to determine the originality of submissions.

Adding/Dropping

You are responsible for enrolling in courses and verifying your schedule on MySacState. The deadline for adding a semester course is Friday, 09/07/2018. The last day to withdraw from a course with a W grade is Friday, 11/16/2018. I do not give "Incomplete" grades to students requesting a drop after the deadline except in extraordinary circumstances.

Disability Services

If you have a documented disability and need accommodations in this course, please register with the Office of Services to Students with Disabilities (https://www.csus.edu/sswd/). They will verify your need for services and make recommendations for the course. I will be happy to discuss any accommodations I can provide to assist your learning with you.

Religious Observation Accommodations

If you cannot satisfy a requirement of the course for religious reasons you must let me know at least two weeks in advance. In some cases you will be required to make up the requirement; in other cases the requirement may be waived with suitable adjustment in grading criteria.

Excused Absences

Students who are unable to attend class due to Sac State sponsored activities (such as sports, band, academic competition, field trips, etc.) or personal religious observances may request reasonable accommodations. Please notify me during the first week of class regarding potential absences so that we can determine alternative methods for you to complete the required work.

Housing & Food Security

If you experience difficulties with financial, housing or food security, please contact Basic Needs Division of Student Affairs (https://www.csus.edu/basicneeds/) for assistance.

Parents & Families

If you are students with children, please feel free to let me know your needs. Also, please reach out to Parents & Families Division of Student Affairs (https://www.csus.edu/student/parents/student-parents/) for all resources available on campus.

Changes to this Document

I reserve the right to change any information on this document or course materials at any time.

Deployment, Maintenance, and Support

Software products are used (by the user) in what is often called the **production environment**, which is the hardware and software systems that the user interacts with through the software product. The process of making the product available in the production environment is known as **deployment**. After the software product has been deployed, the developer (or an agent of the developer) will need to help the customer use it. This kind of interaction is known as **support**. Finally, since faults may be discovered, the production environment may change, and requirements may change, the software product will need to be modified after it is deployed. This process is known as **maintenance**.

All of these activities are important aspects of software engineering that warrant some discussion. However, to understand them it is first necessary to understand physical architectures.

Physical Architectures

A **physical architecture** is the realization of a software product as artifacts (such as files) residing on and executing on computational resources. A product's physical architecture is distinct from its **logical architecture** (the software's major components and the relationships between them) which we discussed before.

Physical architectures can vary in many ways. However, software engineers often categorize them based on three characteristics—where the software is stored or installed, where the software is executed, and where the user's data operated on by the software (including configuration information) are stored. Since each of these characteristics can take one of two possible values (user device or shared device), these three characteristics lead to eight different categories of production environments, the following four of which are archetypical.

> *Personal*—The software is stored (installed) on a user device (such as a personal computer, phone, or tablet) and executed on that device. User data is also stored on that device. (If the device has multiple users, each user's data is associated with that user.)
>
> *Shared*—The software is stored on a shared device (such as a shared disk), temporarily delivered to the user device, and executed on the user device. User data is stored on the user device.
>
> *Mainframe*—The software product is stored on a shared device and executed on that shared device but is accessible from a user device (known as a terminal). User data is stored on the shared device.
>
> *Cloud*—The software product is stored on a shared device, temporarily delivered to the user device, and executed on the user device. User data is stored on the shared device.

Two of the other four possible categories involve the software being stored on the user device and executed on the shared device. Such software is sometimes called a *mobile agent*. The other two involve the software being stored and executed on one device but the user data stored on the other. While such environments exist, they are rare.

Physical architectures can be modeled using UML deployment diagrams, which contain artifacts and nodes. An *artifact* is a physical manifestation of a component of a software system. In other words, an artifact is the physical manifestation of data that is created or used during software development or operation. Examples include files, documents, source code, diagrams, audio clips, and video clips. A *node* is either a physical device or an execution environment (such as a virtual machine or an operating system), both of which are computational resources.

Artifacts are represented as rectangles (with either the stereotype «artifact» or an iconic representation of the type of the artifact) and nodes are represented as boxes (with either the stereotype «device», «execution environment», or a description of the computational resource). Both artifacts and nodes can either be types or instances; instances are distinguished from types by underlining the identifier. The identifier for a node instance is of the form *name*:*type*, but either *name* or :*type* may be omitted.

Communication between different nodes is represented using a solid line called a *communication path*. The deployment relationship (between artifacts and nodes) can be represented in three different ways—an artifact can be placed inside of a node, a node can contain a list of artifact names, or the artifact and node can be connected using a dependency arrow with the stereotype «deploy».

The following UML deployment diagram illustrates a mobile quiz system. The QuizClient, written in JavaScript, resides permanently on the Server. When the user wants to take a quiz, it is temporarily deployed to the SmartPhone. The QuizClient then makes an HTTP request to GET the Questions. The CorrectAnswers never leave the Server. Both the Questions and the CorrectAnswers are managed by a QuizData database.

Deployment

As mentioned above, deployment is the process of making the software product available in the production environment. Deployment, in its idealized form, involves the following steps.

Release—The artifacts are assembled into a distributable package.

Install—The distributable package is brought to the production environment, the artifacts are disassembled, and the artifacts are moved to the appropriate nodes.

Activate—All of the executable artifacts are started.

When the product is no longer needed it is deactivated (all of the executable components are stopped) and uninstalled (the parts, resources, or artifacts are removed from the production environment).

Atomicity and Rollback

The installation and activation steps, by their very nature, change the state of the production environment. To minimize the disruption that is caused by this change of state, it is very important that the installation and activation steps appear to be **atomic**. That is, though they will likely involve several actions that occur over time, they must appear to be a single activity. Also, it is very important to be able to *roll back* (undo) these activities, and return the production environment to its previous state.

Distributable Packages

There are a variety of different ways to create distributable packages, and they have changed considerably over time. In the early days of the software industry, the distributable package contained everything needed to build or make the product and the product had to be installed by someone with considerable technical knowledge and skill. Later, distributable packages became *installers*, executable products whose only job was to install another product. More recently, software is distributed through *stores* (e.g., the Apple Store, Play Store or Windows Store) or using a *package manager* (e.g., dpkg, fink, or RPM). There are also efforts underway to use *virtual machines* (e.g., VMWare, Virtual Box) and *containers* (e.g., Docker) to distribute software.

In any case, it is important to be able to guarantee the integrity of distributable packages; in particular, clients have confidence that a product deployed on their hardware or responsible for storing their data is trustworthy. This is often accomplished through the use of digital signatures. In some cases, distributable packages also make use of *copy protection* or *digital rights management* to protect the intellectual property in the distributable package. Obviously, this latter concern does not arise with free and open source software (FOSS).

Maintenance

Maintenance is any change to a software product after it has been deployed to a production environment. ISO/IEC 14764 identifies four different kinds of maintenance.

Corrective Maintenance—Modifications that correct faults after they give rise to failures (in the production environment).

Preventive Maintenance—Modifications that correct faults (discovered in the development environment after the product has been deployed) before they give rise to failures (in the production environment), or to correct defects of other kinds to improve maintainability, portability, and so forth.

Adaptive Maintenance—Modifications that keep the product usable in a changed or changing environment (e.g., migration to another platform).

Perfective Maintenance—Modifications that satisfy additional functional requirements (e.g., adding new features) or non-functional requirements (e.g., improving performance).

Regardless of the kind of modification, it takes some time to decide on a modification and carry it out. Hence, it is common to collect data both about the number of modifications requested and completed, and the mean or median amount of time required to complete a modification.

Though, by definition, maintenance only occurs after the product is deployed, the timing and frequency of maintenance activities will vary with the software process used. In general, agile

methods deploy more frequently than traditional methods. Hence, maintenance activities tend to occur more frequently. Indeed, some people might argue that, in an agile process, maintenance activities are indistinguishable from design and construction activities. While this might be true for corrective, preventive and perfective maintenance, adaptive maintenance activities seem to be easily identified even in an agile process. Though maintenance activities are less frequent in traditional processes, they are no less significant. In fact, many empirical studies have argued that development accounts for 20% of the total effort devoted to a software product and maintenance accounts for the remaining 80%.[1]

Maintenance is expensive for several reasons. The main reason is that it goes on for so long: a successful product may be in the field for decades with new releases occurring every year or so. Maintenance activities are bound to accrue huge costs over such a long period. But there are other reasons why maintenance is costly, including the following.

- Much software is poorly written to begin with. This makes it expensive to maintain, and the difficulty only grows as the software product gains features over time (as most do). Often products become so impenetrable that they must be completely rewritten.
- As a software system is modified over time, its structure deteriorates. This occurs at all levels of the system, including the architecture, low-level design, and the code itself.
- Maintenance done using traditional processes has the same problems as original development: the new release does not appear for a long time and only then do fundamental problems become obvious. Also, stakeholder needs and desires may have changed during the maintenance activity. Thus efforts to create new releases can fail just as development projects can, and this costs a lot of money.

Solutions to these problems are mostly obvious. Software should be written from the start to be as maintainable as possible, a point we have stressed throughout. This means making and documenting a good design, implementing it with well written and well-documented code, creating thorough tests that are maintained along with the software, and so forth. The problem of code deterioration is addressed by carefully modifying product architectures and designs over time, then implementing these changes in code that is written with the same care as original code, not just by patching here and there. Refactoring to improve code structure is an especially important tool in this regard. Finally, an organization can switch to agile processes for maintenance even though a product was developed using a traditional process, obviating many of the problems of traditional processes.

While maintenance activities are intended to improve a software product, they can interfere with the use of that product. In many cases, maintenance activities may make a software product completely unavailable to users for an extended period of time (for example, the maintenance of an online banking system). Hence, maintenance activities involve significant organizational tradeoffs and compromises.[2] In addition, as with any problem-solving situation, there is always more than one way to make a modification. At the extremes, they are often characterized as the "elegant solution" and the "quick and dirty solution," the latter usually being the one that will have the smallest short-term impact on users. Hence, technical compromises must often be made.

---

[1] A key question that often arises when considering maintenance is the replace versus maintain decision. In other words, should an existing system be maintained or should it be retired and replaced with a new system? Such questions are often answered by calculating the life-cycle cost. Such issues are discussed in the chapter on Financial and Economic Planning.

[2] In some cases, both the software architecture and the physical architecture must be designed with these tradeoffs in mind.

Maintenance activities may be performed by the original development team (which, of course, is likely to evolve over time) or by a distinct maintenance team. Obviously, while the development team will have an excellent understanding of the product, they may not understand the operational environment (e.g., business requirements, user needs) and, because of their technical orientation, may not understand or appreciate the tradeoffs. On the other hand, though a maintenance team will understand the tradeoffs, they may have limited understanding of the product and, hence, may face a steep learning curve. Also, maintenance teams may have, or perceive themselves as having, lower status than development teams. Hence, morale problems may arise and maintainer may have a higher turnover rate than developers.

Maintenance processes generally recapitulate development processes. As noted above, in agile processes maintenance requests are simply folded into the queue of pending work (e.g., the product backlog in Scrum), and in traditional processes a project is mounted to create a new release. The maintenance project is planned, runs through the traditional life cycle phases, and delivers a result just like a from-scratch development project. The only difference is one of starting point: all work products are modifications or elaborations of existing work products.

Support

The goal of the product design phase is to specify software product features, capabilities and interfaces that satisfy the needs and desires of stakeholders. However, no product design is perfect and, as discussed in the section on requirements, the needs and desires of different stakeholders may conflict. Hence, users will almost always require assistance with the product after it is deployed. Interactions between the user and the developer (or an agent of the developer) for the purpose of improving the users' experience with the product are referred to as support activities.

At the two extremes, support activities can be characterized as either professional or community.

> **Professional Support**—The person interacting with the user is employed, either directly or indirectly[3], by the developer or provider of the software product.
>
> **Community Support**—The person interacting with the user is another user or an expert who is not employed by the developer or product provider.

Of course, hybrids (a community system with a small number of professionals) are also both possible and fairly common.

Though professional support teams are employed by the developer or provider of the software product, they are almost always completely distinct from both the development and maintenance teams. There are several reasons for this. First, members of the support team tend to have less technical knowledge and fewer technical skills, and as a result, they are paid significantly less than members of the development or maintenance teams. Second, the members of the development team (and to a lesser extent, the members of the maintenance team) may not have the interpersonal skills necessary to deal with frustrated users. Third, the members of the support team are likely to have greater knowledge of the application domain.[4]

In practice, support teams tend to be segmented. Individual members of the team tend to have or develop specific areas of both technical and domain expertise. In some support organizations this

---

[3] Support activities are sometimes **outsourced**, that is, performed by a company that is employed by the developer/provider of the software product. Such a company is sometimes referred to as a **third party** since it is neither the developer/provider nor the user/client.

[4] In some situations, a member of the development team may be brought in to help with particular technical issues.

segmentation is formalized while in others it is more informal. For example, the initial point of contact may try to resolve the issue using a knowledge base of frequently asked questions (FAQ) and answers. Then, if that isn't possible, the issue may be referred to a member of the support team with more in-depth or specific knowledge.

Support activities obviously involve *communication*, that is, the transmission or exchange of information, knowledge, or ideas, by means of speech, writing, or electronic media. A *communication channel* is a particular means, either physical or logical, of transmitting or exchanging information. Some channels are synchronous and others are asynchronous. In a *synchronous* channel both parties (e.g., the user and the member of the support team) participate in the conversation at the same time and any transmission that is not immediately processed by the recipient is lost. In an *asynchronous* channel, the parties participate in the conversation over time (i.e., each transmission is stored by the recipient and processed at a later time).[5] Support activities can and frequently do use any or all of the following channels, all of which are electronic.

Telephone—A synchronous channel used to transmit voice (or voice and video).

Chat/Messaging—A synchronous channel used to transmit text (or text and pictures).

Electronic Mail—An asynchronous channel used to transmit text (or text and pictures).

Some support systems also use variants of the last two like social networking sites and WWW forms.

The channel used can have a significant impact on both the cost of providing support and the users' assessment of the quality of the support—both cost and quality tend to go down as you move down the list. Because of this, attempts have been made to reduce the cost of telephone support, including telephone trees (i.e., menu systems), voice recognition, and voice FAQs.

Support teams commonly use an **issue tracking system** (sometimes called a **call management system** or **ticketing system**) to accept, prioritize, assign, track and report on requests for support. Such systems are not dissimilar to the bug tracking systems discussed in the chapter on quality assurance in construction.

Support can either be provided for free or there can be a fee. Fee-based support activities tend to fall into one of several categories. Some organizations provide support for free. Other organizations charge per contact or per unit of calendar time (e.g., per year). Still others charge based on the amount of time required to resolve the issue, some charging per issue while others charging for blocks of time (either in advance or as you go). Organizations that charge a fee for support often provide different levels of support to different users. Such systems are said to be *tiered*. Tiers can be defined based on the hours that support is available (e.g., M-F, 9:00-5:00 vs. 24/7), the time to engage and respond (e.g., same-day vs. within five minutes), or the channel used.

Though it seems self-evident, it is worth explicitly noting that support systems should be created and put into place before the software product is deployed. Many software products have failed because of insufficient support. Less obvious, though equally important, it is necessary to estimate the cost of providing support and incorporate it into the pricing of the software product.

References

1. Lehman, M.M. (1980) "Programs, Life Cycles, and the Laws of Software Evolution", *Proceedings of the IEEE*, 68, pp. 1060-1076.

[5] Note that with both a synchronous and an asynchronous channel, the *protocol* that is used may involve turn-taking.

Software Engineering Design

Software Design Fundamentals

Once a software product is designed and specified, plans must be made to actually implement it. **Software engineering design** is the activity of specifying programs, sub-systems, and their constituent parts and workings to meet software product specifications. Software engineering design is the counterpart to software product design in that product design specifies the "external" features and behavior of a product while engineering design specifies the "internal" features and behavior.

When an architect designs a building, different drawings are prepared showing the walls, windows, doors, and so forth (often from different points of view), the heating and cooling system, the lighting and electrical system, and the plumbing system. This is because a single drawing showing all this detail would be much too hard to read. In like fashion, when software engineers design a software system, they often make models showing different *views* of the system. For example, some diagrams show the static decomposition of the software into sub-systems, packages, classes, and so forth. Others might show the state-based behavior of the system. Others might show the hardware nodes in the system and the software components that run on them. And there can be other views as well. Software engineers must be familiar with notations for making these views and the conventions and good practices in these design domains.

As discussed in the chapter on software quality assurance, defects can arise at any step in the software process, and it is important to try to prevent and eliminate them at each step of the process. Software design defects can be prevented in several ways, among them the following.

Design principles—A **design principle** is a statement that certain characteristics of a design make it better. Design principles guide choices among design alternatives by providing criteria to evaluate them. We discuss design principles below.

Design notations—Knowing design notations and how to use them helps engineering designers make good designs that are complete and consistent. The most important design notations are architectural box-and-line diagrams (showing components and their relationships), UML class diagrams (showing classes, attributes, operations, inheritance, and relationships), UML package diagrams (showing how the code for a system is grouped), UML state diagrams (showing system states and transitions between them), and UML sequence diagrams (showing messages passed among collaborating objects). We will learn about box-and-line diagrams and class diagrams below.

Design processes—There are many ways to make designs, but some approaches have proven over time to be more effective. We consider good design processes below.

Design patterns—A **pattern** is a model proposed for imitation. Architectural styles and design patterns are models of good solutions to common design problems. Familiarity with important design patterns results in better solutions, and makes design communication and documentation more concise. We discuss examples of architectural styles and design patterns below. In addition to preventing defects, patterns have other benefits as well: they make design practice easier and faster, and they promote code reuse.

The main technique for detecting and removing design defects is reviewing design specifications and correcting any defects. A particularly effective design review technique is active reviews. An **active review** is an examination of parts of a design by experts who answer questions about the design in their areas of expertise. Active reviews are effective because they employ experts who focus effort on careful and detailed study of a design. The active review process has the following phases.

Preparation—The designers choose the parts of their design that they are least happy with and that they think may contain the most important defects. Experts competent to evaluate these portions of the design are chosen, and their consent to perform the review obtained. The design team prepares questions for the reviewers about the design. These questions should force the reviewers to examine the design carefully and consider its merits critically.

Performance—The reviewers receive the design and the questions prepared for them and write answers to the questions. Although given deadlines, each reviewer is free to work on their task when and how they see fit. Reviewers are also free to work with the design team to obtain additional material or to get help understanding the design.

Completion—Once the reviewers deliver their results, the design team studies them to understand defects, problems, and suggestions from the reviewers. The designers then modify their design in response. The designers can interact with the reviewers to clarify the reviewers' comments or get elaboration about them.

Active reviews are expensive and time consuming, but they often yield valuable results. Remember that a design with major defects that are not detected until far into the implementation or testing phases will cost a lot more to fix than one whose defects are found during design.

Design Principles

As noted, a **design principle** is statement that certain characteristics make a design better. Thus, design principles can be used as quality criteria in generating and evaluating designs. Many of these principles are about modules. A **module** is some sort of program unit. Modules can be small and simple, like blocks in a control structure, or large and complex, like a sub-system of a big program. The things we usually refer to as modules are functions or sub-programs, operations, classes, packages, and source code files.

Simplicity

Simplicity is a fundamental principle in every design discipline. The principle is simply that

Simpler designs are better.

This probably seems reasonable, but the question arises, what makes a design simple? Clearly size has some effect on simplicity: designs with fewer elements in them tend to be simpler. But other factors affect simplicity as well. For example, designs with many connections between their parts tend to be more complicated. Designs with long computations involving several different kinds of objects tend to be complicated. And of course, designs with single threads of execution are usually simpler than designs with concurrent threads or processes. The ultimate test is whether a design is hard to understand. In comparing several design alternatives, if one is easier to understand than the others, then it is simpler, and by this principle better than the others.

Small Modules

The principle of small modules is that

Designs with small modules are better.

Small modules are easier to write, read, understand, and test. Many projects have guidelines about how large modules should be. For example, some guidelines recommend that classes contain no more than a dozen operations and be no more than 500 lines long, and that operations be no more than 50 lines long.

Information Hiding

Information hiding is shielding the internal details of a module's structure and processing from other modules. The principle of information hiding is that

Each module should shield the internal details of its processing from other modules.

The usual way to hide information is to declare program elements to have the smallest scope possible, and in object-oriented programming, to use visibility modifiers to restrict the accessibility of program elements as much as possible.

An instance of the first technique is the rule that variables should be declared in the innermost block possible. For example, if a variable is only used inside a while-loop body, then it should be declared in the body. An instance of the second rule is that class attributes should have private visibility in most situations, protected visibility if they are needed in descendent classes, package visibility on rare occasions, and almost never have public visibility.

Information hiding has many advantages, including the following.

- Modules that hide their internal structure and processing can be changed without affecting the remainder of a program.
- Modules that hide information are easier to understand, test, and reuse because they stand on their own.
- Modules that hide information are less likely to be affected by errors elsewhere in the program, and less susceptible to security breaches.

Information hiding is a fundamental software engineering design principle and should always be considered in design and implementation.

Module Coupling

Module coupling is the degree of connection between two modules. There are many ways that modules may be coupled, several of which are listed below.

- One class is an ancestor of another class
- One class has an attribute whose type is another class
- One class has an operation with a parameter whose type is another class
- One operation invokes another operation

The more of these sorts of connections there are between modules, and the more often they occur, the more strongly coupled two modules are.

When two modules are strongly coupled, it is difficult to understand, test, and reuse one of them without the other. Modifying one may cause problems in the other, so it is harder to change strongly coupled modules. These considerations lead to the following principle.

Module coupling should be minimized.

This principle suggests that in the best case, modules would not be coupled at all. But of course, modules must be coupled to some extent: there would be no point for modules to be in the same program if none were connected to the others in any way. Some module coupling is necessary, but we should try to couple modules as little as possible.

There are several ways to minimize module coupling. Hiding information usually helps, as does making modules cohesive (cohesion is discussed next). Another technique is **delegation**, a tactic in which one module trusts another with some responsibility. For example, suppose module A must read data from some input resource controlled by module B, and then place this data into a collection. Module A could use a loop to read data items one by one from module B until the input was exhausted, or module A could just delegate the job to module B and ask it to provide a collection filled with data from its input resource. The latter approach lessens the coupling between A and B. The use of interfaces can also dramatically decrease coupling.

Module Cohesion

Module cohesion is the degree to which the elements in a module are related to one another. Modules are cohesive when the data that they hold belong together, and the operations in the module process mainly the data held by the module. For example, suppose a class represents maintenance requests at a hospital. This class will hold data about the type of the request, the requestor, the date and time of the request, its severity, and so forth. What sort of data about the requestor should be stored? There needs to be enough data to identify the requestor (such as a name or identifier), but should the maintenance request class store the phone numbers, address, gender, title, and other information about the requestor? Clearly, this data does not belong in such a module, and including it lowers the module's cohesion. Similar considerations apply to other sorts of modules, such as functions and operations, packages, and so forth.

The principle of cohesion is stated as follows.

Module cohesion should be maximized.

Highly cohesive modules are easier to write, read, understand, test, and reuse. When modules are loosely coupled, they are usually also cohesive. Simple modules and modules that hide information well also tend to be cohesive. The main way to insure good module cohesion is to examine each of its elements and consider whether it really belongs with the rest. In general, modules that are difficult to name or that have vacuous names (like "utilities" or "miscellaneous") are not very cohesive.

Design Processes

Design is best thought of as a kind of problem solving. Like any sort of problem solving, the design process begins with **analysis**, which is breaking a problem into parts to understand it. Design analysis often employs various models. Making models for a problem often reveals details, clarifies relationships between parts of the problem, and suggests solutions.

Once a problem is understood, then various solution candidates can be proposed and investigated. Solution ideas can come from many places, including solutions to similar problems, standard solutions to classes of problems (such as design patterns and architectural styles, discussed below), and the results of various brainstorming and idea generation activities. Early solution candidates are rarely perfect; they must be evaluated and the bad ones thrown out while the best are kept. Often a promising but inadequate solution can be improved by combining it with other solutions or modifying in some other ways. In this way solutions are gradually refined and improved until a good design is produced. This is a trial and error process that repeatedly refines and evaluates candidate solutions until the design problem is solved.

The design process ends when the final solution is documented and checked. The diagram in Figure 1 illustrates this general process

Figure 1: The General Software Design Process

Different phases or activities in software design are often distinguished at different levels of abstraction. **Architectural design** is the activity of specifying a program's major parts, including their responsibilities, properties, and interfaces, and the relationships and interactions among them. For example, a network-based product might have a server component running on a server computer, and a client component running on users' computers. The two major components might themselves be divided into several components. Specifying these components, how they communicate with one another, how quickly each has to respond to the other, whether and how they preserve state between interactions, and so forth, would be an architectural design activity.

Detailed design is the activity of specifying the internal elements of all major program constituents, including their structures, relationships, processing, and often their algorithms and data structures. For examples, the detailed design of the client-server system alluded to above would include specifying the classes in each of the components of the product, along with the attributes and operations in the classes, the responsibilities of each class, the way classes collaborate to accomplish tasks, the processes and threads in the components and what runs in them, and so forth.

Top-down design begins with architectural design and moves on to detailed design. Parts of an architecture might be designed down to a low-level of detail before other parts of the architectural design are completed, or an architecture might be designed to include some existing low-level modules. This is how a design process can be partly bottom-up while being mainly top-down. We turn next to a closer look at architectural and detailed design.

Architectural Design

As noted above, **architectural design** is the activity of specifying a program's major parts, including their responsibilities, properties, and interfaces, and the relationships and interactions among them.

Several notations are used for this purpose, but perhaps the most widely used are simple **box-and-line diagrams** that show architectural components with arbitrary icons called *boxes*, and relationships and interactions between them with various connectors called *lines*. Because there is no standard for box-and-line drawings, it is important to make clear what the various boxes and lines in a diagram mean. This can be done by using common symbols (for example, using a drum icon for a data store), or by supplying a legend with the diagram. Examples of box-and-line diagrams appear later in this section.

A **pattern** is a model proposed for imitation. For example, a dress pattern is a paper template for the pieces of cloth cut out to be sewn together to make a garment. The paper templates are a model because they represent the shapes of cloth and how they must be sewn together. They are proposed for imitation by those who would like to make the garment.

Software design patterns are models proposed for imitation in solving a software design problem. Software design patterns occur at several levels of abstraction. At the architectural level, software design patterns are called **architectural styles**, which we can define as models of system components and their relationships. There are many common architectural styles; here we consider two of them.

Layered Style

The **layered architectural style** is a model in which components are arrayed linearly in ranks or levels with some "above" and some "below." These ranks or levels form the layers in the architecture. Each layer provides services to the layers above, and uses services of the layers below. The figure below shows a layered architecture in a box-and-line diagram. In this diagram (sometimes called a *wedding cake diagram*), rectangles represent layers and vertical adjacency represents the *uses* relationship between layers (that is, if layer *x* is above layer *y*, then *x* uses *y*).

Figure 2: A Layered Architecture

In this architecture for a program that controls some hardware, such as an HVAC unit, a Device Interface layer has software that directly interacts with some hardware device(s). The Device Interface layer provides a simple, device-independent interface that hides details about interaction protocols, command formats, and so forth. This simplified interface is used by the Activity Scheduling layer. This layer is responsible for determining what the hardware should be doing and when it should be doing it. For example, a smart HVAC controller might adjust the HVAC thermostat depending on the day of the week and the time of the day—the Activity Scheduling layer keeps track of the time and adjusts the thermostat (through the Device Interface layer) accordingly. The top layer is responsible for interacting with human users. It displays information and collects user inputs to direct the Activity Scheduling layer. For example, the User Interface layer tells the Activity Scheduling layer the times and days to adjust the thermostat based on input from users.

The layered architectural style solves several problems:

- The layers in a layered architecture provide interfaces at different levels of abstraction, the highest at the top, and the lowest at the bottom. The layers thus provide a means of organizing a solution by levels of abstraction.
- It decomposes the program into parts, simplifying design and implementation.
- It (should) divide the program into independent modules (the layers) with well-defined interfaces, thus increasing cohesion and helping hide information.
- It constraints the communication paths among modules, thus decreasing coupling.
- It makes it easy to replace or modify parts of the program (the layers), thus increasing maintainability and encouraging reuse.

There are some minor drawbacks to using a layered style. Often collaborations must begin at the top layer, ripple down to the bottom, and then ripple back up, which may not be very efficient. Also, it may be hard to get the layers right: if too big, a layer can become too complicated. If too small, a layer may not be cohesive. Nevertheless, layered style architectures offer so many advantages that they are very common.

Pipe and Filter Style

In the **pipe and filter architectural style** the principal participants are modules that convert inputs into outputs (called **filters**) and the channels that connect the modules (called **pipes**). Box-and-line diagrams of pipe and filter architectures typically represent the filters as boxes and the pipes as lines or arrows. What distinguishes the pipe and filter style from the layered style is that there are no limits placed on communications—the output of any filter can be piped to any other filter, becoming its input. To make this possible, all filters must accept the same type of input and all filters must produce output of that type as well.[1]

For example, consider the following pipe and filter architecture for a billing system in which the inputs to (and outputs from) all of the filters are lines of ASCII characters (that is, each line is a sequence of ASCII characters terminated by a newline character).

Figure 3: A Pipe and Filter Architecture

The Balance Finder is a filter that accepts a collection of account invoices and account payments, matches them (by account number), and determines which accounts have a remaining balance and which do not. The Receipt Generator is a filter that creates receipts for accounts that do not have a balance and the Reminder Generator is a filter that creates reminders for accounts that do have an

[1] In mathematics, a set is said to be *closed* under an operation if the result of that operation when performed on a member of the set is itself a member of that set. In a pipe and filter architecture, the set of inputs is closed under all filters.

outstanding balance. Invoices and Payments are files acting as data sources, and Receipts and Reminders are files acting as data sinks.

Pipe and filter architectures have been used successfully in a wide variety of areas. Operating systems based on Unix make extensive use of filters (the inputs to which are *streams*, which are sequences of bytes). For example, one can read a file, find all of the lines that contain the string "Overdue," and sort those lines by piping to the `cat`, `grep` and `sort` filters as follows.

```
cat in.txt | grep "Overdue" | sort > out.txt
```

The pipe and filter architectural style has two major advantages.

- It is simple and easy to understand.
- It promotes code reuse (i.e., it is easy to combine filters to provide new services).

The primary drawbacks of the pipe and filter architectural style are the following.

- It forces the lowest common denominator of data transmission.
- All data flows are unidirectional.
- There may be capacity constraints on the pipes.

There are many other architectural styles (see [1]); we have presented these two styles as widely-used examples that illustrate both a static model (a layered style architecture shows system components and relationships rather than behavior during execution), and a dynamic model (a pipe-and-filter architecture shows how data flows through components during execution).

Architectural styles are also often combined to form **hybrid architecture** displaying several styles at once. For example, the Figure 4 below shows two views of a compiler.

Figure 4: Two Views of a Compiler

The top portion of Figure 4 shows a pipe-and-filter view of a compiler. characters from a Source Code file stream into a Tokenizer that recognizes things like numbers, keywords, operators, and so on (tokens), and passes a stream of tokens to a Parser. The Parser recognizes statements, declarations, control structures, and so forth in the program and builds in internal representation called an *abstract syntax tree* (AST). The Parser passes this AST to a Type Checker that makes sure the program follows all the type rules of the language, adds automatic type conversions to the AST, and so forth. The AST is then passed along to the Code Generator that creates object code. The object code is written to an Object Code file.

An alternative layered view of the same program appears in the bottom portion of Figure 4. In this view, there are two layers in the compiler. The top layer consists of the Parser, Type Checker, and Code Generator. This layer makes use of the modules in the bottom layer, namely the Tokenizer, AST, and Machine Coder (which accepts commands to generate machine-level code and produces machine code for a particular computer). The bottom layer modules are accessible to modules that

physically touch it above. For example, the Parser access the Tokenizer and AST but not the Machine Coder.

Both of these views are useful to understand the design of the compiler and provide guidance in detailed design and coding. The compiler implements both a pipe-and-filter style and a layered style, so it has a hybrid architectural style.

Software Architecture and Non-Functional Requirements

You will recall that a non-functional requirement is a property that a software product must have. In general, non-functional requirements are constraints on the manner in which a product can behave or the way it is implemented. Typical non-functional requirements specify constraints on security, reliability, performance, efficiency, scalability, portability, or changeability. It turns out that most non-functional requirements can only be satisfied if a product has the right architecture, so architectural design focuses a lot on satisfying non-functional requirements.

To illustrate the connection between non-functional requirements and architectural design, we consider some design guidelines and practices for helping achieve two different non-functional requirements. More architectural styles can be found in several books, such as Taylor, Medvidović and Dashofy [1].

Achieving Performance Goals

The **performance** of a product is its ability to achieve computational goals without exceeding limits on time, memory, or energy. For example, a product might be required to compute an answer to a certain kind of problem within half a second, or to execute in a memory partition not exceeding one megabyte. There are several ways that architectural designers can meet performance requirements, including the following.

Use Budgets—Each component and connector in an architecture can be assigned a limit on the time, space, or energy it can use to do certain tasks. These limits are based on the total allotted to each function such that if all the components and connectors used to realize the function stay within their limits, then the entire product will meet its performance requirements.

Choose Appropriate Styles—Some styles are better for achieving performance goals than others. For example, if fast performance is a, then a layered style with many layers may not be a good choice.

Modify Styles—There are often modifications or elaborations of styles that can help meet constraints. In a layered style, special operations might be added to a low-level layer and made accessible to a high-level layer so that intermediate layers can be circumvented to speed up processing. Additionally, a cache might be added in a high-level layer to reduce interactions with lower levels to speed up processing.

Emphasize Simplicity—A highly configurable component that provides many operations to its clients, though easy to use (and reuse), may take up too much memory or be too slow to satisfy performance constraints. A simple version of the component with a stripped-down interface, though harder to use, will often be smaller and faster.

Minimize Synchronous Interactions—A **synchronous interaction** is one in which a component, after invoking an operation in another component or sending a message to another component, must stop its processing and wait for a reply. **Asynchronous interactions**, on the other hand, are those in which a component continues processing after invoking an operation in another component or sending a message to another component. If a component must complete its

work in a limited time, it is better for it to be able to continue its work while another component provides a service than to have to wait for the other component to finish.

Achieving Reliability Goals

The **reliability** of a product is the probability that it will behave as specified under normal operating conditions for a given period of time. For example, a product might have a non-functional reliability requirement that its probability of performing without failure under normal use for 24 hours is 99.9%. Among the ways that architects can make systems sufficiently reliable are the following.

Choose Appropriate Styles—Some styles are inherently more reliable than others. For example, a client-server style tends to be quite reliable because failure of all but one of the components does not cause failure of the entire system: any client can fail without consequences for the overall system. Unfortunately, failure of the server is catastrophic; however, steps can be taken to make the server sufficiently reliable.

Modify Styles—One step that can be taken in a client server style system is to modify the style to use redundant servers. Then failure of a redundant server component does not cause overall system failure.

Control Component Interactions—Interactions between architectural components should occur only through explicit interfaces. Otherwise, unexpected interactions (like side-effects or unexpected thread interactions) could cause system failure.

Handle Exceptions—Good exception handling interactions must be defined at the architectural level so that the system as a whole can (at least try to) recover from failures in its components. This guideline is connected to the last one: even if all exceptions are handled well, unexpected interactions that go around the exception handling mechanism can cause system failure.

Monitor System Health—If components run in different threads or processes, and some component becomes unresponsive, the system as a whole need not fail if a new version of the component can be started in a new thread or process. A health monitor component can constantly check on other components and restart those that fail.

Note that some of the practices and guidelines for increasing reliability conflict with those for achieving performance guidelines. For example, architectural styles that promote reliability often are not very efficient, making it harder to meet performance goals, and designs that make systems faster and smaller may be more prone to failure. This is part of what makes architectural design so much fun: the architect gets to make trade-offs between conflicting non-functional requirements.

Detailed Design

Recall that **detailed design** is the activity of specifying the internal elements of all major program constituents, including their structures, relationships, processing, and often their algorithms and data structures. Detailed design spans many levels of abstraction, from specifying several classes and their interactions to choosing data structures and algorithms. Here we focus mainly on the more abstract parts of detailed design having to do mainly with classes and their interactions.

UML Class Diagrams

Several UML notations are widely used for this sort of design.

Class diagrams represent classes, their attributes and operations, interfaces, inheritance relationships among classes, implementation relationships between classes and interfaces, and associations among class instances.

Sequences diagrams represent collaborations between class instances by depicting message passing between instances over time. They also show instance creation and destruction and the flow of control that provides the context for message passing.

Interaction diagrams represent collaborations between class instances by depicting referential relationships between instances and the message traffic supported by these referential links.

State diagrams depict the various states, or modes of being, that a class instance (or a system, sub-system, or other entity) may occupy, the transitions between states and the conditions and events governing transitions, and the actions that occur during transitions and while an entity is in a given state.

In this chapter we present class diagrams. The main symbol in a class diagram is a rectangular **class symbol** consisting of one or more **compartments**, shown as vertical divisions in the class symbol. The top compartment is required and contains the class name. The second compartment is optional and contains class attributes. A class attribute is specified by listing its name, possibly followed by a colon and its type. The third compartment is optional and contains class operations. An operation is specified by listing its name, possibly followed by a comma-separated list of argument names and types in parentheses, possibly followed by a colon and a comma-separated list of the types of values returned by the operation. Both attributes and operations can optionally be preceded by a **visibility modifier**: + for public, # for protected, ~ for package, and – for private. Static attributes and operations are underlined. Figure 5 shows examples of class symbols.

Figure 5: Examples of Class Symbols

In Figure 5, the Bridge class is shown with no attributes or operations. This does not mean that it has none, just that none are shown. The Road class is shown with (some of) its attributes, and the StopLight class with both attributes and operations. The first three StopLight attributes are marked as private and the fourth as protected. All StopLight operations are marked as public.

Interfaces in UML define sets of operations and are used to indicate that the set of operations is provided or must be provided. They are represented by class icons with the stereotype «interface» above the class name. A class that realizes or implements an interface (i.e., a class that provides all of the operations in the interface) is connected to the interface it realizes by a dashed line terminating at a solid triangle attached to the interface symbol. This connector is a variant of the *generalization* connector (UML's symbol for inheritance), which is a solid line from the child class terminating at a solid triangle attached to the parent class. Figure 6 illustrates interface implementation and inheritance relationships shown in UML class diagrams.

Figure 6: Interface Realization and Class Inheritance

This diagram shows that Road, Bridge, Intersection, Source, and Sink are all sub-classes of GridElement (i.e., they are all specializations of GridElement or, alternatively, GridElement is a generalization of each of them), and that Road and Bridge also implement the LinkElement interface.

Class diagrams may also show relationships between class instances, called **associations** in UML. These are indicated by a line between class symbols, usually labeled with the association name. Furthermore, the number of participating instances in each role of an association, called the **multiplicity**, can be shown on the diagram. Multiplicities are specified in comma-separated lists of ranges of the form $m..n$ where m and n are non-negative integers, $m \leq n$, and n may be an asterisk, meaning "arbitrarily many." If $m = n$ then the range may be abbreviated to just a single number. For example, 0..1 means zero or one, 1,3,5,7..10 means one, three, five, or between seven and 10, and * means zero or more. The association "is a biological parent of" between Parent and Child classes has a multiplicity of 2 on the Parent end (because everyone has exactly two biological parents), and 1..* on the Child end (because every Parent has one or more children). Figure 7 is a class diagram illustrating multiplicities.

Figure 7: UML Association Multiplicities

Figure 7 depicts a few classes in a traffic simulation system. In this program, an Intersection is linked to three or more Lanes, and each Lane is linked to 0 to 2 Intersections (a Lane has 2 ends that both might be linked to Intersections, or one or both may be linked to other entities). A Road contains at least one Lane and perhaps more, but every Lane is contained in only a single Road.

In some situations, it is important to indicate that two (or more) classes exhibit a part-whole (or containment) relationship. In UML, this is accomplished using an **aggregation** relationship that is denoted by a line with a diamond head (rather than a triangle head). Figure 8 is a class diagram illustrating aggregation.

Figure 8: UML Aggregation

Figure 8 depicts a model of a Document that consists of two or more Sections, each of which can contain one or more Paragraphs or two or more Subsections (which themselves can contain one or more Paragraphs). Note that the use of aggregation relationships in this example provides additional information over and above the association relationships in the previous example. Specifically, the aggregations indicate that, for example, the Document isn't just associated with Sections but it contains Sections.

Design Patterns

Architectural styles are design patterns at the highest level of abstraction in engineering design. Patterns at the level of classes and modules are simply termed **design patterns**. We will now use UML class diagrams to consider three examples of design patterns, the Composite, Command and Decorator patterns. More patterns can be found in Gamma, Helm, Johnson, and Vlissides [2].

The Composite Pattern—Many things that we model in software have a hierarchical structure. For example, documents can have parts, which themselves can have parts, and so on; windows can have parts that themselves can have parts, and so on; and processes can have parts that themselves can have parts, and so on. When building hierarchies, it must be possible for a hierarchy to have other hierarchies as parts. Furthermore, though often everything in a hierarchy must be treated the same way, at other times the things at the bottom of a hierarchy need to be treated differently. The **Composite pattern** provides a way of conveniently handling hierarchies using classes. The static structure of the Composite pattern is shown in Figure 9 below.

Figure 9: The Composite Pattern

The Component abstract class defines default operations for all components in a hierarchy. A Leaf object has no children while a Composite object is a container for children. Note that both a Composite and a Leaf can be a child of another Composite, so arbitrary hierarchies can be formed. Also, note that a Client object need not know whether a component that it has a reference for is a Leaf or a Composite because they both implement the same interface.

When a Client calls an operation() to manipulate the elements of the hierarchy, a Composite object usually just forwards the call to all its children, while Leaf objects at the bottom of the hierarchy respond directly to the method.

A good example of the use of the Composite pattern is graphical components in a GUI. All implement a Component interface with operations common to all graphical components. Some components, such as buttons and text fields, do not have any sub-components—these are leaf components. Others, such as windows, frames, or panels, are containers for other components—these are composite components. Windows, panels, and frames can be placed within other windows, panels and frames to construct complex GUI layouts. Of course, ground-level components like buttons and text fields are also placed in windows, panels, and frames.

The Composite pattern has several advantages.

- Arbitrary hierarchies can be constructed easily.
- Hierarchy details need not concern clients, which only have to deal with a Component interface.
- It is easy to add new Leaf and Composite classes into the structure without disrupting existing code.

Among the difficulties of using the Composite pattern is that Leaf components have methods that don't make sense for them, such as adding child components. In a variation of the pattern, such methods can be moved from the Component to the Composite class (where, strictly speaking, they belong), but doing so removes the uniform interface that all Components share. Some designers make one decision when using this pattern, and some make the other.

The Composite pattern occurs very frequently in a variety of applications, and it is a good pattern for designers to have in their toolkits.

The Command Pattern—In graphical user interfaces (GUIs), code must be executed when users click buttons, move scrollbars, select items from menus, and so forth. The code presenting GUI components and implementing their behavior is part of libraries or frameworks used by all programs with GUIs. How is the code implementing particular behavior associated with GUI components? The usual technique is to associate functions, called *listeners*, *handlers*, or *callbacks*, with GUI components. When some action, such as a button press, scrollbar motion, or menu selection occurs, the listener function associated with the button, scrollbar, or menu item is invoked. This is a general solution that nicely separates and decouples the general from the implementation-specific parts of user interface code, is easy to work with, and is efficient.

Encapsulating functionality and passing it to components or storing it for later use is a common idiom in many kinds of software. In many object-oriented languages, functions cannot be manipulated as pure values, so they must be implemented as operations in objects and then the objects are passed or stored. This is the basis of the **Command** pattern, whose static structures is pictured in Figure 10.

Figure 10: The Command Pattern

A ConcreteCommand class is written to encapsulate some desired functionality in its execute() operation. A Client class instance creates an instance of ConcreteCommand and passes it as the parameter of the register() operation of the Invoker class. When the time comes, the Invoker class instance invokes the execute() operation of the Command instance, putting into play the encapsulated functionality registered with it.

The Command pattern has the following advantages.

- The Client and the Invoker are loosely coupled. The Invoker knows nothing of the Client or the ConcreteCommand that it invokes, except that the latter implements the Command interface. Similarly, the Client does not depend on any aspects of the Invoker other than its guarantee to invoke the Command interface execute() operation under certain circumstances.
- The Client is decomposed and simplified because part of its functionality is encapsulated in the ConcreteCommand class.
- Command instances can be passed and stored as regular values, even if functions cannot be.
- ConcreteCommand classes can store all sorts of data and can support various operations, making them highly configurable for the Client.
- The Command interface can contain many operations to be called under a variety of circumstances, so this pattern can be generalized to work in more complex situations.
- The Invoker can support registration of several different Commands, several instances of the same Command, and Command deregistration.

The Command pattern is found throughout GUI implementations in object-oriented languages that do not allow functions as values (like Java, for example). It is also found in other event-driven contexts.

The Decorator Pattern—Sub-classes of a class add attributes and operations that will attach to all instances of the sub-class. Classes must be chosen when programs are written, so the extra attributes and operations of a sub-class cannot be added or removed at runtime (in many languages). If many different collections of attributes and operations are possible, then a complicated inheritance hierarchy must be created to accommodate all the possibilities. The **Decorator** pattern offers a way to add (and remove) attributes and operations to arbitrary class instances at runtime, and to add various collections of attributes and operations without creating a complex class hierarchy.

The structure of the Decorator pattern appears in Figure 11.

Figure 11: The Decorator Pattern

In Figure 11, the class whose instances are to be decorated is ConcreteComponent. It implements various operations signified by the operation() method in the Component interface. The Decorators all share the Component interface and record the decorated component in the attribute c. By default they simply call the operations of the component c that they decorate. However, they may also add their own attributes and operations, and override Component operations if they wish, adding processing before, after, or in place of the Component's operation processing. Furthermore, a Decorator may decorate another Decorator, so an entire chain of Decorators can be used to add various features in arbitrary ways.

To illustrate this pattern, suppose we are writing a program to support order and production of cakes at a bakery. Consider the design in Figure 12.

The BasicCake class implements the price() and invoice() methods without calling any other methods. Suppose a customer wants a cake with a happy birthday message and rosettes. We could create a BasicCake instance and a chain of Decorators to represent this order with code like the following.

```
Cake cake = new Message("Happy Birthday", new Rosettes(new BasicCake()))
```

As noted, the Decorator pattern has several advantages, including the ability to (apparently) add or remove class features at runtime and to simplify class hierarchies. Its disadvantages are that it adds lots of little classes and it may be difficult or confusing to use when there are a lot of basic components and decorators.

Figure 12: Cake and Decoration Classes

The Decorator pattern is widely used. For example, the Java IO library uses the Decorator pattern to add buffering and filtering, and to make data sources and sinks into IO streams.

References

1. Richard Taylor, Nenad Medvidović and Eric Dashofy. *Software Architecture: Foundations, Theory, Practice*. John Wiley and Sons, 2010.

2. Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Execution and Control

Execution

In traditional projects, managers are in charge of acquiring resources for a project, including hiring people and preparing them for the project. Managers also set up the work processes, quality assurance processes, data collection and reporting practices, risk monitoring and response processes, and so forth. Usually this is done by mandating certain standard processes and practices and putting people in charge of overseeing them.

In a Scrum project, new teams may be formed at the start of the project, or agile teams already formed and experienced from other projects may be brought into a new project. New teams may need to be trained in Scrum, and arranging for this falls to a manager. In an ongoing Scrum project, the overall processes and practices are already well established, although there are many details that vary among organizations. Deciding in detail how an agile method like Scrum will work is mainly left to the teams, none of whom are managers. Thus most management responsibilities for development process execution are devolved to the various players in the Scrum process.

Control

Control in Traditional Projects

In a traditional project, managers are responsible for tracking the progress of work and, if progress deviates from the plan, either changing the plan or changing what people are doing. The main way to do this is to monitor scheduled task completion: if a task is done early, then the project is ahead of schedule, and if done late, it is behind. This job is made more complex by the fact that many tasks are completed or underway simultaneously, some behind schedule and some ahead of schedule, so it is hard to get a good picture of the overall status of the project. A technique for supplying this overall status evaluation is *earned value analysis*, which uses actual and planned task completion data to compute numbers that give measures of project status. We will learn about earned value analysis later.

As noted earlier, the project management iron triangle (reproduced in Figure 1 below) imposes limits on how project plans and work can be adjusted when things have gone awry.

Figure 1: Project Management Iron Triangle

If a project is behind schedule, then scope may be adjusted by reducing the features and functions delivered. The delivery date and cost can then be maintained. Alternatively, the scope may be kept constant. If the delivery date is also maintained, then more people might be brought into the project, increasing its cost. If the delivery date is delayed, then because many costs depend on how long a project lasts, then costs will increase as well. If a project must reduce its costs, then scope can be reduced and either the delivery date moved closer or people can be let go, or both. If cost is reduced but scope is maintained, the people may have to be eliminated, which will delay delivery.

But there are limits beyond the iron triangle to the ways that adjustments can be made to heal an ailing project. A famous maxim called **Brooks' Law** states that adding programmers to a late project makes it later. The justification for this claim is that bringing new people into a project requires that the people already on the project spend time educating the new hires about the project and helping them become productive; this is especially true in projects that are nearly done because they have lots for the new people to learn, including requirements, designs, code, tests, and so forth. The time that project personnel spend helping the new hires is time that they cannot spend finishing the project, so the project ends up being even later than it otherwise would have been.

Another constraining factor is that there seems to be a limit on the time to develop any product, regardless of how many resources are provided to do it, as noted in the discussion of effort estimation. After a certain point, it does not matter how many people are set to work on a project: it won't get done any faster. This suggests that there is an optimal team size for every project (though we don't know what it is). Therefore, adding more people to speed up a project may just increase costs without making it finish sooner.

A final point about making adjustments to project plans or processes is that managers should do so in consultation with stakeholders. Most stakeholders become very upset when told without warning at the last minute that (a) the product they have been waiting so long for will not be available for a while, or (b) the product will not do what they expected it to do, or (c) the product will cost a lot more than they expected it to. When it becomes clear that a project is in trouble, mangers should present the situation to stakeholders, outline possible courses of action, and try to reach consensus about the best course. People who are party to a decision are much more likely to accept it (though they may not be happy about it), than are those who have it foisted upon them.

The difficulty of meeting the schedule, scope, and cost constraints of any project help explain why so many traditional projects have disappointing results or fail outright.

Control in Scrum

Progress in Scrum projects is also monitored by earned value analysis in the form of a burn chart. A **burn chart** is a graphic whose vertical axis is work (in person-effort units or story points) and whose horizontal axis is time. A **burn down chart** shows work remaining while a **burn up chart** shows work completed. Usually a burn down chart is made in every sprint so that progress during the sprint can be tracked. Adjustments can be made during the sprint if necessary, but process changes are perhaps best made in the sprint retrospective.

A burn up chart is usually made to track progress for the current release. The burn up chart can indicate that adjustments should to be made in the PBIs planned for the release, or in the release date. As noted before, changes to these two variables can go in either direction depending on whether the release is ahead of or behind schedule. These are changes in scope and time; the affect on cost is simple: a longer release is more expensive by the number of extra sprints times the cost of a sprint. A shorter release is cheaper by the number of fewer sprints times the cost of a sprint. We will discuss burn charts in more detail later.

Another tool used to control progress in Scrum is the **task board**. A task board allows team members to document all the tasks that need to be done to finish a sprint in one place, and then track each task as it is worked on and eventually completed during the sprint. A glance at the task board reveals how much progress has been made at any point, the tasks that are complete, those that are still not finished, and often how far along unfinished tasks are. Task boards are easy to make and maintain, and the provide more detail than burn charts. We will discuss task boards later on as well.

Earned Value Management (EVM)

Suppose that you have estimated that a job will require 40 hours of effort. After working for a while, you realize that you have put in 20 hours of effort. If you conclude from this that you are half done, then you may be wildly mistaken. For example, you may have over-estimated the effort and in fact be 80% finished after 20 hours of effort; or you may have under-estimated the effort and be only 20% finished after 20 hours of effort. Similarly, suppose that you have spent 50% of the budget for a particular project. This does not mean that you are 50% done. You may have over-estimated or under-estimated, or your spending rate may not be linear.

These problems, which are a traditional difficulty in project management, are addressed by a technique called **earned value management** (**EVM**). In EVM, *progress* is gauged by how much of the overall job is actually complete at a point in time. The *health* of the project is gauged by comparing how much you estimated you would have gotten done with how much you have actually gotten done by some point in time.

EVM measures **value** as either effort (such as person-days) or money (like dollars). Suppose that you have decomposed the work of a project into tasks and estimated the cost of each task (in effort or money). Each estimated task cost is the **planned value** (**PV**) of that task. You can also estimate duration for each task, figure out task dependencies, and use a Gantt chart of CPM to make a schedule. Then the total time for the project, called the **planned duration** (**PD**) of the project, is known as well. From the schedule it is not very difficult to figure out the accumulated PV of the project over time. Figure 2 below shows such a graph. The point on the Value axis where the PV line ends (at the PD of the project) is the **budget at completion** (**BAC**); this is, of course, the total estimated cost of the project.

Figure 2: EVM Chart Showing PV

As a project progresses, you can collect two kinds of data: the planned value of completed tasks, called the **earned value** (**EV**), and the effort or money actually expended on completed tasks, called the **actual cost** (**AC**). Note that these will often be different. For example, if some completed task

was estimated to require five person days (its PV) but actually needed seven person days to finish (its AC), then the EV of this task is five, but its AC is seven. Both cumulative EV and cumulative AC can be plotted on a graph along with cumulative PV up to the present time. Figure 3 shows an example of such a graph.

Figure 3: EVM Chart Showing PV, EV, and AC

The EV line shows that after starting out above the PV line, EV is less than PV at the present time. This means that, at the present time, it was planned that a certain value would have accrued (the PV) based on the value of the tasks that should have been done by now, but in fact the estimated value of tasks really completed at this time (the EV) is less than anticipated. So the project is behind schedule. Whenever the EV line is below the PV line, the project is behind, and whenever the EV line is above the PV line, the project is ahead of schedule.

Similarly, the AC line is above the EV line at the present time. This means that the effort or money actually expended up to now (the AC) is more than the effort or money planned to be expended for the work completed at this time (the EV). In other words, the project is currently over budget. Whenever the AC line is above the EV line, the project is over budget, and whenever the AC line is below the EV line, the project is under budget.

These conditions can be quantified using two indices.

> **Schedule Performance Index** (**SPI**), computed as EV/PV. If this index is one, then earned value is exactly as planned and the project is right on schedule. If it is less than one, then the project is behind schedule, and if it is greater than one, the project is ahead of schedule. The degree a project is ahead of or behind schedule is indicated by how far this index is from one. For example, suppose that in the project depicted in Figure 3, the current EV is 380 while the PV at this point in time was expected to be 400. Then the SPI is 360/400 = 0.9, indicating that the project is behind schedule (as also indicated by the graph).

Cost Performance Index (**CPI**), computed as EV/AC. If this index is one, then the amount spent on the project so far is exactly as much as it should be, so the project is right on budget. If this index is greater than one, then the project is under budget, and if it is less than one, then the project is over budget. Suppose that for the project in Figure 3, the AC is currently 378.94 and the EV is 360. Then CPI is 360/378.94 = 0.95, indicating that the project is over budget (as also indicated by the graph).

Besides providing numerical indice for the health of a project, these indices can be used to make predictions. The **Forecast Project Duration** (**FPD**) is computed as PD/SPI. This is an estimate of project duration based on how far behind or ahead of schedule the project is right now. Continuing the example from Figure 3, suppose that the PD is 30 months. Then the FPD is 30/0.9 = 33.33 months, indicating that if progress continues at the current rate, the project will finish three and one third months late.

The **Estimate At Completion** (**EAC**) is computed as BAC/CPI. This is an estimate of the total project cost based on how much has been spent for the amount of work done so far. Again using the example from Figure 3, suppose the BAC is 1140. Then the EAC is 1140/0.95 = 1200, so at the current rate of expenditure, the project will cost 60 units more than anticipated.

Managers can use EVM charts like the one in Figure 2 above to track whether a project is on schedule and on budget, and use FPD and EAC to make new estimates of project duration and cost. These indicators can help decide what actions to take to either bring a project back on schedule and budget (by changing scope, for example), or to revise estimates of project time and cost.

Burn Charts

Scrum projects traditionally use a simplified kind of earned value management in the form of graphs called burn charts. A **burn chart** shows value on the vertical axis against time on the horizontal axis. The value on the vertical axis can be the number of PBIs, story points, ideal days, or ideal hours, or it can be monetary units (which could be useful if business value was estimated in monetary units when determining PBI priorities). The time on the horizontal axis can be either days or sprints. To measure earned value, work is only plotted on a burn chart when a PBI is complete, and it is plotted as the PBI size estimate rather than the actual effort expended during the sprint (which would be actual cost).

At the start of a project, release, or sprint, the planned value over time is plotted (using the same units that will be used for the earned value). For example, if the number of story points is going to be used to measure the value, this curve illustrates the planned number of story points that will be completed during the project, release, or sprint. Thus the relationship between the earned and planned value lines indicates the health of the project, release, or sprint. We illustrate this below.

Two kinds of burn charts are used in Scrum. A **burn-up chart** shows how much value has been *completed* on the vertical axis against time on the horizontal axis. A **burn-down chart** shows the value *remaining* to be completed on the vertical axis against time on the horizontal axis. Burn-down charts tend to be used more often to chart progress in projects or releases, and burn-down charts more often to track progress in sprints.

To illustrate, consider the burn-up chart in Figure 4 below. This chart shows story points in a release against sprints. The chart shows the state of the project after the seventh sprint. At this time 13 sprints are planned to complete the release. The dashed-and-dotted horizontal line at the top shows that at first only about 350 story points were planned for the release. After the first sprint, many feature estimates were increased so that there were about 425 story points in the release. After the third sprint, some changes in the features for the release increased the story points to just under 450.

The dashed line shows the planned value for the project, that is, how many story points are expected to be completed at each sprint. Since story points are a measure of effort, and the team completes work at a more or less constant velocity, this line has a constant slope.

Figure 4: Release Earned Value Burn-Up Chart

The solid line with the black dots show the number of story points completed after each sprint, which is the earned value. The project started off well, with earned value above planned value for the first two sprints. But the team's velocity slowed in sprints three and four, bringing earned value below planned value (in other words, the project fell behind schedule). Velocity picked up again in the last several sprints, but not enough to make up for lost production. It may be necessary to either reduce the scope of the release by getting rid of some features (and hence story points), or adding a sprint or two to have time to finish all the features.

Figure 5 shows a sprint burn-down chart. This chart shows an entire three-week sprint. Note that there are 15 days in three work weeks, so there are 15 days on the horizontal axis. The sprint backlog started with 31 story points. By the second day of the sprint, PBIs with a total size of five story points were finished, so a point was plotted to show that 26 story points remained to complete after day two. After day three, another PBI with a size of two story points was completed, so a point was plotted showing that 24 story points remained. The other points were plotted similarly. Note that points are only plotted when PBIs are completed, and that the story points plotted are those remaining as estimated at the start of the sprint. This means that the solid line shows earned value and the dashed line shows planned value for the sprint.

Figure 5: Earned Value Sprint Burn-Down Chart

The chart shows that earned value was ahead of planned value at the start of the sprint, then fell behind planned value in the middle of the sprint, but then earned value converged with planned value by the end of the sprint. If the solid earned value line had met the horizontal axis before the fifteenth day, then all the items in the sprint backlog would have been completed faster then expected, and the team could negotiate with the PO to augment the sprint backlog. If the earned value line had not met the horizontal axis by the end of the sprint, then one of more PBIs would not have been completed, and they would have gone back into the product backlog. In any case, the number of story points completed by the end of the sprint would be the team's velocity for the sprint. From the chart above, we see that the team's velocity for this sprint was 31 story points.

In addition to plotting earned value compared to planned value, it is also helpful to track the estimated effort still remaining as work progresses. Sometimes earned value management can be misleading because it does not reflect new knowledge about the extent of the work, and because it does not count work until a PBI is completely finished. Continuously estimating effort remaining provides a means of projecting where a team will end up at the end of a sprint, which may influence the way people arrange their work. For example, a team that thinks it will not finish the work in its sprint backlog might forego some background research or other tasks to concentrate on building the product, or the team might decide to concentrate on a few higher-priority PBIs so that it maximizes the value it generates from the sprint. Consequently Scrum teams often forego earned value management when tracking progress in sprints, and instead use burn-down charts to plot estimated effort remaining.

Figure 6 shows such a burn-down chart (in which the number of ideal hours per day is constant). This chart displays the state of the sprint after 11 days. Note that the vertical axis is estimated ideal hours of effort remaining. The team estimates the effort remaining and plots a point every day. This is not as onerous as it seems, because in sprint planning the team has already generated and estimated all the tasks for each PBI in the sprint backlog. The team only needs to adjust the estimated time remaining for the tasks that it has worked on that day, and perhaps modify estimates

for tasks that it now understands to be different. In this case, the team appears to have realized on the third day of the sprint that it had considerably under-estimated some tasks, and so the estimated time remaining actually increased quite a bit that day. Despite progress, the team is clearly not on a trajectory to finish all the work in the sprint by day 15. Presumably the team would make adjustments in anticipation of not completing all PBIs in the sprint backlog.

Figure 6: Sprint Effort Remaining Burn-Down Chart

In general, earned value burn-up charts are used more for project and release tracking, while estimated effort remaining burn-down charts are used more for sprint tracking. However, either kind of burn chart can be used for either kind of tracking.

Task Boards

A **task board** is simply a board showing the current status of all tasks to be done in a sprint. In its simplest version, a task board is a portion of a wall in a public location divided into four columns labelled "PBI," "To Do," "In Progress," and "Done." The wall is also divided into rows, one for each PBI in the sprint. At the start of the sprint, all the PBIs and the tasks to be done to complete each PBI are written on cards or sticky notes. The PBI cards define rows, and the task cards are placed in the PBI rows in the To Do column. When a team member is free to work on a task, he or she takes a task card from the To Do column and moves it to the In Progress column (in the same row), perhaps writing his or her name on the card. When the team member finishes a task, he or she moves it from the In Progress column to the Done column. The figure below shows an example task board.

Figure 7: A Task Board

As this description should make clear, task boards are very easy to set up and maintain. They show the status of every task in the sprint, and a quick look at the board reveals where the bulk of the tasks are, and hence how far the work of the sprint has progressed. They are thus a valuable tool for tracking progress. They can also reveal problems. For example, if the bulk of the tasks are in the To Do column late in the sprint, then there may be a problem finishing many PBIs on time. Or if a task is stuck in the In Progress column for a while, the developer working on this task may need help.

Task boards also help solve problems. For example, a team may be able to see from the task board that it will not complete several of the PBIs in a sprint, but that it can finish some of them if it concentrates its efforts on the tasks for certain PBIs. The team may then decide to concentrate on certain PBIs to maximize the value produced from the sprint. In the task board above, the team is closer to finishing the first PBI than the second. If the end of the sprint were approaching the team could concentrate on moving the tasks in the first row across the board, thus completing that PBI by the end of the sprint.

Tasks board may be elaborated in several ways. The In Progress column may be further divided to indicate steps in task completion. For example, it is common to add a column labelled "To Verify" or "In Testing" between In Progress and Done. The PBI cards might also show estimated size, giving even more detail that might help in tracking progress or changing course.

Introduction to Software Engineering

This is a book about software engineering, so we begin by defining our topic, first by defining "software," and then "engineering." Then we will talk about why software engineering is both hard and important, its status as a discipline and a profession, and how successful software development is after almost fifty years of progress in software engineering. We finish up buy surveying in more detail the concerns that software engineers must address in their work.

What is Software Engineering?

The word "software" is relatively easy to define, but the context matters. To the hardware, software is a sequence of instructions to be executed by a processor. To the developer, software is a collection of human-readable statements (in one or more languages) that can be converted into a sequence of machine-executable instructions. Of course, human-readable statements are converted into instructions executable by a processor, so these two perspectives are strongly related. We will use the term "software" to refer to whichever of these two senses is appropriate in the context, so **software** is a sequence of instructions that can be executed on a computer, or a collection of statements that can be converted into such a sequence.

The word "program" is slightly more difficult to define now than it used to be. In the early days, the word "program" referred both to a file (or deck of cards) containing the human-readable statements necessary to perform one or more specific tasks, and to the file that contained the corresponding sequence of machine-executable instructions. Now, things are much more complicated both because some programs are compiled while others are interpreted, and because both the human-readable and machine-executable instructions are almost always in multiple files. Even worse, mostly for marketing and advertising reasons, a variety of other words have been used over the years like "utility," "tool," "script," and most recently, "app." It is almost impossible to find universally accepted definitions for any of these terms, though most people would agree that they are software and many would agree that they are programs. We will use the word **program** as a generic term for any piece of software that can run on its own.

Besides programs, software can be organized in a variety of ways. One important distinction is between programs and sub-programs and libraries. A sub-program is a collection of instructions or statements that implement an algorithm for accomplishing a specific task or tasks. A **library** is a group of related sub-programs for accomplishing a specific collection of (usually related) tasks. Programs are made up of sub-programs often combined with (parts of) libraries.

People interested in obtaining software typically want more than just programs, sub-programs, or libraries; usually they want everything that they need to solve problems with software. We call such things software products. A **software product** is one or more programs, sub-programs, or libraries, along with data and supporting materials and services, that a client can use to solve problems or achieve goals. For example, MS Word is a software product, as is the software that runs in a car to control its carburetor.

Software products are generally classified as either bespoke or generic. **Bespoke software products** are developed (usually under contract) for a specific customer. For example, a car manufacturer might contract with a software development company for the software that controls the carburetors in its cars. **Generic software products** are developed (usually speculatively) and then sold in a market (either a mass market or a niche market). The consumer products that Microsoft makes and sells are examples of generic software products.

Engineering is the application of (scientific) theories, methods and tools to the specification, design, creation, verification and validation, deployment, operation, and maintenance of products.

For example, civil engineers apply principles of chemistry and physics and methods and tools developed over thousands of years to build and maintain roads, bridges, parks, and other large public structures.

Combining our characterizations of software and engineering, we have that software engineering is the application of engineering discipline to building and software. But customers purchase software products, not just software. Furthermore, software engineering as a discipline tends to emphasize building and maintaining products more than the science behind its techniques. So our final definition is that **software engineering** is the technical and managerial discipline concerned with the systematic and disciplined development, operation, and maintenance of high quality software products delivered on time, at minimum cost. Software engineering applies scientific principles from computer science, management science, psychology, and economics to achieve its goals.

Software engineering has two main areas of concern: managerial concerns and technical concerns.

> *Managerial Concerns*—Managerial concerns are about organizing and controlling the job of building a software product. Such concerns include project cost and time estimation; project scheduling and tracking; team organization and management; risk management; and product quality tracking and improvement.
>
> *Technical Concerns*—Technical concerns are about the tools and techniques we use to determine what product to build, how to build it, and then actually building it. Such concerns include software requirements elicitation and representation techniques and tools; software design techniques and tools; programming language tools and environments; coding standards and practices; defect prevention, detection, and removal tools and techniques; version and configuration management; documentation tools and techniques; and maintenance and debugging tools and techniques.

We will look at these concerns in more detail at the end of this chapter.

Making Software Products

Suppose you get a job working for a small construction company as their first software specialist. You will be expected to build and maintain a web site, and to write programs to help out with various tasks in running the business. Of course, the company already has software for doing inventory, taxes, payroll, and so forth, but occasionally some tasks will arise for which there is no commercial product and you will be asked to develop some software. Because you are the only software developer in the company, you won't just be writing code; you will have to do all the work to develop an entire software product. This means that besides developing the software, you will have to write its documentation, set up databases or configuration files, train the users, and so forth. You will also have to work with management to provide estimates of the time and cost for developing products, report on your progress, explain delays or extra costs, and so forth.

Here are some of the questions that you will have answer to develop software products in your new job.

- What exactly should the new program do? If several people have ideas for the program, you can bet that their ideas will be different. How should these different ideas be reconciled?
- How will the program fit into the company's current ways of operating?
- How exactly should the program interact with its users?
- What parts should the program have and how should they interact? What properties must the program and its parts have? What data structures and algorithms should be used?

- What languages and systems should be used to implement the program?
- What standards and practices should be used in writing the code?
- Does the program do what you expect it to do (that is, does it have bugs)? Does the product satisfy the needs and desires of your coworkers? How will you determine answers to these questions?
- How much effort will be required to build the product? What knowledge and skills are needed? How long will it take you to build it? What other resources will be needed?
- What sorts of documentation or other user support products and services should be part of the product? How will you train your coworkers to use the new program?
- Will the product need to be modified, ported to other platforms, improved, and fixed over time? If so, how will bug reports and change requests be collected? How should the redevelopment effort be different from an initial development effort?
- During development, how will you tell whether you are going to finish on time and within budget? What will you do if things are not going well?

These are questions that must be answered when building small products. Besides these, other issues arise when large products are built. The following list summarizes these issues.

- Specifications for large programs or collections of programs are extensive (thousands or tens of thousands of requirements) and complicated (related to each other in many ways that may be hard to understand).
- Designs for large programs are big (thousands of classes and tens or hundreds of thousands of methods) and complicated (classes related to one another in many ways that may be hard to understand).
- Code for large programs is enormous (millions of lines). For example, the Linux kernel has over 13 million lines of code.
- Testing large programs is extremely difficult because there are so many possible execution paths.
- The larger a programs grows, and the more people are involved in creating it, the harder it is to estimate how much effort will be required to complete it, and to determine whether adequate progress is being made.
- As the number of people involved in a project increases, the communication paths between them increase quadratically, making communication and coordination much more difficult.
- Estimating effort, tracking progress, and communicating and coordinating, all require large efforts in big projects, so extra people (managers and support personnel) must be hired to do all this extra work, which increases the size and cost of the project even more.

Software engineering attempts to deal with the issues involved in building software products of all sizes. As these lists make clear, these issues are challenging, so software engineering is hard. Studying software engineering will make you aware of these issues, and give you the skills and tools you need to resolve them (or at least to have a good chance of resolving them). If building software is important (and the size of the software industry and the pervasiveness of software in out lives suggest that it is), then software engineering is important too.

Software Engineering as a Discipline

Though most people don't think about it much, disciplines have a tendency to proliferate. Most commonly, a discipline will split into new or more disciplines when the specialists in that discipline can no longer agree on the core knowledge and skills in the discipline.

Electronic computers were first invented and came into practical use during the second world war, when they were used to help decipher encoded German military messages. Early computers were hugely expensive and not very capable, and were mostly used to solve scientific and engineering problems. Programming them was relatively easy compared to the challenge of getting the hardware to work, and the people who worked on computers early on concentrated mostly on the hardware. But as computers grew in speed, reliability, and storage capacity through the 1950s, they began to be used for much more demanding applications, and issues about data structures and algorithms began to be very important. The people focused on the hardware began to disagree with the people focused on the software about the core of the discipline, and so the field split into computer engineering, a sub-field of electronics and electrical engineering, and computer science, which is heavily influenced by mathematics and concentrates on discrete mathematics, algorithmic thinking, and numerical methods.

Computers continued to be applied in more and more challenging domains, and by the late 1960s and 1970s practical problems of building massive programs with large teams of developers came to the fore as many software development projects were failing, or finishing late and over budget. Some computer scientists advocated the inclusion of material drawn from project management, psychology, human factors, and other disciplines in computer science, and there was once again disagreement over the core content of the field. This gave rise to the emergence of software engineering as a distinct discipline.

The birth of software engineering is conventionally dated to a NATO conference convened in Garmisch, Germany in 1968 to consider the many problems faced by large software projects. The term *software crisis* was coined at this conference to describe the inability of software developers to complete many projects successfully. The conference was entitled "software engineering," a new term chosen to suggest that software developers needed to establish the same theoretical foundations and practical techniques as traditional engineers, a provocative idea at the time.

Software engineering and computer science remain closely affiliated, and most software engineering courses are taught in computer science departments, though there are a few software engineering department and degree programs around the world. Software engineering is still an emerging discipline.

Software Engineering as a Profession

Most disciplines with "engineering" in their names are recognized professions with legal status and rigorous certification required for practitioners. Software engineering does not have this status (an embarrassing state of affairs). However, efforts over the years have produced some of the artifacts characteristic of a recognized engineering profession, such as a code of ethics, a body of knowledge, and an accepted college curriculum. A survey of software engineering is part of the computer science and software engineering curricula; this book is the text for such a course. The material in this book is based largely on the Software Engineering Body of Knowledge (SWEBOK) [1].

The State of the Software Industry

Although software engineering aims to improve software development, and it has succeeded in doing so to some extent, developing software is difficult and many projects are not successful. In

fact, the software industry continues to have many failures, some of them quite spectacular. For example, consider the following recent findings from various groups studying the software industry.

- The Standish Group, which collects data on thousands of projects annually, found that in 2012 (a) 39% of all projects finished on time, on budget and delivered expected features and functions; (b) 43% were late, over budget, or failed to deliver all expected features and functions; (c) 18% failed outright, meaning that they were cancelled or that they delivered a product that was never used [2]. More recent findings from the Standish group are similar.
- A study of 5,400 large scale information technology (IT) projects (projects with initial budgets greater than $15M) found that (a) on average, large IT projects ran 66% over budget 33% of the time, while delivering 17% less value than predicted; (b) 17% of large IT projects went so badly wrong that they threatened the existence of the company [3].
- Interviews with 600 executives overseeing software development projects in 2010 and 2011 found that even at the start of a project 75% of them expected their projects to fail. Furthermore, 78% of respondents reported that project requirements were usually or always out of sync with business needs [4].
- A General Accounting Office study found that 413 federally funded IT projects with budgets totaling at least $25.2 billion in fiscal year 2008 were either poorly planned, poorly performing or both [5]. For example, Project Railhead, an effort to upgrade the National Counterterrorism Center's (NCTC's) terrorist watch list, failed utterly at a cost of $500M [6].

As these results show, software development project failures are quite common and very costly. On the other hand, failure rates and costs have declined steadily (though slowly) since the field began in 1968, so things are getting better. Though no one can prove it, it appears that this steady improvement is due to gradual refinement of software engineering methods and techniques, and their wider adoption and application across the software industry. But problems remain for the following reasons.

- Many software project managers have not been educated in software engineering or project management—poor management is the biggest source of project failures;
- Managers are unwilling to fund and staff projects adequately, and often demand results impossible to achieve in the time allotted—this is another management failure and amounts to an unwillingness to acknowledge reality;
- Even well-planned and well-run projects fail when they encounter technical obstacles or prove too ambitious—these are technical failures.

In short, knowing and applying good software engineering practices pays off.

More About Managerial and Technical Concerns

To begin our study of software engineering, we look in more detail at the things that software engineers must address in their work, both on the managerial and the technical sides. This brief overview of the topic introduces basic terminology and frames the more detailed discussions that follow in the later chapters.

Managerial Concerns

A **project** is a one-time effort to achieve a particular goal, subject to specific time or cost constraints. For example, building a doghouse, writing a term paper, or setting up a savings plan, are all projects. Ongoing activities, or activities that do not have any constraints, are not projects. So

running a household, maintaining a car, and the pursuit of happiness are not projects. Software is almost always built and maintained in software projects.

Management is the activity of assembling, directing, and supporting human and other resources in achieving substantial goals under time and cost constraints. Management is not necessary for trivial tasks or when time and money are not at issue, but it becomes increasingly important as goals, and the time and money needed to achieve them, become larger. Both ongoing activities and projects may need to be managed, but here we focus on managing projects.

There are several things that need to be managed in a project if it is to succeed, among them the following.

Scope—The **scope** of a project is the work to be done in the project. If scope is not controlled, then project goals may not be achievable within time and cost constraints. It is easy for scope to *creep*, that is for the work to increase gradually during the project until it gets out of hand. This happens on software development projects, for example, when developers add requirements unrequested by external stakeholders because they think it will improve the product.

Time—The time for project completion is a constraint that must be set at a reasonable level given the goals of the project, or the project will fail. Project participants must monitor progress and make adjustment to ensure that projects finish on time.

Cost—Like time, cost must be set at a reasonable level given the goals of the project, or the project will fail. Also like time, project participants must monitor costs and make adjustments to ensure that projects finish on budget.

Quality—The level of quality of a product or service may be thought of as one of the goals of the project; usually a product or service is not considered acceptable if its quality is poor. We can also think of the level of quality as an additional constraint. No matter how we think of it, however, acceptable quality is not achieved unless quality is assured throughout a project.

Resources—Resources (human and otherwise) must be obtained and be ready when needed during a project, and maintained and perhaps improved over the course of the project.

Risks—Things always go wrong. Projects are more likely to succeed if project participants anticipate risks, rule them out or reduce their likelihood as much as possible, and make plans to deal with them if they occur.

A important relationship exists between the first four of these items. The project management **iron triangle** is pictured in Figure 1.

Figure 1: Project Management Iron Triangle

In a project one is attempting to achieve project goals (scope), finish by the due date (time), and finish the project on budget (cost). If scope increases (decreases), then time or cost must increase (decrease) as well. If time increases (decreases), then scope may increase (decrease) or cost may

decrease (increase). If cost increases (decreases), then scope may increase (decrease) or time may decrease (increase). Thus these three factors are strongly linked, as indicated by the lines in the triangle. It is assumed that quality will never be compromised (though it obviously can be), so it is pictured in the middle of the triangle.

Resources are closely linked with costs, for obvious reasons. Risks are a cross-cutting concern, because risks can arise with respect to scope, time, cost, quality, and resources.

Traditional and Agile Methods

Since its inception in 1968 software engineering has adopted many approaches and techniques from other disciplines, notably other engineering fields. Engineering management has typically used processes and management practices emphasizing careful planning, close oversight by managers, and thorough documentation; we call such processes and practices **traditional methods**. Since about 2000, different ideas have emerged that emphasize rapid adjustment to change, self-directed teams, and minimal documentation; these processes and practices are called **agile methods**. The field of software engineering is in the midst of a transition from traditional to agile methods. A recent large surveys found that about half of 3,000 projects used agile methods and about half used traditional methods [7]. Consequently we will discuss both because both are in use throughout the software industry.

Although there are several agile methods, the most popular is called *Scrum*. Currently about 70% of agile projects use Scrum [7] so it will be the focus our discussion of agile methods.

Management Responsibilities

Traditional project management assigns management responsibilities to individuals, called **project managers**, who are given authority to collect data and information, make decisions, and control resources. Agile methods devolve responsibility and authority to teams and rely more on processes that minimize the need for management oversight.

Traditionally, managers are responsible for the following activities.

Planning is the activity of setting goals, establishing constraints, setting requirements, determining tasks, estimating costs, formulating schedules, assigning resources, choosing quality assurance tools and techniques, and managing risks.

Execution is the activity of acquiring resources, training and educating people, establishing processes for doing work, monitoring work, monitoring and mitigating risks, and assuring quality.

Control is the activity of collecting and analyzing data, determining progress, reporting progress, and adjusting project scope, schedules, and budgets.

Leading is the activity of directing resources, motivating people, facilitating resolution of conflicts and problems, removing obstacles, and supporting people in their work.

In agile methods like Scrum, most of these responsibilities are taken away from managers and given to the team. Some planning activities are done when adjustments are made to the list of product features by the Scrum customer representative (the product owner). Scrum project facilitators (Scrum masters) perform most leadership activities. Traditional managers at levels above the Scrum team are still responsible for acquiring resources for Scrum teams, providing education and training, establishing some constraints, and setting business requirements. But everything else in the lists above is the province of the Scrum team.

We will consider roles and responsibilities in traditional and agile methods more in later chapters.

Technical Concerns

The first thing that software engineers must do when building or modifying a software product is to determine what the product must do; one cannot make something without a good idea of what to make. A software product **requirement** is some function, characteristic, or property that a software product must have. Product requirements can come from anywhere, but usually they come from whoever is paying for the product (**customers**) or whoever is using it (**users**). Other people, for example developers, managers, funders, and the public, may also have an interest in what a product does and how it does it, so all these **stakeholders** influence requirements. The development team uses various tools and techniques to elicit and record stakeholder needs and desires, record and analyze them, and eventually generate and document product requirements. We will learn some of these tools and techniques.

Having determined product requirements, developer are in a position to concoct a software system to satisfy them. A software **engineering design** is a specification of the parts of a software system, and their properties, relationships, and workings. Typically, software designs are large and complicated, so they are specified at several levels of abstraction from a variety of viewpoints. This requires various notations, several of which we will learn. Designers rely on a design process and design principles to choose between alternative deigns; we will study a generic design process and fundamental design principles. Designers also rely on standard solutions to problems, called **design patterns**, to help them make designs. We will study several high-level and mid-level design patterns.

Once software has been designed it can be implemented by writing code in some programming language. Besides the **compilers** and **interpreters** that realize languages, coders use tools to write code (**editors**), to keep track of code as it changes (**version control** tools), to make sure it follows standards and guidelines (**code checkers**), and to find and correct defects (**debuggers**). We will learn about such tools.

Quality must be assured during development. This can be done by adopting practices that help prevent defects. It also requires using various techniques to detect and correct defects in products, such as manual **reviews**, and **testing** code by executing it to see if its output is as expected. We will learn several review and testing techniques, as well as **unit testing** tools for testing code as it is written, and **code coverage** tools for ensuring most of the code in a program has been exercised by test cases. We will also study the capabilities and limitations of quality assurance activities and how to select a suite of quality assurance tools and techniques to achieve good results.

Finally, we will learn about **integrated development environments** that combine many tools into a single program that runs various tools automatically and coordinates and displays their outputs to enhance programmer productivity.

Summary

Software engineering is a discipline and an emerging profession concerned with the systematic and disciplined development, operation, and maintenance of high quality software products delivered on time, at minimum cost. As such it has both managerial and technical aspects. Managing a software project requires planning, execution, control, and leadership; we will study both traditional and agile techniques for managing a project. Technical work in a software project includes determining and recording stakeholder needs and desires, creating and documenting an overall software design, coding the product, and finally delivering and maintaining it. Along the way developers must ensure quality by both preventing defects and by detecting and removing them. We will study this development process and the tools and techniques used to make projects succeed.

Software engineering difficult and project often fail or fall short of their goals. However, application of software engineering tools and techniques has increased the likelihood of project success over several decades, so it is worthwhile to study software engineering and apply its findings in software development.

References

1. Pierre Bourque and Richard E. (Dick) Fairley, *Guide to the Software Engineering Body of Knowledge, Version 3*. IEEE, 2014. (www.swebok.org)
2. The Standish Group, *CHAOS Manifesto 2013*. February, 2013. (www.standishgroup.com)
3. McKinsey & Company, "Delivering Large-Scale IT Projects On Time, On Budget, and On Value." http://www.mckinsey.com/insights/business_technology/delivering_large-scale_it_projects_on_time_on_budget_and_on_value, accessed July 2, 2014.
4. Geneca, "Up to 75% of Business and IT Executives Anticipate Their Software Projects Will Fail." http://www.geneca.com/75-business-executives-anticipate-software-projects-fail/, accessed July 2, 2014.
5. General Accounting Office, "OMB and Agencies Need to Improve Planning, Management, and Oversight of Projects Totaling Billions of Dollars." http://www.gao.gov/new.items/d081051t.pdf, accessed July 2, 2014.
6. Patrick Thibodeau, "US Terror Threat System 'Crippled' by Technical Flaws." *ComputerWorld UK*, August 28, 2008. http://www.computerworlduk.com/news/applications/10723/us-terror-threat-system-crippled-by-technical-flaws/, accessed July 2, 2014.
7. Donald J. Reifer, "Quantitative Analysis of Agile Methods Study (2017): Twelve Major Findings." *InfoQ*, August 10, 2017. https://www.infoq.com/articles/reifer-agile-study-2017.

Software Processes

Processes and Models

A **process** is a collection of related actions that transforms a set of inputs into a set of outputs. For example, the process of making a sandwich transforms various ingredients, like bread, meat, vegetables, and condiments, into a single edible item. Steps in this process include spreading condiments on the bread, putting the meat on the bread, slicing vegetables and putting them on the bread, and so forth.

To describe a process we need to

- Specify the inputs to and outputs from the process as a whole;
- Specify that actions that constitute the process;
- Specify the inputs to and outputs from each action within the process (object flow);
- Specify the conditions under which actions occur and the order in which they occur (control flow).

There are various notations for describing processes, including UML activity diagrams [1]. For example, the UML activity diagram below shows a process for playing a round of golf.

Playing a Round of Golf

In UML activity diagrams, processes and actions are represented by rounded rectangles. Data and materials (*objects*) are represented by rectangles. An object icon on the boundary of a process icon means that the object is a process input or output. In this example, Money and Clubs & Balls are

process inputs, and Scorecard is a process output. Object icons connected by arrows to actions are action inputs or outputs. For example, Scorecard is shown as an output of the Pay Green Fee action, where it originates. It is subsequently shown as inputs and outputs of other actions, indicating that the arrows connecting these actions are object flows rather than control flows. The diamonds show where flows diverge (at *decision nodes* with boolean *guards* on their outgoing arrows) or converge (at *merge nodes* with several incoming and only one outgoing flow arrow).

Often, the actions that constitute the steps of a processes are themselves complicated enough to be regarded as processes. Hence actions in a process may themselves be shown in other activity diagrams, forming a hierarchy of diagrams that decompose a very complicated process into much simpler actions.

A **software process** is a process used in the creation or support of a software product. A **software lifecycle process** is a process depicting the steps that occur from software product inception through retirement from service—that is, a process that shows all the steps in the life of a software product. Lifecycle processes are good examples of very complicated processes with actions that are themselves processes. Variation in lifecycle processes comes not only from differences in the way their component actions are related to one another, but from the way that component actions are themselves decomposed as processes.

A **model** is an entity used to represent another entity, called the *target*. For example, a model railroad represents a real, full-sized railroad. There are many different sorts of models. For example, a parts diagram is a static 2D drawing that models how the parts of a real 3D thing look and how they go together. A model railroad is a 3D dynamic physical model of a (much larger) 3D physical thing. A graph of the equation $y = x^2$ is a 2D model of a function, which does not exist in space at all.

A **software process model** is a model of a software process. Usually software process models are static 2D diagrams (like UML activity diagrams) of the tasks of many people in the real world. They are usually quite abstract because they leave out many details. Nevertheless, software process models are very important because they either ***describe*** what actually happens in software development or (more usually) because they ***prescribe*** what is supposed to happen.

In the remainder of this chapter we present standard software lifecycle process models that prescribe what is supposed to be done when a software product is created and maintained. These models have a long history in software engineering and inform how we think about software development.

Waterfall Model

The waterfall lifecycle process model is the oldest prescription for the main tasks in the development of a software product. It was proposed by Winston Royce in 1970 based on processes in traditional engineering disciplines and it quickly became a staple of software engineering management. The following activity diagram shows a typical version of the waterfall model.

Waterfall Model

The process begins when a Product Vision document is provided to the developers. This document outlines the business goals for the product, its major features, constraints on its development, and so forth. The developers perform a Requirements action during which they create a **software requirements specification** (**SRS**) document that states in complete detail the behavior, appearance, and other properties (such as reliability and speed) that the product must have. The SRS is the input for the Design action during which the developers formulate a complete software design specification and record it in a **software design document** (**SDD**). The software design document is the input to the Implementation action during which the product's software is written. The Code is then passed on to a Testing action during which it is tested. Once Testing is complete, the finished Software Product is delivered to customers. Inevitably, defects are found and improvement suggested, so the product enters a Maintenance action that results in a new release of the product. Maintenance cycles continue indefinitely, though most products ultimately are retired from service (not shown in the diagram).

The waterfall process model is so named because work flows from one action to the next without returning to earlier actions, much as water goes over a series of waterfalls. Even projects that follow this model never wholly refrain from returning to earlier actions because it is impossible to always do every action completely and correctly the first time. However, the goal of the waterfall model is to complete each phase as thoroughly as possible so that return to earlier phases is minimized.

The waterfall model has several advantages.

- The software product is wholly specified and the project to create it is entirely planned early, so everything is predictable. This is especially valuable for large and complicated projects because managers need to know a big project's scope, delivery date, and budget up-front to make a good decision about whether to go ahead with it.
- It is easy to tell whether a project is on-time and on-budget by monitoring activity and comparing it to the plan. Managers can then take appropriate action to fix problems, adjust resource allocations, modify schedules, change scope, and adjust plans.

- Problems can (in principle) be found and corrected cheaply. For example, if a product requirement is incorrect, but it is not detected until the incorrect product specification has already been designed into the system, coded, and tested, it will be very expensive to fix because of all the effort wasted in building the mistake into the product, and all the effort required to fix it. On the other hand, if the requirements specification is done completely and correctly before any other work is done, and all incorrect specifications are found and fixed, then no time or effort will be lost.
- The waterfall model emphasizes production of complete and correct documentation at every stage. If people leave the project, or if new people must be brought in to augment the team, documentation is available to bring them up to speed.
- The waterfall model divides development work into distinct and relatively independent phases that can be performed by independent teams. This allows organizations to form teams of specialists to carry out tasks efficiently. Furthermore, because projects have detailed plans, these teams can be scheduled into different projects at the right times, maximizing resource utilization.

The waterfall model also has several disadvantages.

- The waterfall model relies heavily on being able to produce complete and correct product specifications that do not change appreciably during the project. But this is usually impossible, for many reasons that we will discuss when we consider requirements later on. Without stable requirements, all the advantages of predictability vanish.
- Even when requirements are stable, it is almost impossible to make them complete and correct. People are simply unable to envision complicated products and anticipate how all their features will work and interact, what problems will arise, and so forth. Making complete and correct designs is similarly difficult. This plays havoc with plans and leads to many defects in product specifications and designs that are not detected until much later in the process when they are very expensive to fix.
- Producing and maintaining all the documentation needed for the waterfall model is enormously expensive.
- Passing a product from team to team of specialists during development means that every team must study all the project documentation to understand what needs to be done. In contrast, a single team that carries the product through its life cycle is relieved of this heavy burden. Hence the advantages of using teams of specialists is reduced because of the communications overhead.
- The waterfall model typically uses many people in large teams who must coordinate their activities using a lot of documentation over a long period of time. This requires a lot of management. The upshot of this approach is high costs. Processes with a lot of documentation and management overhead (and attendant costs) are called **heavyweight** processes.
- The waterfall model does not deliver a product until completion of the development project, which may take several years. Often it is only then that bad product specifications, design problems, inadequate testing, and so forth, come to light. Furthermore, clients have not been able to use the product in all that time, and they may decide once they get it that they no longer need or want it.

Is the waterfall software process model a good one or not? For several decades after the birth of software engineering, the waterfall model was the only defined life cycle process. During that time, waterfall model projects succeeded about a quarter of the time, failed completely about a quarter of the time, and had middling success the rest of the time. Success rates gradually increased and failure rates gradually decreased, but even today, waterfall projects fail quite often.

As noted, one key to the success of waterfall projects is the stability of requirements: this process only works with very well-defined and unchanging product specifications. Another is the size and importance of the product: the overhead costs of the waterfall process are not justified for small and simple products or for products that do not have stringent safety, security, or reliability requirements. The consensus is that a waterfall lifecycle process should only be used for fairly large projects with stable requirements or stringent safety, security, or reliability requirements.

Prototyping

One way to help overcome some of the problems of the waterfall life cycle model is to use prototypes to explore product and design specifications. A **prototype** is a working model of some or all of a finished product. For example, product specifications might be explored by mocking up a user interface for part of a product and showing it to users. Users can then get a much better idea of what they will and will not be able to do with the product and how it will work. This helps them decide what is good and bad about the envisioned product. Also, prototypes are easy to change, so several can be put together or changed rapidly to explore product ideas. Developers can also use prototypes to try out program design alternatives to help them make good design choices.

Prototypes may be **throwaway prototypes**, which are made just to help nail down specifications and then disposed of, or **evolutionary prototypes**, which are modified into a final product. Either kind of prototype can be incorporated into any software life cycle process. Prototyping can also be used as the basis for a life cycle process, such as the one pictured below.

A Prototype Evolution Process

This process begins when the team receives a Product Vision. The team makes an initial, very simple product design for a first prototype, then implements it. The team presents the Prototype to users and collects reactions and ideas in the Demo & Evaluation step. Based on this feedback, a new cycle of Design, Implementation, and Evaluation occurs. Cycles repeat until the Software Product is complete. As we will see, this process closely resembles incremental and agile processes, which we will consider more thoroughly later.

Prototype evolution has several advantages.

- Changes to product specifications are easy to handle, and customers are more likely to get what they want.
- Customers can get useful software very quickly.
- There is typically not a lot of documentation or management oversight required (such processes are called **lightweight** processes).

The disadvantages of the prototype evolution process are as follows.

- It is very hard to predict when an adequate product will be finished and how much it will cost.
- The design of the finished product may be very bad because it may have evolved chaotically, so the product may be unmaintainable.
- There is little discipline in this process, so quality control may be lax, resulting in an unreliable or very buggy product.

Although prototype evolution is initially very attractive to customers because it is so responsive to their needs and desires and delivers products so quickly, lack of predictability, quality, and maintainability often cause products developed in this way to fail.

Risk Management

In software engineering, a **risk** is an occurrence with negative consequences. For example, losing the latest copy of a piece of software, losing a team member, finding a serious design flaw that causes a major program restructuring, or mis-estimating how long it will take to write some code, are all risks. **Risk management** is the practice of identifying, analyzing, and controlling or mitigating risks. Risk management can (and should) be incorporated into any life cycle process.

One well known life cycle process model incorporates risk as a fundamental activity. The spiral life cycle process model was proposed in 1988 by Barry Boehm [2]. The **spiral model** proposes a cycle driven by risk management. Each cycle begins with consideration of objectives of the next period of work in the project, alternative solutions (such as alternative product specifications or product designs), and constraints imposed on the project. The next step is to evaluate the alternatives in light of the objective and constraints, which leads to identification of risks. These risks are analyzed and steps are taken to remove or lessen them. These steps might involve prototyping, simulation, various requirements elicitation techniques (discussed in a later chapter), etc. Commitments are then made about the next steps in light of risks: developers might decide to create only a part of a product in the cycle, pursue a phase of the waterfall model, and so forth. The committed activities are completed to end the cycle, and another cycle of the spiral begins. The spiral model is intended to be adaptable to any other life cycle process. The figure below shows how it can be used in a waterfall life cycle process (note that not every item in the diagram must be created; for example, not every prototype in the diagram must be created).

The Spiral Model

The spiral model is important because it demonstrated the central place of risk management in software development, and it showed how to incorporate risk management into several life cycle process models. It also emphasizes the importance of tailoring development practices to the project at hand, and how to decide what should be done and what should not be done. Specifically, if a risk is likely, then work should be done to remove or reduce the risk, but if a risk is unlikely, then it should not. For example, if a particular user interface is important for the success of a product, then considerable effort should be put into its design, but if it is not very important, then little design effort should be expended.

The drawbacks of the spiral model are the following.

- It is centered on risk management, but not very many people are trained in and good at risk management;
- It is very general and adaptable, and so it demands expertise in tailoring software processes that is not very common.

Iterative and Incremental Processes

The spiral model is also important because it is an early example of an iterative process. To see this, consider the following activity diagram representation of the overall structure of the spiral model.

Activity Diagram for the Spiral Model

As this representation makes clear, the spiral model contains many tasks that are repeated. This leads to the following definition.

> An **iterative process** is one that contains one or more repeated tasks.

For example, when debugging code one typically runs some tests, makes changes to the code, reruns the tests, makes more changes, and so forth. This process is iterative.

Iterative processes should not be confused with incremental processes.

> An **incremental process** is one that produces its output in parts or phases. An incremental process produces its output a bit at a time.

For example, when a customer orders the breakfast special, the chef can first cook the eggs, then cook the bacon, and then cook the hash browns. Alternatively, the chef can cook the eggs, bacon, and hash browns at the same time. The first process is incremental while the second is not.

Any software processes can be made iterative, and most are "naturally" iterative because it is difficult to get correct result after only one try (any process that works by trial and error is iterative). The waterfall process model is ideally non-iterative, but in practice, a certain amount of iteration is inevitable. The prototype evolution process is iterative, and the spiral process is iterative in its overall structure, particularly with respect to risk management.

The advantage of iteration is that it is the primary means of achieving work products of sufficient quality. As noted, it is usually very difficult to produce adequate specifications, design, code, documentation, and so forth on the first try, so iteration is necessary. The drawback of iteration is

that it so often involves **rework**, which is discarding or redoing previous work products. Rework is a major source of expense and delay in any process, and eliminating it is always desirable.

Software processes are sharply distinguished by whether they are incremental. The waterfall process model is not incremental while the prototype evolution process is incremental. The spiral process model, since it may be applied to any base process, is not inherently incremental, but may be used with an incremental process.

To understand why the waterfall process is not incremental, it is important to understand the difference between what economists call *intermediate* and *final goods*. Each action in the waterfall process produces an output, but the product specification and the software design are intermediate goods that are used in the production of the final good, the software product. So, while there are multiple actions each of which produces an output, one could not deliver the product specification and claim to have delivered a version of the software product. The cooking example might help clarify this distinction. While a restaurant using an incremental process to deliver breakfast to its customers might not make them happy, it can claim to be producing the breakfast in phases. In other words, the eggs alone can be considered a version of the breakfast.

It is important to note that the distinction between incremental and non-incremental processes is not always clear and depends on what constitutes the final product. Returning to the cooking example, can a sandwich without bread be considered a version of the final product?

An important iterative and incremental process is the unified process from Rational Software (now part of IBM) [3]. The **Rational Unified Process (RUP)** produces software products in parts called *releases* created during a *cycle*—each release is a product increment. The RUP is incremental because each release can be considered a final good (as opposed to an intermediate good). Each cycle has four phases called *inception*, *elaboration*, *construction*, and *transition*. Each phase is divided into iterations, and work in each iteration is divided into five workflows called *requirements*, *analysis*, *design*, *implementation*, and *test*. The work in each phase is summarized as follows.

Inception —The product vision and business case are developed. Preliminary work is done on the product specifications and design.

Elaboration —The product specifications are set out in detail along with the high-level design of the system. Code is written and tested for core functions. Plans for the rest of the cycle are made.

Construction —The bulk of the code is written and tested. Designs are elaborated as necessary and corrections are made where needed.

Transition —The product is tested with customers and made ready for release.

The phases can be represented in the following activity diagram.

Actions in the Rational Unified Process

Note that this diagram has a loop showing the cycles (increments) during which releases are produced. In addition, there are iterations during each phase that are not shown in this diagram. The RUP workflows are also not shown in this activity diagram.

The unified process is depicted below in a diagram showing phases, cycles, workflows, and iterations.

Workflows in the Rational Unified Process

The humps in the diagram show roughly how much effort is devoted to the various workflows during each phase.

The unified process has the following advantages.

- It is fully specified and comes with templates, checklists, and other process aids, and it incorporates best practices, such as risk management, so it is a complete package that developers can use to manage their life cycle process.
- It is adaptable to projects of different sizes with different needs.
- Because it is incremental, it lessens the risk of project failure, and it delivers usable product versions to customers fairly quickly.
- Because it is iterative, work is easier to schedule and monitor, work products can be modified to make them better, and risk management is supported.

The disadvantages of the unified process are the following.

- The unified process is complex and requires knowledgeable and experienced developers to use it properly.

- Although it is adaptable to different kinds of projects, it tends to impose a lot of documentation and management effort (it is a heavyweight process).
- Requirements change is not well-supported because development cycles tend to be fairly long, and requirements are assumed to be stable during a cycle.

The unified process has been adopted by many organizations successfully. It has also been the basis for many other life cycle process models that modify it in various ways.

Agile Processes

The waterfall life cycle process model, and various elaborations and refinements of it, dominated practice in software development from the 1970s through the 1990s. By the end of that period, it was clear that although there had been steady improvement in project outcomes, project failures rates were still too high and customers (and many developers) were very dissatisfied with the state of the software development enterprise. A group of influential developers decided to change this state of affairs and proposed a radical departure from traditional practice. Their philosophy was proclaimed in the *Agile Manifesto* [4]. The manifesto stated that its authors "had come to value

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan"

In other words, the writers of the *Agile Manifesto* wanted to change software engineering practice so that products could be created quickly and cheaply by streamlining development processes and practices and working more closely with customers. They also wanted to emphasize the "human" aspects of software development, taking into account teamwork, interesting work, and work-life balance.

Several agile methods have been proposed, the best known being Extreme Programming (XP), The Chrystal Methods, Dynamic System Development Method (DSDM), and Scrum. All these methods have similar life cycle process models with the following characteristics.

Incremental—All use an incremental process with short increments, ranging from a week to a few months. Short increments allow rapid response to changes in customer needs and preferences.

Customer involvement—All build-in close and continuous involvement with customers as a way to track customer preferences and promote informal communication.

Lightweight—All try to avoid documentation and project management tasks as much as possible to speed up work and avoid rework.

Test Driven—All try to avoid the problems that might arise as a result of frequent changes by using test suites that are created early in the process and maintained throughout.

In addition to their distinctive life-cycle processes, agile methods utilize many practices that accommodate change, impose discipline on work, and promote quality. We will discuss these practices later on when we discuss one agile method, Scrum, in detail.

The following diagram illustrates a generic agile process.

A Generic Agile Process Model

The process begins by formulating initial product specifications based on the Product Vision in the Refine Specifications activity. These specifications are the basis for designing, implementing, and testing a version of a Deliverable Product. This version is then evaluated by customers and other stakeholders in the Evaluate Increment activity. The work done to create the increment is also considered and used to make changes to the process in the Improve Process activity. If the Deliverable Product is deemed ready to release, then it is delivered as the Software Product. Otherwise, the Deliverable Product and the results of its evaluation are used to Refine Specifications and another cycle to produce a new product increment is begun.

Agile processes have many advantages, including the following.

- Product specifications can be changed frequently without disrupting the process.
- A version of the product is delivered to customers soon after development begins, and new versions with gradually increasing capabilities can be delivered frequently, if desired.
- Bad projects can be recognized and cancelled early.
- The process is lightweight, so much time and effort is saved.
- Waste and duplication of effort are usually greatly reduced.

Of course, agile processes have disadvantages as well.

- Customers and users must be involved throughout development but it is often difficult to get customers to commit so much time and effort.

- Although continuously refining the product design is a common agile practice, designs developed incrementally may not be very good, degrading product quality and increasing development effort.
- Agile processes are difficult to use on large projects because it is hard to coordinate the activities of many teams that are evolving different parts of the product in unpredictable directions, especially since minimal documentation is used.
- Although agile methods include planning activities, it is harder to predict the outcomes of agile projects.

Agile processes have grown in popularity since 2000, and they are on the verge of replacing the waterfall process as the dominant process used by developers. We will discuss Scrum, currently the most popular agile method, as our paradigm agile method in a later chapter.

References

1. Pilone, Dan. *UML 2.0 in a Nutshell, 2nd Edition*. O'Reilly, 2005.
2. Boehm, Barry. "A Spiral Model of Software Development and Enhancement," *IEEE Computer* 21(5), May 1988, pp. 61-72.
3. Jacobson, Ivar, Grady Booch and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
4. *The Agile Manifesto*. http://www.agilealliance.org/the-alliance/the-agile-manifesto/, accessed July 17, 2014.

Software Product Requirements

Before we can design and build a software product, we must know what it is we are trying to build—or at least, we must know enough about it to begin building it, even though many details may still not be clear. This chapter discusses requirements for software products, which we will often just call *software requirements*, or more often simply *requirements*.

A **requirement** is a property or behavior that something must exhibit—it is something demanded or necessary. A **specification** is a precise description of something. Thus a **requirements specification** is a precise description of properties or behaviors that something must have. For example, a credit card company's web site must show users how much their balance is—the software product *must* display this information, so this is a requirement. A statement to this effect, such as "The web site must display the user's current balance" is a requirements specification. Usually we will not take pains to distinguish requirements from the specifications that express them, and this distinction is often blurred or ignored when requirements are discussed, but you need to understand it.

Required According to Whom?

Requirements are statements about characteristics that a product must have, but who gets to decide this? It seems clear that the people and organizations affected by a product ought to have something to say about it. A **product stakeholder**, or just a **stakeholder** for short, is anyone affected by a product or its development. Most software products have quite a few stakeholders, including the following.

Customers are people or organizations that pay for a product.

Users are people who directly interact with a software product. Often users are not the ones who pay for software products, but they are often the most affected by them.

Clients are people or organizations on whose behalf software products are created. Clients are customers, users, or both; this is a blanket term for customers and users.

Developers of all kinds, including requirements engineers, designers, programmers, and testers, as well as the people who manage and support the developers, are all stakeholders.

Regulators of all kinds responsible for ensuring that software products meet various standards are stakeholders. For example, governments typically have all sorts of regulations to which software products and their development must conform.

Marketers act as stand-ins for clients when mass-market products are created. Marketers also often represent clients when they cannot participate closely in product development.

Even the public can be a stakeholder. For example, software for flying planes or assisting air traffic controllers can affect almost anyone if it results in airplane crashes. Government regulators usually represent the interests of the public.

Stakeholders have a say in the requirements for a product, but they can't have the final word, for several reasons. First, various stakeholders are almost guaranteed to have conflicting desires and ideas about a new product. For example, consider the requirements for an online college course registration system. Students (one kind of user) might want a system that makes it easy to see who has registered for classes so that they can take classes with their friends, classes at the same time as other students who can give them rides to campus, and so forth. The registrar (as a regulator) has to ensure student privacy, which includes class enrollments, so the registrar does *not* want students to see who is enrolled in classes. These sorts of conflicts are frequent.

A second reason that stakeholder desires cannot simply be treated as requirements is that they are invariably incomplete and sometimes even incorrect. Stakeholders typically have vague ideas about what they want a system to do, but very rarely do they have lists of specific properties, characteristics, and features produced with thoroughness and care. Producing such lists is hard work, takes a long time, and employs techniques to help ensure that nothing important is missed. Furthermore, stakeholders often think they understand what they need a software product to do, but it turns out they are wrong. People are often mistaken about their own business processes and what their co-workers actually do. It is not unusual for clients to learn a lot about their own organization and its processes when they help developers produce requirements.

A third reason that stakeholders cannot have the final say about requirements is that their desires and ideas tend to be too abstract to serve as the basis for software design and implementation, and stakeholders typically do not have the necessary knowledge and skills to add the needed details. For example, suppose all stakeholders for a healthy lifestyle web site agree that users must be able to enter information about themselves (like sex, height, weight, blood pressure, exercise habits, etc.) and in response receive recommendations about how to live a healthier lifestyle. But how should the web page be laid out? How should the prompts and responses be worded? What should the steps in the user interaction be? All of this must be determined if code is to be written, so these are vital requirements. Settling these questions calls for user interaction design expertise that few stakeholders possess.

In summary, stakeholders clearly have a say in requirements, but they cannot have the final say. We can clarify the situation by distinguishing between *stakeholder needs* and *requirements*. A **stakeholder need** (or just a **need**) is a feature, function, or property needed or desired by one or more stakeholders. Stakeholder needs reflect what various stakeholders think the product should be and what it should do. As noted above, a collection of stakeholder needs will typically be inconsistent, incomplete, abstract, and sometimes will reflect an incorrect understanding of the situation. Stakeholder needs must be adjusted, augmented, refined, and corrected to produce software product requirements acceptable to stakeholders that can serve as the basis for development.

Like requirements, needs are expressed in detailed descriptions, or **needs specifications**. As noted, often needs are quite vague or abstract, so needs specifications may be descriptions without a lot of detail, but we call them specifications anyway to keep our terminology uniform.

Developers are responsible for transforming needs into requirements and getting all the stakeholders to agree to them. In traditional development processes, certain developers specialize in this job and have job titles such as *requirements analyst*, *requirements specialist*, or *user interaction designer*. In agile methods, teams are cross-functional, so transforming needs to requirements may be done by anyone on the team. Nevertheless, many agile teams have members specializing in this job.

A collection of stakeholder needs poses the problem that requirements analysts must solve. Specifically, the requirements analyst must formulate requirements that resolve the conflicts and correct the errors in stakeholder needs in a way that satisfies stakeholders as much as possible and specifies a good product in sufficient detail to be implemented within the project budget and schedule by the developers available. Thus a collection of needs specifications describes a problem, and a requirements specification describes a solution to this problem.

To recap, stakeholder needs are features or properties that particular stakeholders desire in a product, and requirements are product features or properties agreed by all. Developers work with stakeholders to understand their needs and transform them into requirements that all stakeholders agree to. Who sets product requirements? Stakeholders set the requirements, but only with the help and guidance of product developers.

Functional and Non-Functional Needs and Requirements

Requirements specifications must describe all sorts of things about software products if the right product is to be created by its developers. Commonly, needs and requirements are divided into two broad categories, as follows.

> **Functional needs and requirements** are about how a software product must map inputs to outputs. In other words, functional specifications stipulate the *behavior* of a software product. This category is very broad—it includes such things as the appearance and actions of the user interface; the way data is read, modified, stored, transformed, and output; what a program does in response to the passage of time; and so forth. The bulk of product requirements are functional requirements.
>
> **Non-functional needs and requirements** are about *properties* that a software product must have. Non-functional specifications include stipulations about speed of processing, amount of memory used, communications bandwidth used, frequency of failure, level of security, ease of modification, cost of development, platforms where the product must run, and so on.

Levels of Abstraction in Specifications

Consider the following specifications for a software product.

> The product must provide an online bookstore shopping experience.
>
> The product must provide titles and descriptions of books from which users may choose books to purchase.
>
> The product must allow shoppers to buy books by placing them in a shopping cart.
>
> The product must display a button with each book description that when pressed places books in the shopper's cart.
>
> The product must display a button labeled "Buy" that maintains a constant position (despite scrolling) at the upper right-hand corner of a book-description web page. When pressed, this button must place the book whose description is displayed into the shopper's cart.

Notice that these specifications describe the product at different levels of abstraction, each with more detail than the one before. Note too the huge range of abstraction comprehended by product specifications, from those like the first that encompass entire systems, to those like the last that describe the smallest implementation detail. When working with needs and requirements, developers typically begin with quite abstract specifications for whole products and gradually refine them by adding more and more detail until they are specific enough to implement in programs.

Software engineers have terms for specifications at various levels of abstraction. **Business requirement specifications** are statements of client or development organization objectives that a product must meet. For example, the first specification above states a business requirement. Other examples include the following.

> The initial version of the product must be delivered by May first of this year.
>
> The product must generate sales of one million dollars during its first year of operation.
>
> The product must reduce the cost of order fulfillment by half within a year of delivery.

Business requirements tend to be very abstract, particularly about what a product actually does, but more specific about what benefits the product is expected to provide for the organization or the client. Business requirements usually are based on needs from marketing or management

stakeholders. They also tend to be non-functional requirements. In some sense business requirements are often hardly software requirements at all because they specify characteristics not definitive of particular products, such as sales, profit, or market penetration goals. Nevertheless, they provide context for the software product that may have consequences for more specific requirements.

One rung down on the ladder of abstraction from business requirements are **user-level needs and requirements**. These are about how a product must support user needs (hence the name), though in fact needs of other stakeholders besides users may appear at this level of abstraction as well. Typically user-level needs and requirements indicate that a product must enable users to achieve certain goals or complete certain tasks. Some examples are the following.

> The product must produce earning reports suitable for financial auditing.
>
> The product must allow personnel clerks to set employee salaries and wages.
>
> The product must allow employees and their managers to view customer evaluations of their performance.
>
> The product must authenticate all users and assign them roles governing access to information.

Notice that user-level specifications may still be quite abstract, but they do indicate particular product properties or behaviors. Notice too that they may be functional or non-functional specifications.

Operational-level needs and requirements are about individual inputs, outputs, computations, operations, characteristics, or properties of a product. These can be quite specific. The following are examples of operational-level specifications.

> The product must allow users to enter polynomial equations, trigonometric equations, logarithmic equations, and exponential equations, all of one variable.
>
> The product must support the six standard trigonometric function, logarithmic functions with arbitrary bases, and exponential functions with arbitrary bases.
>
> The product must display two-dimensional graphs of entered equations.
>
> The product must display a graph of any well-formed equation in no more than a fifth of a second.

Finally, we can distinguish **physical-level needs and requirements**. These are about details of the physical layout and behavior of the product's user interface, the format of input and output data, and other low-level details. The following are examples of physical-level specifications.

> The product must provide a text-box for equation input. The text box must display 50 characters and scroll horizontally up to 800 lines.
>
> The product must support cut-and-paste of text into all text boxes.
>
> The product must display numbers rounded to three decimal places with terminal zeros truncated. If a number has no decimal digits then the decimal point must not be displayed.

There are no hard and fast distinctions between levels of abstraction in specifications. There is nothing to be gained by trying to classify particular examples of requirements or needs statements as being user-level, operational-level, or physical-level specifications. We distinguish levels of abstraction in specifications to illustrate and emphasize the point that specifications are stated across a broad range of levels of abstraction. Furthermore, needs and requirements are usually conceived and stated at high levels of abstraction and refined over time to lower levels of abstraction. This is

an essential, important, and expensive task in software development requiring considerable skill and quite a lot of time. We discuss refinement further below.

It is also important to appreciate that requirements are realized at the physical-level even if they are never explicitly refined to that level. It may be the case, for example, that no stakeholders particularly care exactly how many characters are displayed in a text-box, or what font is used in a button label, and so no one bothers to include such things in the requirements for the product. But all these details must be decided when software is actually written, so someone (perhaps the programmer when she is actually writing the code) *will* decide on all requirements at the lowest level of abstraction.

Requirements in Traditional Processes

The framework and terminology we have been discussing so far reflect the traditional approach to software requirements that has been used in software engineering for decades. The context for this approach is some form of the waterfall life-cycle process in which the work for a particular phase of development is mostly done at one time. The first thing that one needs to do is understand what to develop, so stakeholder needs and product requirements are determined first. In the waterfall model, *all* the requirements for the product (or as near as possible) are determined before almost anything else is done. Once set, (or **frozen**, as is typically said) requirements are difficult and expensive to change, so requirements change is resisted.

Unfortunately, this has often not worked well, for the following reasons.

- It is very difficult (perhaps impossible) to determine all requirements for non-trivial products before any implementation is done. Stakeholders have a hard time even figuring out their needs without having some idea of what the product will do and without seeing early versions of it. Analysts likewise find it difficult to choose requirements without exploring alternatives as they are realized in the product.
- The world changes rapidly, and so do stakeholder needs and desires. Making requirements impossible or hard to change leads to products stakeholders don't want when they are delivered.
- Writing all the requirements for a significant product is an enormous and difficult job, but it must be done if developers won't be implementing the product until well after requirements are formulated—there is no way that anyone could remember so many details. Hence waterfall approaches necessitate large, unwieldy requirements documents that cost a lot of time and money to produce.
- Following a waterfall process usually means that a product is not delivered for months or years after its development project is started. Many projects are cancelled or never even get started because of this huge delay. Developers also find this delay demoralizing, which damages their productivity.

These problems have posed difficulties for software engineering since its beginning. Many ways of addressing these problems have been tried, such as making heavy use of product prototypes so that stakeholders and analysts can get a better idea of what a product might do and how it might work as they formulate requirements. Another approach has been to develop products in smaller increments so that the time between project start and product delivery is reduced; a product with minimal function is delivered early and then evolved into a full-featured product over time with several releases.

Requirements in Agile Processes

Agilists try to avoid writing traditional software requirements specifications altogether as a way of solving the problems with traditional requirements specifications discussed above. However, as noted, requirements *must* be present, even if only implicitly, before code can be written. Scrum uses several approaches to make requirements specification and change as easy and cheap as possible, and we will discuss them now.

Scrum projects have stakeholders with needs and desires just like all other projects, and they begin with business requirements like other projects. But rather than proceeding from these to user-level requirements that are then elaborated into operational-level and then physical-level requirements recorded in a large requirements specification document, Scrum projects have lists, called *product backlogs*, that guide development. Product backlog items are added based on stakeholder needs and desires. A person called the **product owner** acts as a stand-in for the product's stakeholders and adds backlog items and sets their priorities. The highest priority items are selected for implementation in each development iteration, or **sprint**. Product backlog items are added, removed, changed, and reprioritized by the product owner continuously, thus altering the direction of development. The fluidity and ease of change of the product backlog is a large part of the justification for calling Scrum an "agile" method.

Items added to the product backlog capture needs and desires of various stakeholders, so they express stakeholder needs. Product backlog items are prioritized in terms of the value they add to the product. Thus a **product backlog item** (**PBI**) is a statement of a stakeholder need, and the **product backlog** is a prioritized list of unimplemented stakeholder need specifications. When a PBI is selected for implementation in a sprint, it is because the item has high priority and the decision has been made that it is time to incorporate it into the product. This decision is tantamount to the determination that a stakeholder need is of sufficient importance, and that its inclusion in the product is sufficiently valuable, that the need should become a requirement. PBIs selected for implementation go into a **sprint backlog**. Thus a **sprint backlog item** is a requirements specification.

With this brief introduction to the Scrum product backlog, we can catalog ways that Scrum makes requirements specification and change as easy and cheap as possible.

> *Delay choosing requirements as long as possible*—Stakeholder needs can be added to or removed from the product backlog, or changed, on a whim. Product requirements, however, are only set once PBIs are selected for implementation. It is easy and cheap to manipulate needs statements, but harder to change requirements because they represent important product design decisions. In Scrum requirements are chosen just before they are implemented, and they cannot be changed until a sprint is completed. Delaying choosing requirements avoids the cost of changing them.
>
> *Delay refinement as long as possible*—PBIs are initially stated at a high level of abstraction, at about the user-level. Such high-level items typically describe product features that cannot be implemented in a single sprint. These large PBIs are refined into smaller and more detailed PBIs until they are small enough to be implemented in a sprint, at a level of abstraction that roughly correspond to operational-level needs. User-level PBIs are not refined until just before they are expected to be implemented in a sprint, so needs refinement at this level of abstraction is delayed as much as possible. Finally, some operational-level needs are selected for implementation during a sprint, thus becoming product requirements. At this point, developers interact with stakeholders to refine these operational-level requirements to the lowest-level of abstraction, physical level. Again, refinement is delayed as much as possible.

Delaying specification refinement avoids much rework and speeds product delivery. Refinement occurs as late as possible, so product delivery is not delayed by (perhaps unnecessary) requirements analysis work. Needs in the product backlog can change at any time—if a PBI is removed or altered, the work done to refine it need not be redone because it was (usually) never done in the first place—user-level needs are refined only just before they are implemented, and operational-level requirements are refined only when they are implemented. Of course, an implemented requirement may ultimately be rejected and all the work done refining and building it wasted, but this is (ideally) a relatively rare occurrence.

Avoid writing requirements altogether—In Scrum and most other agile methods, operational-level PBIs are written down, but physical-level requirements are usually not. When a PBI is chosen for implementation in a sprint, developers converse with stakeholders and the product owner to understand their needs and then make decisions about detailed requirements. Discussions with stakeholders and the product owner about detailed requirements may be extensive, and sometimes result in written physical-level requirements, but usually developers just go ahead and implement the results of their discussions immediately in the sprint. The bulk of traditional requirements documents are detailed (often physical-level) requirements specifications. By not writing such requirements, agile developers avoid a great deal of requirements documentation work.

Determine requirements in light of current product features—As noted, stakeholders have trouble identifying their needs and desires and developers have trouble choosing requirements without seeing how they might fit into the product. In Scrum, stakeholders see the latest product increment at the end of every sprint in a sprint review. Stakeholders and developers can then consider needs and requirements in light of the latest version of the product. Feedback from stakeholders is used by the product owner to alter the product backlog and prioritize PBIs, and also provides input to decision about what PBIs to implement in the next sprint. Thus the Scrum process provides stakeholders and developers with product information to guide them in their choices of needs and requirements.

In summary, Scrum tackles the problems of traditional waterfall approaches to requirements specification by (a) delaying choosing requirements as long as possible, thus avoiding requirements change rework; (b) delaying refinement as much as possible, thus speeding product delivery and facilitating product change by avoiding rework, (c) not writing detailed requirements specifications, thus speeding delivery and avoiding work, and (d) providing product increments that help stakeholders identify needs and developers analyze requirements. Studies have shown that these practices are effective in solving the problems resulting from writing large requirements specifications early on.

Stating Specifications in Traditional Processes

Traditional waterfall-style process needs and requirements specifications are stated in declarative English sentences. Despite the availability of several mathematical specification languages (such as Z [1], for example), most specifications are stated in English (or other natural languages) so that they are easy for everyone to read. Unfortunately, English specifications may be vague or ambiguous, and still hard to understand. For example, consider the following specification.

> The product must draw parts with and without notations showing sub-assemblies.

Does this mean that part drawings can include notations, and the notations show sub-assemblies, or that the part drawings show sub-assemblies and in addition may have notations? Dividing it into several simpler specifications, rewording it, and defining its terms can disambiguate this specification.

We try to make requirements specifications clear and unambiguous by following rules of good technical writing, such as the following.

- Write complete, simple sentences in the active voice.
- Define terms clearly and use them consistently.
- Use the same words for the same thing—avoid synonyms.
- Group related material into sections.
- Use tables, lists, indentation, white space, and other formatting aids to present and organize material clearly and concisely.

Following these rules makes specifications easier to find, understand, and change.

In addition, we follow some rules particular to specification writing that help make them easier to understand, refer to, verify, and validate. The first of these rules is to express all specifications using the words "must" or "shall." This reflects the fact that needs and requirements are statements about features or characteristics that a product must have. It also helps distinguish specifications from other statements that provide additional information or state features or characteristics that a product *may* but need not have. If you look back over the examples presented so far, as well as those below, you will see that all of them use "must," so they all serve as examples of this convention.

Another rule about writing specifications is that they should be **testable** or **verifiable**, which means that each specification should be written in such a way that there is a definitive procedure for determining whether it has been satisfied. To illustrate, the following requirement specifications are not testable.

> The product must display query results quickly.
>
> The product must produce error messages that are easy to understand.
>
> The product must be able to handle several users at once.

How would a tester determine whether results are displayed quickly enough, or that a product can handle enough users at once? Testable requirements include clear and measurable properties that can be used to determine whether the requirement is satisfied. The requirements above could be restated as follows.

> The product must display query results in less than one second.
>
> The product must produce error messages that (a) state the problem without referring to implementation details known only to developers, (b) state whether the program can continue and, if it can continue, the consequences of continuing, and (c) explain how to get help responding to the error.
>
> The product must be able to handle up to 18 users at once.

Specifications should be easy to understand and to change. It should also be easy to associate individual requirements with parts of design that realize them, code that implements them, tests that verify them, and so forth. This latter activity is called **requirements traceability**. Specifications that state several needs or requirements are harder to understand, harder to change, and difficult to associate with other artifacts for traceability. Thus another rule for writing good specifications is that each specification should state only a single need or requirement. Such specifications are called **atomic**. For example, consider the following specifications.

The product must display a list of previous commands and the results of commands, each in its own window.

If the user presses an up or down arrow key or selects a previous command with a pointer device, then previous commands must be scrolled through or the selected command displayed.

The first specification clearly states two needs or requirements. The second also states two needs or requirements, and conflates them so that neither one is clear. The following atomic versions of these specifications are better.

The product must display a list of entered commands in an entered commands window.

The product must display the result of all commands in a results window.

The user must be able to scroll through previous commands in the command textbox using the up and down arrow keys.

The user must be able to insert a previous command into the command textbox by selecting it in the previous commands window using a pointer device.

A final rule about specifications, and especially requirements, is that each one should have a unique identifier. Needs and requirements identifiers make it easier to refer to them in discussions and when tracing requirements. A typical form for identifiers is a dotted-number scheme that uses requirements document sections. For example, 4.2.8.3 might be the identifier for the third requirement in chapter 4, section 2, subsection 8 of the requirements document. Assigning every requirement its own identifier makes it easier to refer to requirements in other parts of the requirements specification document, or in other documents.

Despite all these rules and recommendation about writing requirements, it is simply the case that writing unambiguous, detailed, concise, and complete requirements is very difficult, and furthermore that even the best-written requirements are easy to misunderstand. Both requirements writers and readers must work hard to express and understand requirements specifications, being willing to rewrite and reread them over and over again.

Stating Specifications in Agile Processes

As noted before, agile developers try to avoid writing requirements specifications. But stakeholder needs and requirements specifications are still written down in various documents, like product vision statement, and certainly as product and sprint backlog items. These specifications may be written in the traditional manner as discussed above. However, agilists have almost universally adopted alternative means of stating specifications, which we consider now.

Agile methods usually express functional needs and requirements in user stories. A **user story** is a description of a function a product must provide for a user. User stories can be expressed at many levels of abstraction. At the highest level of abstraction, user stories describe functions that cover a large fraction of an entire application, or perhaps even an entire application, and would take many months or years to implement. Such stories are called **epics**. User stories that would take several sprints to implement (at about the user-level of abstraction) are called **features**. User stories small enough to be implemented in a sprint (at the operational-level of abstraction) are **sprintable stories** or **implementable stories**. Some Scrum experts prefer to restrict the term **user story** to just implementable stories, but we will use the broader meaning of this term here.

User stories are sometimes expressed simply as lists of functions that a product might provide. For example, a product vision statement might include the following in its product description.

Display drawings of individual parts.

Display drawings of assemblies with exploded views.

Rotate drawings in any direction.

Zoom in and out of drawings.

Such lists can be interpreted as specifications by understanding them as lists of functions that a product must provide.

There are two shortcomings of simply listing functions. The first is that when they express stakeholder needs, they do not identify the stakeholders. The second problem is that they are typically not testable. This is not a problem for abstract needs statements, but it becomes more of a problem as specifications become more detailed.

The most widely used way of expressing user stories is called the **user voice form.** It has the following format.

As a <*role*>, I want to <*activity*> so that <*benefit*>.

The <*role*> is filled in by a **user role**, which is a name for a class of product users. The <*activity*> is filled in by a function performed by a system. The <*benefit*>, which is optional, specifies the value of the activity for that class of users. The following examples illustrate this form of user story.

As a payroll clerk, I want to enter salary data so that payrolls will use adjusted salaries.

As a design engineer, I want to display an expanded parts drawing so that I can understand how parts fit together.

As a bank teller, I want to see account balances so that I can answer customer inquiries.

As a department head, I want to edit project portfolios.

If all product backlog items are user stories, how does a stakeholder who it not a user put a need into the product backlog? One solution is to broaden the <*role*> entry in user stories to include all stakeholders rather than just user roles. Another is to abandon the user voice form. Another is to allow product backlog items that are not user stories. Any of these alternatives may be used, but here we elect the first and so characterize the user voice form as naming a stakeholder, the product function that stakeholder desires, and why the stakeholder desires it.

The user voice form provides information about stakeholders, but these statements may still not be testable. Scrum experts recommend that user stories, in particular sprintable user stories, also have **acceptance criteria** or **conditions of satisfaction** stating the product behavior regarded as fulfilling the user story. Acceptance criteria should be specific enough that it will be clear whether the product fulfills the user story. Acceptance criteria are a vehicle for adding details to somewhat abstract user story specifications.

Agile methods often employ the "three Cs of user stories:" card, conversation, and confirmation.

Card—A card is an index card or post-it note with a user story title and a statement of the user story on one side. Typically user stories employ the user voice form, but additional information may be included as well, though part of the point of using a card is to keep the specification short.

Conversation—The card is regarded as a placeholder for various conversations amongst developers and stakeholders to discuss the need when it is modified, refined, estimated, prioritized, and (perhaps) implemented. These conversations may alter the need or fill in details as necessary during development. Many of these details will not be written down because the emphasis in agile methods is on face-to-face discussion rather than documentation.

Confirmation—The back of the card contains the acceptance criteria for the user story. Because it is small, many details may be left out, which again can be derived during conversations and captured in tests.

Actual physical cards can be used and they can constitute the product backlog. Alternatively, the product backlog can be kept in a spreadsheet with "cards" being rows with cells for user stories and acceptance criteria. There are also many agile tools that automate product backlogs. The format of PBIs and the product backlog is not important as long as documentation overhead is minimal and it is easy to add, remove, change, and shuffle PBIs in the product backlog.

There is nothing wrong with supplementing PBIs with additional information either. For example, there may be detailed interface specifications relevant to PBIs about interfaces, or papers about algorithms or implementation strategies connected with certain PBIs. Such information (or references to it), can supplement PBIs as needed.

Eliciting Stakeholder Needs in Traditional Processes

In a traditional development process in which most requirements are determined before further development work is done, eliciting all (or most) stakeholder needs must be done early and thoroughly. The size and importance of this effort have given rise to many techniques that we catalog in this section.

Eliciting stakeholder needs for software products is similar to extracting needs and preferences in many other disciplines. For example, pollsters elicit voter preferences, market researchers try to determine what consumers are willing to buy, and TV ratings researchers try to find out what people like to watch on television. Techniques used in these disciplines are sophisticated and highly refined; we can only survey them here. These techniques have been used widely in traditional development; agile developers do things somewhat differently, as we discuss below.

The following are the most frequently used techniques used to elicit stakeholder needs in traditional processes.

Interviews—An **interview** is a question and answer session. Developers ask stakeholders what functions and characteristics they need, and record the answers. The developers are trying to understand the stakeholders' perspectives, so usually it helps to also ask stakeholders to explain the domain of application, their vision for the product, how they think it could be valuable, and so forth. Some questions should be prepared before the interview, but it is valuable to allow interviewers to formulate questions during the session as new ideas and issues arise. Answers should be recorded. It is much easier to record the session on audio or video recording equipment than to try to take notes.

Observation—If a product supports users in doing particular tasks, or is intended to replace a system currently in place, then it is often useful for developers to watch users doing the tasks or using the system to be replaced. This is called **observation**. It allows developers to understand the context and goals of the product, and it provides a wealth of information leading directly to detailed requirements. Developers should watch people do the same task several times, and watch several people do the same task so they understand how its performance can vary. It is easier to record subjects with videotape than to take notes. Subjects can be asked to explain what they are doing as they are doing it. This technique, called the *think-aloud protocol*, enables observers to understand better what subjects are doing, though it may also distort what the subject actually does.

Focus groups—A **focus group** is an informal discussion among six to nine people led by a facilitator who keeps the discussion on topic. Stakeholders can meet to discuss the product and

its features and characteristics led by a developer or a trained facilitator. Other developers can observe the sessions or learn about them second hand. Focus groups are a good way to collect a broad spectrum of ideas and perspectives, with feedback from other stakeholders, in a short period of time. They are especially useful early in needs elicitation.

Workshops—A **workshop** is a meeting whose participants focus intently on a particular topic or goal. Elicitation workshops are meetings with stakeholders aimed at discovering and documenting stakeholder needs and desires. Workshops require effort from their participants, but they can be very productive.

Prototypes—A **prototype** is a working model of part or all of a final product. We have noted how difficult it is for stakeholders to decide what products should do and how they should work. Prototypes give stakeholders a chance to try out different versions of a product. This is usually very helpful in nailing down good requirements.

Document studies—Developers can learn about a product problem domain, the needs of stakeholders, product constraints, and so on, by studying various documents describing the business, its policies and procedures, quality improvement studies, bug reports, suggestions for improvement, and so forth. Studying documents cannot by itself provide everything developers need to know, but it can provide a lot of the context necessary to really understand stakeholders.

Competitive product studies—If a product has competitors already on the market, analyzing them is a prerequisite for generating requirements for a new product. Obviously, new products must avoid the weaknesses and match the strengths of existing products or they won't be successful.

Requirements specialists usually use several of these techniques over the course of the requirements analysis phase of the traditional life cycle, often returning to stakeholders for follow-up interviews or observations, demonstrations of new prototypes, and so forth, as requirements are developed over time.

Eliciting Stakeholder Needs in Agile Processes

Agile methods do not attempt to determine most requirements early, so the need to elicit all stakeholder needs early goes away. Instead, needs are elicited continuously during development. The main advantage of this approach is that stakeholders can express needs that have only just arisen in response to the current situation. Also, stakeholders can see the current state of the product and react to it in formulating needs. In effect, the product itself becomes a continuously evolving prototype. This usually makes it easier for stakeholders to understand the product, and helps ensure that needs to not become obsolete. But there are drawbacks too, particularly the following.

- Agile methods do not advocate systematic means for eliciting needs (such as those listed above). Stakeholders may over-react to recent events in stating and prioritizing needs, possibly losing track of the big picture and making poor decisions about what to include in the product.
- Agile methods usually place great trust in a few stakeholders or stakeholder representatives. In Scrum, for example, the product owner has primary responsibility for the product backlog, with input from whichever stakeholders make time to attend sprint reviews or make their opinions known in other ways. There is considerable risk that the product will not reflect the needs of all stakeholders, or that important stakeholders' opinions will not be properly taken into account.

There is no reason that agile methods cannot use the several systematic techniques for collecting stakeholder needs used in traditional processes, but these techniques are not part of agile culture.

Verifying and Validating Requirements in Traditional Processes

Requirements specifications must be nearly complete before moving on to implementation in traditional processes, so it is important to ensure that they are right. There are five important characteristics to be checked.

Clarity—If requirements are vague or ambiguous, then questions will arise during development, or perhaps the product will not embody the intentions of stakeholders. So checking for clarity is important. A good way to check clarity is to ensure that requirements are testable.

Consistency—Requirements that contradict one another can obviously not be implemented.

Completeness—Requirements specifications that fail to fully characterize the product will result in a product lacking features or properties that may be important to stakeholders. Requirements also need to be sufficiently detailed.

Correctness—Requirements specifications that do not accurately reflect stakeholder needs will result in a poor product.

Well-formedness—As noted, requirements specifications should be stated using "must" or "shall," should be atomic, and should have unique identifiers.

The main technique for checking requirements specification is **reviews**, which are examinations and evaluations of work product by qualified individuals. There are various kinds of reviews, which we discuss elsewhere, and any of them can be used for requirements. All reviews are expensive because they require several people to perform a painstaking task that usually requires a lot of time. Nevertheless, studies have shown it is less expensive to do reviews and detect and correct errors during the requirements process than to not do them and detect and correct the errors much later in the development process.

Of course the real test of requirements specifications occurs when the product is delivered to its customers. All too often, it becomes clear that important requirements have been missed or misunderstood, but at that point, a great deal of money and time have been spent, and even more would have to be spent to fix the problems. This is the Achilles heal of traditional processes.

Verifying and Validating Requirements in Agile Processes

Requirements are under continuous scrutiny in agile processes. In Scrum, for example, the product backlog is always being groomed, so requirements are always being reconsidered. The most careful check of requirements, however, is during the reviews at the conclusion of each sprint. The team presents the latest version of the product to the product owner and any other stakeholders who care to attend the review. Requirements problems should manifest themselves in the form of wrong, missing, or extraneous product behaviors or properties. These problems can then be traced to erroneous requirements that can be fixed immediately. Stories to alter the product and correct problems can be added to the product backlog.

The greatest weakness of this approach is that requirements errors will not be detected because a stakeholder who could recognize errors is not present at sprint review. Not every stakeholder can attend every sprint review. The product owner stands-in for missing stakeholders, but if the product owner's understanding of stakeholder needs is incorrect or incomplete, then errors may go undetected.

Requirements Management in Traditional Processes

In traditional processes, a project is launched with a **product mission statement** or some other document that provides high-level (business) requirements for a product. One or more requirements specialists then conduct an extensive requirements analysis. **Requirements analysis** is the activity of eliciting, examining, and understanding stakeholder needs, developing requirements specifications, and evaluating them to ensure that they are clear, complete, consistent, correct, and well formed. The output of this activity is a **software requirements specification (SRS)**, a document containing all the requirements for a product. The SRS is usually divided into sections consisting of atomized requirements statements. The figure below shows a template for an SRS.

1. Introduction
   - 1.1 Product Vision
   - 1.2 Project Scope
   - 1.3 Stakeholders
   - 1.4 Design and Implementation Constraints
2. Functional Requirements
   - 2.1 Product Behavior
   - 2.2 User Interfaces
   - 2.3 System Interfaces
   - 2.4 Data Requirements
3. Non-Functional Requirements
4. Other Requirements
5. Glossary

Figure 1: An SRS Template

Sometimes an SRS will also contain tables, figures, and various sorts of models. During its writing the specifications in the SRS will be reviewed and corrected; often there will be a final review at its completion as well. Once it is finished, the SRS is the basis for design, implementation, and testing.

Although great effort is expended to ensure that the SRS is clear, complete, consistent, correct, and well formed, shortcomings in all these areas will inevitably arise during the rest of the development process. Furthermore, stakeholders' needs and desires may change. Hence, the SRS will need to be changed. Usually a special process for requirements modification is instituted to ensure that requests for change are logged, evaluated, and decided upon with due consideration of their importance and costs. Then the changes are made and traced through designs, code, and tests to make sure that the changes are realized and checked. Clearly, this is an expensive endeavor.

The entire activity of developing requirements and then overseeing them throughout the software development process is called **requirements management**.

Requirements Management in Agile Processes

Agile methods do not contain an explicit requirements management process, but we can sketch the role of requirements in agile processes, and in particular in Scrum. Agile projects begin with some sort of project mission statement stating high-level needs in the form of epics or features that become the start of the product backlog. The product owner, usually with the help of the team and other stakeholders, refines these high-level features into user stories and prioritizes them to the point that some of the stories are sprintable. Then the team begins sprinting.

During each sprint, operational-level requirements in the form of user stories are refined into physical-level requirements in conversations with stakeholders and the product owner, and then

implemented. At the conclusion of each sprint, stakeholders and the product owner validate requirements by participating in product demonstrations in the sprint review.

As the team sprints, the product owner continues to groom the product backlog, refining and reconsidering needs. Needs will typically be added, deleted, and modified as the product backlog is changed.

As this summary makes clear, requirements management is a core activity of agile methods, but it is incorporated into the broader processes of backlog grooming and sprinting, and not distinguished as a separate process.

Requirements and Product Design

Suppose you work for an automobile company as an automotive designer. Your job is to come up with ideas for appealing new cars with state-of-the-art features and then specify them in every detail so that they can be manufactured. In the automobile industry (and most similar industries) your job is known as *product design*. If you think about it, you will realize that your automotive design job is very much like the software requirements analyst's. In fact, requirements analysis *is* product design for software. It is not clear why the software industry has adopted terminology so different from other industries, but it has. This is unfortunate for two reasons:

- It hides the fact that software product development is very similar to other kinds of product development. Many product design practices, tools, and techniques in other engineering disciplines are perfectly applicable in software, but this is not obvious because of the difference in terminology. Therefore, software developers do not use many of these practices, tools, and techniques.
- Perhaps because it is called *requirements analysis* rather than *product design*, specifying the requirements of a software product has not been considered a design activity. Being sloppy about distinguishing needs and requirements also encourages people to think that requirements analysis is mostly about gathering stakeholder needs, writing them down, and calling the result a specification. This tendency, which originated in traditional processes, has carried over into agile processes as well: the product owner is really the chief product designer, and team members do product design when working with stakeholders to formulate detailed specifications during sprints. As discussed, there is a lot more to requirements analysis, backlog grooming, and requirements refinement than just writing down stakeholder needs. In particular, good developers must generate alternative designs, consider their relative merits in light of stakeholder needs, improve them if possible, and ultimately settle on the best ones as the requirements. This is a design process. Calling what developers do when pinning down requirements *product design* would not only be more accurate, but would encourage these designers to *do* design, not just gather and document needs.

As an illustration of why recognizing requirements analysis as a design activity would be beneficial, lets consider some of the characteristics of design. A fruitful way of thinking about design is as problem solving: stakeholder needs and project constraints present a design problem, and a product specification is a solution to the problem. Problem solving has several salient characteristics.

Trial and error—Problem solving usually involves a certain amount of trial and error. Rarely are acceptable product specifications created on the first try. Typically several attempts are needed before a good solution emerges.

Iterative—Requirements analysis is an iterative activity. This is another way of saying that it involves trial and error, except now we are focusing on the process. First, one or more tentative

solutions are generated, then they are evaluated to determine the advantages and disadvantages of each, and finally a decision is made whether a solution is acceptable. If not, then we return to step one, using what we have learned from previous attempts to generate better solutions.

No unique solution—There is often more than one way to solve a problem. Different analysts may come up with equally good ideas. If it is not apparent which of several specifications is best, then pick one of the good ones and move on (perhaps noting the alternatives in case problems arise with the chosen solution later on).

If requirements analysts, product owners, and agile team members think of themselves (at least in part) as product designers, then they would know to expect to do a lot of iteration as they explore design alternatives, that a lot of their product designs will have to be thrown away, or at least greatly changed, and that there may be several good ways to solve their problems (as well as a lot of bad ways).

References

1. J. M. Spivey, *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

Scheduling

Introduction

It is natural to think that effort estimates for each task are all that are necessary to determine how long a project will take to complete. However, a little additional thought makes it readily apparent that this is not the case.

Units Matter

It is easy to be careless about units, especially since few programming languages take them into account and, as a result, programmers tend to just think about unitless numbers. However, it is important to realize that effort and time are not measured in the same units; effort is commonly measured in person-days or person-months and time is commonly measured in days or months.

Thinking only about the units, we can ask what needs to be done to convert person-days to days and the answer is obvious -- divide by the number of people working on the task. That is, thinking only about units

$$T = E / N$$

where T denote the time, E denotes the effort, and N denotes the number of people.

Task Details Matter

Unfortunately, the relationship between E and T described above does not hold in general. To see why, consider an extreme example unrelated to software engineering. Specifically, suppose the task is to thread a nut onto a that is 1 meter long, and suppose that the nut can be spun at a rate of 1 centimeter per second. Then, the effort required to complete the task is 100 person-seconds. In other words, one person would have to work for 100 seconds to complete the task. Now suppose 20 people are assigned to the task. The relationship above says that the task can be completed in 100 person-seconds / 20 persons = 5 seconds, but this is obviously ridiculous. There is no way 20 people could work on this task at the same time. Indeed, it's hard to imagine more than one person working on this task at a time. So, clearly, the details of the task matter.

Very small tasks can typically only be worked on by one person at a time. Hence, the relationship above is applicable only for $N = 1$, in which case the effort required to complete the task (in person-days) is equal to the time required to complete the task (in days).

Larger sized tasks often exhibit the proportional relationship described above for small value of N. However, for larger values of N the benefit of additional people declines and ultimately become negative. In other words, throwing more people at a task can actually make things worse.

Task Dependencies Matter

Even when we know all of the tasks and have a clear relationship between E and T for each one it is not easy to calculate the time required to complete a project that consists of multiple tasks dependencies between tasks. This can easily be illustrated using two examples.

At one extreme, it might be possible to work on all of the tasks simultaneously. In this case, if there are enough people available, the project will only take as long to complete as the longest task will take to complete. For example, if there are five tasks, requiring efforts of 4, 1, 3, 1, and 2 person months, and there are five people available to work on the project, then the project will take 4 months to complete. In addition, task 2 can slip 3 months (since it only requires 1 person-month of

effort and the longest task requires 4 person-months), task 2 can slip 1 month, task three can slip 3 months, and task 5 can slip 3 months, all without the project being delayed.

At the other extreme, it might not be possible to complete task $i + 1$ until task i has been completed. In this case, regardless of the number of people available, the project will take as long to complete as the sum of the times required for each task. Continuing with the same example, regardless of the number of people available, the project will take 4 + 1 + 3 + 1 + 2 = 11 months to complete. In addition, if any of the tasks slips, the project will be delayed by the length of the slip.

Generalizing from these two extremes, it is clear that the amount of time required to complete a project depends on both the effort required for each task, and the dependencies between tasks. Current and former college students have some experience with this kind of problem already. Specifically, think about the problem of determining how long it will take to complete a particular major. The answer depends on the required courses, the effort required to complete each course (e.g., one person-semester), and the prerequisite structure (i.e., which courses must be completed before which other courses can be attempted).

Personnel Capabilities Matter

The dependencies discussed above were intrinsic to the tasks themselves. For example, the unit testing of a particular module cannot possibly begin until after the unit tests have been constructed and the module has been constructed. (In test-driven development the unit tests are constructed first and in traditional development the module is constructed first, but in neither case can the actual testing be conducted until both of the other tasks have been completed.) However, the capabilities of team members can also place constraints on the order in which tasks can be completed. For example, a team may have only one person who has expertise with GUI construction technologies. Hence, GUI construction tasks must be worked on serially, they cannot be worked on in parallel.

The Problem to be Considered

We assume that relationship between effort and time is known. Most commonly, this means that the number of people assigned to a task is appropriate for the task (e.g., tasks are very small and assigned to a single individual). We also assume that personnel capabilities have been captured by the task dependencies (e.g., if tasks i and j can only be completed by person p then there is a task dependency between i and j).

Given these two assumptions, the basic data needed to calculate the time required to complete a project can be summarized in a table that includes the **duration** (i.e., the time required to complete each task) and the **dependencies** (i.e., which must be completed before a specific task of interest can be started). For example, consider the following table describing a product that requires the finish of 14 distinct tasks.

Task Number	Duration (Days)	Prerequisite Tasks
1	6	—
2	5	1
3	2	1
4	6	1

5	4	2, 3
6	1	4
7	2	4
8	4	4
9	3	2
10	4	5, 6
11	1	7, 8
12	4	9, 10
13	2	6, 11
14	1	12, 13

For example, task 1 has a duration of 6 days and has no prerequisite tasks. As another example, task 5 has a duration of 4 days and can't be initiated until after both tasks 2 and 3 have been completed.

While this table completely describes the tasks required to complete the product, it is sometimes useful to look at the dependency data in another way. Specifically, instead of thinking about the prerequisite tasks we can think about the subsequent tasks, which are summarized in the following table.

Task Number	Subsequent Tasks
1	2, 3, 4
2	5, 9
3	5
4	6, 7, 8
5	10
6	10, 13
7	11
8	11
9	12
10	12
11	13
12	14
13	14
14	—

For example, this table shows that task 1 (and perhaps others) must be completed before tasks 2, 3 and 4 can be initiated and that task 7 (and perhaps others) must be completed before task 11 can be initiated.'.

Note that, *tasks must be numbered/ordered in such a way that lower numbered tasks do not depend on higher numbered tasks* (i.e., tasks with a higher number are not prerequisites for tasks with a lower number). This is necessary to ensure that there are no dependency cycles, which would make completion of all tasks impossible. Obviously, the data about tasks times and task dependencies can be combined into one table but, especially at first, it is usually clearer to use two.

Note also that not all projects have a natural first task. In other words, it may be possible at the outset of a project to initiate several tasks simultaneously. In such cases, it is sometimes convenient

to create a *dummy* task that has no duration that is imagined to be a prerequisite for these tasks. Similarly, not all projects have a natural last task. In other words, it may be possible at the end of a project to complete several tasks simultaneously. In such cases, it is again sometimes convenient to create a dummy task that has no duration that is imagined to be subsequent to these tasks.

With these data in hand we can consider the problem of determining how long it will take to complete a project. The techniques we consider all involve the identification of the **critical tasks**—the tasks that determines the minimum amount of time required to complete the project. They also involve the calculation of **slack** (sometimes called **float**)—the amount of time that a task can slip without the project falling behind.

Gantt Charts

Gantt charts have a long history, having been first developed in the 1910s. In a Gantt chart, tasks are represented as rectangles and dependencies are represented as arrows. Time increases in the horizontal direction (traditionally, from left to right) and tasks are added in the vertical direction (with the first task at the top). The left side of each task is positioned at its earliest possible start time. A task's horizontal extent is determined by the time required to complete it, its vertical extent is arbitrary.

So, continuing with the example above, since task 1 starts at time 0 and requires 6 time units to complete, it is represented as follows

Task 2 has task 1 as a prerequisite so the earliest it can start is at time 6. Further, it requires 5 time units to complete, so it is represented as follows.

The arrow from the end of task 1 to the start of task 2 indicates that task 2 cannot start until after task 1 has been completed (i.e., that task 2 depends on task 1). Similarly, tasks 3 and 4 have task 1 as a prerequisite and require 2 and 6 days respectively. Hence, they are represented as follows

Since task 5 has both tasks 2 and 3 as prerequisites, it can't be started until after both are completed. This means that the earliest it can start is after task 2 (i.e., at time 11). Hence, it is added as follows.

Continuing in this fashion, the other tasks are added as follows.

The critical tasks can be found by working backwards from the task with the latest finish time. Then, the predecessor that is part of the the critical sequence is the predecessor that itself has the latest end time.

For this example, task 14 is clearly critical. Then, looking at its two predecessors (12 and 13), task 12 has the later end time, so it is critical. Task 12 has two predecessors, tasks 10 and 11. Since task 10 has the later end time, it too is critical. Task 10 has two predecessors, tasks 5 and 6. Since task 5 has the later end time, it is critical. Continuing in this fashion, task 5 has two predecessors, task 2 having the later end time. Finally, task 2 has only one predecessor. The critical sequence of tasks can be represented as follows.

Once the critical tasks have been identified, it is an easy matter to identify the slack times. For example task 13 can slip 4 time units without causing task 14 to be delayed. Similarly, task 9 can slip five time units without causing task 12 to be delayed. This can be illustrated as follows.

Not surprisingly, there is no reason to create Gantt charts by hand; many software products exist that can create them. One example is shown below.

Name	Duration	Predecessors
1	6 days	
2	5 days	1
3	2 days	1
4	6 days	1
5	4 days	2;3
6	1 day	4
7	2 days	4
8	4 days	4
9	3 days	2
10	4 days	5;6
11	1 day	7;8
12	4 days	9;10
13	2 days	6;11
14	1 day	12;13

Note that the software product used to create this Gantt chart shortens tasks slightly so that the dependency arrow can extend horizontally from the prerequisite task even when there is no slack. Thus, task durations should not be measured from the chart, they should be read from the textual description.

The Critical Path Method (CPM)

In the 1950s, a considerable amount of attention was devoted to the use of graph theoretic algorithms (e.g., path finding algorithms) and numerical optimization algorithms (e.g., the Simplex Method) to solve planning the problems of various kinds, including scheduling and assignment problems. On such effort resulted in the development of the Program Evaluation and Review Technique (PERT). In PERT, the data from the task table and dependency table are used to construct a graph in which the vertices represent **milestones** (i.e., events that can be used to mark the progress of the project) and the edges represent tasks (each edge having a weight that corresponds to the duration of the task). The critical sequence of tasks is determined by finding the longest path from the first event to the last event. As a result, the critical sequence of tasks is usually called the critical path.

At around the same time, another group of researchers developed a technique that uses both time and cost estimates. Specifically, it constructs a time-cost curve for each task based on the ***normal time*** and the ***crash time*** for that task (to account for the relationship between effort and time/cost). This technique, which was called the Critical Path Method (CPM), requires the solution of a constrained linear optimization problem to identify a schedule that results in the minimum cost way to complete the project at a particular time.

Subsequently, a variety of other methods have been developed, some of which solve different variants of the problem and others of which use different graph representations (e.g., some represent the tasks as edges and some as vertices, some use dummy edges and vertices for different purposes), all of which make use of the term "critical path." We will consider one such Critical Path Method (CPM).

In this version of the CPM, the data from the duration column and dependency column are used to construct a graph in which the vertices represent the tasks and the directed edges represent the dependencies. Each task is visualized as a rectangle that contains seven pieces of information, the task identifier (*ID*), task duration (*D*), earliest start time (*ES*), earliest finish time (*EF*), latest start time (*LS*) and latest finish time (*LF*), and slack (*S*) as follows.

The duration is taken directly from the table, and the five other pieces of information, which are described below, are calculated in an iterative fashion. So, before any iterations are performed, task 1 is represented as follows.

The dependency column is used to construct the directed edges in the graph. Specifically, if task *i* must be completed before task *j* then the graph includes a directed edge from *i* to *j*. For example, since task 2 cannot be completed until after task 1, there is a directed edge from vertex 1 to vertex 2.

Similarly, since tasks 3 and 4 also can't be completed until after task 1, there are directed edges from task 1 to both tasks 3 and 4 as follows.

Now, since task 5 cannot be completed until after tasks 2 and task 3, there is one directed edge from 2 to 5 and another from 3 to 5. In other words, task 5 has two inbound edges as follows.

The entire graph showing all of the information from the durations and dependencies table is shown below.

The **earliest start time** for task *i*, denoted by ES_i, is the start time for task *i* assuming the preceding tasks are completed as early as possible. The calculation of these values starts by assigning the initial tasks (the tasks that have no prerequisites) an earliest start time of 0. In this example, this means $ES_1 = 0$.

The **earliest finish time** for task *i*, denoted by EF_i,is the sum of the earliest start time and the duration. That is,

$$EF_i = ES_i + D_i$$

So, the earliest finish time for task 1 is $EF_1 = ES_1 + D_1 = 0 + 6 = 6$.

The earliest start time for each task other than the initial tasks is the maximum of the earliest finish time for all of its prerequisite tasks. So, letting $\mathcal{P}_j$ denote the set of prerequisite tasks of task j.

$$ES_j = \max_{i \in \mathcal{P}_j} EF_i$$

So, for example, consider task 2. It has only one prerequisite (task 1), so $\mathcal{P}_2 = \{1\}$ and, hence, $ES_2 = 6$. That is, since task 2 can't start until after task 1, it's earliest start time is task 1's earliest finish time (i.e., time 6). It's earliest finish time is simply $EF_2 = ES_2 + D_2 = 6 + 5 = 11$. Similarly, tasks 3 and 4 have earliest start times of 6 and earliest finish times of 6 + 2 = 8 and 6 + 6 = 12, respectively.

Now consider task 5.Since task 5 can't be completed until after tasks 2 and 3 (i.e., tasks 2 and 3 are prerequisites) $\mathcal{P}_5 = \{2, 3\}$. So, it's earliest start time is the latest finish time for those two tasks. Specifically, since the earliest finish time for 2 is 11 and the earliest finish time for 3 is 8, the earliest start time for 5 is 11.

Continuing in this fashion yields the following.

The **latest finish time** for task i, denoted by LF_i, is the last time the task can be completed without delaying the project beyond its earliest time. For the last task on the critical path the latest finish time is the same as the earliest finish time. So, in this example $LF_{14} = EF_{14} = 24$ as illustrated below.

The **latest start time** is the latest finish time minus the duration. That is

$$LS_i = LF_i - T_i$$

So, for task 14, $LS_{14} = LF_{14} - T_{14} = 24 - 1 = 23$ as shown below.

For tasks other than the last task on the critical path, the latest finish time is the minimum start time for all (immediately) subsequent tasks. That is, letting S_i denote the set of tasks that are subsequent to task *i,* the latest finish time for task *i* is given by

$$LF_i = \min_{j \in S_i} LS_j$$

For example, consider task 13. Since the only task subsequent to task 13 is task 14, it follows that $S_{13} = \{14\}$. Hence, $LF_{13} = LS_{14} = 23$. The latest start time is then $LS_{13} = LF_{13} - T_{13} = 23 - 2 = 21$.

The calculation continue in exactly the same way until (working backwards) task 6 (which is different from tasks 7 through 13 in that it has more than one subsequent task.

Specifically, task 6 must be completed before tasks 10 and 13. So, at first glance, you might think that it's latest end time can be either task 10's latest start time (i.e., 15) or task 13's latest start time (i.e., 21). However, if task 10 doesn't end until 21 then task 10 will be delayed. Hence, task 6's latest end time must be the minimum of the latest start time of all successor tasks (i.e., 15).

Formally, $S_6 = \{10, 13\}$ and $LF_6 = \min\{LS_{10}, LS_{13}\} = \min\{15, 21\} = 15$. The latest start time is then $LS_6 = LF_6 - T_6 = 15 - 1 = 14$.

Continuing in this fashion (paying particular attention to tasks 4, 2 and 1) yields the following.

The **slack time** for task *i*, denoted by S_i, is the difference its latest finish time and its earliest finish **time**. That is

$$S_i = LF_i - EF_i$$

So, not surprisingly, the slack time for task 14 is $S_{14} = LF_{14} - EF_{14} = 24 - 24 = 0$. Indeed, this will be true of all of the tasks that are on the critical path (since that's precisely what it means to be critical). Specifically, the slack for tasks 1, 3, 5, 10, 12, and 14 are all 0, meaning they are the vertices in the critical path. On the other hand, all of the other tasks can slip somewhat, without delaying the overall project. For example, the latest finish time for task 13 is 23 while the earliest finish time is 19. Hence, task 13 can slip 4 days without delaying the overall project. All of the slack times are shown below.

6
0 1 6
0 0
0
5
6 2 11
6 11
0
2
6 3 8
9 11
3
6
6 4 12
8 14
2
4
11 5 15
11 15
0
1
12 6 13
14 15
2
2
12 7 14
18 20
6
4
12 8 16
16 20
4
3
11 9 14
16 19
5
4
15 10 19
15 19
0
4
19 12 23
19 23
0
2
17 13 19
21 23
4
1
16 11 17
20 21
4
1
23 14 24
23 24
0

Scrum

Introduction

There are several agile methods, some more popular than others. In this chapter we focus on the most popular agile method: Scrum. Part of the reason for its popularity is that Scrum is a process framework rather than a detailed process specification, which makes it more adaptable to the needs and preferences of different groups. Even though there is quite a lot of cross-fertilization among agile methods, with good ideas from one being adopted by the others, the generality of Scrum makes adding ideas from other agile methods especially easy. In this section we will consider Scrum and also mention some of the practices from other agile methods that Scrum practitioners frequently adopt.

Some ideas behind Scrum go back to an article in the Harvard Business Review about what is now called *concurrent product development* [1]. These ideas were transmogrified and made more applicable to software development by Jeff Sutherland in 1993. The first Scrum paper appeared in 1995 [2], and the seminal book *Agile Software Development with Scrum* appeared in 2001 [3]. Since then Scrum has become more and more popular. The current version of Scrum is documented online in the *Scrum Guide* [4], and in various books, such as *Essential Scrum* by Kenneth Rubin [5]. You are encouraged to consult these (or other) resources for more information about Scrum.

Scrum Overview

Scrum specifies a high level *process* consisting of *activities* that create and modify certain *artifacts*. The work is divided among developers playing certain *roles*. We thus begin by discussing Scrum's process, activities, artifacts, and roles.

Process

The Scrum process is pictured in Figure 1 below. This process contains the activities and artifacts discussed subsequently, so you will have to refer back to this diagram as you read the remainder of the overview. The process begins with the creation of an initial product backlog, which contains specifications for the target product. During sprint planning, a subset of the product backlog is selected for implementation, the tasks needed to do this are identified, and the effort to do them estimated. This information together forms the sprint backlog. The items in the sprint backlog are implemented during sprint execution, resulting in a potentially shippable product. At the end of the sprint, a sprint review demonstrates the new product features to stakeholders, and the sprint retrospective provides the team with a chance to look back over the sprint and decide how to do the next one better. Then this cycle, which is called a **sprint**, repeats. Because the cycle repeats, Scrum is an *iterative* process. More importantly, since a shippable product is produced in each cycle, Scrum is also an *incremental* process.

Figure 1: The Scrum Process

Roles

A Scrum project has at least one development team consisting of people playing the roles of *product owner*, *scrum master* and *team members*.

The **product owner** (PO) is responsible for deciding what will be done and in what order. The PO is in charge of the composition of the product, and as such represents the customers and users to the other members of the development team. Also, the PO collaborates with the scrum master and team members to make sure that the right features appear in the product in the right order as quickly as possible. Thus the PO is the pivotal individual mediating between the team

and the other stakeholders in the project. The PO's responsibility is largely exercised through control of the product backlog.

The **scrum master** (SM) is responsible for guiding the team in following the Scrum development process. The SM helps everyone embrace Scrum values, principles and practices and use them properly. The SM is a facilitator and coach. Also the SM helps remove impediments and protects the team from being interfered with by people outside the team.

The **team members** are responsible for deciding how to build the product and for building it. Team members must have all the skills necessary to build the product. They are thus a cross-functional team. The team decides for itself how to organize to get the job done. A common Scrum practice is to have everyone work on everything so as to maximize productivity and to help team members develop new skills.

Artifacts

There are three main artifacts in Scrum: the product backlog, sprint backlogs, and the shippable product.

The **product backlog** is a prioritized list of not-yet-implemented product features or characteristics. Its elements are **product backlog items** (**PBIs**), each of which is a product feature or function, a product change or fix, a technical improvement, and so forth. The PO is responsible for the product backlog; PBI priorities are typically based on the business value of the feature described in the PBI.

A **sprint backlog** is a collection of PBIs, the tasks needed to complete them, and estimates of how much effort each task will require. The PO and the rest of the team choose items from the product backlog to be implemented in a sprint during sprint planning. The sprint backlog guides the team in its work during sprint execution.

The overarching goal of every sprint is to produce a **potentially shippable product**, that is, a product increment that could actually be shipped to customers. Usually potentially shippable products are not deployed, but the point is that they *could* be. This means that a PBI can only be included in a product increment if it is actually completely done: designed, built, tested, and documented. A team may not succeed in completing every sprint backlog PBI in a sprint. PBIs that are not done are returned to the product backlog and may be included in a subsequent sprint.

Activities

Scrum activities consist of the steps in the process as well as a continuous background grooming activity. A **sprint** is a product development iteration whose goal is to add something to the product of value to a customer or user. Once the product backlog is created, the team is sprinting continuously: as soon as one sprint ends, the next begins. Sprints stop when the project is finished. Sprints have four parts: planning, execution, reviews and retrospectives.

Product Backlog Creation—The product backlog is initially a list of high-level features and functions based on whatever product descriptions the PO has available. There may be a written product vision statement the PO can use for guidance. The PO can also discuss the product with the managers sponsoring the product, salespeople, potential customers and users, and so forth. The product backlog need not have a lot of detail at first because its refinement will be a major activity during the entire development process.

Grooming—PBIs are constantly being added to and deleted from the product backlog, and also modified, refined, and reprioritized in the product backlog; this is called **grooming**. The PO works with stakeholders and the team to groom the product backlog. The PO grooms the product backlog more or less continuously, and typically schedules meetings with the team members during sprints to get help with this task.

Sprint Planning—Sprints begin with sprint planning, during which the PO, SM, and other team members select PBIs from the product backlog to accomplish in a sprint. This choice may be guided by an agreed **sprint goal**, which is the main thing to accomplish in the sprint. PBIs are chosen based on their priority and the effort expected to complete them. The idea is to choose the most valuable PBIs that can be completed in the sprint with everyone working at a sustainable pace. To accomplish this, the team determines for each PBI the tasks required to accomplish it and how long each task will take. These tasks and estimates, along with the PBIs, comprise the sprint backlog.

Sprint Execution—After sprint planning, sprint execution occurs; in other words, the tasks are performed and the PBIs implemented. Various tools and practices (discussed below) may be used to track progress and to maintain productivity during a sprint.

Sprint Review—At the end of a sprint, a sprint review occurs. All stakeholders are invited to the sprint review to see a product demonstration and to discuss the features or functions added, changes made, and so forth, in the sprint. (Remember, each sprint should result in a shippable product so, at the end of the sprint, it should be possible to demonstrate it.) The sprint review thus serves to bring stakeholders up-to-date on the state of development, and to provide feedback to the team about the product and its features and characteristics.

Sprint Retrospective—After the sprint review, the team conducts a sprint retrospective during which they discuss what went well, what did not, and how the next sprint can be done better. The sprint retrospective provides a means to improve the development process. In other words, the sprint review is concerned with the product and the sprint retrospective is concerned with the process. Once the retrospective is done, the sprint is complete and a new one begins.

Scrum Core Practices

In this section we consider in greater detail how scrum is supposed to work by considering how some of the central practices of Scrum are carried out.

Managing the Product Backlog

As we have already noted, the product backlog is a prioritized collection of **product backlog items**, or PBIs. Each PBI should consist of (a) a specification, (b) a priority, (c) an estimate of effort, and (d) acceptance criteria. We consider each of these in turn.

PBI specifications can take any form; for example, they may be traditional requirements statements, user interface diagrams, use cases, user stories (discussed below), defects that need to be fixed, a design task, a research task, or something else. Regardless of their form, it is essential that PBIs be expressible at different levels of abstraction. PBIs often start out as very broad, abstract statements about features or functions or modifications, and then are refined over time into several more detailed PBIs. The product backlog starts out with several very abstract PBIs. The highest priority PBIs are refined (during grooming) into more detailed PBIs that are small enough to be implemented in a sprint. Lower priority PBIs remain abstract until shortly before they are implemented, and then they are refined in preparation for sprinting. In general, a product backlog should contain enough refined PBIs to support two to three sprints, and the remaining PBIs should

be more abstract. This deferred refinement helps eliminate waste and rework because several abstract PBIs can be modified more easily than many detailed PBIs. Fortunately, most ways of stating requirements accommodate a wide range of abstraction.

Even when PBIs have been refined to the point that they are small enough to be implemented in a sprint, they are often still not detailed enough to serve as the basis for writing code. It is understood in Scrum that the team will often need to meet with the PO and perhaps customers and users to iron out PBI details during a sprint. Thus even a detailed PBI should be considered a placeholder for (or a promise to conduct) an interaction or conversation among the developers and other stakeholders aimed at nailing down the final details about a portion of the system just about to be implemented.

Perhaps the most popular form of specifying product features in Scrum is the user story. A **user story** is a description of a product feature or characteristic in the form

As a <*user role*> I want to <*goal*> so that <*benefit*>.

The *goal* is what the user wants to accomplish and the *benefit* explains why. For example, the following are user stories.

> As a book critic I want to be able to search for words and phases in a book so that I can analyze the book.
>
> As an author I want to create an index for my document so that I can make it easier to find material.
>
> As a course scheduler I want to determine whether students can take other sections of a course so that I can see if I can cancel a section with students already enrolled in it.
>
> As a shopper I want to see whether an item is still on sale so that I can buy it more cheaply.
>
> As an internet user I want to secure my devices so that I can protect my private information.
>
> As an instructor I want to combine sections of the same course in the course management system so that I only have to maintain one site.
>
> As an electric utility customer I want to see my usage over several years so that I can analyze it to budget my electricity costs more exactly.
>
> As a student I want to analyze my coursework and graduation requirements to plan my schedule for the remainder of my time at school so that I can see what courses I need to graduate on time.
>
> As a traveller I want to find the limits on checked luggage size so that I avoid luggage charges.
>
> As an investor I want to see all my investments from different companies on the same web page so that I can track my portfolio.

As you can see, even though all these sentences are quite similar in size and grammatical complexity, they are at very different levels of abstraction. For example, seeing whether an item is on sale or searching for text in a book are fairly detailed tasks, while displaying investments in a portfolio or creating an index is a mid-level task, and protecting devices or planning an entire four years of school are higher level tasks. The largest, most abstract user stories are sometimes called **epics**; they describe product functions that would take months or entire releases to produce. User stories smaller than epics that still span several sprints are sometimes called **features**. User stories that are small and detailed enough to do in a single sprint are called **sprintable stories** or sometimes just

stories. A product backlog may have a few epics at the bottom, many features in the middle, and enough sprintable stories at the top to support a few sprints.

Note also that even sprintable stories do not have nearly enough detail to actually design and write code from. For example, in searching for text, there are all sorts of questions that come up about how the text should be specified, what search options there might be, and how the results should be displayed. These are all details that the team must determine in conversation with the PO and other stakeholders. Agilists sometimes say that a user story is a placeholder for a conversation; in fact they can be placeholders for several conversations. In principle, the results of these conversations can be documented along with the user story (in any format desired) so that details are not forgotten. However, the emphasis on conversation rather than documentation is consistent with the agile effort to avoid waste and rework in the form of obsolete documentation.

In addition to a specification of the feature, each PBI must have a priority. A **priority** can be a number or a classification (low, medium, high, very high, for example). The priority determines when the PBI will be implemented. Priorities are set by the product owner with feedback from the team and stakeholders. Priorities should take dependencies into account—in other words, if PBI X cannot be done before PBI Y is in place, then Y should have a priority at least as high as X. High-priority PBIs must be small enough that they can be included in a sprint (as discussed above), and they should have more precise effort estimates and acceptance criteria (discussed below), while low-priority items may be larger and more abstract, may have rough estimates, and may not have acceptance criteria.

PBIs must also include **effort estimates**. These are needed for planning sprints and product releases, and for determining priorities. Sprint planning must be done quite precisely, so effort estimates for high-priority items must be in numeric units (such as person days). Low-priority items need only rough effort estimates for use in release planning and prioritizing, so categorical estimates (like small, medium, large, and extra-large) are adequate. As PBIs are refined during grooming (as they near the top of the product backlog), their effort estimates are refined as well. Detailed estimates are deferred to avoid the perhaps wasted work of making detailed effort estimates on low-priority items that may be radically changed, eliminated, or never make it to the top of the product backlog. As PBIs get close to the top of the product backlog, they are refined and their effort estimated carefully so that sprint planning will be more accurate.

Finally, PBIs may also have acceptance criteria. **Acceptance criteria** are checks that a user can make to determine whether a PBI is implemented correctly and completely. They are useful because they help flesh out the user story, making it clearer. These checks can often be used as part of the suite of tests developed to ensure that the PBI has been implemented correctly (more about this below when we discus the definition of done). PBIs not near the top of the product backlog do not need to have acceptance criteria (though they may), but detailed stories near the top of the product backlog should have acceptance criteria developed when they are refined. The acceptance criteria can also be expanded during the sprint in light of further conversations with the PO and other stakeholders.

As noted above, the product backlog must be groomed regularly. Grooming involves adding, removing, or modifying PBIs, elaborating PBIs that are approaching the top of the product backlog, re-estimating PBIs, and re-prioritizing PBIs. Grooming occurs naturally during the sprint review and sometimes during sprint planning. It should also be done at least once during a sprint so that there are enough small and detailed items for the next sprint(s). Some organizations schedule weekly or daily grooming sessions. Grooming requires participation of the team for generating effort estimates, recognizing dependencies, and so forth. Thus it is important that the team factor in time for helping the PO groom the product backlog when planning a sprint.

There are several commercial tools that POs can use to record the PBIs in a product backlog. Some POs just use a spreadsheet or a word processor document. A product backlog can even be kept in paper form on notecards. The tools are not that important; the important point about product backlogs is that they be groomed frequently so that a good selection of sprintable PBIs is ready for sprint planning at the start of every sprint.

A final note about completed PBIs: Once a PBI is done, it is no longer part of the product backlog, which is supposed to be the collection of features, functions, modifications, and so forth yet to be implemented. However, many POs are loath to throw PBIs away. Although finished PBIs should be removed from the product backlog, there is no reason not to put them in some other repository, called perhaps a *product feature list*, that contains all the completed items from the product backlog.

In conclusion, the product backlog is the central artifact of Scrum: it is essentially a stakeholder needs and desires list, and as such it is the lever by which the PO controls development and drives the entire process. Taking care of the product backlog is the most important job the PO has.

Estimating

When developing a product, it is important to be able to answer questions about how long it will take to develop the product, how much it will cost, and when it can be released. To answer questions like this, we need two basic sorts of information: the size of the job, and how quickly the team can get a job of a certain size done. The first requires size estimates, and the second requires data about how much work the team can get done in a sprint.

Scrum uses PBIs to specify products, so estimates of the size of the job are made by estimating the sizes of the PBIs comprising the product. PBI sizes are estimated in either story points or ideal hours. A **story point** is a unit of relative effort. Small PBIs are chosen as representative of one or two story points, and then other PBIs are assigned story point estimates based on how much bigger they are than the representative PBIs. An **ideal hour** or **person hour** is how much an average developer can accomplish in one uninterrupted hour of work. It is usually easier to estimate using story points than using ideal hours, so most Scrum estimators use story points.

Velocity is the amount of work completed by a team in a sprint. Once PBI sizes are estimated, a team need only add up the sizes of the PBIs completed in a sprint to compute its velocity. Thus velocity is measured in either story points per sprint or ideal hours per sprint.

PBI estimates can be used to predict how much effort will be required to complete a product or a subset of the product backlog. Using velocity, we can also estimate how many sprints will be required to implement a product or a subset of the product backlog.

Size estimates and velocity are also important for sprint planning. When preparing for a sprint, the team and PO must agree on the PBIs for the sprint backlog. Obviously, the total estimated size of the PBIs in a sprint backlog should be approximately the team velocity. However, the team typically will want to make more detailed estimates before committing to a set of PBIs for the sprint backlog. We discuss this next.

Creating the Sprint Backlog

Ideally, a team will complete all the PBIs in its sprint backlog on the last day of the sprint, thus having no slack time in the sprint and also having no unfinished PBIs. It is not easy to time things so perfectly, even with a short sprint. During sprint planning, teams work to make more precise estimates before finalizing the sprint backlog. This is done by considering, for each PBI, every task needed to complete it and estimating the effort of each task in ideal hours. For example, suppose a PBI is captured by the following user story.

> As an auditor I want to toggle an invoice annotation display using an annotation checkbox so that I can understand product and service codes.

This feature will require modifying the user interface to include checkboxes for each invoice, then writing code to scan an invoice, extract its product and service codes, look them up in a master code list, and display the results in a window off to the side of the invoice display. The team divides this work into four different coding tasks, and each is estimated. The coding tasks are assumed to include unit testing activities, but the entire feature must be integrated and tested (including PBI acceptance criteria testing—see below), which the team records and estimates as another task. The team inspects all its code, so the inspection task is estimated and included. Finally, modifying the user documentation to include this feature is estimated and added as a final task. Thus the original PBI, with its size estimate in ideal hours or story points, has been augmented with seven tasks needed to complete the PBI, each with a more precise estimate in ideal hours. The team can then be much more confident about how much effort completion of this PBI will entail, and it can factor this information into its decision-making about what will go into the sprint backlog.

In summary then, during sprint planning the team and the PO make an initial stab at the contents of the sprint backlog based on the sizes of the PBIs at the top of the product backlog. The team then analyzes the PBIs in more detail, generating tasks and task effort estimates for each PBI. Based on this analysis, the team reconvenes with the PO and comes to final agreement on the contents of the sprint backlog. The sprint backlog then consists of (a) PBIs, (b) the tasks necessary to complete them, and (c) effort estimates for all task, usually in ideal hours. The team is then ready to execute the sprint.

Sprinting

When a person has some work to do, he or she may either set the scope of the work and then expend as much time as necessary to do the job, or he or she may set the time to be spent and adjust the scope of the work to fit the time. The latter approach is called **time-boxing**. Sprints are time-boxed, meaning that they are of fixed duration, typically one week to one month long. Furthermore, sprints should have a consistent duration, except for occasional special sprints, typically at the start or beginning of a project or a release.

These characteristics have several advantages. Short, consistent, time-boxed sprints are easier to plan and track. They provide rapid feedback about development so that mistakes and misunderstandings are caught quickly. They provide lots of information about the current state of development and lots of opportunities for feedback about the developing product.

Sprints should not be shortened or extended. Furthermore, the PO should never try to alter the sprint backlog during a sprint. If a team cannot finish the items in its sprint backlog, then the unfinished items go back in the product backlog. If a team finishes all PBIs before the end of the sprint, then they can consult with the PO for another PBI to work on. Inaccuracies in time estimates should be identified during the retrospective, and steps should be taken to improve their accuracy in the future.

Tracking Progress with Burn Charts and Task Boards

Any project, even an agile project, must have some way to monitor whether work is progressing as expected. Agile methods, being light-weight, use techniques that do not require a lot of effort to collect data or produce management artifacts. One of the main tools for tracking progress in Scrum projects is a **burn chart**, which is a graph displaying work on the vertical axis and time on the horizontal axis. If a burn chart shows how much work remains against time, then it is a **burn-down**

chart; if it shows how much work has been accomplished, then it is a **burn-up** chart. Both kinds are used in Scrum projects, but burn-down charts are usually preferred for sprints.

A burn-down chart for a sprint has the total effort (usually expressed in story points) for the sprint marked on the vertical axis, and the horizontal axis marked with work days in the sprint. On the day that one or more PBIs are finished, there is that much less work to do in the sprint, so a point is marked showing the story points remaining on that day. Connecting these points produces a line showing progress in the sprint. A straight diagonal line from the starting point on the vertical axis to the last day of the sprint on the horizontal axis serves as a reference line showing ideal progress.

Consider the burn-down chart in the figure below. This burn-down chart shows the progress of work in a three-week sprint (with 15 work days) that began with PBIs totaling 28 story points of effort. On day two of the sprint, one PBI worth three story points was completed, so a point was plotted at that spot. Further points were plotted as the sprint progressed. The last PBI was completed on day 14; one PBI worth three story points was not completed in the sprint, so the progress line does not meet the horizontal axis. Comparing the solid progress line with the dashed ideal progress line shows that work in the sprint was a bit behind right from the start, though progress was steady. It may be that the team simply took on a bit too much in this sprint.

Figure 2: Sprint Burn-Down Chart

Burn-down charts are easy to make: once they are laid out, a point is plotted at the end of any day that some PBIs are completed. Teams usually post them in a prominent spot in their work area so that everyone can see how work in the sprint is going at any time.

Burn-down charts show the progress of work during a sprint in terms of story points, but they can only show progress at the level of PBIs. During a sprint, work has been planned in smaller units of tasks, so it is useful to be able to see the status of all the tasks in the sprint. A **task board** does this. It is laid out as a table whose rows are the sprint backlog PBIs and whose columns are areas showing tasks that have not yet been started, are being worked on, or are done. Task board column headers are "PBI," "To Do," "In Progress," and "Done." Cards or sticky notes are filled out for each PBI in

the sprint backlog and for all the tasks for each PBI. The PBI cards serve as row headers, and the task cards all start in the To Do column at the start of the sprint. When a team-member is ready to work on a task, he or she chooses one from the To Do column and moves it into the In Progress column. When that task is finished, it is moved into the Done column.

Figure 3 below shows the layout of a task board. In this particular task board there are four PBIs in the sprint backlog. Each has several tasks associated with it, all appearing in that PBI's row. The task board shows that Story4 is nearly finished because all its tasks are either done or in progress. Work on Story3 has clearly just gotten started, however, because only one of its five task cards is in progress.

Figure 3: A Task Board Layout

As this schematic example shows, it is very easy to use task board, and easy to tell by looking at them the state of the tasks in the sprint. Teams can use this tool to decide how to concentrate their effort, to detect when they are getting behind in their work, to identify tasks that are giving a team-member trouble, and so forth.

Making and Using a Definition of Done

One pitfall that teams sometimes fall into is to count a PBI as finished during a sprint when really it is not. The goal is to have a potentially shippable product after every sprint, so if a PBI is not really finished, then it should not be counted as part of a potentially shippable product. More pragmatically, counting a PBI as done when it really is not gives a misleading impression about the state of the project, and leaves work to be done that does not fit in anywhere—later sprints will be concerned with other PBIs, so when will the unfinished PBI be completed?

To help keep this from happening, each team should devise its own definition of done. A **definition of done** is a checklist of what needs to be accomplished for a BPI to be considered completed. Typical checklists include the following items.

- The design is complete and reviewed.
- The code has been formatted and commented.
- The code has passed inspection.
- The code has passed all PBI acceptance criteria (usually embodied in tests).
- The code has passed all unit tests (and regression tests).
- User documentation has been updated.
- The code has been integrated and passed all integration and system tests (and regression tests).

A team adopts a definition of done for all PBIs and a PBI is not considered done until it meets *all* items in the checklist. The best way to manage this is to create a definition of done checklist for each PBI and use it as a guide during the sprint to ensure that all work needed to complete each PBI is actually performed.

Finishing a Sprint

The two activities at the conclusion of a sprint are both aimed at reflection and improvement: the sprint review reflects on the product to improve it, and the sprint retrospective reflects on the process to improve it.

At a sprint review the team presents what it accomplished in the sprint. Only completed aspects of the product should be presented. All stakeholders should be invited to sprint reviews. Ideally, sprint reviews are held at a regular time so that stakeholders can put them on their calendars as recurring meetings, thus making it easier for all interested parties to attend.

Sprint reviews should be informal but follow more or less the following agenda.

1. The meeting begins with a team member (usually the PO) presenting the overall sprint goal (if any) and the PBIs in the sprint backlog. The team also lists the PBIs completed and explains any discrepancy between the planned and actual work accomplished. This is for informational purposes only, not as a basis for a discussion about team performance.
2. One or more team members demonstrate the new aspects of the product. This demonstration can include tests showing that parts of the product work better or correctly even if improvement don't result in anything evident to users. In this way there will be something to demonstrate even if most changes are improvements without much obvious change in product behavior.
3. All participants discuss the product and its development path. For example, the stakeholders may say whether they are happy with the product, whether its features are what they need and expect, what features might need to be changed, what features might still be needed, and so forth. The team might suggest changes or propose ideas they may have had for the product. The goal is to decide on the next steps in development to ensure that the team is always working on the most important and valuable changes that can be made to the product.

The results of the sprint review are fed forward into planning for the next sprint and should be reflected in the product backlog.

The sprint retrospective is a meeting for the team only, but it should include the entire team, including the PO and the SM. Its purpose is to devote time to analyzing the way the team works, identifying ways to improve, and making plans to incorporate the improvements into the way the team works.

When teams start using Scrum, enough problems come up that there is always at least one thing from the most recent sprint that the team wants to improve. Usually it is a good idea to work on only one improvement at a time. It may take several sprints for an improvement to be fully incorporated into a team's practice. After a team has been sprinting together for some time, however, they may become quite comfortable with what they are doing, and obvious improvements may not present themselves. At that point, the team may need to start working harder to brainstorm improvement opportunities, collect and analyze data about what they are doing, research possible improvements, and figure out how to make them. Improvement activities may even be incorporated as team goals with time allocated for working on them outside regular development activities.

Other Scrum Practices

There are several other practices that are common in Scrum. Many of them come from other agile methods and have been adopted by Scrum teams.

Daily scrum—A short (maximum 15 minute), often stand-up (no chairs) meeting in which each member states three things: (a) what did I do since the last meeting, (b) what will I do today, (c) what is impeding my progress. These meetings allow team members to decide how to spend their time that day. For example, if all is well, a team member may choose a task that still needs to be done in the sprint and work on it. But if a team-mate is struggling with a problem, a team member may choose to help them, or if there is some impediment to progress, several team members may meet after the daily scrum to figure out how to resolve it. The daily scrum is thus a just-in-time technique to help a team organize its daily activities.

Story time—This is a regularly scheduled meeting to groom the product backlog. As noted, although the PO is responsible for the product backlog, grooming often requires input from the team; story time provides an opportunity for the PO and the team to meet.

Cross-functional teams—Scrum teams usually include people with all the skills necessary to carry a product increment from requirements to deployment, including designers, testers, coders, documenters, and so forth. But if there is too much of one kind of work for specialists to accomplish in a sprint, then everyone else helps out. This is viewed as a way for team members to increase their skills, to keep growing, and to stay interested.

Sustainable pace—Although this idea comes from XP (XP is short for *Extreme Programming*, one of the first and most influential agile methods), it is common to all agile methods. Each sprint should be planned to keep everyone busy until the end, but without a mad rush to finish. Furthermore, developers are not expected to work overtime. Sprints are not extended and only shortened if the team asks them to be. Sprint backlogs cannot be changed during a sprint. If something is not complete, at the end of the sprint it goes back into the product backlog. This helps avoid pressure to over-work.

Planning poker—This is a technique that a team can use for jointly estimating PBIs. It uses story point cards in a format resembling poker. After describing a PBI, each team member throws an estimate card on the table. If there is disagreement, each team member explains the basis for their estimate, and the cards are collected and then thrown again, presumably reflecting estimates modified in light of the discussion. This continues until there is consensus. Planning poker is discussed in more detail in another chapter.

Bidding—This is also a technique for estimating the effort required to complete PBIs. In essence, team members take ownership of particular tasks by bidding on them in ideal hours. If one team member places a bid that is too high, another can come along and offer to complete it in less time.

Pair programming—This technique also comes from XP. Two team members sit together at one terminal, with one person typing. The two collaborate writing code (or whatever else they are doing), with the non-typing person checking what the typing person is doing. The idea is to incorporate instantaneous inspection and correction into the development process.

Thanks to its flexibility as a process framework rather than a strict process specification, Scrum can incorporate any technique that a team feels would benefit their work. Consequently there is a wide range of practices used by Scrum teams including and going beyond those listed above.

References

1. Takeuchi, Hirotaka, and Ikujiro Nonaka. 1986. "The New New Product Development Game." *Harvard Business Review*, January, 137–146.
2. Schwaber, Ken. 1995. "Scrum Development Process." In *OOPSLA Business Object Design and Implementation Workshop*, ed. J. Sutherland et al. Springer.
3. Schwaber, Ken, and Mike Beedle. 2001. *Agile Software Development with Scrum*. Prentice Hall.
4. *The Scrum Guide*. http://www.scrumguides.org/.
5. Rubin, Kenneth S. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley, 2012.

The Scrum Guide™

The Definitive Guide to Scrum:
The Rules of the Game

July 2013

Developed and sustained by Ken Schwaber and Jeff Sutherland

Table of Contents

Purpose of the Scrum Guide 3
Definition of Scrum 3
Scrum Theory 3
The Scrum Team 4
The Product Owner 5
The Development Team 5
The Scrum Master 6
Scrum Events 7
The Sprint 7
Sprint Planning 8
Daily Scrum 10
Sprint Review 11
Sprint Retrospective 12
Scrum Artifacts 12
Product Backlog 12
Sprint Backlog 14
Increment 15
Artifact Transparency 15
Definition of "Done" 15
End Note 16
Acknowledgements 16
People 16
History 16

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Purpose of the Scrum Guide

Scrum is a framework for developing and sustaining complex products. This Guide contains the definition of Scrum. This definition consists of Scrum's roles, events, artifacts, and the rules that bind them together. Ken Schwaber and Jeff Sutherland developed Scrum; the Scrum Guide is written and provided by them. Together, they stand behind the Scrum Guide.

Definition of Scrum

Scrum (n): A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Scrum is:

- Lightweight
- Simple to understand
- Difficult to master

Scrum is a process framework that has been used to manage complex product development since the early 1990s. Scrum is not a process or a technique for building products; rather, it is a framework within which you can employ various processes and techniques. Scrum makes clear the relative efficacy of your product management and development practices so that you can improve.

The Scrum framework consists of Scrum Teams and their associated roles, events, artifacts, and rules. Each component within the framework serves a specific purpose and is essential to Scrum's success and usage.

The rules of Scrum bind together the events, roles, and artifacts, governing the relationships and interaction between them. The rules of Scrum are described throughout the body of this document.

Specific tactics for using the Scrum framework vary and are described elsewhere.

Scrum Theory

Scrum is founded on empirical process control theory, or empiricism. Empiricism asserts that knowledge comes from experience *and* making decisions based on what is known. Scrum employs an iterative, incremental approach to optimize predictability and control risk.

Three pillars uphold every implementation of empirical process control: transparency, inspection, and adaptation.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Transparency

Significant aspects of the process must be visible to those responsible for the outcome. Transparency requires those aspects be defined by a common standard so observers share a common understanding of what is being seen.

For example:

- A common language referring to the process must be shared by all participants; and,
- Those performing the work and those accepting the work product must share a common definition of "Done".

Inspection

Scrum users must frequently inspect Scrum artifacts and progress toward a Sprint Goal to detect undesirable variances. Their inspection should not be so frequent that inspection gets in the way of the work. Inspections are most beneficial when diligently performed by skilled inspectors at the point of work.

Adaptation

If an inspector determines that one or more aspects of a process deviate outside acceptable limits, and that the resulting product will be unacceptable, the process or the material being processed must be adjusted. An adjustment must be made as soon as possible to minimize further deviation.

Scrum prescribes four formal events for inspection and adaptation, as described in the *Scrum Events* section of this document:

- Sprint Planning
- Daily Scrum
- Sprint Review
- Sprint Retrospective

The Scrum Team

The Scrum Team consists of a Product Owner, the Development Team, and a Scrum Master. Scrum Teams are self-organizing and cross-functional. Self-organizing teams choose how best to accomplish their work, rather than being directed by others outside the team. Cross-functional teams have all competencies needed to accomplish the work without depending on others not part of the team. The team model in Scrum is designed to optimize flexibility, creativity, and productivity.

Scrum Teams deliver products iteratively and incrementally, maximizing opportunities for feedback. Incremental deliveries of "Done" product ensure a potentially useful version of working product is always available.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

The Product Owner

The Product Owner is responsible for maximizing the value of the product and the work of the Development Team. How this is done may vary widely across organizations, Scrum Teams, and individuals.

The Product Owner is the sole person responsible for managing the Product Backlog. Product Backlog management includes:

- Clearly expressing Product Backlog items;
- Ordering the items in the Product Backlog to best achieve goals and missions;
- Optimizing the value of the work the Development Team performs;
- Ensuring that the Product Backlog is visible, transparent, and clear to all, and shows what the Scrum Team will work on next; and,
- Ensuring the Development Team understands items in the Product Backlog to the level needed.

The Product Owner may do the above work, or have the Development Team do it. However, the Product Owner remains accountable.

The Product Owner is one person, not a committee. The Product Owner may represent the desires of a committee in the Product Backlog, but those wanting to change a Product Backlog item's priority must address the Product Owner.

For the Product Owner to succeed, the entire organization must respect his or her decisions. The Product Owner's decisions are visible in the content and ordering of the Product Backlog. No one is allowed to tell the Development Team to work from a different set of requirements, and the Development Team isn't allowed to act on what anyone else says.

The Development Team

The Development Team consists of professionals who do the work of delivering a potentially releasable Increment of "Done" product at the end of each Sprint. Only members of the Development Team create the Increment.

Development Teams are structured and empowered by the organization to organize and manage their own work. The resulting synergy optimizes the Development Team's overall efficiency and effectiveness.

Development Teams have the following characteristics:

- They are self-organizing. No one (not even the Scrum Master) tells the Development Team how to turn Product Backlog into Increments of potentially releasable functionality;
- Development Teams are cross-functional, with all of the skills as a team necessary to create a product Increment;

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

- Scrum recognizes no titles for Development Team members other than Developer, regardless of the work being performed by the person; there are no exceptions to this rule;
- Scrum recognizes no sub-teams in the Development Team, regardless of particular domains that need to be addressed like testing or business analysis; there are no exceptions to this rule; and,
- Individual Development Team members may have specialized skills and areas of focus, but accountability belongs to the Development Team as a whole.

Development Team Size

Optimal Development Team size is small enough to remain nimble and large enough to complete significant work within a Sprint. Fewer than three Development Team members decrease interaction and results in smaller productivity gains. Smaller Development Teams may encounter skill constraints during the Sprint, causing the Development Team to be unable to deliver a potentially releasable Increment. Having more than nine members requires too much coordination. Large Development Teams generate too much complexity for an empirical process to manage. The Product Owner and Scrum Master roles are not included in this count unless they are also executing the work of the Sprint Backlog.

The Scrum Master

The Scrum Master is responsible for ensuring Scrum is understood and enacted. Scrum Masters do this by ensuring that the Scrum Team adheres to Scrum theory, practices, and rules.

The Scrum Master is a servant-leader for the Scrum Team. The Scrum Master helps those outside the Scrum Team understand which of their interactions with the Scrum Team are helpful and which aren't. The Scrum Master helps everyone change these interactions to maximize the value created by the Scrum Team.

Scrum Master Service to the Product Owner

The Scrum Master serves the Product Owner in several ways, including:

- Finding techniques for effective Product Backlog management;
- Helping the Scrum Team understand the need for clear and concise Product Backlog items;
- Understanding product planning in an empirical environment;
- Ensuring the Product Owner knows how to arrange the Product Backlog to maximize value;
- Understanding and practicing agility; and,
- Facilitating Scrum events as requested or needed.

Scrum Master Service to the Development Team

The Scrum Master serves the Development Team in several ways, including:

- Coaching the Development Team in self-organization and cross-functionality;
- Helping the Development Team to create high-value products;
- Removing impediments to the Development Team's progress;

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

- Facilitating Scrum events as requested or needed; and,
- Coaching the Development Team in organizational environments in which Scrum is not yet fully adopted and understood.

Scrum Master Service to the Organization

The Scrum Master serves the organization in several ways, including:

- Leading and coaching the organization in its Scrum adoption;
- Planning Scrum implementations within the organization;
- Helping employees and stakeholders understand and enact Scrum and empirical product development;
- Causing change that increases the productivity of the Scrum Team; and,
- Working with other Scrum Masters to increase the effectiveness of the application of Scrum in the organization.

Scrum Events

Prescribed events are used in Scrum to create regularity and to minimize the need for meetings not defined in Scrum. All events are time-boxed events, such that every event has a maximum duration. Once a Sprint begins, its duration is fixed and cannot be shortened or lengthened. The remaining events may end whenever the purpose of the event is achieved, ensuring an appropriate amount of time is spent without allowing waste in the process.

Other than the Sprint itself, which is a container for all other events, each event in Scrum is a formal opportunity to inspect and adapt something. These events are specifically designed to enable critical transparency and inspection. Failure to include any of these events results in reduced transparency and is a lost opportunity to inspect and adapt.

The Sprint

The heart of Scrum is a Sprint, a time-box of one month or less during which a “Done”, useable, and potentially releasable product Increment is created. Sprints best have consistent durations throughout a development effort. A new Sprint starts immediately after the conclusion of the previous Sprint.

Sprints contain and consist of the Sprint Planning, Daily Scrums, the development work, the Sprint Review, and the Sprint Retrospective.

During the Sprint:

- No changes are made that would endanger the Sprint Goal;
- Quality goals do not decrease; and,
- Scope may be clarified and re-negotiated between the Product Owner and Development Team as more is learned.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Each Sprint may be considered a project with no more than a one-month horizon. Like projects, Sprints are used to accomplish something. Each Sprint has a definition of what is to be built, a design and flexible plan that will guide building it, the work, and the resultant product.

Sprints are limited to one calendar month. When a Sprint's horizon is too long the definition of what is being built may change, complexity may rise, and risk may increase. Sprints enable predictability by ensuring inspection and adaptation of progress toward a Sprint Goal at least every calendar month. Sprints also limit risk to one calendar month of cost.

Cancelling a Sprint

A Sprint can be cancelled before the Sprint time-box is over. Only the Product Owner has the authority to cancel the Sprint, although he or she may do so under influence from the stakeholders, the Development Team, or the Scrum Master.

A Sprint would be cancelled if the Sprint Goal becomes obsolete. This might occur if the company changes direction or if market or technology conditions change. In general, a Sprint should be cancelled if it no longer makes sense given the circumstances. But, due to the short duration of Sprints, cancellation rarely makes sense.

When a Sprint is cancelled, any completed and "Done" Product Backlog items are reviewed. If part of the work is potentially releasable, the Product Owner typically accepts it. All incomplete Product Backlog Items are re-estimated and put back on the Product Backlog. The work done on them depreciates quickly and must be frequently re-estimated.

Sprint cancellations consume resources, since everyone has to regroup in another Sprint Planning to start another Sprint. Sprint cancellations are often traumatic to the Scrum Team, and are very uncommon.

Sprint Planning

The work to be performed in the Sprint is planned at the Sprint Planning. This plan is created by the collaborative work of the entire Scrum Team.

Sprint Planning is time-boxed to a maximum of eight hours for a one-month Sprint. For shorter Sprints, the event is usually shorter. The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches the Scrum Team to keep it within the time-box.

Sprint Planning answers the following:

- What can be delivered in the Increment resulting from the upcoming Sprint?
- How will the work needed to deliver the Increment be achieved?

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Topic One: What can be done this Sprint?

The Development Team works to forecast the functionality that will be developed during the Sprint. The Product Owner discusses the objective that the Sprint should achieve and the Product Backlog items that, if completed in the Sprint, would achieve the Sprint Goal. The entire Scrum Team collaborates on understanding the work of the Sprint.

The input to this meeting is the Product Backlog, the latest product Increment, projected capacity of the Development Team during the Sprint, and past performance of the Development Team. The number of items selected from the Product Backlog for the Sprint is solely up to the Development Team. Only the Development Team can assess what it can accomplish over the upcoming Sprint.

After the Development Team forecasts the Product Backlog items it will deliver in the Sprint, the Scrum Team crafts a Sprint Goal. The Sprint Goal is an objective that will be met within the Sprint through the implementation of the Product Backlog, and it provides guidance to the Development Team on why it is building the Increment.

Topic Two: How will the chosen work get done?

Having set the Sprint Goal and selected the Product Backlog items for the Sprint, the Development Team decides how it will build this functionality into a "Done" product Increment during the Sprint. The Product Backlog items selected for this Sprint plus the plan for delivering them is called the Sprint Backlog.

The Development Team usually starts by designing the system and the work needed to convert the Product Backlog into a working product Increment. Work may be of varying size, or estimated effort. However, enough work is planned during Sprint Planning for the Development Team to forecast what it believes it can do in the upcoming Sprint. Work planned for the first days of the Sprint by the Development Team is decomposed by the end of this meeting, often to units of one day or less. The Development Team self-organizes to undertake the work in the Sprint Backlog, both during Sprint Planning and as needed throughout the Sprint.

The Product Owner can help to clarify the selected Product Backlog items and make trade-offs. If the Development Team determines it has too much or too little work, it may renegotiate the selected Product Backlog items with the Product Owner. The Development Team may also invite other people to attend in order to provide technical or domain advice.

By the end of the Sprint Planning, the Development Team should be able to explain to the Product Owner and Scrum Master how it intends to work as a self-organizing team to accomplish the Sprint Goal and create the anticipated Increment.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Sprint Goal

The Sprint Goal is an objective set for the Sprint that can be met through the implementation of Product Backlog. It provides guidance to the Development Team on why it is building the Increment. It is created during the Sprint Planning meeting. The Sprint Goal gives the Development Team some flexibility regarding the functionality implemented within the Sprint. The selected Product Backlog items deliver one coherent function, which can be the Sprint Goal. The Sprint Goal can be any other coherence that causes the Development Team to work together rather than on separate initiatives.

As the Development Team works, it keeps the Sprint Goal in mind. In order to satisfy the Sprint Goal, it implements the functionality and technology. If the work turns out to be different than the Development Team expected, they collaborate with the Product Owner to negotiate the scope of Sprint Backlog within the Sprint.

Daily Scrum

The Daily Scrum is a 15-minute time-boxed event for the Development Team to synchronize activities and create a plan for the next 24 hours. This is done by inspecting the work since the last Daily Scrum and forecasting the work that could be done before the next one. The Daily Scrum is held at the same time and place each day to reduce complexity. During the meeting, the Development Team members explain:

- What did I do yesterday that helped the Development Team meet the Sprint Goal?
- What will I do today to help the Development Team meet the Sprint Goal?
- Do I see any impediment that prevents me or the Development Team from meeting the Sprint Goal?

The Development Team uses the Daily Scrum to inspect progress toward the Sprint Goal and to inspect how progress is trending toward completing the work in the Sprint Backlog. The Daily Scrum optimizes the probability that the Development Team will meet the Sprint Goal. Every day, the Development Team should understand how it intends to work together as a self-organizing team to accomplish the Sprint Goal and create the anticipated Increment by the end of the Sprint. The Development Team or team members often meet immediately after the Daily Scrum for detailed discussions, or to adapt, or replan, the rest of the Sprint's work.

The Scrum Master ensures that the Development Team has the meeting, but the Development Team is responsible for conducting the Daily Scrum. The Scrum Master teaches the Development Team to keep the Daily Scrum within the 15-minute time-box.

The Scrum Master enforces the rule that only Development Team members participate in the Daily Scrum.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Daily Scrums improve communications, eliminate other meetings, identify impediments to development for removal, highlight and promote quick decision-making, and improve the Development Team's level of knowledge. This is a key inspect and adapt meeting.

Sprint Review

A Sprint Review is held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed. During the Sprint Review, the Scrum Team and stakeholders collaborate about what was done in the Sprint. Based on that and any changes to the Product Backlog during the Sprint, attendees collaborate on the next things that could be done to optimize value. This is an informal meeting, not a status meeting, and the presentation of the Increment is intended to elicit feedback and foster collaboration.

This is a four-hour time-boxed meeting for one-month Sprints. For shorter Sprints, the event is usually shorter. The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches all to keep it within the time-box.
The Sprint Review includes the following elements:

- Attendees include the Scrum Team and key stakeholders invited by the Product Owner;
- The Product Owner explains what Product Backlog items have been "Done" and what has not been "Done";
- The Development Team discusses what went well during the Sprint, what problems it ran into, and how those problems were solved;
- The Development Team demonstrates the work that it has "Done" and answers questions about the Increment;
- The Product Owner discusses the Product Backlog as it stands. He or she projects likely completion dates based on progress to date (if needed);
- The entire group collaborates on what to do next, so that the Sprint Review provides valuable input to subsequent Sprint Planning;
- Review of how the marketplace or potential use of the product might have changed what is the most valuable thing to do next; and,
- Review of the timeline, budget, potential capabilities, and marketplace for the next anticipated release of the product.

The result of the Sprint Review is a revised Product Backlog that defines the probable Product Backlog items for the next Sprint. The Product Backlog may also be adjusted overall to meet new opportunities.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Sprint Retrospective

The Sprint Retrospective is an opportunity for the Scrum Team to inspect itself and create a plan for improvements to be enacted during the next Sprint.

The Sprint Retrospective occurs after the Sprint Review and prior to the next Sprint Planning. This is a three-hour time-boxed meeting for one-month Sprints. For shorter Sprints, the event is usually shorter. The Scrum Master ensures that the event takes place and that attendants understand its purpose. The Scrum Master teaches all to keep it within the time-box. The Scrum Master participates as a peer team member in the meeting from the accountability over the Scrum process.

The purpose of the Sprint Retrospective is to:

- Inspect how the last Sprint went with regards to people, relationships, process, and tools;
- Identify and order the major items that went well and potential improvements; and,
- Create a plan for implementing improvements to the way the Scrum Team does its work.

The Scrum Master encourages the Scrum Team to improve, within the Scrum process framework, its development process and practices to make it more effective and enjoyable for the next Sprint. During each Sprint Retrospective, the Scrum Team plans ways to increase product quality by adapting the definition of "Done" as appropriate.

By the end of the Sprint Retrospective, the Scrum Team should have identified improvements that it will implement in the next Sprint. Implementing these improvements in the next Sprint is the adaptation to the inspection of the Scrum Team itself. Although improvements may be implemented at any time, the Sprint Retrospective provides a formal opportunity to focus on inspection and adaptation.

Scrum Artifacts

Scrum's artifacts represent work or value to provide transparency and opportunities for inspection and adaptation. Artifacts defined by Scrum are specifically designed to maximize transparency of key information so that everybody has the same understanding of the artifact.

Product Backlog

The Product Backlog is an ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product. The Product Owner is responsible for the Product Backlog, including its content, availability, and ordering.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

A Product Backlog is never complete. The earliest development of it only lays out the initially known and best-understood requirements. The Product Backlog evolves as the product and the environment in which it will be used evolves. The Product Backlog is dynamic; it constantly changes to identify what the product needs to be appropriate, competitive, and useful. As long as a product exists, its Product Backlog also exists.

The Product Backlog lists all features, functions, requirements, enhancements, and fixes that constitute the changes to be made to the product in future releases. Product Backlog items have the attributes of a description, order, estimate and value.

As a product is used and gains value, and the marketplace provides feedback, the Product Backlog becomes a larger and more exhaustive list. Requirements never stop changing, so a Product Backlog is a living artifact. Changes in business requirements, market conditions, or technology may cause changes in the Product Backlog.

Multiple Scrum Teams often work together on the same product. One Product Backlog is used to describe the upcoming work on the product. A Product Backlog attribute that groups items may then be employed.

Product Backlog refinement is the act of adding detail, estimates, and order to items in the Product Backlog. This is an ongoing process in which the Product Owner and the Development Team collaborate on the details of Product Backlog items. During Product Backlog refinement, items are reviewed and revised. The Scrum Team decides how and when refinement is done. Refinement usually consumes no more than 10% of the capacity of the Development Team. However, Product Backlog items can be updated at any time by the Product Owner or at the Product Owner's discretion.

Higher ordered Product Backlog items are usually clearer and more detailed than lower ordered ones. More precise estimates are made based on the greater clarity and increased detail; the lower the order, the less detail. Product Backlog items that will occupy the Development Team for the upcoming Sprint are refined so that any one item can reasonably be "Done" within the Sprint time-box. Product Backlog items that can be "Done" by the Development Team within one Sprint are deemed "Ready" for selection in a Sprint Planning. Product Backlog items usually acquire this degree of transparency through the above described refining activities.

The Development Team is responsible for all estimates. The Product Owner may influence the Development Team by helping it understand and select trade-offs, but the people who will perform the work make the final estimate.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Monitoring Progress Toward a Goal

At any point in time, the total work remaining to reach a goal can be summed. The Product Owner tracks this total work remaining at least every Sprint Review. The Product Owner compares this amount with work remaining at previous Sprint Reviews to assess progress toward completing projected work by the desired time for the goal. This information is made transparent to all stakeholders.

Various projective practices upon trending have been used to forecast progress, like burn-downs, burn-ups, or cumulative flows. These have proven useful. However, these do not replace the importance of empiricism. In complex environments, what will happen is unknown. Only what has happened may be used for forward-looking decision-making.

Sprint Backlog

The Sprint Backlog is the set of Product Backlog items selected for the Sprint, plus a plan for delivering the product Increment and realizing the Sprint Goal. The Sprint Backlog is a forecast by the Development Team about what functionality will be in the next Increment and the work needed to deliver that functionality into a "Done" Increment.

The Sprint Backlog makes visible all of the work that the Development Team identifies as necessary to meet the Sprint Goal.

The Sprint Backlog is a plan with enough detail that changes in progress can be understood in the Daily Scrum. The Development Team modifies the Sprint Backlog throughout the Sprint, and the Sprint Backlog emerges during the Sprint. This emergence occurs as the Development Team works through the plan and learns more about the work needed to achieve the Sprint Goal.

As new work is required, the Development Team adds it to the Sprint Backlog. As work is performed or completed, the estimated remaining work is updated. When elements of the plan are deemed unnecessary, they are removed. Only the Development Team can change its Sprint Backlog during a Sprint. The Sprint Backlog is a highly visible, real-time picture of the work that the Development Team plans to accomplish during the Sprint, and it belongs solely to the Development Team.

Monitoring Sprint Progress

At any point in time in a Sprint, the total work remaining in the Sprint Backlog can be summed. The Development Team tracks this total work remaining at least for every Daily Scrum to project the likelihood of achieving the Sprint Goal. By tracking the remaining work throughout the Sprint, the Development Team can manage its progress.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Increment

The Increment is the sum of all the Product Backlog items completed during a Sprint and the value of the increments of all previous Sprints. At the end of a Sprint, the new Increment must be "Done," which means it must be in useable condition and meet the Scrum Team's definition of "Done." It must be in useable condition regardless of whether the Product Owner decides to actually release it.

Artifact Transparency

Scrum relies on transparency. Decisions to optimize value and control risk are made based on the perceived state of the artifacts. To the extent that transparency is complete, these decisions have a sound basis. To the extent that the artifacts are incompletely transparent, these decisions can be flawed, value may diminish and risk may increase.

The Scrum Master must work with the Product Owner, Development Team, and other involved parties to understand if the artifacts are completely transparent. There are practices for coping with incomplete transparency; the Scrum Master must help everyone apply the most appropriate practices in the absence of complete transparency. A Scrum Master can detect incomplete transparency by inspecting the artifacts, sensing patterns, listening closely to what is being said, and detecting differences between expected and real results.

The Scrum Master's job is to work with the Scrum Team and the organization to increase the transparency of the artifacts. This work usually involves learning, convincing, and change. Transparency doesn't occur overnight, but is a path.

Definition of "Done"

When a Product Backlog item or an Increment is described as "Done", everyone must understand what "Done" means. Although this varies significantly per Scrum Team, members must have a shared understanding of what it means for work to be complete, to ensure transparency. This is the definition of "Done" for the Scrum Team and is used to assess when work is complete on the product Increment.

The same definition guides the Development Team in knowing how many Product Backlog items it can select during a Sprint Planning. The purpose of each Sprint is to deliver Increments of potentially releasable functionality that adhere to the Scrum Team's current definition of "Done." Development Teams deliver an Increment of product functionality every Sprint. This Increment is useable, so a Product Owner may choose to immediately release it. If the definition of "done" for an increment **is** part of the conventions, standards or guidelines of the development organization,

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

all Scrum Teams must follow it as a minimum. If "done" for an increment is **not** a convention of the development organization, the Development Team of the Scrum Team must define a definition of “done” appropriate for the product. If there are multiple Scrum Teams working on the system or product release, the development teams on all of the Scrum Teams must mutually define the definition of “Done.”

Each Increment is additive to all prior Increments and thoroughly tested, ensuring that all Increments work together.

As Scrum Teams mature, it is expected that their definitions of “Done” will expand to include more stringent criteria for higher quality. Any one product or system should have a definition of “Done” that is a standard for any work done on it.

End Note

Scrum is free and offered in this Guide. Scrum’s roles, artifacts, events, and rules are immutable and although implementing only parts of Scrum is possible, the result is not Scrum. Scrum exists only in its entirety and functions well as a container for other techniques, methodologies, and practices.

Acknowledgements

People

Of the thousands of people who have contributed to Scrum, we should single out those who were instrumental in its first ten years. First there was Jeff Sutherland working with Jeff McKenna, and Ken Schwaber working with Mike Smith and Chris Martin. Many others contributed in the ensuing years and without their help Scrum would not be refined as it is today.

History

Ken Schwaber and Jeff Sutherland first co-presented Scrum at the OOPSLA conference in 1995. This presentation essentially documented the learning that Ken and Jeff gained over the previous few years applying Scrum.

The history of Scrum is already considered long. To honor the first places where it was tried and refined, we recognize Individual, Inc., Fidelity Investments, and IDX (now GE Medical).

The Scrum Guide documents Scrum as developed and sustained for 20-plus years by Jeff Sutherland and Ken Schwaber. Other sources provide you with patterns, processes, and insights that complement the Scrum framework. These optimize productivity, value, creativity, and pride.

©2014 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons, accessible at http://creativecommons.org/licenses/by-sa/4.0/legalcode and also described in summary form at http://creativecommons.org/licenses/by-sa/4.0/. By utilizing this Scrum Guide you acknowledge and agree that you have read and agree to be bound by the terms of the Attribution Share-Alike license of Creative Commons.

Software Quality Assurance

The Definition of Software Quality

Many attempts have been made to define quality, none of which are completely satisfying. However, one simple and quite widely accepted definition is that **quality** is the ability of a product or service to meet the evolving needs and desires of its stakeholders. Note first that this definition applies to both products and services, so it is quite broad. A **stakeholder** in something is anyone affected by or involved in or influencing it. The intuitive notion of quality is that it is about how "good" something is. Presumably, something is good or bad for someone, and who better to use as a standard for how good or bad something is than its stakeholders? But stakeholders are a broad group, so once more this definition casts a wide net in characterizing quality. Finally, note that the word *evolving* is present in this definition. The idea is that the quality of a product or service changes over time because the needs and desires of stakeholders change. The quality of something may decline (or increase) over time even if it has not changed because its stakeholders need or want something different from it. Hence quality must be evaluated continuously.

Software quality is the quality of a software product or service, that is, the ability of a software product or service to meet the evolving needs and desires of its stakeholders. Although pithy, this definition is somewhat vague. Consequently, many people have identified *dimensions* or *aspects* of software quality delineating particular ways in which a product or service succeeds or fails to meet stakeholder needs and desires. One specification of quality dimensions that has become particularly widespread was developed by the International Standards Organization (ISO) as part of its effort to define and promote quality assurance processes. The standard, designated ISO/IEC 25010 [1], defines software quality along eight dimensions: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.

Functional suitability is the degree to which the product satisfies user needs. It includes the functional completeness, correctness, and appropriateness of the software product.

Performance efficiency is related to the processing times and throughput rates of the product, the resources used by the product, and the capacity of the product.

Compatibility is the degree to which the product can co-exist with and interoperate with other products.

Usability is the degree to which specified users can learn a product and actually use it to achieve specified goals to their satisfaction. It includes the ability of users to recognize that a product is appropriate for their needs, learn to use the system, and operate the system.

Reliability is the degree to which the product performs specified functions under specified conditions. It includes the availability of the product (the proportion of time that the product is operating properly when needed) and its ability to recover from interruptions.

Security is the degree to which the product provides confidentiality (access only to legitimate users), integrity (prevention of unauthorized modifications), authenticity (identifying users), non-repudiation (proving that changes have occurred), and accountability (ability to associate changes with particular agents).

Maintainability is the degree to which the product can be modified, improved, corrected, and adapted, and components of the product can be re-used.

Portability is the ease with which a product can be made to operate in a new or altered computing environment.

These dimensions of quality are standard and we will use this terminology throughout the book.

Quality Assurance

Quality assurance (QA) is a systematic pattern of activities intended to ensure that a product has adequate quality, that is, that it properly satisfies the needs and desires of its stakeholders. Though there is sometimes overlap, QA activities can be characterized as either validation-oriented or verification-oriented.[1]

> **Validation** is the process of determining if a product (or its specification) satisfies stakeholders' needs and desires.
>
> **Verification** is the process of determining if a product (or its specification) satisfies those needs and desires properly.

Validation is sometimes described as the process of answering the question "Are we building the proper or right product?" and verification is sometimes described as the process of answering the question "Are we building the product properly or right?"[2] In both cases, the objective is to determine if the product contains **defects**, which are any undesirable aspect of a product.

Perhaps the easiest way to understand the distinction between validation and verification is with an example. Suppose you are a newspaper reporter and you are told to write an article about homelessness (the product). You must both write about homelessness (the topic that you were given; the proper product) and you must write properly (use proper grammar, follow the newspaper's style guide, etc…). If you write an article about homelessness then your product can be validated. If you write an article properly then your product can be verified.

All validation activities are product-specific. In other words, validation is the process of determining if a specific product satisfies needs and desires. In the context of the newspaper example, an editor will not be able to validate your article if she/he does not know the product you were told to create.

On the other hand, some verification activities are product-specific and some are not. That is, some verification activities require information about the product and some do not. Again in the context of the newspaper example, a copy editor will be able to determine if your article is grammatical and follows the style guide without knowing what product you were told to create. However, an assignment editor will not be able to determine if your argument is compelling without knowing what product you were told to create.

Validation is sometimes described as involving customer satisfaction and verification is sometimes described as involving conformance to specifications. However, while intuitive, this is too simplistic. First, it ignores the fact that verification can occur at all phases of the process. That is, needs lists,

---

[1] Unfortunately, there are many different definitions of these terms. IEEE-610 defines validation as the "process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements" and verification is the "process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase". ANSI/EIA-632 defines validation as "confirmation by examination and provision of objective evidence that the specific intended use of a product … is accomplished in an intended usage environment and verification as "confirmation by examination and provision of objective evidence that the specified requirements to which a product was built, coded or assembled have been fulfilled".

[2] The word "proper" is used here, rather than the word "correct", to avoid confusion later. When referring to needs lists, the word "correct" is used to refer to items that accurately reflect a stakeholder's need and are prioritized appropriately. When referring to software requirements specifications, the word "correct" is used to refer to a product that is a reasonable compromise among conflicting needs and desires.

requirements specifications, engineering designs and code can (and should) all be verified. Second, it ignores the fact that some aspects of verification are independent of the particular product (e.g., readable code).

Note that it may be impossible to validate a product that can't be verified. Continuing with the example of the reporter, it may not be possible to determine whether the story is about the right topic if it is so ungrammatical as to be unreadable.

Note, also, that a product can fail to satisfy needs and desires properly for different reasons, and that these reasons may be readily apparent in some circumstances but difficult to ascertain in others. For example, consider a word processor in which the users want to have misspelled words rendered in red. Now suppose that the product, in fact, renders misspelled words in blue. Clearly, this is a simple matter of the wrong color being used. However, now suppose that the misspelled words are rendered in black. This could either be because spelling isn't being checked at all, the spell-checking algorithm is not working properly, or because the wrong color is being used.

Defect Elimination

Given that QA is the systematic pattern of activities intended to ensure that a product properly satisfies the needs and desires of its stakeholders, and that a defect is any undesirable aspect of a product, it is clear than an important aspect of QA is the elimination of defects. In general, defects can be eliminated in one of two ways: they can be detected and removed, or they can be prevented.

Defect detection and removal is easy to understand and, hence, widely used. Unfortunately, defect detection and removal has many drawbacks. First, while it is easy to understand, it is difficult and costly to do. Both detection and removal require special training, special tools, and time. Second, when too many defects are detected products may be cancelled. When defects go undetected customer satisfaction problems often arise. Hence relying only on defect detection and removal can lead to serious problems.

In contrast, when defects are prevented, then no cost and time are spent finding and fixing them, and there are no bad consequences from failing to find them (because they are not there to be found). Hence, defect prevention is almost always superior to defect detection and removal.

The Centrality of Processes

Defects can be prevented only if the processes delivering a product or service make it unlikely that defects will occur. Also, defects can be found and eliminated only if there are defect detection and removal tasks built into product or service delivery processes. Therefore, processes are central to achieving quality. **Process improvement** is the analysis and modification of a process to enhance its efficiency, productivity, or the quality of its output. Process improvement is the fundamental means for improving and sustaining quality.

The centrality of processes for quality means that quality assurance is process driven, working towards the continuous improvement of an organization's processes. But processes cannot be improved using a single method or technique, and improvement cannot be done by QA specialists alone. Instead, an entire organization must be committed to quality, using a variety of tools and techniques and involving everyone in the organization. Furthermore, because of the relative costs of prevention and defect detection and removal, quality improvement should be oriented towards defect prevention, while recognizing that some defects will occur and incorporating detection and removal steps as necessary.

Companies committed to quality assurance are data driven, but not narrowly. They collect data about products, but also about processes and customers. Ultimately, such companies are *customer focused*—no data is more important that data about customer satisfaction.

A successful quality culture involves periodic training and education for all members of the organization. It also involves specific evolving quality goals and the clear communication of those goals.

Defect Prevention

A variety of different tools and techniques have been developed that attempt to prevent defects in software products.

Of particular interest are the tools and techniques that can be used during multiple phases of the process. These tools and techniques can be thought of as **process guides**: standards and guidelines (e.g., documentation standards, programming style guides, natural language usage guides) that describe how people should behave as well as templates and checklists that make it easier to do so. Process guides are important because they prevent common mistakes from being made. However, they are also important because they lead to behaviors that reduce the number of uncommon mistakes.

Closely related, but somewhat more specific, are **analysis and design methodologies**. These are well-codified approaches to understanding and solving software-related problems. Perhaps the best known example is object-oriented analysis and design.

Another way to prevent defects is to use well-studied solutions to problems or, to use a colloquial expression, not reinventing the wheel. In software quality assurance this approach to defect prevention can take many forms. At a high level of abstraction, it includes the use of **reference architectures**, which are configurable architectures for an application domain. At a lower level of abstraction, it includes **design patterns**, which are customizable solutions to a design problems. At a still lower level of abstraction are well-studied **algorithms and data structures**. The most extreme example of this is the reuse of code and other assets.

To the extent that you must reinvent the wheel, there are two different approaches that can be taken to prevent defects in software. First, formal methods can be used to prove that a software product will behave correctly. These methods rely on the use of mathematical models to determine whether a program meets requirements either without executing the program or by simulating the execution of the program. Second, **prototypes** can be used to identify likely defects before they arise.

Throwaway/exploratory prototypes are developed to help the design process and then discarded; relatively little time and effort are devoted to them. **Evolutionary prototypes**, on the other hand, are created as part of an iterative process and ultimately become part of the final product; they must be well-designed and flexible.

Since defects arise over time, **version control** and **configuration management tools** can also be an effective way to prevent defects. These tools can be used to control change and ensure consistency in many different kinds of documents (e.g., requirements specifications, design documents, and code).

Similarly, **computer-aided software engineering** (CASE) tools can play an important role in preventing defects. These tools range from the fairly simple (e.g., color-coding programming language constructs) to the fairly sophisticated (e.g., round-trip integration of UML diagrams and implementations in code).

Finally, all defect prevention techniques discussed above rely on people knowing about them and using them correctly and effectively. Thus in some sense the most fundamental defect prevention technique of all is **training and education** in software processes, good practices, methods, and tools.

Defect Detection and Removal

Obviously, not all defects can be prevented. Hence, some aspects of SQA are concerned with the detection and removal of defects. These techniques can be classified as "review and correct" or "test and debug."

Review and Correct

Review and correct techniques can be used during all phases of the software engineering process. In many ways, review and correct techniques are very similar to one another; they vary principally in their levels of formality and in their degree of automation.

Automated tools include style and standards checkers (e.g., spelling and grammar checkers, and program syntax and style checkers) and consistency and completeness checkers. These will be discussed more fully in the chapter on implementation.

Manual techniques involve one or more individuals examining the work product (be it a requirements specification, design, or code). The most informal such technique is the **desk check**, which involves a single individual (usually the author of the product) looking for and correcting defects. Slightly more formal is the **walkthrough**, which involves one individual (usually the author of the product) guiding one or more others through the examination. For example, a walkthrough might be done by showing a group of developers some code or part of a design or requirements specification on an overhead projector and then having a discussion about any defects and how to correct them.

An **inspection** is a formal examination of a product by a team of trained inspectors. Because of their effectiveness and applicability to most software development work products, inspections are particularly important.

Each participant in an inspection has particular role. The process is overseen by a **moderator** who schedules and runs meetings, organizes and distributes the materials to be used, reports on results, and monitors follow-up activities. The **author** of the inspected work product is a meeting participant. The team meeting is guided by a **reader**. In some cases, the reader literally reads the product line-by-line; in other cases, the reader simply instructs the other inspectors to consider a particular part of the product. The **recorder** takes notes during inspection meetings. One inspector may be the recorder for all meetings or the recorder may change for each meeting. In general, the recorder should not be the moderator. The remaining participants are **inspectors** who check the work product beforehand and help find defects during the meeting.

The inspection process proceeds as follows.

1. **Readiness check**. The moderator ensures that the product has been through all other reviews (whether automated or manual) and any defects that were found have been corrected. In other words, at the time of an inspection, the product is thought to be free of defects.
2. **Overview meeting**. The author introduces and distributes the work product (and supporting materials) to the rest of the team. In many cases, this is done electronically and the team members do not actually meet.

3. **Preparation**. Each inspector (guided by a checklist) reviews the work product individually, making notes on either the checklist or the work product, as appropriate.
4. **Team inspection**. The reader guides the other inspectors through the work product. They comment on it (using the notes they made while preparing), and the recorder takes notes (recording the location of the problem, its nature, and its severity).
5. **Correction**. The moderator provides the results of the inspection to the author, who corrects all the defects.
6. **Follow-up**. The moderator ensures that the defects are, in fact, corrected, and determines if another inspection is necessary.

Inspections are most successful when the guidelines below are followed.

- Inspectors are guided by a good checklist. Checklists become good by being improved over time: items that don't help find defects are removed, and defects that are missed in inspections have items added to help find them.
- Information about the defects is specific.
- Inspectors attend at most one team meeting per day.
- Inspection meetings last at most two hours.
- The moderator is not a manager.
- The report and meeting interactions are not judgmental—attention is focussed on defects, not on the abilities of the author of the work product.
- The report is written and delivered to the author within 24 hours.

In 1976, Michael Fagan, who invented this technique and used it to inspect code, found that 67% of all faults found in a development effort were found in code inspections, and that an inspected product had 38% fewer faults after seven months in the field than a comparable product developed with walkthroughs. Later experiments and studies have confirmed the efficacy of inspections right down to today.

Test and Debug

While review and correct techniques can be used throughout all phases of the software engineering process, given the complexity of most software products, they are unlikely to detect and remove all defects. Hence, it is also necessary to test software.

> **Testing** is a validation and verification process that makes use of the system or product (including prototypes) while it is in operation or being operated on.

In the context of software products, the phrase "in operation" refers to an executable product (e.g., a word processor) whereas the phrase "being operated on" refers to an input to an executable product (e.g., a document template or stylesheet). Because software testing involves the product while it is "operating" or "being operated on," it is only useful in detecting certain kinds of defects. These defects are referred to as faults and can be defined in terms of failures as follows.

> A **failure** is a deviation between a product's actual behavior and its intended behavior (e.g., a feature that is missing or incorrect).
>
> A **fault** is a defect that could (or does) give rise to a failure.

In the testing of software products it is important to distinguish between cause and effect, and the manifestations of causes and effects. To that end, the following definition is often used.

A **trigger** is a condition that causes a fault to result in a failure.

Taking this into account, software testing can be defined more concretely as follows.

Software testing is the process of identifying failures by creating "inputs" and applying them to the software product.

A particular choice of "inputs" is referred to as a **test case** (or **test point**, or **test**). A collection of test cases or points is referred to as a **test suite**.

After testing has identified a fault it must be corrected. This process, which can be fairly complicated, is known as debugging.

Debugging is the process of using trigger conditions to identify and correct faults.

It is relatively inexpensive (compared to other engineering disciplines) to test software (after it has been implemented), but relatively expensive (compared to testing) to debug software. Hence, because it is easier to debug small units, software should be tested extensively and at various times throughout the process. The different kinds of tests and testing (the details of which are described in subsequent chapters) are often characterized as illustrated in the following diagram.

Unit testing exercises a single sub-program in isolation from any other sub-programs, **integration testing** exercises multiple sub-programs in combination, and **alpha** and **beta testing** (sometimes referred to collectively as **system testing**) exercise the entire software product.

Unit testing is conducted by the author of the sub-program and sometimes also dedicated testers (i.e., members of the organization whose main job is testing). Integration testing can involve the authors, testers of both. Alpha testing (which is sometimes subdivided into function testing and

performance testing) typically involves dedicated testers, and beta testing (which is sometimes subdivided into acceptance testing and installation testing) always involves users.

Obviously, a test can't be applied until after the relevant modules have been implemented. However, a test can be created before the relevant modules have been implemented. In fact, a test can be created before the relevant module has even been designed (in a detailed sense). As discussed below, the implications of this observation are somewhat different for traditional and agile processes.

Test and Debug in a Traditional Process

As noted in the chapter on requirements, in a traditional process, specifications should be testable or verifiable. Thus, the requirement specification is a good place to start when developing tests for a traditional process—each requirement should be mapped to one or more tests. Some requirements will be mapped to unit tests and others will be mapped to system tests. (In practice, few requirements tend to be mapped to integration tests.) While the requirements specification is a good place to start when developing tests, care must still be taken to ensure that the test suite is complete.

Unit testing is part of the implementation or construction phase. Hence, it tends to occur early and often in a traditional process. Unit authors are intimately involved with unit testing in a traditional process but, because of the length of cycles, testers are also often involved in the creation and execution of unit tests. Since integration testing and system testing occur much later, rigorous unit testing (and the debugging that results) is critical in a traditional process.

Integration testing often doesn't begin until fairly late in traditional processes. In other words, it is common for a large number of units to be constructed before they are combined and tested. As a result, in a traditional process it is often necessary to create **stubs**—placeholder code that stands in for units that have not yet been constructed. A stub has the same interface as the code it is standing in for, but provides none (or very little) of the required functionality. Unfortunately, when the stub is replaced with the fully-functional unit during integration testing, faults are often uncovered.

System testing often can't begin until the end of a traditional process. This is because traditional processes make no effort to have a working (if limited) version of the product at all times. Since the system can't be tested in its entirety until it exists, and it doesn't exist until the end of a traditional process, system testing can't occur until very late. Furthermore, traditional processes tend to have long cycle times, so there is typically a long period between system specification and system test. This is, perhaps, the biggest reason that traditional processes are not considered agile.

Because of the length of the cycles in traditional processes, the product released at the end of each cycle is often thought of as the "final product" (even though the product has a version number), and changes are described in relation to it. Of course, though one would like to think that all such changes are improvements, any change to a software product might introduce new faults. **Regression testing** is used to determine if changes to a product have introduced faults. In general, regression testing does not involve new tests. Rather, in regression testing the new product is subjected to the same tests as the original product.

Test and Debug in Scrum

In a traditional process, tests are created from the requirements specification. Since the Scrum process does not make use of a formal requirements specifications, test development in Scrum proceeds differently from test development in a traditional process.

Perhaps most importantly, in Scrum integration testing and system testing are conducted each and every sprint. Indeed, proponents of most agile processes argue that frequent integration and system

testing is a big part of what it means to be agile. In Scrum, integration and system testing are used to demonstrate that features in the product backlog have been completed.

Unit testing in Scrum is typically left entirely to the authors. Those who argue for this approach believe that unit testing will find implementation failures and integration and system testing will find design failures. Alternatively, one can specify *acceptance criteria* (also known as *conditions of satisfaction*) as unit tests. Using this approach, it becomes very clear when a user-level PBI has been completed.

The Efficiency of Defect Detection and Removal Techniques

The **defect removal efficiency** of an activity is the number of defects removed from a work product by the activity divided by the number of defects present in the work product at the start of the activity. Different techniques have different defect removal efficiencies. The numbers in the table below show defect removal efficiencies for several activities in a study from 2013 [2].

Technique	Minimum (%)	Median (%)	Maximum (%)
Requirements Review (informal)	20	30	50
Top-level design reviews (informal)	30	40	60
Detailed functional design inspection	30	65	80
Detailed logic design inspection	35	65	75
Code inspection or static analysis	35	60	90
Unit tests	10	25	50
Integration tests	25	45	60
System test	25	50	65
External Beta tests	15	40	75

It is important to note that different defect detection and removal techniques tend to find different kinds of defects, so a range of detection and removal techniques is needed. For example, requirements reviews tend to find failures to meet customer needs, while design reviews tend to find failures to meet requirements.

It is also important to note that, though many of these techniques have low efficiencies, because they are likely to be independent (in a probabilistic sense), using multiple techniques in sequence will likely produce a large defect removal efficiency. For example, if four independent defect detection and removal techniques each work with a probability of 0.6 (i.e., an efficiency of 60%) and they are used in sequence, the cumulative defect removal efficiency will be $1-(1-0.6)^4 \approx 97.5\%$.

A Defect Elimination Taxonomy

We have now discussed many defect elimination techniques, in various categories. The diagram below summarizes this material in a taxonomic tree.

References

1. ISO/IEC, *Software Product Quality*, http://iso25000.com/index.php/en/iso-25000-standards/iso-25010.
2. Jones, Capers. *Software Quality In 2013: A Survey Of The State Of The Art*. Http://Namcookanalytics.Com/Software-Quality-Survey-State-Art/.

Version Control with Subversion

For Subversion 1.7

(Compiled from r4880)

Ben Collins-Sussman
Brian W. Fitzpatrick
C. Michael Pilato

Version Control with Subversion: For Subversion 1.7: (Compiled from r4880)

by Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato

Copyright © 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011 Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Table of Contents

Foreword xii
Preface xiv
- What Is Subversion? xiv
  - Is Subversion the Right Tool? xiv
  - Subversion's History xv
  - Subversion's Architecture xvi
  - Subversion's Components xviii
  - What's New in Subversion xviii
- Audience xix
- How to Read This Book xix
- Organization of This Book xx
- This Book Is Free xxi
- Acknowledgments xxii

1. Fundamental Concepts 1
- Version Control Basics 1
  - The Repository 1
  - The Working Copy 2
  - Versioning Models 2
- Version Control the Subversion Way 7
  - Subversion Repositories 7
  - Revisions 7
  - Addressing the Repository 8
  - Subversion Working Copies 9
- Summary 14

2. Basic Usage 15
- Help! 15
- Getting Data into Your Repository 16
  - Importing Files and Directories 16
  - Recommended Repository Layout 17
  - What's In a Name? 17
- Creating a Working Copy 18
- Basic Work Cycle 19
  - Update Your Working Copy 20
  - Make Your Changes 20
  - Review Your Changes 22
  - Fix Your Mistakes 25
  - Resolve Any Conflicts 26
  - Commit Your Changes 33
- Examining History 34
  - Examining the Details of Historical Changes 34
  - Generating a List of Historical Changes 36
  - Browsing the Repository 38
  - Fetching Older Repository Snapshots 40
- Sometimes You Just Need to Clean Up 41
  - Disposing of a Working Copy 42
  - Recovering from an Interruption 42
- Dealing with Structural Conflicts 42
  - An Example Tree Conflict 43
- Summary 48

3. Advanced Topics 49
- Revision Specifiers 49
  - Revision Keywords 49
  - Revision Dates 50

Peg and Operative Revisions 51
Properties 55
Why Properties? 56
Manipulating Properties 57
Properties and the Subversion Workflow 60
Automatic Property Setting 62
File Portability 63
File Content Type 63
File Executability 64
End-of-Line Character Sequences 65
Ignoring Unversioned Items 66
Keyword Substitution 69
Sparse Directories 72
Locking 77
Creating Locks 78
Discovering Locks 80
Breaking and Stealing Locks 81
Lock Communication 83
Externals Definitions 84
Changelists 90
Creating and Modifying Changelists 90
Changelists As Operation Filters 92
Changelist Limitations 94
Network Model 94
Requests and Responses 94
Client Credentials 95
Summary 97
4. Branching and Merging 99
What's a Branch? 99
Using Branches 99
Creating a Branch 101
Working with Your Branch 102
The Key Concepts Behind Branching 105
Basic Merging 105
Changesets 105
Keeping a Branch in Sync 106
Reintegrating a Branch 111
Mergeinfo and Previews 113
Undoing Changes 116
Resurrecting Deleted Items 117
Advanced Merging 119
Cherrypicking 119
Merge Syntax: Full Disclosure 121
Merges Without Mergeinfo 122
More on Merge Conflicts 123
Blocking Changes 125
Keeping a Reintegrated Branch Alive 126
Merge-Sensitive Logs and Annotations 127
Noticing or Ignoring Ancestry 129
Merges and Moves 130
Blocking Merge-Unaware Clients 130
The Final Word on Merge Tracking 131
Traversing Branches 132
Tags 134
Creating a Simple Tag 134
Creating a Complex Tag 134
Branch Maintenance 135
Repository Layout 135

Data Lifetimes ... 136
Common Branching Patterns ... 137
Release Branches ... 137
Feature Branches ... 138
Vendor Branches ... 138
General Vendor Branch Management Procedure ... 139
svn_load_dirs.pl ... 141
To Branch or Not to Branch? ... 142
Summary ... 143
5. Repository Administration ... 145
The Subversion Repository, Defined ... 145
Strategies for Repository Deployment ... 146
Planning Your Repository Organization ... 146
Deciding Where and How to Host Your Repository ... 148
Choosing a Data Store ... 149
Creating and Configuring Your Repository ... 152
Creating the Repository ... 152
Implementing Repository Hooks ... 153
Berkeley DB Configuration ... 154
FSFS Configuration ... 154
Repository Maintenance ... 155
An Administrator's Toolkit ... 155
Commit Log Message Correction ... 159
Managing Disk Space ... 159
Berkeley DB Recovery ... 163
Migrating Repository Data Elsewhere ... 164
Filtering Repository History ... 168
Repository Replication ... 171
Repository Backup ... 178
Managing Repository UUIDs ... 179
Moving and Removing Repositories ... 180
Summary ... 181
6. Server Configuration ... 182
Overview ... 182
Choosing a Server Configuration ... 183
The svnserve Server ... 183
svnserve over SSH ... 183
The Apache HTTP Server ... 184
Recommendations ... 184
svnserve, a Custom Server ... 185
Invoking the Server ... 185
Built-in Authentication and Authorization ... 189
Using **svnserve** with SASL ... 191
Tunneling over SSH ... 193
SSH Configuration Tricks ... 195
httpd, the Apache HTTP Server ... 196
Prerequisites ... 197
Basic Apache Configuration ... 197
Authentication Options ... 199
Authorization Options ... 202
Protecting network traffic with SSL ... 205
Extra Goodies ... 207
Path-Based Authorization ... 214
High-level Logging ... 219
Server Optimization ... 221
Data Caching ... 221
Network Compression of Data ... 222
Supporting Multiple Repository Access Methods ... 222

7. Customizing Your Subversion Experience ... 225
Runtime Configuration Area ... 225
Configuration Area Layout ... 225
Configuration and the Windows Registry ... 226
Configuration Options ... 227
Localization ... 232
Understanding Locales ... 233
Subversion's Use of Locales ... 233
Using External Editors ... 234
Using External Differencing and Merge Tools ... 235
External diff ... 236
External diff3 ... 237
External merge ... 238
Summary ... 239
8. Embedding Subversion ... 240
Layered Library Design ... 240
Repository Layer ... 241
Repository Access Layer ... 244
Client Layer ... 245
Using the APIs ... 246
The Apache Portable Runtime Library ... 246
Functions and Batons ... 247
URL and Path Requirements ... 247
Using Languages Other Than C and C++ ... 247
Code Samples ... 248
Summary ... 254
9. Subversion Complete Reference ... 255
svn—Subversion Command-Line Client ... 255
svn Options ... 255
svn Subcommands ... 262
svnadmin—Subversion Repository Administration ... 339
svnadmin Options ... 339
svnadmin Subcommands ... 340
svnlook—Subversion Repository Examination ... 363
svnlook Options ... 363
svnlook Subcommands ... 365
svnsync—Subversion Repository Mirroring ... 385
svnsync Options ... 385
svnsync Subcommands ... 387
svnrdump—Remote Subversion Repository Data Migration ... 395
svnrdump Options ... 395
svnrdump Subcommands ... 396
svnserve—Custom Subversion Server ... 399
svnserve Options ... 400
svndumpfilter—Subversion History Filtering ... 401
svndumpfilter Options ... 401
svndumpfilter Subcommands ... 402
svnversion—Subversion Working Copy Version Info ... 407
mod_dav_svn—Subversion Apache HTTP Server Module ... 409
mod_authz_svn—Subversion Apache HTTP Authorization Module ... 412
Subversion Properties ... 413
Versioned Properties ... 413
Unversioned Properties ... 414
Repository Hooks ... 415
A. Subversion Quick-Start Guide ... 425
Installing Subversion ... 425
High-Speed Tutorial ... 426
B. Subversion for CVS Users ... 428

Revision Numbers Are Different Now 428
Directory Versions 428
More Disconnected Operations 429
Distinction Between Status and Update 429
Status 429
Update 430
Branches and Tags 430
Metadata Properties 431
Conflict Resolution 431
Binary Files and Translation 431
Versioned Modules 431
Authentication 432
Converting a Repository from CVS to Subversion 432
C. WebDAV and Autoversioning 433
What Is WebDAV? 433
Autoversioning 434
Client Interoperability 435
Standalone WebDAV Applications 436
File-Explorer WebDAV Extensions 437
WebDAV Filesystem Implementation 438
D. Copyright 440
Index 445

List of Figures

1. Subversion's architecture xvi
1.1. A typical client/server system 1
1.2. The problem to avoid 2
1.3. The lock-modify-unlock solution 3
1.4. The copy-modify-merge solution 5
1.5. The copy-modify-merge solution (continued) 5
1.6. Tree changes over time 7
1.7. The repository's filesystem 11
4.1. Branches of development 99
4.2. Starting repository layout 100
4.3. Repository with new copy 101
4.4. The branching of one file's history 103
8.1. Files and directories in two dimensions 242
8.2. Versioning time—the third dimension! 243

List of Tables

1.1. Repository access URLs 8
2.1. Common log requests 36
4.1. Branching and merging commands 143
5.1. Repository data store comparison 149
6.1. Comparison of subversion server options 182
C.1. Common WebDAV clients 435

List of Examples

4.1. Merge-tracking gatekeeper start-commit hook script 131
5.1. txn-info.sh (reporting outstanding transactions) 160
5.2. Mirror repository's pre-revprop-change hook script 172
5.3. Mirror repository's start-commit hook script 173
6.1. A sample svnserve launchd job definition 188
6.2. A sample configuration for anonymous access 203
6.3. A sample configuration for authenticated access 204
6.4. A sample configuration for mixed authenticated/anonymous access 204
6.5. Disabling path checks altogether 205
7.1. Sample registration entries (.reg) file 226
7.2. diffwrap.py 236
7.3. diffwrap.bat 237
7.4. diff3wrap.py 237
7.5. diff3wrap.bat 238
7.6. mergewrap.py 238
7.7. mergewrap.bat 239
8.1. Using the repository layer 248
8.2. Using the repository layer with Python 250
8.3. A Python status crawler 252

Foreword

Karl Fogel
Chicago, March 14, 2004.

A bad Frequently Asked Questions (FAQ) sheet is one that is composed not of the questions people actually ask, but of the questions the FAQ's author *wishes* people would ask. Perhaps you've seen the type before:

> Q: How can I use Glorbosoft XYZ to maximize team productivity?
>
> A: Many of our customers want to know how they can maximize productivity through our patented office groupware innovations. The answer is simple. First, click on the `File` menu, scroll down to `Increase Productivity`, then…

The problem with such FAQs is that they are not, in a literal sense, FAQs at all. No one ever called the tech support line and asked, “How can we maximize productivity?” Rather, people asked highly specific questions, such as “How can we change the calendaring system to send reminders two days in advance instead of one?” and so on. But it's a lot easier to make up imaginary Frequently Asked Questions than it is to discover the real ones. Compiling a true FAQ sheet requires a sustained, organized effort: over the lifetime of the software, incoming questions must be tracked, responses monitored, and all gathered into a coherent, searchable whole that reflects the collective experience of users in the wild. It calls for the patient, observant attitude of a field naturalist. No grand hypothesizing, no visionary pronouncements here—open eyes and accurate note-taking are what's needed most.

What I love about this book is that it grew out of just such a process, and shows it on every page. It is the direct result of the authors' encounters with users. It began with Ben Collins-Sussman's observation that people were asking the same basic questions over and over on the Subversion mailing lists: what are the standard workflows to use with Subversion? Do branches and tags work the same way as in other version control systems? How can I find out who made a particular change?

Frustrated at seeing the same questions day after day, Ben worked intensely over a month in the summer of 2002 to write *The Subversion Handbook*, a 60-page manual that covered all the basics of using Subversion. The manual made no pretense of being complete, but it was distributed with Subversion and got users over that initial hump in the learning curve. When O'Reilly decided to publish a full-length Subversion book, the path of least resistance was obvious: just expand the Subversion handbook.

The three coauthors of the new book were thus presented with an unusual opportunity. Officially, their task was to write a book top-down, starting from a table of contents and an initial draft. But they also had access to a steady stream—indeed, an uncontrollable geyser—of bottom-up source material. Subversion was already in the hands of thousands of early adopters, and those users were giving tons of feedback, not only about Subversion, but also about its existing documentation.

During the entire time they wrote this book, Ben, Mike, and Brian haunted the Subversion mailing lists and chat rooms incessantly, carefully noting the problems users were having in real-life situations. Monitoring such feedback was part of their job descriptions at CollabNet anyway, and it gave them a huge advantage when they set out to document Subversion. The book they produced is grounded firmly in the bedrock of experience, not in the shifting sands of wishful thinking; it combines the best aspects of user manual and FAQ sheet. This duality might not be noticeable on a first reading. Taken in order, front to back, the book is simply a straightforward description of a piece of software. There's the overview, the obligatory guided tour, the chapter on administrative configuration, some advanced topics, and of course, a command reference and troubleshooting guide. Only when you come back to it later, seeking the solution to some specific problem, does its authenticity shine out: the telling details that can only result from encounters with the unexpected, the examples honed from genuine use cases, and most of all the sensitivity to the user's needs and the user's point of view.

Of course, no one can promise that this book will answer every question you have about Subversion. Sometimes the precision with which it anticipates your questions will seem eerily telepathic; yet occasionally, you will stumble into a hole in the community's knowledge and come away empty-handed. When this happens, the best thing you can do is email `<users@subversion.apache.org>` and present your problem. The authors are still there and still watching, and the authors include not just the three listed on the cover, but many others who contributed corrections and original material. From the community's point of view, solving your problem is merely a pleasant side effect of a much larger project—namely, slowly adjusting this book, and ultimately Subversion itself, to more closely match the way people actually use it. They are eager to hear from you, not only because they can help you, but because you can help them. With Subversion, as with all active free software projects, *you*

are not alone.

Let this book be your first companion.

Preface

> “It is important not to let the perfect become the enemy of the good, even when you can agree on what perfect is. Doubly so when you can't. As unpleasant as it is to be trapped by past mistakes, you can't make any progress by being afraid of your own shadow during design.”
>
> —Greg Hudson, Subversion developer

In the world of open source software, the Concurrent Versions System (CVS) was the tool of choice for version control for many years. And rightly so. CVS was open source software itself, and its nonrestrictive modus operandi and support for networked operation allowed dozens of geographically dispersed programmers to share their work. It fit the collaborative nature of the open source world very well. CVS and its semi-chaotic development model have since become cornerstones of open source culture.

But CVS was not without its flaws, and simply fixing those flaws promised to be an enormous effort. Enter Subversion. Subversion was designed to be a successor to CVS, and its originators set out to win the hearts of CVS users in two ways—by creating an open source system with a design (and “look and feel”) similar to CVS, and by attempting to avoid most of CVS's noticeable flaws. While the result wasn't—and isn't—the next great evolution in version control design, Subversion *is* very powerful, very usable, and very flexible.

This book is written to document the 1.7 series of the Apache Subversion™[1] version control system. We have made every attempt to be thorough in our coverage. However, Subversion has a thriving and energetic development community, so already a number of features and improvements are planned for future versions that may change some of the commands and specific notes in this book.

What Is Subversion?

Subversion is a free/open source *version control system* (VCS). That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of “time machine.”

Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that conduit—if some incorrect change is made to the data, just undo that change.

Some version control systems are also *software configuration management* (SCM) systems. These systems are specifically tailored to manage trees of source code and have many features that are specific to software development—such as natively understanding programming languages, or supplying tools for building software. Subversion, however, is not one of these systems. It is a general system that can be used to manage *any* collection of files. For you, those files might be source code—for others, anything from grocery shopping lists to digital video mixdowns and beyond.

Is Subversion the Right Tool?

If you're a user or system administrator pondering the use of Subversion, the first question you should ask yourself is: "Is this the right tool for the job?" Subversion is a fantastic hammer, but be careful not to view every problem as a nail.

As a first step, you need to decide if version control in general is required for your purposes. If you need to archive old versions of files and directories, possibly resurrect them, and examine logs of how they've changed over time, then version control tools can do that. If you need to collaborate with people on documents (usually over a network) and keep track of who made which changes, a version control tool can do that, too. In fact, this is why version control tools such as Subversion are so often used in software development environments—working on a development team is an inherently social activity where changes to source code files are constantly being discussed, made, evaluated, and even sometimes unmade. Version control tools facilitate that sort of collabora-

[1] We'll refer to it simply as “Subversion” throughout this book. You'll thank us when you realize just how much space that saves!

tion.

There is cost associated with using version control, too. Unless you can outsource the administration of your version control system to a third-party, you'll have the obvious costs of performing that administration yourself. When working with the data on a daily basis, you won't be able to copy, move, rename, or delete files the way you usually do. Instead, you'll have to do all of those things through the version control system.

Even assuming that you are okay with the cost/benefit tradeoff afforded by a version control system, you shouldn't choose to use one merely because it *can* do what you want. Consider whether your needs are better addressed by other tools. For example, because Subversion replicates data to all the collaborators involved, a common misuse is to treat it as a generic distribution system. People will sometimes use Subversion to distribute huge collections of photos, digital music, or software packages. The problem is that this sort of data usually isn't changing at all. The collection itself grows over time, but the individual files within the collection aren't being changed. In this case, using Subversion is "overkill."[2] There are simpler tools that efficiently replicate data *without* the overhead of tracking changes, such as **rsync** or **unison**.

Once you've decided that you need a version control solution, you'll find no shortage of available options. When Subversion was first designed and released, the predominant methodology of version control was *centralized version control*—a single remote master storehouse of versioned data with individual users operating locally against shallow copies of that data's version history. Subversion quickly emerged after its initial introduction as the clear leader in this field of version control, earning widespread adoption and supplanting installations of many older version control systems. It continues to hold that prominent position today.

Much has changed since that time, though. In the years since the Subversion project began its life, a newer methodology of version control called *distributed version control* has likewise garnered widespread attention and adoption. Tools such as Git (http://git-scm.com/) and Mercurial (http://mercurial.selenic.com/) quickly rose to the tops of the distributed version control system (DVCS) ranks. Distributed version control harnesses the growing ubiquity of high-speed network connections and low storage costs to offer an approach which differs from the centralized model in key ways. First and most obvious is the fact that there is no remote, central storehouse of versioned data. Rather, each user keeps and operates against very deep—complete, in a sense—local version history data stores. Collaboration still occurs, but is accomplished by trading *changesets* (collections of changes made to versioned items) directly between users' local data stores, not via a centralized master data store. In fact, any semblance of a canonical "master" source of a project's versioned data is by convention only, a status attributed by the various collaborators on that project.

There are pros and cons to each version control approach. Perhaps the two biggest benefits delivered by the DVCS tools are incredible performance for day-to-day operations (because the primary data store is locally held) and vastly better support for merging between branches (because merge algorithms serve as the very core of how DVCSes work at all). The downside is that distributed version control is an inherently more complicated model, which can present a non-negligible challenge to comfortable collaboration. Also, DVCS tools do what they do well in part because of a certain degree of control withheld from the user which centalized systems freely offer—the ability to implement path-based access control, the flexibility to update or backdate individual versioned data items, etc. Fortunately, many wise organizations have discovered that this needn't be a religious debate, and that Subversion and a DVCS tool such as Git can be used together harmoniously within the organization, each serving the purposes best suited to the tool.

Alas, this book is about Subversion, so we'll not attempt a full comparison of Subversion and other tools. Readers who have the option of choosing their version control system are encouraged to research the available options and make the determination that works best for themselves and their fellow collaborators. And if, after doing so, Subversion is the chosen tool, there's *plenty* of detailed information about how to use it successfully in the chapters that follow!

Subversion's History

In early 2000, CollabNet, Inc. (http://www.collab.net) began seeking developers to write a replacement for CVS. CollabNet offered[3] a collaboration software suite called CollabNet Enterprise Edition (CEE), of which one component was version control. Although CEE used CVS as its initial version control system, CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better. Unfortunately, CVS had become the de facto standard in the open source world largely because there *wasn't* anything better, at least not under a free license. So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

[2] Or as a friend puts it, "swatting a fly with a Buick."
[3] CollabNet Enterprise Edition has since been replaced by a new product line called CollabNet TeamForge.

In February 2000, they contacted Karl Fogel, the author of *Open Source Development with CVS* (Coriolis, 1999), and asked if he'd like to work on this new project. Coincidentally, at the time Karl was already discussing a design for a new version control system with his friend Jim Blandy. In 1995, the two had started Cyclic Software, a company providing CVS support contracts, and although they later sold the business, they still used CVS every day at their jobs. Their frustration with CVS had led Jim to think carefully about better ways to manage versioned data, and he'd already come up with not only the Subversion name, but also the basic design of the Subversion data store. When CollabNet called, Karl immediately agreed to work on the project, and Jim got his employer, Red Hat Software, to essentially donate him to the project for an indefinite period of time. CollabNet hired Karl and Ben Collins-Sussman, and detailed design work began in May 2000. With the help of some well-placed prods from Brian Behlendorf and Jason Robbins of CollabNet, and from Greg Stein (at the time an independent developer active in the WebDAV/DeltaV specification process), Subversion quickly attracted a community of active developers. It turned out that many people had encountered the same frustrating experiences with CVS and welcomed the chance to finally do something about it.

The original design team settled on some simple goals. They didn't want to break new ground in version control methodology, they just wanted to fix CVS. They decided that Subversion would match CVS's features and preserve the same development model, but not duplicate CVS's most obvious flaws. And although it did not need to be a drop-in replacement for CVS, it should be similar enough that any CVS user could make the switch with little effort.

After 14 months of coding, Subversion became “self-hosting” on August 31, 2001. That is, Subversion developers stopped using CVS to manage Subversion's own source code and started using Subversion instead.

While CollabNet started the project, and still funds a large chunk of the work (it pays the salaries of a few full-time Subversion developers), Subversion is run like most open source projects, governed by a loose, transparent set of rules that encourage meritocracy. In 2009, CollabNet worked with the Subversion developers towards the goal of integrating the Subversion project into the Apache Software Foundation (ASF), one of the most well-known collectives of open source projects in the world. Subversion's technical roots, community priorities, and development practices were a perfect fit for the ASF, many of whose members were already active Subversion contributors. In early 2010, Subversion was fully adopted into the ASF's family of top-level projects, moved its project web presence to http://subversion.apache.org, and was rechristened “Apache Subversion”.

Subversion's Architecture

Figure 1, “Subversion's architecture” illustrates a “mile-high” view of Subversion's design.

Figure 1. Subversion's architecture

On one end is a Subversion repository that holds all of your versioned data. On the other end is your Subversion client program, which manages local reflections of portions of that versioned data. Between these extremes are multiple routes through a Repository Access (RA) layer, some of which go across computer networks and through network servers which then access the repository,

others of which bypass the network altogether and access the repository directly.

Subversion's Components

Subversion, once installed, has a number of different pieces. The following is a quick overview of what you get. Don't be alarmed if the brief descriptions leave you scratching your head—*plenty* more pages in this book are devoted to alleviating that confusion.

svn
: The command-line client program

svnversion
: A program for reporting the state (in terms of revisions of the items present) of a working copy

svnlook
: A tool for directly inspecting a Subversion repository

svnadmin
: A tool for creating, tweaking, or repairing a Subversion repository

mod_dav_svn
: A plug-in module for the Apache HTTP Server, used to make your repository available to others over a network

svnserve
: A custom standalone server program, runnable as a daemon process or invokable by SSH; another way to make your repository available to others over a network

svndumpfilter
: A program for filtering Subversion repository dump streams

svnsync
: A program for incrementally mirroring one repository to another over a network

svnrdump
: A program for performing repository history dumps and loads over a network

What's New in Subversion

The first edition of this book was published by O'Reilly Media in 2004, shortly after Subversion had reached 1.0. Since that time, the Subversion project has continued to release new major releases of the software. Here's a quick summary of major new changes since Subversion 1.0. Note that this is not a complete list; for full details, please visit Subversion's web site at http://subversion.apache.org.

Subversion 1.1 (September 2004)
: Release 1.1 introduced FSFS, a flat-file repository storage option for the repository. While the Berkeley DB backend is still widely used and supported, FSFS has since become the default choice for newly created repositories due to its low barrier to entry and minimal maintenance requirements. Also in this release came the ability to put symbolic links under version control, auto-escaping of URLs, and a localized user interface.

Subversion 1.2 (May 2005)
: Release 1.2 introduced the ability to create server-side locks on files, thus serializing commit access to certain resources. While Subversion is still a fundamentally concurrent version control system, certain types of binary files (e.g. art assets) cannot be merged together. The locking feature fulfills the need to version and protect such resources. With locking also came a complete WebDAV auto-versioning implementation, allowing Subversion repositories to be mounted as network folders. Finally, Subversion 1.2 began using a new, faster binary-differencing algorithm to compress and retrieve old versions of files.

Subversion 1.3 (December 2005)
: Release 1.3 brought path-based authorization controls to the **svnserve** server, matching a feature formerly found only in the Apache server. The Apache server, however, gained some new logging features of its own, and Subversion's API bindings to other languages also made great leaps forward.

Subversion 1.4 (September 2006)
: Release 1.4 introduced a whole new tool—**svnsync**—for doing one-way repository replication over a network. Major parts of the working copy metadata were revamped to no longer use XML (resulting in client-side speed gains), while the Berkeley DB repository backend gained the ability to automatically recover itself after a server crash.

Subversion 1.5 (June 2008)
: Release 1.5 took much longer to finish than prior releases, but the headliner feature was gigantic: semi-automated tracking of branching and merging. This was a huge boon for users, and pushed Subversion far beyond the abilities of CVS and into the ranks of commercial competitors such as Perforce and ClearCase. Subversion 1.5 also introduced a bevy of other user-focused features, such as interactive resolution of file conflicts, sparse checkouts, client-side management of changelists, powerful new syntax for externals definitions, and SASL authentication support for the **svnserve** server.

Subversion 1.6 (March 2009)
: Release 1.6 continued to make branching and merging more robust by introducing tree conflicts, and offered improvements to several other existing features: more interactive conflict resolution options; de-telescoping and outright exclusion support for sparse checkouts; file-based externals definitions; and operational logging support for **svnserve** similar to what **mod_dav_svn** offered. Also, the command-line client introduced a new shortcut syntax for referring to Subversion repository URLs.

Subversion 1.7 (October 2011)
: Release 1.7 was primarily a delivery vehicle for two big plumbing overhauls of existing Subversion components. The largest and most impactful of these was the so-called “WC-NG”—a complete rewrite of the **libsvn_wc** working copy management library. The second change was the introduction of a sleeker HTTP protocol for Subversion client/server interaction. Subversion 1.7 delivered a handful of additional features, many bug fixes, and some notable performance improvements, too.

Audience

This book is written for computer-literate folk who want to use Subversion to manage their data. While Subversion runs on a number of different operating systems, its primary user interface is command-line-based. That command-line tool (**svn**), and some additional auxiliary programs, are the focus of this book.

For consistency, the examples in this book assume that the reader is using a Unix-like operating system and is relatively comfortable with Unix and command-line interfaces. That said, the **svn** program also runs on non-Unix platforms such as Microsoft Windows. With a few minor exceptions, such as the use of backward slashes (`\`) instead of forward slashes (`/`) for path separators, the input to and output from this tool when run on Windows are identical to that of its Unix counterpart.

Most readers are probably programmers or system administrators who need to track changes to source code. This is the most common use for Subversion, and therefore it is the scenario underlying all of the book's examples. But Subversion can be used to manage changes to any sort of information—images, music, databases, documentation, and so on. To Subversion, all data is just data.

While this book is written with the assumption that the reader has never used a version control system, we've also tried to make it easy for users of CVS (and other systems) to make a painless leap into Subversion. Special sidebars may mention other version control systems from time to time, and Appendix B, *Subversion for CVS Users* summarizes many of the differences between CVS and Subversion.

Note also that the source code examples used throughout the book are only examples. While they will compile with the proper compiler incantations, they are intended to illustrate a particular scenario and not necessarily to serve as examples of good programming style or practices.

How to Read This Book

Technical books always face a certain dilemma: whether to cater to *top-down* or to *bottom-up* learners. A top-down learner prefers to read or skim documentation, getting a large overview of how the system works; only then does she actually start using the software. A bottom-up learner is a “learn by doing” person—someone who just wants to dive into the software and figure it out as she goes, referring to book sections when necessary. Most books tend to be written for one type of person or the other, and this book is undoubtedly biased toward top-down learners. (And if you're actually reading this section, you're probably already a top-down learner yourself!) However, if you're a bottom-up person, don't despair. While the book may be laid out as a broad survey of Subversion topics, the content of each section tends to be heavy with specific examples that you can try-by-doing. For the impatient folks who just want to get going, you can jump right to Appendix A, *Subversion Quick-Start Guide*.

Regardless of your learning style, this book aims to be useful to people of widely different backgrounds—from those with no previous experience in version control to experienced system administrators. Depending on your own background, certain chapters may be more or less important to you. The following can be considered a “recommended reading list” for various types of readers:

Experienced system administrators
: The assumption here is that you've probably used version control before and are dying to get a Subversion server up and running ASAP. Chapter 5, *Repository Administration* and Chapter 6, *Server Configuration* will show you how to create your first repository and make it available over the network. After that's done, Chapter 2, *Basic Usage* and Appendix B, *Subversion for CVS Users* are the fastest routes to learning the Subversion client.

New users
: Your administrator has probably set up Subversion already, and you need to learn how to use the client. If you've never used a version control system, then Chapter 1, *Fundamental Concepts* is a vital introduction to the ideas behind version control. Chapter 2, *Basic Usage* is a guided tour of the Subversion client.

Advanced users
: Whether you're a user or administrator, eventually your project will grow larger. You're going to want to learn how to do more advanced things with Subversion, such as how to use Subversion's property support (Chapter 3, *Advanced Topics*), how to use branches and perform merges (Chapter 4, *Branching and Merging*), how to configure runtime options (Chapter 7, *Customizing Your Subversion Experience*), and other things. These chapters aren't critical at first, but be sure to read them once you're comfortable with the basics.

Developers
: Presumably, you're already familiar with Subversion, and now want to either extend it or build new software on top of its many APIs. Chapter 8, *Embedding Subversion* is just for you.

The book ends with reference material—Chapter 9, *Subversion Complete Reference* is a reference guide for all Subversion commands, and the appendixes cover a number of useful topics. These are the chapters you're mostly likely to come back to after you've finished the book.

Organization of This Book

The chapters that follow and their contents are listed here:

Chapter 1, *Fundamental Concepts*
: Explains the basics of version control and different versioning models, along with Subversion's repository, working copies, and revisions.

Chapter 2, *Basic Usage*
: Walks you through a day in the life of a Subversion user. It demonstrates how to use a Subversion client to obtain, modify, and commit data.

Chapter 3, *Advanced Topics*
: Covers more complex features that regular users will eventually come into contact with, such as versioned metadata, file locking, and peg revisions.

Chapter 4, *Branching and Merging*
: Discusses branches, merges, and tagging, including best practices for branching and merging, common use cases, how to undo changes, and how to easily swing from one branch to the next.

Chapter 5, *Repository Administration*
: Describes the basics of the Subversion repository, how to create, configure, and maintain a repository, and the tools you can use to do all of this.

Chapter 6, *Server Configuration*
: Explains how to configure your Subversion server and offers different ways to access your repository: `HTTP`, the `svn` protocol, and local disk access. It also covers the details of authentication, authorization and anonymous access.

Chapter 7, *Customizing Your Subversion Experience*
: Explores the Subversion client configuration files, the handling of internationalized text, and how to make external tools cooperate with Subversion.

Chapter 8, *Embedding Subversion*
: Describes the internals of Subversion, the Subversion filesystem, and the working copy administrative areas from a programmer's point of view. It also demonstrates how to use the public APIs to write a program that uses Subversion.

Chapter 9, *Subversion Complete Reference*
: Explains in great detail every subcommand of **svn**, **svnadmin**, and **svnlook** with plenty of examples for the whole family!

Appendix A, *Subversion Quick-Start Guide*
: For the impatient, a whirlwind explanation of how to install Subversion and start using it immediately. You have been warned.

Appendix B, *Subversion for CVS Users*
: Covers the similarities and differences between Subversion and CVS, with numerous suggestions on how to break all the bad habits you picked up from years of using CVS. Included are descriptions of Subversion revision numbers, versioned directories, offline operations, **update** versus **status**, branches, tags, metadata, conflict resolution, and authentication.

Appendix C, *WebDAV and Autoversioning*
: Describes the details of WebDAV and DeltaV and how you can configure your Subversion repository to be mounted read/write as a DAV share.

Appendix D, *Copyright*
: A copy of the Creative Commons Attribution License, under which this book is licensed.

This Book Is Free

This book started out as bits of documentation written by Subversion project developers, which were then coalesced into a single work and rewritten. As such, it has always been under a free license (see Appendix D, *Copyright*). In fact, the book was written in the public eye, originally as part of the Subversion project itself. This means two things:

- You will always find the latest version of this book in the book's own Subversion repository.
- You can make changes to this book and redistribute it however you wish—it's under a free license. Your only obligation is to maintain proper attribution to the original authors. Of course, we'd much rather you send feedback and patches to the Subversion developer community, instead of distributing your private version of this book.

The online home of this book's development and most of the volunteer-driven translation efforts regarding it is http://svnbook.red-bean.com. There you can find links to the latest releases and tagged versions of the book in various formats, as well as instructions for accessing the book's Subversion repository (where its DocBook XML source code lives). Feedback is welcomed—encouraged, even. Please submit all comments, complaints, and patches against the book sources to `<svnbook-dev@red-bean.com>`.

Acknowledgments

This book would not be possible (nor very useful) if Subversion did not exist. For that, the authors would like to thank Brian Behlendorf and CollabNet for the vision to fund such a risky and ambitious new open source project; Jim Blandy for the original Subversion name and design—we love you, Jim; and Karl Fogel for being such a good friend and a great community leader, in that order.[4]

Thanks to O'Reilly and the team of professional editors who have helped us polish this text at various stages of its evolution: Chuck Toporek, Linda Mui, Tatiana Apandi, Mary Brady, and Mary Treseler. Your patience and support has been tremendous.

Finally, we thank the countless people who contributed to this book with informal reviews, suggestions, and patches. An exhaustive listing of those folks' names would be impractical to print and maintain here, but may their names live on forever in this book's version control history!

[4]Oh, and thanks, Karl, for being too overworked to write this book yourself.

Chapter 1. Fundamental Concepts

This chapter is a short, casual introduction to Subversion and its approach to version control. We begin with a discussion of general version control concepts, work our way into the specific ideas behind Subversion, and show some simple examples of Subversion in use.

Even though the examples in this chapter show people sharing collections of program source code, keep in mind that Subversion can manage any sort of file collection—it's not limited to helping computer programmers.

Version Control Basics

A version control system (or revision control system) is a system that tracks incremental versions (or revisions) of files and, in some cases, directories over time. Of course, merely tracking the various versions of a user's (or group of users') files and directories isn't very interesting in itself. What makes a version control system useful is the fact that it allows you to explore the changes which resulted in each of those versions and facilitates the arbitrary recall of the same.

In this section, we'll introduce some fairly high-level version control system components and concepts. We'll limit our discussion to modern version control systems—in today's interconnected world, there is very little point in acknowledging version control systems which cannot operate across wide-area networks.

The Repository

At the core of the version control system is a repository, which is the central store of that system's data. The repository usually stores information in the form of a *filesystem tree*—a hierarchy of files and directories. Any number of *clients* connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others. Figure 1.1, “A typical client/server system” illustrates this.

Figure 1.1. A typical client/server system

Why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository *is* a kind of file server, but it's not your usual breed. What makes the repository special is that as the files in the repository are changed, the repository remembers each version of those files.

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But what makes a version control client interesting is that it also has the ability to request previous states of the filesystem from the repository. A version control client can ask historical questions such as “What did this directory contain last Wednesday?” and “Who was the last person to change this file, and what changes did he make?” These are the sorts of questions that are at the heart of any version control sys-

tem.

The Working Copy

A version control system's value comes from the fact that it tracks versions of files and directories, but the rest of the software universe doesn't operate on “versions of files and directories”. Most software programs understand how to operate only on a *single* version of a specific type of file. So how does a version control user interact with an abstract—and, often, remote—repository full of multiple versions of various files in a concrete fashion? How does his or her word processing software, presentation software, source code editor, web design software, or some other program—all of which trade in the currency of simple data files—get access to such files? The answer is found in the version control construct known as a *working copy*.

A working copy is, quite literally, a local copy of a particular version of a user's VCS-managed data upon which that user is free to work. Working copies[1] appear to other software just as any other local directory full of files, so those programs don't have to be “version-control-aware” in order to read from and write to that data. The task of managing the working copy and communicating changes made to its contents to and from the repository falls squarely to the version control system's client software.

Versioning Models

If the primary mission of a version control system is to track the various versions of digital information over time, a very close secondary mission in any modern version control system is to enable collaborative editing and sharing of that data. But different systems use different strategies to achieve this. It's important to understand these different strategies, for a couple of reasons. First, it will help you compare and contrast existing version control systems, in case you encounter other systems similar to Subversion. Beyond that, it will also help you make more effective use of Subversion, since Subversion itself supports a couple of different ways of working.

The problem of file sharing

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Consider the scenario shown in Figure 1.2, “The problem to avoid”. Suppose we have two coworkers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made *won't* be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost—or at least missing from the latest version of the file—and probably by accident. This is definitely a situation we want to avoid!

Figure 1.2. The problem to avoid

[1]The term “working copy” can be generally applied to any one file version's local instance. When most folks use the term, though, they are referring to a whole directory tree containing files and subdirectories managed by the version control system.

The lock-modify-unlock solution

Many version control systems use a *lock-modify-unlock* model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must “lock” a file before he can begin making changes to it. If Harry has locked a file, Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is read the file and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, Sally can take her turn by locking and editing the file. Figure 1.3, “The lock-modify-unlock solution” demonstrates this simple solution.

Figure 1.3. The lock-modify-unlock solution

The problem with the lock-modify-unlock model is that it's a bit restrictive and often becomes a roadblock for users:

- *Locking may cause administrative problems.* Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied. And then Harry goes on vacation. Now Sally has to get an administrator to release Harry's lock. The situation ends up causing a lot of unnecessary delay and wasted time.
- *Locking may cause unnecessary serialization.* What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together. There's no need for them to take turns in this situation.
- *Locking may create a false sense of security.* Suppose Harry locks and edits file A, while Sally simultaneously locks and edits file B. But what if A and B depend on one another, and the changes made to each are semantically incompatible? Suddenly A and B don't work together anymore. The locking system was powerless to prevent the problem—yet it somehow provided a false sense of security. It's easy for Harry and Sally to imagine that by locking files, each is beginning a safe, insulated task, and thus they need not bother discussing their incompatible changes early on. Locking often becomes a substitute for real communication.

The copy-modify-merge solution

Subversion, CVS, and many other version control systems use a *copy-modify-merge* model as an alternative to locking. In this model, each user's client contacts the project repository and creates a personal working copy. Users then work simultaneously and

independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately, a human being is responsible for making it happen correctly.

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently and make changes to the same file A within their copies. Sally saves her changes to the repository first. When Harry attempts to save his changes later, the repository informs him that his file A is *out of date*. In other words, file A in the repository has somehow changed since he last copied it. So Harry asks his client to *merge* any new changes from the repository into his working copy of file A. Chances are that Sally's changes don't overlap with his own; once he has both sets of changes integrated, he saves his working copy back to the repository. Figure 1.4, “The copy-modify-merge solution” and Figure 1.5, “The copy-modify-merge solution (continued)” show this process.

Figure 1.4. The copy-modify-merge solution

Figure 1.5. The copy-modify-merge solution (continued)

But what if Sally's changes *do* overlap with Harry's changes? What then? This situation is called a *conflict*, and it's usually not much of a problem. When Harry asks his client to merge the latest repository changes into his working copy, his copy of file A is somehow flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making the necessary intelligent choices. Once Harry has manually resolved the overlapping changes—perhaps after a discussion with Sally—he can safely save the merged file back to the repository.

The copy-modify-merge model may sound a bit chaotic, but in practice, it runs extremely smoothly. Users can work in parallel, never waiting for one another. When they work on the same files, it turns out that most of their concurrent changes don't overlap at all; conflicts are infrequent. And the amount of time it takes to resolve conflicts is usually far less than the time lost by a locking system.

In the end, it all comes down to one critical factor: user communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts. So there's no point in being lulled into a false sense of security that a locking system will somehow prevent conflicts; in practice, locking seems to inhibit productivity more than anything else.

When Locking Is Necessary

While the lock-modify-unlock model is considered generally harmful to collaboration, sometimes locking is appropriate.

The copy-modify-merge model is based on the assumption that files are contextually mergeable—that is, that the majority of the files in the repository are line-based text files (such as program source code). But for files with binary formats, such as

artwork or sound, it's often impossible to merge conflicting changes. In these situations, it really is necessary for users to take strict turns when changing the file. Without serialized access, somebody ends up wasting time on changes that are ultimately discarded.

While Subversion is primarily a copy-modify-merge system, it still recognizes the need to lock an occasional file, and thus provides mechanisms for this. We discuss this feature in the section called “Locking”.

Version Control the Subversion Way

We've mentioned already that Subversion is a modern, network-aware version control system. As we described in the section called “Version Control Basics” (our high-level version control overview), a repository serves as the core storage mechanism for Subversion's versioned data, and it's via working copies that users and their software programs interact with that data. In this section, we'll begin to introduce the specific ways in which Subversion implements version control.

Subversion Repositories

Subversion implements the concept of a version control repository much as any other modern version control system would. Unlike a working copy, a Subversion repository is an abstract entity, able to be operated upon almost exclusively by Subversion's own libraries and tools. As most of a user's Subversion interactions involve the use of the Subversion client and occur in the context of a working copy, we spend the majority of this book discussing the Subversion working copy and how to manipulate it. For the finer details of the repository, though, check out Chapter 5, *Repository Administration*.

Revisions

A Subversion client commits (that is, communicates the changes made to) any number of files and directories as a single atomic transaction. By atomic transaction, we mean simply this: either all of the changes are accepted into the repository, or none of them is. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

Each time the repository accepts a commit, this creates a new state of the filesystem tree, called a *revision*. Each revision is assigned a unique natural number, one greater than the number assigned to the previous revision. The initial revision of a freshly created repository is numbered 0 and consists of nothing but an empty root directory.

Figure 1.6, “Tree changes over time” illustrates a nice way to visualize the repository. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a “snapshot” of the way the repository looked after a commit.

Figure 1.6. Tree changes over time

Global Revision Numbers

Unlike most version control systems, Subversion's revision numbers apply to *the entire repository tree*, not individual files. Each revision number selects an entire tree, a particular state of the repository after some committed change. Another way to think about it is that revision N represents the state of the repository filesystem after the Nth commit. When Subversion users talk about "revision 5 of `foo.c`," they really mean "`foo.c` as it appears in revision 5." Notice that in general, revisions N and M of a file do *not* necessarily differ! Many other version control systems use per-file revision numbers, so this concept may seem unusual at first. (Former CVS users might want to see Appendix B, *Subversion for CVS Users* for more details.)

Addressing the Repository

Subversion client programs use URLs to identify versioned files and directories in Subversion repositories. For the most part, these URLs use the standard syntax, allowing for server names and port numbers to be specified as part of the URL.

- http://svn.example.com/svn/project
- http://svn.example.com:9834/repos

Subversion repository URLs aren't limited to only the `http://` variety. Because Subversion offers several different ways for its clients to communicate with its servers, the URLs used to address the repository differ subtly depending on which repository access mechanism is employed. Table 1.1, "Repository access URLs" describes how different URL schemes map to the available repository access methods. For more details about Subversion's server options, see Chapter 6, *Server Configuration*.

Table 1.1. Repository access URLs

Schema	Access method
`file:///`	Direct repository access (on local disk)

Schema	Access method
`http://`	Access via WebDAV protocol to Subversion-aware Apache server
`https://`	Same as `http://`, but with SSL encryption
`svn://`	Access via custom protocol to an `svnserve` server
`svn+ssh://`	Same as `svn://`, but through an SSH tunnel

Subversion's handling of URLs has some notable nuances. For example, URLs containing the `file://` access method (used for local repositories) must, in accordance with convention, have either a server name of `localhost` or no server name at all:

- file:///var/svn/repos
- file://localhost/var/svn/repos

Also, users of the `file://` scheme on Windows platforms will need to use an unofficially “standard” syntax for accessing repositories that are on the same machine, but on a different drive than the client's current working drive. Either of the two following URL path syntaxes will work, where `X` is the drive on which the repository resides:

- file:///X:/var/svn/repos
- file:///X|/var/svn/repos

Note that a URL uses forward slashes even though the native (non-URL) form of a path on Windows uses backslashes. Also note that when using the `file:///X|/` form at the command line, you need to quote the URL (wrap it in quotation marks) so that the vertical bar character is not interpreted as a pipe.

You cannot use Subversion's `file://` URLs in a regular web browser the way typical `file://` URLs can. When you attempt to view a `file://` URL in a regular web browser, it reads and displays the contents of the file at that location by examining the filesystem directly. However, Subversion's resources exist in a virtual filesystem (see the section called “Repository Layer”), and your browser will not understand how to interact with that filesystem.

The Subversion client will automatically encode URLs as necessary, just like a web browser does. For example, the URL `http://host/path with space/project/españa` — which contains both spaces and upper-ASCII characters — will be automatically interpreted by Subversion as if you'd provided `http://host/path%20with%20space/project/espa%C3%B1a`. If the URL contains spaces, be sure to place it within quotation marks at the command line so that your shell treats the whole thing as a single argument to the program.

There is one notable exception to Subversion's handling of URLs which also applies to its handling of local paths in many contexts, too. If the final path component of your URL or local path contains an at sign (`@`), you need to use a special syntax—described in the section called “Peg and Operative Revisions”—in order to make Subversion properly address that resource.

In Subversion 1.6, a new caret (`^`) notation was introduced as a shorthand for “the URL of the repository's root directory”. For example, you can use the `^/tags/bigsandwich/` to refer to the URL of the `/tags/bigsandwich` directory in the root of the repository. Note that this URL syntax works only when your current working directory is a working copy—the command-line client knows the repository's root URL by looking at the working copy's metadata. Also note that when you wish to refer precisely to the root directory of the repository, you must do so using `^/` (with the trailing slash character), not merely `^`.

Subversion Working Copies

A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. You can edit these

files however you wish, and if they're source code files, you can compile your program from them in the usual way. Your working copy is your own private work area: Subversion will never incorporate other people's changes, nor make your own changes available to others, until you explicitly tell it to do so. You can even have multiple working copies of the same project.

After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to “publish” your changes to the other people working with you on your project (by writing to the repository). If other people publish their own changes, Subversion provides you with commands to merge those changes into your working copy (by reading from the repository).

A working copy also contains some extra files, created and maintained by Subversion, to help it carry out these commands. In particular, each working copy contains a subdirectory named `.svn`, also known as the working copy's *administrative directory*. The files in the administrative directory help Subversion recognize which of your versioned files contain unpublished changes, and which files are out of date with respect to others' work.

Prior to version 1.7, Subversion maintained `.svn` administrative subdirectories in *every* versioned directory of your working copy. Subversion 1.7 offers a completely new approach to how working copy metadata is stored and maintained, and chief among the visible changes to this approach is that each working copy now has only one `.svn` subdirectory which is an immediate child of the root of that working copy.

While `.svn` is the de facto name of the Subversion administrative directory, Windows users may run into problems with the ASP.NET Web application framework disallowing access to directories whose names begin with a dot (`.`). As a special consideration to users in such situations, Subversion will instead use `_svn` as the administrative directory name if it finds a variable named `SVN_ASP_DOT_NET_HACK` in its operating environment. Throughout this book, any reference you find to `.svn` applies also to `_svn` when this “ASP.NET hack” is in use.

How the working copy works

For each file in a working directory, Subversion records (among other things) two essential pieces of information:

- What revision your working file is based on (this is called the file's *working revision*)
- A timestamp recording when the local copy was last updated by the repository

Given this information, by talking to the repository, Subversion can tell which of the following four states a working file is in:

Unchanged, and current
: The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its working revision. An **svn commit** of the file will do nothing, and an **svn update** of the file will do nothing.

Locally changed, and current
: The file has been changed in the working directory, and no changes to that file have been committed to the repository since you last updated. There are local changes that have not been committed to the repository; thus an **svn commit** of the file will succeed in publishing your changes, and an **svn update** of the file will do nothing.

Unchanged, and out of date
: The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated in order to make it current with the latest public revision. An **svn commit** of the file will do nothing, and an **svn update** of the file will fold the latest changes into your working copy.

Locally changed, and out of date
: The file has been changed both in the working directory and in the repository. An **svn commit** of the file will fail with an “out-of-date” error. The file should be updated first; an **svn update** command will attempt to merge the public changes with

the local changes. If Subversion can't complete the merge in a plausible way automatically, it leaves it to the user to resolve the conflict.

Fundamental working copy interactions

A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement, a user's working copy will usually correspond to a particular subtree of the repository.

For example, suppose you have a repository that contains two software projects, `paint` and `calc`. Each project lives in its own top-level subdirectory, as shown in Figure 1.7, “The repository's filesystem”.

Figure 1.7. The repository's filesystem

To get a working copy, you must *check out* some subtree of the repository. (The term *check out* may sound like it has something to do with locking or reserving resources, but it doesn't; it simply creates a working copy of the project for you.) For example, if you check out `/calc`, you will get a working copy like this:

```
$ svn checkout http://svn.example.com/repos/calc
A    calc/Makefile
A    calc/integer.c
```

```
A    calc/button.c
Checked out revision 56.
$ ls -A calc
Makefile  button.c integer.c .svn/
$
```

The list of letter A s in the left margin indicates that Subversion is adding a number of items to your working copy. You now have a personal copy of the repository's `/calc` directory, with one additional entry—`.svn`—which holds the extra information needed by Subversion, as mentioned earlier.

Suppose you make changes to `button.c`. Since the `.svn` directory remembers the file's original modification date and contents, Subversion can tell that you've changed the file. However, Subversion does not make your changes public until you explicitly tell it to. The act of publishing your changes is more commonly known as *committing* (or *checking in*) changes to the repository.

To publish your changes to others, you can use Subversion's **svn commit** command:

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending        button.c
Transmitting file data .
Committed revision 57.
$
```

Now your changes to `button.c` have been committed to the repository, with a note describing your change (namely, that you fixed a typo). If another user checks out a working copy of `/calc`, she will see your changes in the latest version of the file.

Suppose you have a collaborator, Sally, who checked out a working copy of `/calc` at the same time you did. When you commit your change to `button.c`, Sally's working copy is left unchanged; Subversion modifies working copies only at the user's request.

To bring her project up to date, Sally can ask Subversion to *update* her working copy, by using the **svn update** command. This will incorporate your changes into her working copy, as well as any others that have been committed since she checked it out.

```
$ pwd
/home/sally/calc
$ ls -A
Makefile button.c integer.c .svn/
$ svn update
Updating '.':
U    button.c
Updated to revision 57.
$
```

The output from the **svn update** command indicates that Subversion updated the contents of `button.c`. Note that Sally didn't need to specify which files to update; Subversion uses the information in the `.svn` directory as well as further information in the repository, to decide which files need to be brought up to date.

Mixed-revision working copies

As a general principle, Subversion tries to be as flexible as possible. One special kind of flexibility is the ability to have a working copy containing files and directories with a mix of different working revision numbers. Subversion working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose

you check out a working copy from a repository whose most recent revision is 4:

```
calc/
  Makefile:4
  integer.c:4
  button.c:4
```

At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to `button.c`, and commit that change. Assuming no other commits have taken place, your commit will create revision 5 of the repository, and your working copy will now look like this:

```
calc/
  Makefile:4
  integer.c:4
  button.c:5
```

Suppose that, at this point, Sally commits a change to `integer.c`, creating revision 6. If you use **svn update** to bring your working copy up to date, it will look like this:

```
calc/
  Makefile:6
  integer.c:6
  button.c:6
```

Sally's change to `integer.c` will appear in your working copy, and your change will still be present in `button.c`. In this example, the text of `Makefile` is identical in revisions 4, 5, and 6, but Subversion will mark your working copy of `Makefile` with revision 6 to indicate that it is still current. So, after you do a clean update at the top of your working copy, it will generally correspond to exactly one revision in the repository.

Updates and commits are separate

One of the fundamental rules of Subversion is that a “push” action does not cause a “pull” nor vice versa. Just because you're ready to submit new changes to the repository doesn't mean you're ready to receive changes from other people. And if you have new changes still in progress, **svn update** should gracefully merge repository changes into your own, rather than forcing you to publish them.

The main side effect of this rule is that it means a working copy has to do extra bookkeeping to track mixed revisions as well as be tolerant of the mixture. It's made more complicated by the fact that directories themselves are versioned.

For example, suppose you have a working copy entirely at revision 10. You edit the file `foo.html` and then perform an **svn commit**, which creates revision 15 in the repository. After the commit succeeds, many new users would expect the working copy to be entirely at revision 15, but that's not the case! Any number of changes might have happened in the repository between revisions 10 and 15. The client knows nothing of those changes in the repository, since you haven't yet run **svn update**, and **svn commit** doesn't pull down new changes. If, on the other hand, **svn commit** were to automatically download the newest changes, it would be possible to set the entire working copy to revision 15—but then we'd be breaking the fundamental rule of “push” and “pull” remaining separate actions. Therefore, the only safe thing the Subversion client can do is mark the one file—`foo.html`—as being at revision 15. The rest of the working copy remains at revision 10. Only by running **svn update** can the latest changes be downloaded

and the whole working copy be marked as revision 15.

Mixed revisions are normal

The fact is, *every time* you run **svn commit** your working copy ends up with some mixture of revisions. The things you just committed are marked as having larger working revisions than everything else. After several commits (with no updates in between), your working copy will contain a whole mixture of revisions. Even if you're the only person using the repository, you will still see this phenomenon. To examine your mixture of working revisions, use the **svn status** command with the `--verbose` (`-v`) option (see the section called "See an overview of your changes" for more information).

Often, new users are completely unaware that their working copy contains mixed revisions. This can be confusing, because many client commands are sensitive to the working revision of the item they're examining. For example, the **svn log** command is used to display the history of changes to a file or directory (see the section called "Generating a List of Historical Changes"). When the user invokes this command on a working copy object, he expects to see the entire history of the object. But if the object's working revision is quite old (often because **svn update** hasn't been run in a long time), the history of the *older* version of the object is shown.

Mixed revisions are useful

If your project is sufficiently complex, you'll discover that it's sometimes nice to forcibly *backdate* (or update to a revision older than the one you already have) portions of your working copy to an earlier revision; you'll learn how to do that in Chapter 2, *Basic Usage*. Perhaps you'd like to test an earlier version of a submodule contained in a subdirectory, or perhaps you'd like to figure out when a bug first came into existence in a specific file. This is the "time machine" aspect of a version control system—the feature that allows you to move any portion of your working copy forward and backward in history.

Mixed revisions have limitations

However you make use of mixed revisions in your working copy, there are limitations to this flexibility.

First, you cannot commit the deletion of a file or directory that isn't fully up to date. If a newer version of the item exists in the repository, your attempt to delete will be rejected to prevent you from accidentally destroying changes you've not yet seen.

Second, you cannot commit a metadata change to a directory unless it's fully up to date. You'll learn about attaching "properties" to items in Chapter 3, *Advanced Topics*. A directory's working revision defines a specific set of entries and properties, and thus committing a property change to an out-of-date directory may destroy properties you've not yet seen.

Finally, beginning in Subversion 1.7, you cannot by default use a mixed-revision working copy as the target of a merge operation. (This new requirement was introduced to prevent common problems which stem from doing so.)

Summary

We covered a number of fundamental Subversion concepts in this chapter:

- We introduced the notions of the central repository, the client working copy, and the array of repository revision trees.
- We saw some simple examples of how two collaborators can use Subversion to publish and receive changes from one another, using the "copy-modify-merge" model.
- We talked a bit about the way Subversion tracks and manages information in a working copy.

At this point, you should have a good idea of how Subversion works in the most general sense. Armed with this knowledge, you should now be ready to move into the next chapter, which is a detailed tour of Subversion's commands and features.

Chapter 2. Basic Usage

Theory is useful, but its application is just plain fun. Let's move now into the details of using Subversion. By the time you reach the end of this chapter, you will be able to perform all the tasks you need to use Subversion in a normal day's work. You'll start with getting your files into Subversion, followed by an initial checkout of your code. We'll then walk you through making changes and examining those changes. You'll also see how to bring changes made by others into your working copy, examine them, and work through any conflicts that might arise.

This chapter will not provide exhaustive coverage of all of Subversion's commands—rather, it's a conversational introduction to the most common Subversion tasks that you'll encounter. This chapter assumes that you've read and understood Chapter 1, *Fundamental Concepts* and are familiar with the general model of Subversion. For a complete reference of all commands, see Chapter 9, *Subversion Complete Reference*.

Also, this chapter assumes that the reader is seeking information about how to interact in a basic fashion with an existing Subversion repository. No repository means no working copy; no working copy means not much of interest in this chapter. There are many Internet sites which offer free or inexpensive Subversion repository hosting services. Or, if you'd prefer to set up and administer your own repositories, check out Chapter 5, *Repository Administration*. But don't expect the examples in this chapter to work without the user having access to a Subversion repository.

Finally, any Subversion operation that contacts the repository over a network may potentially require that the user authenticate. For the sake of simplicity, our examples throughout this chapter avoid demonstrating and discussing authentication. Be aware that if you hope to apply the knowledge herein to an existing, real-world Subversion instance, you'll probably be forced to provide at least a username and password to the server. See the section called “Client Credentials” for a detailed description of Subversion's handling of authentication and client credentials.

Help!

It goes without saying that this book exists to be a source of information and assistance for Subversion users new and old. Conveniently, though, the Subversion command-line is self-documenting, alleviating the need to grab a book off the shelf (wooden, virtual, or otherwise). The **svn help** command is your gateway to that built-in documentation:

```
$ svn help
Subversion command-line client, version 1.7.0.
Type 'svn help <subcommand>' for help on a specific subcommand.
Type 'svn --version' to see the program version and RA modules
  or 'svn --version --quiet' to see just the version number.

Most subcommands take file and/or directory arguments, recursing
on the directories.  If no arguments are supplied to such a
command, it recurses on the current directory (inclusive) by default.

Available subcommands:
   add
   blame (praise, annotate, ann)
   cat
…
```

As described in the previous output, you can ask for help on a particular subcommand by running **`svn help SUBCOMMAND`**. Subversion will respond with the full usage message for that subcommand, including its syntax, options, and behavior:

```
$ svn help help
help (?, h): Describe the usage of this program or its subcommands.
```

```
usage: help [SUBCOMMAND...]

Global options:
  --username ARG           : specify a username ARG
  --password ARG           : specify a password ARG
…
```

Options and Switches and Flags, Oh My!

The Subversion command-line client has numerous command modifiers. Some folks refer to such things as “switches” or “flags”—in this book, we'll call them “options”. You'll find the options supported by a given **svn** subcommand, plus a set of options which are globally supported by all subcommands, listed near the bottom of the built-in usage message for that subcommand.

Subversion's options have two distinct forms: short options are a single hyphen followed by a single letter, and long options consist of two hyphens followed by several letters and hyphens (e.g., `-s` and `--this-is-a-long-option`, respectively). Every option has at least one long format. Some, such as the `--changelist` option, feature an abbreviated long-format alias (`--cl`, in this case). Only certain options—generally the most-used ones—have an additional short format. To maintain clarity in this book, we usually use the long form in code examples, but when describing options, if there's a short form, we'll provide the long form (to improve clarity) and the short form (to make it easier to remember). Use the form you're more comfortable with when executing your own Subversion commands.

Many Unix-based distributions of Subversion include manual pages of the sort that can be invoked using the **man** program, but those tend to carry only pointers to other sources of real help, such as the project's website and to the website which hosts this book. Also, several companies offer Subversion help and support, too, usually via a mixture of web-based discussion forums and fee-based consulting. And of course, the Internet holds a decade's worth of Subversion-related discussions just begging to be located by your favorite search engine. Subversion help is never too far away.

Getting Data into Your Repository

You can get new files into your Subversion repository in two ways: **svn import** and **svn add**. We'll discuss **svn import** now and will discuss **svn add** later in this chapter when we review a typical day with Subversion.

Importing Files and Directories

The **svn import** command is a quick way to copy an unversioned tree of files into a repository, creating intermediate directories as necessary. **svn import** doesn't require a working copy, and your files are immediately committed to the repository. You typically use this when you have an existing tree of files that you want to begin tracking in your Subversion repository. For example:

```
$ svn import /path/to/mytree \
             http://svn.example.com/svn/repo/some/project \
             -m "Initial import"
Adding         mytree/foo.c
Adding         mytree/bar.c
Adding         mytree/subdir
Adding         mytree/subdir/quux.h

Committed revision 1.
$
```

The previous example copied the contents of the local directory `mytree` into the directory `some/project` in the repository. Note that you didn't have to create that new directory first—**svn import** does that for you. Immediately after the commit, you can see your data in the repository:

```
$ svn list http://svn.example.com/svn/repo/some/project
bar.c
foo.c
subdir/
$
```

Note that after the import is finished, the original local directory is *not* converted into a working copy. To begin working on that data in a versioned fashion, you still need to create a fresh working copy of that tree.

Recommended Repository Layout

Subversion provides the ultimate flexibility in terms of how you arrange your data. Because it simply versions directories and files, and because it ascribes no particular meaning to any of those objects, you may arrange the data in your repository in any way that you choose. Unfortunately, this flexibility also means that it's easy to find yourself “lost without a roadmap” as you attempt to navigate different Subversion repositories which may carry completely different and unpredictable arrangements of the data within them.

To counteract this confusion, we recommend that you follow a repository layout convention (established long ago, in the nascency of the Subversion project itself) in which a handful of strategically named Subversion repository directories convey valuable meaning about the data they hold. Most projects have a recognizable “main line”, or *trunk*, of development; some *branches*, which are divergent copies of development lines; and some *tags*, which are named, stable snapshots of a particular line of development. So we first recommend that each project have a recognizable *project root* in the repository, a directory under which all of the versioned information for that project—and only that project—lives. Secondly, we suggest that each project root contain a `trunk` subdirectory for the main development line, a `branches` subdirectory in which specific branches (or collections of branches) will be created, and a `tags` subdirectory in which specific tags (or collections of tags) will be created. Of course, if a repository houses only a single project, the root of the repository can serve as the project root, too.

Here are some examples:

```
$ svn list file:///var/svn/single-project-repo
trunk/
branches/
tags/
$ svn list file:///var/svn/multi-project-repo
project-A/
project-B/
$ svn list file:///var/svn/multi-project-repo/project-A
trunk/
branches/
tags/
$
```

We talk much more about tags and branches in Chapter 4, *Branching and Merging*. For details and some advice on how to set up repositories when you have multiple projects, see the section called “Repository Layout”. Finally, we discuss project roots more in the section called “Planning Your Repository Organization”.

What's In a Name?

Subversion tries hard not to limit the type of data you can place under version control. The contents of files and property values are stored and transmitted as binary data, and the section called “File Content Type” tells you how to give Subversion a hint that “textual” operations don't make sense for a particular file. There are a few places, however, where Subversion places restrictions on information it stores.

Subversion internally handles certain bits of data—for example, property names, pathnames, and log messages—as UTF-8-encoded Unicode. This is not to say that all your interactions with Subversion must involve UTF-8, though. As a general rule, Subversion clients will gracefully and transparently handle conversions between UTF-8 and the encoding system in use on your computer, if such a conversion can meaningfully be done (which is the case for most common encodings in use today).

In WebDAV exchanges and older versions of some of Subversion's administrative files, paths are used as XML attribute values, and property names in XML tag names. This means that pathnames can contain only legal XML (1.0) characters, and properties are further limited to ASCII characters. Subversion also prohibits `TAB`, `CR`, and `LF` characters in path names to prevent paths from being broken up in diffs or in the output of commands such as **svn log** or **svn status**.

While it may seem like a lot to remember, in practice these limitations are rarely a problem. As long as your locale settings are compatible with UTF-8 and you don't use control characters in path names, you should have no trouble communicating with Subversion. The command-line client adds an extra bit of help—to create “legally correct” versions for internal use it will automatically escape illegal path characters as needed in URLs that you type.

Creating a Working Copy

Most of the time, you will start using a Subversion repository by performing a *checkout* of your project. Checking out a directory from a repository creates a working copy of that directory on your local machine. Unless otherwise specified, this copy contains the youngest (that is, most recently created or modified) versions of the directory and its children found in the Subversion repository:

```
$ svn checkout http://svn.example.com/svn/repo/trunk
A    trunk/README
A    trunk/INSTALL
A    trunk/src/main.c
A    trunk/src/header.h
…
Checked out revision 8810.
$
```

Although the preceding example checks out the trunk directory, you can just as easily check out a deeper subdirectory of a repository by specifying that subdirectory's URL as the checkout URL:

```
$ svn checkout http://svn.example.com/svn/repo/trunk/src
A    src/main.c
A    src/header.h
A    src/lib/helpers.c
…
Checked out revision 8810.
$
```

Since Subversion uses a copy-modify-merge model instead of lock-modify-unlock (see the section called “Versioning Models”), you can immediately make changes to the files and directories in your working copy. Your working copy is just like any other collection of files and directories on your system. You can edit the files inside it, rename it, even delete the entire working copy and forget about it.

While your working copy is “just like any other collection of files and directories on your system,” you can edit files at will, but you must tell Subversion about *everything else* that you do. For example, if you want to copy or move an item in a working copy, you should use **svn copy** or **svn move** instead of the copy and move commands provided by your operating system. We'll talk more about them later in this chapter.

Unless you're ready to commit the addition of a new file or directory or changes to existing ones, there's no need to further notify the Subversion server that you've done anything.

What Is This .svn Directory?

The topmost directory of a working copy—and prior to version 1.7, every versioned subdirectory thereof—contains a special administrative subdirectory named `.svn`. Usually, your operating system's directory listing commands won't show this subdirectory, but it is nevertheless an important directory. Whatever you do, don't delete or change anything in the administrative area! Subversion uses that directory and its contents to manage your working copy.

Notice that in the previous pair of examples, Subversion chose to create a working copy in a directory named for the final component of the checkout URL. This occurs only as a convenience to the user when the checkout URL is the only bit of information provided to the **svn checkout** command. Subversion's command-line client gives you additional flexibility, though, allowing you to optionally specify the local directory name that Subversion should use for the working copy it creates. For example:

```
$ svn checkout http://svn.example.com/svn/repo/trunk my-working-copy
A    my-working-copy/README
A    my-working-copy/INSTALL
A    my-working-copy/src/main.c
A    my-working-copy/src/header.h
…
Checked out revision 8810.
$
```

If the local directory you specify doesn't yet exist, that's okay—**svn checkout** will create it for you.

Basic Work Cycle

Subversion has numerous features, options, bells, and whistles, but on a day-to-day basis, odds are that you will use only a few of them. In this section, we'll run through the most common things that you might find yourself doing with Subversion in the course of a day's work.

The typical work cycle looks like this:

1. *Update your working copy.* This involves the use of the **svn update** command.
2. *Make your changes.* The most common changes that you'll make are edits to the contents of your existing files. But sometimes you need to add, remove, copy and move files and directories—the **svn add**, **svn delete**, **svn copy**, and **svn move** commands handle those sorts of structural changes within the working copy.
3. *Review your changes.* The **svn status** and **svn diff** commands are critical to reviewing the changes you've made in your working copy.
4. *Fix your mistakes.* Nobody's perfect, so as you review your changes, you may spot something that's not quite right. Sometimes the easiest way to fix a mistake is start all over again from scratch. The **svn revert** command restores a file or directory to its

unmodified state.

5. *Resolve any conflicts (merge others' changes).* In the time it takes you to make and review your changes, others might have made and published changes, too. You'll want to integrate their changes into your working copy to avoid the potential out-of-dateness scenarios when you attempt to publish your own. Again, the **svn update** command is the way to do this. If this results in local conflicts, you'll need to resolve those using the **svn resolve** command.

6. *Publish (commit) your changes.* The **svn commit** command transmits your changes to the repository where, if they are accepted, they create the newest versions of all the things you modified. Now others can see your work, too!

Update Your Working Copy

When working on a project that is being modified via multiple working copies, you'll want to update your working copy to receive any changes committed from other working copies since your last update. These might be changes that other members of your project team have made, or they might simply be changes you've made yourself from a different computer. To protect your data, Subversion won't allow you commit new changes to out-of-date files and directories, so it's best to have the latest versions of all your project's files and directories before making new changes of your own.

Use **svn update** to bring your working copy into sync with the latest revision in the repository:

```
$ svn update
Updating '.':
U    foo.c
U    bar.c
Updated to revision 2.
$
```

In this case, it appears that someone checked in modifications to both `foo.c` and `bar.c` since the last time you updated, and Subversion has updated your working copy to include those changes.

When the server sends changes to your working copy via **svn update**, a letter code is displayed next to each item to let you know what actions Subversion performed to bring your working copy up to date. To find out what these letters mean, run **`svn help update`** or see svn update (up) in Chapter 9, *Subversion Complete Reference*.

Make Your Changes

Now you can get to work and make changes in your working copy. You can make two kinds of changes to your working copy: *file changes* and *tree changes*. You don't need to tell Subversion that you intend to change a file; just make your changes using your text editor, word processor, graphics program, or whatever tool you would normally use. Subversion automatically detects which files have been changed, and in addition, it handles binary files just as easily as it handles text files—and just as efficiently, too. Tree changes are different, and involve changes to a directory's structure. Such changes include adding and removing files, renaming files or directories, and copying files or directories to new locations. For tree changes, you use Subversion operations to “schedule” files and directories for removal, addition, copying, or moving. These changes may take place immediately in your working copy, but no additions or removals will happen in the repository until you commit them.

Versioning Symbolic Links

On non-Windows platforms, Subversion is able to version files of the special type *symbolic link* (or “symlink”). A symlink is a file that acts as a sort of transparent reference to some other object in the filesystem, allowing programs to read and write to those objects indirectly by performing operations on the symlink itself.

When a symlink is committed into a Subversion repository, Subversion remembers that the file was in fact a symlink, as well

as the object to which the symlink "points." When that symlink is checked out to another working copy on a non-Windows system, Subversion reconstructs a real filesystem-level symbolic link from the versioned symlink. But that doesn't in any way limit the usability of working copies on systems such as Windows that do not support symlinks. On such systems, Subversion simply creates a regular text file whose contents are the path to which the original symlink pointed. While that file can't be used as a symlink on a Windows system, it also won't prevent Windows users from performing their other Subversion-related activities.

Here is an overview of the five Subversion subcommands that you'll use most often to make tree changes:

`svn add FOO`

Use this to schedule the file, directory, or symbolic link `FOO` to be added to the repository. When you next commit, `FOO` will become a child of its parent directory. Note that if `FOO` is a directory, everything underneath `FOO` will be scheduled for addition. If you want only to add `FOO` itself, pass the `--depth=empty` option.

`svn delete FOO`

Use this to schedule the file, directory, or symbolic link `FOO` to be deleted from the repository. If `FOO` is a file or link, it is immediately deleted from your working copy. If `FOO` is a directory, it is not deleted, but Subversion schedules it for deletion. When you commit your changes, `FOO` will be entirely removed from your working copy and the repository.[1]

`svn copy FOO BAR`

Create a new item `BAR` as a duplicate of `FOO` and automatically schedule `BAR` for addition. When `BAR` is added to the repository on the next commit, its copy history is recorded (as having originally come from `FOO`). **svn copy** does not create intermediate directories unless you pass the `--parents` option.

`svn move FOO BAR`

This command is exactly the same as running **`svn copy FOO BAR; svn delete FOO`**. That is, `BAR` is scheduled for addition as a copy of `FOO`, and `FOO` is scheduled for removal. **svn move** does not create intermediate directories unless you pass the `--parents` option.

`svn mkdir FOO`

This command is exactly the same as running **`mkdir FOO; svn add FOO`**. That is, a new directory named `FOO` is created and scheduled for addition.

Changing the Repository Without a Working Copy

Subversion *does* offer ways to immediately commit tree changes to the repository without an explicit commit action. In particular, specific uses of **svn mkdir**, **svn copy**, **svn move**, and **svn delete** can operate directly on repository URLs as well as on working copy paths. Of course, as previously mentioned, **svn import** always makes direct changes to the repository.

There are pros and cons to performing URL-based operations. One obvious advantage to doing so is speed: sometimes, checking out a working copy that you don't already have solely to perform some seemingly simple action is an overbearing cost. A disadvantage is that you are generally limited to a single, or single type of, operation at a time when operating directly on URLs. Finally, the primary advantage of a working copy is in its utility as a sort of "staging area" for changes. You can make sure that the changes you are about to commit make sense in the larger scope of your project before committing them. And, of course, these staged changes can be as complex or as a simple as they need to be, yet result in but a single new revision when committed.

[1] Of course, nothing is ever totally deleted from the repository—just from its `HEAD` revision. You may continue to access the deleted item in previous revisions. Should you desire to resurrect the item so that it is again present in `HEAD`, see the section called "Resurrecting Deleted Items".

Review Your Changes

Once you've finished making changes, you need to commit them to the repository, but before you do so, it's usually a good idea to take a look at exactly what you've changed. By examining your changes before you commit, you can compose a more accurate *log message* (a human-readable description of the committed changes stored alongside those changes in the repository). You may also discover that you've inadvertently changed a file, and that you need to undo that change before committing. Additionally, this is a good opportunity to review and scrutinize changes before publishing them. You can see an overview of the changes you've made by using the **svn status** command, and you can dig into the details of those changes by using the **svn diff** command.

Look Ma! No Network!

You can use the commands **svn status**, **svn diff**, and **svn revert** without any network access even if your repository *is* across the network. This makes it easy to manage and review your changes-in-progress when you are working offline or are otherwise unable to contact your repository over the network.

Subversion does this by keeping private caches of pristine, unmodified versions of each versioned file inside its working copy administrative area (or prior to version 1.7, potentially multiple administrative areas). This allows Subversion to report—and revert—local modifications to those files *without network access*. This cache (called the *text-base*) also allows Subversion to send the user's local modifications during a commit to the server as a compressed *delta* (or “difference”) against the pristine version. Having this cache is a tremendous benefit—even if you have a fast Internet connection, it's generally much faster to send only a file's changes rather than the whole file to the server.

See an overview of your changes

To get an overview of your changes, use the **svn status** command. You'll probably use **svn status** more than any other Subversion command.

Because the **cvs status** command's output was so noisy, and because **cvs update** not only performs an update, but also reports the status of your local changes, most CVS users have grown accustomed to using **cvs update** to report their changes. In Subversion, the update and status reporting facilities are completely separate. See the section called “Distinction Between Status and Update” for more details.

If you run **`svn status`** at the top of your working copy with no additional arguments, it will detect and report all file and tree changes you've made.

```
$ svn status
?       scratch.c
A       stuff/loot
A       stuff/loot/new.c
D       stuff/old.c
M       bar.c
$
```

In its default output mode, **svn status** prints seven columns of characters, followed by several whitespace characters, followed by a file or directory name. The first column tells the status of a file or directory and/or its contents. Some of the most common codes that **svn status** displays are:

`? item`
: The file, directory, or symbolic link `item` is not under version control.

`A item`
: The file, directory, or symbolic link `item` has been scheduled for addition into the repository.

`C item`
: The file `item` is in a state of conflict. That is, changes received from the server during an update overlap with local changes that you have in your working copy (and weren't resolved during the update). You must resolve this conflict before committing your changes to the repository.

`D item`
: The file, directory, or symbolic link `item` has been scheduled for deletion from the repository.

`M item`
: The contents of the file `item` have been modified.

If you pass a specific path to **svn status**, you get information about that item alone:

```
$ svn status stuff/fish.c
D       stuff/fish.c
```

svn status also has a `--verbose` (`-v`) option, which will show you the status of *every* item in your working copy, even if it has not been changed:

```
$ svn status -v
M               44        23    sally     README
                44        30    sally     INSTALL
M               44        20    harry     bar.c
                44        18    ira       stuff
                44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
                44        21    sally     stuff/things
A                0         ?     ?        stuff/things/bloo.h
                44        36    harry     stuff/things/gloo.c
```

This is the “long form” output of **svn status**. The letters in the first column mean the same as before, but the second column shows the working revision of the item. The third and fourth columns show the revision in which the item last changed, and who changed it.

None of the prior invocations to **svn status** contact the repository—they merely report what is known about the working copy items based on the records stored in the working copy administrative area and on the timestamps and contents of modified files. But sometimes it is useful to see which of the items in your working copy have been modified in the repository since the last time you updated your working copy. For this, **svn status** offers the `--show-updates` (`-u`) option, which contacts the repository and adds information about items that are out of date:

```
$ svn status -u -v
M      *        44        23    sally     README
M               44        20    harry     bar.c
       *        44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
A                0         ?     ?        stuff/things/bloo.h
Status against revision:   46
```

Notice in the previous example the two asterisks: if you were to run **`svn update`** at this point, you would receive changes to `README` and `trout.c`. This tells you some very useful information—because one of those items is also one that you have locally modified (the file `README`), you'll need to update and get the servers changes for that file before you commit, or the repository will reject your commit for being out of date. We discuss this in more detail later.

svn status can display much more information about the files and directories in your working copy than we've shown here—for an exhaustive description of **svn status** and its output, run **`svn help status`** or see svn status (stat, st) in Chapter 9, *Subversion Complete Reference*.

Examine the details of your local modifications

Another way to examine your changes is with the **svn diff** command, which displays differences in file content. When you run **`svn diff`** at the top of your working copy with no arguments, Subversion will print the changes you've made to human-readable files in your working copy. It displays those changes in *unified diff* format, a format which describes changes as “hunks” (or “snippets”) of a file's content where each line of text is prefixed with a single-character code: a space, which means the line was unchanged; a minus sign (`-`), which means the line was removed from the file; or a plus sign (`+`), which means the line was added to the file. In the context of **svn diff**, those minus-sign- and plus-sign-prefixed lines show how the lines looked before and after your modifications, respectively.

Here's an example:

```
$ svn diff
Index: bar.c
===================================================================
--- bar.c (revision 3)
+++ bar.c (working copy)
@@ -1,7 +1,12 @@
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <unistd.h>
+
+#include <stdio.h>

 int main(void) {
-  printf("Sixty-four slices of American Cheese...\n");
+  printf("Sixty-five slices of American Cheese...\n");
 return 0;
 }

Index: README
===================================================================
--- README (revision 3)
+++ README (working copy)
@@ -193,3 +193,4 @@
+Note to self:  pick up laundry.

Index: stuff/fish.c
===================================================================
--- stuff/fish.c (revision 1)
+++ stuff/fish.c (working copy)
-Welcome to the file known as 'fish'.
-Information on fish will be here soon.

Index: stuff/things/bloo.h
===================================================================
```

```
--- stuff/things/bloo.h (revision 8)
+++ stuff/things/bloo.h (working copy)
+Here is a new file to describe
+things about bloo.
```

The **svn diff** command produces this output by comparing your working files against its pristine text-base. Files scheduled for addition are displayed as files in which every line was added; files scheduled for deletion are displayed as if every line was removed from those files. The output from **svn diff** is somehwat compatible with the **patch** program—more so with the **svn patch** subcommand introduced in Subversion 1.7. Patch processing commands such as these read and apply *patch files* (or “patches”), which are files that describe differences made to one or more files. Because of this, you can share the changes you've made in your working copy with someone else without first committing those changes by creating a patch file from the redirected output of **svn diff**:

```
$ svn diff > patchfile
$
```

Subversion uses its internal diff engine, which produces unified diff format, by default. If you want diff output in a different format, specify an external diff program using `--diff-cmd` and pass any additional flags that it needs via the `--extensions` (`-x`) option. For example, you might want Subversion to defer its difference calculation and display to the GNU **diff** program, asking that program to print local modifications made to the file `foo.c` in context diff format (another flavor of difference format) while ignoring changes made only to the case of the letters used in the file's contents:

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i" foo.c
…
$
```

Fix Your Mistakes

Suppose while viewing the output of **svn diff** you determine that all the changes you made to a particular file are mistakes. Maybe you shouldn't have changed the file at all, or perhaps it would be easier to make different changes starting from scratch. You could edit the file again and unmake all those changes. You could try to find a copy of how the file looked before you changed it, and then copy its contents atop your modified version. You could attempt to apply those changes to the file again in reverse using **`patch -R`**. And there are probably other approaches you could take.

Fortunately in Subversion, undoing your work and starting over from scratch doesn't require such acrobatics. Just use the **svn revert** command:

```
$ svn status README
M       README
$ svn revert README
Reverted 'README'
$ svn status README
$
```

In this example, Subversion has reverted the file to its premodified state by overwriting it with the pristine version of the file cached in the text-base area. But note that **svn revert** can undo *any* scheduled operation—for example, you might decide that you

don't want to add a new file after all:

```
$ svn status new-file.txt
?       new-file.txt
$ svn add new-file.txt
A         new-file.txt
$ svn revert new-file.txt
Reverted 'new-file.txt'
$ svn status new-file.txt
?       new-file.txt
$
```

Or perhaps you mistakenly removed a file from version control:

```
$ svn status README
$ svn delete README
D         README
$ svn revert README
Reverted 'README'
$ svn status README
$
```

The **svn revert** command offers salvation for imperfect people. It can save you huge amounts of time and energy that would otherwise be spent manually unmaking changes or, worse, disposing of your working copy and checking out a fresh one just to have a clean slate to work with again.

Resolve Any Conflicts

We've already seen how **`svn status -u`** can predict conflicts, but dealing with those conflicts is still something that remains to be done. Conflicts can occur any time you attempt to merge or integrate (in a very general sense) changes from the repository into your working copy. By now you know that **svn update** creates exactly that sort of scenario—that command's very purpose is to bring your working copy up to date with the repository by merging all the changes made since your last update into your working copy. So how does Subversion report these conflicts to you, and how do you deal with them?

Suppose you run **`svn update`** and you see this sort of interesting output:

```
$ svn update
Updating '.':
U    INSTALL
G    README
Conflict discovered in 'bar.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options:
```

The `U` (which stands for “Updated”) and `G` (for “merGed”) codes are no cause for concern; those files cleanly absorbed changes from the repository. A file marked with `U` contains no local changes but was updated with changes from the repository. One marked with `G` had local changes to begin with, but the changes coming from the repository didn't overlap with those local changes.

It's the next few lines which are interesting. First, Subversion reports to you that in its attempt to merge outstanding server changes into the file `bar.c`, it has detected that some of those changes clash with local modifications you've made to that file in your working copy but have not yet committed. Perhaps someone has changed the same line of text you also changed. Whatever the reason, Subversion instantly flags this file as being in a state of conflict. It then asks you what you want to do about the problem, allowing you to interactively choose an action to take toward resolving the conflict. The most commonly used options are displayed, but you can see all of the options by typing `s`:

```
…
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options: s

  (e)  edit             - change merged file in an editor
  (df) diff-full        - show all changes made to merged file
  (r)  resolved         - accept merged version of file

  (dc) display-conflict - show all conflicts (ignoring merged version)
  (mc) mine-conflict    - accept my version for all conflicts (same)
  (tc) theirs-conflict  - accept their version for all conflicts (same)

  (mf) mine-full        - accept my version of entire file (even non-conflicts)
  (tf) theirs-full      - accept their version of entire file (same)

  (p)  postpone         - mark the conflict to be resolved later
  (l)  launch           - launch external tool to resolve conflict
  (s)  show all         - show this list

Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options:
```

Let's briefly review each of these options before we go into detail on what each option means.

`(e) edit`
: Open the file in conflict with your favorite editor, as set in the environment variable `EDITOR`.

`(df) diff-full`
: Display the differences between the base revision and the conflicted file itself in unified diff format.

`(r) resolved`
: After editing a file, tell **svn** that you've resolved the conflicts in the file and that it should accept the current contents—basically that you've “resolved” the conflict.

`(dc) display-conflict`
: Display all conflicting regions of the file, ignoring changes which were successfully merged.

`(mc) mine-conflict`
: Discard any newly received changes from the server which conflict with your local changes to the file under review. However, accept and merge all non-conflicting changes received from the server for that file.

`(tc) theirs-conflict`
: Discard any local changes which conflict with incoming changes from the server for the file under review. However, preserve all non-conflicting local changes to that file.

`(mf) mine-full`
: Discard all newly received changes from the server for the file under review, but preserve all your local changes for that file.

`(tf) theirs-full`
: Discard all your local changes to the file under review and use only the newly received changes from the server for that file.

`(p) postpone`
: Leave the file in a conflicted state for you to resolve after your update is complete.

`(l) launch`
: Launch an external program to perform the conflict resolution. This requires a bit of preparation beforehand.

`(s) show all`
: Show the list of all possible commands you can use in interactive conflict resolution.

We'll cover these commands in more detail now, grouping them together by related functionality.

Viewing conflict differences interactively

Before deciding how to attack a conflict interactively, odds are that you'd like to see exactly what is in conflict. Two of the commands available at the interactive conflict resolution prompt can assist you here. The first is the "diff-full" command (**df**), which displays all the local modifications to the file in question plus any conflict regions:

```
…
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options: df
--- .svn/text-base/sandwich.txt.svn-base      Tue Dec 11 21:33:57 2007
+++ .svn/tmp/tempfile.32.tmp     Tue Dec 11 21:34:33 2007
@@ -1 +1,5 @@
-Just buy a sandwich.
+<<<<<<< .mine
+Go pick up a cheesesteak.
+=======
+Bring me a taco!
+>>>>>>> .r32
…
```

The first line of the diff content shows the previous contents of the working copy (the `BASE` revision), the next content line is your change, and the last content line is the change that was just received from the server (*usually* the `HEAD` revision).

The second command is similar to the first, but the "display-conflict" (**dc**) command shows only the conflict regions, not all the changes made to the file. Additionally, this command uses a slightly different display format for the conflict regions which allows you to more easily compare the file's contents in those regions as they would appear in each of three states: original and unedited; with your local changes applied and the server's conflicting changes ignored; and with only the server's incoming changes applied and your local, conflicting changes reverted.

After reviewing the information provided by these commands, you're ready to move on to the next action.

Resolving conflict differences interactively

There are several different ways to resolve conflicts interactively—two of which allow you to selectively merge and edit changes, the rest of which allow you to simply pick a version of the file and move along.

If you wish to choose some combination of your local changes, you can use the "edit" command (**e**) to manually edit the file with conflict markers in a text editor (configured per the instructions in the section called "Using External Editors"). After you've edited the file, if you're satisfied with the changes you've made, you can tell Subversion that the edited file is no longer in conflict by using the "resolved" command (`r`).

Regardless of what your local Unix snob will likely tell you, editing the file by hand in your favorite text editor is a somewhat low-tech way of remedying conflicts (see the section called "Merging conflicts by hand" for a walkthrough). For this reason, Subversion provides the "launch" resolution command (**`l`**) to fire up a fancy graphical merge tool instead (see the section called "External merge").

If you decide that you don't need to merge any changes, but just want to accept one version of the file or the other, you can either choose your changes (a.k.a. "mine") by using the "mine-full" command (**`mf`**) or choose theirs by using the "theirs-full" command (**`tf`**).

Finally, there is also a pair of compromise options available. The "mine-conflict" (**`mc`**) and "theirs-conflict" (**`tc`**) commands instruct Subversion to select your local changes or the server's incoming changes, respectively, as the "winner" for all conflicts in the file. But, unlike the "mine-full" and "theirs-full" commands, these commands preserve both your local changes and changes received from the server in regions of the file where no conflict was detected.

Postponing conflict resolution

This may sound like an appropriate section for avoiding marital disagreements, but it's actually still about Subversion, so read on. If you're doing an update and encounter a conflict that you're not prepared to review or resolve, you can type **`p`** to postpone resolving a conflict on a file-by-file basis when you run **`svn update`**. If you know in advance that you don't want to resolve any conflicts interactively, you can pass the `--non-interactive` option to **svn update**, and any file in conflict will be marked with a `C` automatically.

The `C` (for "Conflicted") means that the changes from the server overlapped with your own, and now you have to manually choose between them after the update has completed. When you postpone a conflict resolution, **svn** typically does three things to assist you in noticing and resolving that conflict:

- Subversion prints a `C` during the update and remembers that the file is in a state of conflict.
- If Subversion considers the file to be mergeable, it places *conflict markers*—special strings of text that delimit the "sides" of the conflict—into the file to visibly demonstrate the overlapping areas. (Subversion uses the `svn:mime-type` property to decide whether a file is capable of contextual, line-based merging. See the section called "File Content Type" to learn more.)
- For every conflicted file, Subversion places three extra unversioned files in your working copy:

  `filename.mine`
  This is the file as it existed in your working copy before you began the update process. It contains any local modifications you had made to the file up to that point. (If Subversion considers the file to be unmergeable, the `.mine` file isn't created, since it would be identical to the working file.)

  `filename.r`*`OLDREV`*
  This is the file as it existed in the `BASE` revision—that is, the unmodified revision of the file in your working copy *before* you began the update process—where *`OLDREV`* is that base revision number.

  `filename.r`*`NEWREV`*
  This is the file that your Subversion client just received from the server via the update of your working copy, where *`NEWREV`* corresponds to the revision number to which you were updating (`HEAD`, unless otherwise requested).

For example, Sally makes changes to the file `sandwich.txt`, but does not yet commit those changes. Meanwhile, Harry commits changes to that same file. Sally updates her working copy before committing and she gets a conflict, which she postpones:

```
$ svn update
Updating '.':
Conflict discovered in 'sandwich.txt'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
```

```
        (s) show all options: p
C    sandwich.txt
Updated to revision 2.
Summary of conflicts:
  Text conflicts: 1
$ ls -1
sandwich.txt
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```

At this point, Subversion will *not* allow Sally to commit the file `sandwich.txt` until the three temporary files are removed:

```
$ svn commit -m "Add a few more things"
svn: E155015: Commit failed (details follow):
svn: E155015: Aborting commit: '/home/sally/svn-work/sandwich.txt' remains in conflict
```

If you've postponed a conflict, you need to resolve the conflict before Subversion will allow you to commit your changes. You'll do this with the **svn resolve** command and one of several arguments to the `--accept` option.

If you want to choose the version of the file that you last checked out before making your edits, choose the *`base`* argument.

If you want to choose the version that contains only your edits, choose the *`mine-full`* argument.

If you want to choose the version that your most recent update pulled from the server (and thus discarding your edits entirely), choose the *`theirs-full`* argument.

However, if you want to pick and choose from your changes and the changes that your update fetched from the server, merge the conflicted text “by hand” (by examining and editing the conflict markers within the file) and then choose the *`working`* argument.

svn resolve removes the three temporary files and accepts the version of the file that you specified with the `--accept` option, and Subversion no longer considers the file to be in a state of conflict:

```
$ svn resolve --accept working sandwich.txt
Resolved conflicted state of 'sandwich.txt'
```

Merging conflicts by hand

Merging conflicts by hand can be quite intimidating the first time you attempt it, but with a little practice, it can become as easy as falling off a bike.

Here's an example. Due to a miscommunication, you and Sally, your collaborator, both edit the file `sandwich.txt` at the same time. Sally commits her changes, and when you go to update your working copy, you get a conflict and you're going to have to edit `sandwich.txt` to resolve the conflict. First, let's take a look at the file:

```
$ cat sandwich.txt
Top piece of bread
Mayonnaise
```

```
Lettuce
Tomato
Provolone
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
```

The strings of less-than signs, equals signs, and greater-than signs are conflict markers and are not part of the actual data in conflict. You generally want to ensure that those are removed from the file before your next commit. The text between the first two sets of markers is composed of the changes you made in the conflicting area:

```
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
```

The text between the second and third sets of conflict markers is the text from Sally's commit:

```
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
```

Usually you won't want to just delete the conflict markers and Sally's changes—she's going to be awfully surprised when the sandwich arrives and it's not what she wanted. This is where you pick up the phone or walk across the office and explain to Sally that you can't get sauerkraut from an Italian deli.[2] Once you've agreed on the changes you will commit, edit your file and remove the conflict markers:

```
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
Salami
Mortadella
Prosciutto
Creole Mustard
Bottom piece of bread
```

[2]And if you ask them for it, they may very well ride you out of town on a rail.

Now use **svn resolve**, and you're ready to commit your changes:

```
$ svn resolve --accept working sandwich.txt
Resolved conflicted state of 'sandwich.txt'
$ svn commit -m "Go ahead and use my sandwich, discarding Sally's edits."
```

Note that **svn resolve**, unlike most of the other commands we deal with in this chapter, requires that you explicitly list any filenames that you wish to resolve. In any case, you want to be careful and use **svn resolve** only when you're certain that you've fixed the conflict in your file—once the temporary files are removed, Subversion will let you commit the file even if it still contains conflict markers.

If you ever get confused while editing the conflicted file, you can always consult the three files that Subversion creates for you in your working copy—including your file as it was before you updated. You can even use a third-party interactive merging tool to examine those three files.

Discarding your changes in favor of a newly fetched revision

If you get a conflict and decide that you want to throw out your changes, you can run **`svn resolve --accept theirs-full`** ***`CONFLICTED-PATH`*** and Subversion will discard your edits and remove the temporary files:

```
$ svn update
Updating '.':
Conflict discovered in 'sandwich.txt'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options: p
C    sandwich.txt
Updated to revision 2.
Summary of conflicts:
  Text conflicts: 1
$ ls sandwich.*
sandwich.txt  sandwich.txt.mine  sandwich.txt.r2  sandwich.txt.r1
$ svn resolve --accept theirs-full sandwich.txt
Resolved conflicted state of 'sandwich.txt'
$
```

Punting: using svn revert

If you decide that you want to throw out your changes and start your edits again (whether this occurs after a conflict or anytime), just revert your changes:

```
$ svn revert sandwich.txt
Reverted 'sandwich.txt'
$ ls sandwich.*
sandwich.txt
$
```

Note that when you revert a conflicted file, you don't have to use **svn resolve**.

Commit Your Changes

Finally! Your edits are finished, you've merged all changes from the server, and you're ready to commit your changes to the repository.

The **svn commit** command sends all of your changes to the repository. When you commit a change, you need to supply a log message describing your change. Your log message will be attached to the new revision you create. If your log message is brief, you may wish to supply it on the command line using the `--message` (`-m`) option:

```
$ svn commit -m "Corrected number of cheese slices."
Sending        sandwich.txt
Transmitting file data .
Committed revision 3.
```

However, if you've been composing your log message in some other text file as you work, you may want to tell Subversion to get the message from that file by passing its filename as the value of the `--file` (`-F`) option:

```
$ svn commit -F logmsg
Sending        sandwich.txt
Transmitting file data .
Committed revision 4.
```

If you fail to specify either the `--message` (`-m`) or `--file` (`-F`) option, Subversion will automatically launch your favorite editor (see the information on `editor-cmd` in the section called “Config”) for composing a log message.

If you're in your editor writing a commit message and decide that you want to cancel your commit, you can just quit your editor without saving changes. If you've already saved your commit message, simply delete all the text, save again, and then abort:

```
$ svn commit
Waiting for Emacs...Done

Log message unchanged or not specified
(a)bort, (c)ontinue, (e)dit
a
$
```

The repository doesn't know or care whether your changes make any sense as a whole; it checks only to make sure nobody else has changed any of the same files that you did when you weren't looking. If somebody *has* done that, the entire commit will fail with a message informing you that one or more of your files are out of date:

```
$ svn commit -m "Add another rule"
Sending        rules.txt
svn: E155011: Commit failed (details follow):
svn: E155011: File '/home/sally/svn-work/sandwich.txt' is out of date
…
```

(The exact wording of this error message depends on the network protocol and server you're using, but the idea is the same in all cases.)

At this point, you need to run **`svn update`**, deal with any merges or conflicts that result, and attempt your commit again.

That covers the basic work cycle for using Subversion. Subversion offers many other features that you can use to manage your repository and working copy, but most of your day-to-day use of Subversion will involve only the commands that we've discussed so far in this chapter. We will, however, cover a few more commands that you'll use fairly often.

Examining History

Your Subversion repository is like a time machine. It keeps a record of every change ever committed and allows you to explore this history by examining previous versions of files and directories as well as the metadata that accompanies them. With a single Subversion command, you can check out the repository (or restore an existing working copy) exactly as it was at any date or revision number in the past. However, sometimes you just want to *peer into* the past instead of *going into* it.

Several commands can provide you with historical data from the repository:

svn diff
: Shows line-level details of a particular change

svn log
: Shows you broad information: log messages with date and author information attached to revisions and which paths changed in each revision

svn cat
: Retrieves a file as it existed in a particular revision number and displays it on your screen

svn annotate
: Retrieves a human-readable file as it existed in a particular revision number, displaying its contents in a tabular form with last-changed information attributed to each line of the file.

svn list
: Displays the files in a directory for any given revision

Examining the Details of Historical Changes

We've already seen **svn diff** before—it displays file differences in unified diff format; we used it to show the local modifications made to our working copy before committing to the repository.

In fact, it turns out that there are *three* distinct uses of **svn diff**:

- Examining local changes
- Comparing your working copy to the repository
- Comparing repository revisions

Examining local changes

As we've seen, invoking **`svn diff`** with no options will compare your working files to the cached “pristine” copies in the `.svn` area:

```
$ svn diff
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparing working copy to repository

If a single `--revision` (`-r`) number is passed, your working copy is compared to the specified revision in the repository:

```
$ svn diff -r 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparing repository revisions

If two revision numbers, separated by a colon, are passed via `--revision` (`-r`), the two revisions are directly compared:

```
$ svn diff -r 2:3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

A more convenient way of comparing one revision to the previous revision is to use the `--change` (`-c`) option:

```
$ svn diff -c 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth open
$
```

Lastly, you can compare repository revisions even when you don't have a working copy on your local machine, just by including the appropriate URL on the command line:

```
$ svn diff -c 5 http://svn.example.com/repos/example/trunk/text/rules.txt
…
$
```

Generating a List of Historical Changes

To find information about the history of a file or directory, use the **svn log** command. **svn log** will provide you with a record of who made changes to a file or directory, at what revision it changed, the time and date of that revision, and—if it was provided—the log message that accompanied the commit:

```
$ svn log
------------------------------------------------------------------------
r3 | sally | 2008-05-15 23:09:28 -0500 (Thu, 15 May 2008) | 1 line

Added include lines and corrected # of cheese slices.
------------------------------------------------------------------------
r2 | harry | 2008-05-14 18:43:15 -0500 (Wed, 14 May 2008) | 1 line

Added main() methods.
------------------------------------------------------------------------
r1 | sally | 2008-05-10 19:50:31 -0500 (Sat, 10 May 2008) | 1 line

Initial import
------------------------------------------------------------------------
```

Note that the log messages are printed in *reverse chronological order* by default. If you wish to see a different range of revisions in a particular order or just a single revision, pass the `--revision` (`-r`) option:

Table 2.1. Common log requests

Command	Description
`svn log -r 5:19`	Display logs for revisions 5 through 19 in chronological order
`svn log -r 19:5`	Display logs for revisions 5 through 19 in reverse chronological order
`svn log -r 8`	Display logs for revision 8 only

You can also examine the log history of a single file or directory. For example:

```
$ svn log foo.c
…
$ svn log http://foo.com/svn/trunk/code/foo.c
…
```

These will display log messages *only* for those revisions in which the working file (or URL) changed.

Why Does svn log Not Show Me What I Just Committed?

If you make a commit and immediately type **`svn log`** with no arguments, you may notice that your most recent commit doesn't show up in the list of log messages. This is due to a combination of the behavior of **svn commit** and the default behavior of **svn log**. First, when you commit changes to the repository, **svn** bumps only the revision of files (and directories) that it commits, so usually the parent directory remains at the older revision (See the section called "Updates and commits are separate" for an explanation of why). **svn log** then defaults to fetching the history of the directory at its current revision, and thus you don't see the newly committed changes. The solution here is to either update your working copy or explicitly provide a revision number to **svn log** by using the `--revision` (`-r`) option.

If you want even more information about a file or directory, **svn log** also takes a `--verbose` (`-v`) option. Because Subversion allows you to move and copy files and directories, it is important to be able to track path changes in the filesystem. So, in verbose mode, **svn log** will include a list of changed paths in a revision in its output:

```
$ svn log -r 8 -v
------------------------------------------------------------------------
r8 | sally | 2008-05-21 13:19:25 -0500 (Wed, 21 May 2008) | 1 line
Changed paths:
   M /trunk/code/foo.c
   M /trunk/code/bar.h
   A /trunk/code/doc/README

Frozzled the sub-space winch.

------------------------------------------------------------------------
```

svn log also takes a `--quiet` (`-q`) option, which suppresses the body of the log message. When combined with `--verbose` (`-v`), it gives just the names of the changed files.

Why Does svn log Give Me an Empty Response?

After working with Subversion for a bit, most users will come across something like this:

```
$ svn log -r 2
------------------------------------------------------------------------
$
```

At first glance, this seems like an error. But recall that while revisions are repository-wide, **svn log** operates on a path in the repository. If you supply no path, Subversion uses the current working directory as the default target. As a result, if you're operating in a subdirectory of your working copy and attempt to see the log of a revision in which neither that directory nor any of its children was changed, Subversion will show you an empty log. If you want to see what changed in that revision, try pointing **svn log** directly at the topmost URL of your repository, as in **`svn log -r 2 ^/`**.

As of Subversion 1.7, users of the Subversion command-line can also take advantage of a special output mode for **svn log** which integrates a difference report such as is generated by the **svn diff** command we introduced earlier. When you invoke **svn log** with the `--diff` option, Subversion will append to each revision log chunk in the log report a **diff**-style difference report. This is a very convenient way to see both the high-level, semantic changes and the line-based modifications of a revision all at the same time!

Browsing the Repository

Using **svn cat** and **svn list**, you can view various revisions of files and directories without changing the working revision of your working copy. In fact, you don't even need a working copy to use either one.

Displaying file contents

If you want to examine an earlier version of a file and not necessarily the differences between two files, you can use **svn cat**:

```
$ svn cat -r 2 rules.txt
Be kind to others
Freedom = Chocolate Ice Cream
Everything in moderation
Chew with your mouth open
$
```

You can also redirect the output directly into a file:

```
$ svn cat -r 2 rules.txt > rules.txt.v2
$
```

Displaying line-by-line change attribution

Very similar to the **svn cat** command we discussed in the previous section is the **svn annotate** command. This command also displays the contents of a versioned file, but it does so using a tabular format. Each line of output shows not only a line of the file's content but also the username, the revision number and (optionally) the datestamp of the revision in which that line was last modified.

When used with a working copy file target, **svn annotate** will by default show line-by-line attribution of the file as it currently ap-

pears in the working copy.

```
$ svn annotate rules.txt
     1      harry Be kind to others
     3      sally Freedom = Responsibility
     1      harry Everything in moderation
     -          - Chew with your mouth closed
     -          - Listen when others are speaking
```

Notice that for some lines, there is no attribution provided. In this case, that's because those lines have been modified in the working copy's version of the file. In this way, **svn annotate** becomes another way for you to see which lines in the file you have changed. You can use the `BASE` revision keyword (see the section called “Revision Keywords”) to instead see the unmodified form of the file as it resides in your working copy.

```
$ svn annotate rules.txt@BASE
     1      harry Be kind to others
     3      sally Freedom = Responsibility
     1      harry Everything in moderation
     1      harry Chew with your mouth open
```

The `--verbose (-v)` option causes **svn annotate** to also include on each line the datestamp associated with that line's reported revision number. (This adds a significant amount of width to each line of ouput, so we'll skip the demonstration here.)

As with **svn cat**, you can also ask **svn annotate** to display previous versions of the file. This can be a handy trick when, after finding out who most recently modified a particular line of interest in the file, you then wish to see who modified the same line prior to that.

```
$ svn blame rules.txt -r 2
     1      harry Be kind to others
     1      harry Freedom = Chocolate Ice Cream
     1      harry Everything in moderation
     1      harry Chew with your mouth open
```

Unlike the **svn cat** command, the functionality of **svn annotate** is tied heavily to the idea of “lines” of text in a human-readable file. As such, if you attempt to run the command on a file that Subversion has determined is *not* human-readable (per the file's `svn:mime-type` property—see the section called “File Content Type” for details), you'll get an error message.

```
$ svn annotate images/logo.png
Skipping binary file: 'images/logo.png'
$
```

As revealed in the error message, you can use the `--force` option to disable this check and proceed with the annotation as if the file's contents are, in fact, human-readable and line-based. Naturally, if you force Subversion to try to perform line-based annotation on a nontextual file, you'll get what you asked for: a screenful of nonsense.

```
$ svn annotate images/logo.png --force
     6      harry \211PNG
     6      harry ^Z
     6      harry
     7      harry \274\361\MI\300\365\353^X\300…
```

Depending on your mood at the time you execute this command and your reasons for doing so, you may find yourself typing **`svn blame …`** or **`svn praise …`** instead of using the canonical **svn annotate** command form. That's okay—the Subversion developers anticipated as much, so those particular command aliases work, too!

Finally, as with many of Subversion's informational commands, you can also reference files in your **svn annotate** command invocations by their repository URLs, allowing access to this information even when you don't have ready access to a working copy.

Listing versioned directories

The **svn list** command shows you what files are in a repository directory without actually downloading the files to your local machine:

```
$ svn list http://svn.example.com/repo/project
README
branches/
tags/
trunk/
```

If you want a more detailed listing, pass the `--verbose` (`-v`) flag to get output like this:

```
$ svn list -v http://svn.example.com/repo/project
  23351 sally                 Feb 05 13:26 ./
  20620 harry            1084 Jul 13  2006 README
  23339 harry                 Feb 04 01:40 branches/
  23198 harry                 Jan 23 17:17 tags/
  23351 sally                 Feb 05 13:26 trunk/
```

The columns tell you the revision at which the file or directory was last modified, the user who modified it, the size if it is a file, the date it was last modified, and the item's name.

The **`svn list`** command with no arguments defaults to the *repository URL* of the current working directory, *not* the local working copy directory. After all, if you want a listing of your local directory, you could use just plain **ls** (or any reasonable non-Unixy equivalent).

Fetching Older Repository Snapshots

In addition to all of the previous commands, you can use the `--revision` (`-r`) option with **svn update** to take an entire working copy “back in time”:[3]

```
# Make the current directory look like it did in r1729.
$ svn update -r 1729
Updating '.':
…
$
```

Many Subversion newcomers attempt to use the preceding **svn update** example to “undo” committed changes, but this won't work as you can't commit changes that you obtain from backdating a working copy if the changed files have newer revisions. See the section called “Resurrecting Deleted Items” for a description of how to “undo” a commit.

If you'd prefer to create a whole new working copy from an older snapshot, you can do so by modifying the typical **svn checkout** command. As with **svn update**, you can provide the `--revision` (`-r`) option. But for reasons that we cover in the section called “Peg and Operative Revisions”, you might instead want to specify the target revision as part of Subversion's expanded URL syntax.

```
# Checkout the trunk from r1729.
$ svn checkout http://svn.example.com/svn/repo/trunk@1729 trunk-1729
…
# Checkout the current trunk as it looked in r1729.
$ svn checkout http://svn.example.com/svn/repo/trunk -r 1729 trunk-1729
…
$
```

Lastly, if you're building a release and wish to bundle up your versioned files and directories, you can use **svn export** to create a local copy of all or part of your repository without any `.svn` administrative directories included. The basic syntax of this subcommand is identical to that of **svn checkout**:

```
# Export the trunk from the latest revision.
$ svn export http://svn.example.com/svn/repo/trunk trunk-export
…
# Export the trunk from r1729.
$ svn export http://svn.example.com/svn/repo/trunk@1729 trunk-1729
…
# Export the current trunk as it looked in r1729.
$ svn export http://svn.example.com/svn/repo/trunk -r 1729 trunk-1729
…
$
```

Sometimes You Just Need to Clean Up

Now that we've covered the day-to-day tasks that you'll frequently use Subversion for, we'll review a few administrative tasks relating to your working copy.

[3] See? We told you that Subversion was a time machine.

Disposing of a Working Copy

Subversion doesn't track either the state or the existence of working copies on the server, so there's no server overhead to keeping working copies around. Likewise, there's no need to let the server know that you're going to delete a working copy.

If you're likely to use a working copy again, there's nothing wrong with just leaving it on disk until you're ready to use it again, at which point all it takes is an **svn update** to bring it up to date and ready for use.

However, if you're definitely not going to use a working copy again, you can safely delete the entire thing using whatever directory removal capabilities your operating system offers. We recommend that before you do so you run **`svn status`** and review any files listed in its output that are prefixed with a `?` to make certain that they're not of importance.

Recovering from an Interruption

When Subversion modifies your working copy—either your files or its own administrative state—it tries to do so as safely as possible. Before changing the working copy, Subversion logs its intentions in a private “to-do list”, of sorts. Next, it performs those actions to effect the desired change, holding a lock on the relevant part of the working copy while it works. This prevents other Subversion clients from accessing the working copy mid-change. Finally, Subversion releases its lock and cleans up its private to-do list. Architecturally, this is similar to a journaled filesystem. If a Subversion operation is interrupted (e.g, if the process is killed or if the machine crashes), the private to-do list remains on disk. This allows Subversion to return to that list later to complete any unfinished operations and return your working copy to a consistent state.

This is exactly what **svn cleanup** does: it searches your working copy and runs any leftover to-do items, removing working copy locks as it completes those operations. If Subversion ever tells you that some part of your working copy is “locked,” run **svn cleanup** to remedy the problem. The **svn status** command will inform you about administrative locks in the working copy, too, by displaying an `L` next to those locked paths:

```
$ svn status
  L     somedir
M       somedir/foo.c
$ svn cleanup
$ svn status
M       somedir/foo.c
```

Don't confuse these working copy administrative locks with the user-managed locks that Subversion users create when using the lock-modify-unlock model of concurrent version control; see the sidebar The Three Meanings of “Lock” for clarification.

Dealing with Structural Conflicts

So far, we have only talked about conflicts at the level of file content. When you and your collaborators make overlapping changes within the same file, Subversion forces you to merge those changes before you can commit.[4]

But what happens if your collaborators move or delete a file that you are still working on? Maybe there was a miscommunication, and one person thinks the file should be deleted, while another person still wants to commit changes to the file. Or maybe your collaborators did some refactoring, renaming files and moving around directories in the process. If you were still working on these files, those modifications may need to be applied to the files at their new location. Such conflicts manifest themselves at the directory tree structure level rather than at the file content level, and are known as *tree conflicts*.

Tree conflicts prior to Subversion 1.6

Prior to Subversion 1.6, tree conflicts could yield rather unexpected results. For example, if a file was locally modified, but

[4]Well, you *could* mark files containing conflict markers as resolved and commit them, if you really wanted to. But this is rarely done in practice.

had been renamed in the repository, running **svn update** would make Subversion carry out the following steps:

- Check the file to be renamed for local modifications.
- Delete the file at its old location, and if it had local modifications, keep an on-disk copy of the file at the old location. This on-disk copy now appears as an unversioned file in the working copy.
- Add the file, as it exists in the repository, at its new location.

When this situation arises, there is the possibility that the user makes a commit without realizing that local modifications have been left in a now-unversioned file in the working copy, and have not reached the repository. This gets more and more likely (and tedious) if the number of files affected by this problem is large.

Since Subversion 1.6, this and other similar situations are flagged as conflicts in the working copy.

As with textual conflicts, tree conflicts prevent a commit from being made from the conflicted state, giving the user the opportunity to examine the state of the working copy for potential problems arising from the tree conflict, and resolving any such problems before committing.

An Example Tree Conflict

Suppose a software project you were working on currently looked like this:

```
$ svn list -Rv svn://svn.example.com/trunk/
     13 harry                 Sep 06 10:34 ./
     13 harry              27 Sep 06 10:34 COPYING
     13 harry              41 Sep 06 10:32 Makefile
     13 harry              53 Sep 06 10:34 README
     13 harry                 Sep 06 10:32 code/
     13 harry              54 Sep 06 10:32 code/bar.c
     13 harry             130 Sep 06 10:32 code/foo.c
$
```

Later, in revision 14, your collaborator Harry renames the file `bar.c` to `baz.c`. Unfortunately, you don't realize this yet. As it turns out, you are busy in your working copy composing a different set of changes, some of which also involve modifications to `bar.c`:

```
$ svn diff
Index: code/foo.c
===================================================================
--- code/foo.c (revision 13)
+++ code/foo.c (working copy)
@@ -3,5 +3,5 @@
 int main(int argc, char *argv[])
 {
     printf("I don't like being moved around!\n%s", bar());
-    return 0;
+    return 1;
 }
Index: code/bar.c
===================================================================
```

```
--- code/bar.c (revision 13)
+++ code/bar.c (working copy)
@@ -1,4 +1,4 @@
 const char *bar(void)
 {
-    return "Me neither!\n";
+    return "Well, I do like being moved around!\n";
 }
$
```

You first realize that someone else has changed `bar.c` when your own commit attempt fails:

```
$ svn commit -m "Small fixes"
Sending        code/bar.c
svn: E155011: Commit failed (details follow):
svn: E155011: File '/home/svn/project/code/bar.c' is out of date
svn: E160013: File not found: transaction '14-e', path '/code/bar.c'
$
```

At this point, you need to run **svn update**. Besides bringing our working copy up to date so that you can see Harry's changes, this also flags a tree conflict so you have the opportunity to evaluate and properly resolve it.

```
$ svn update
Updating '.':
   C code/bar.c
A    code/baz.c
U    Makefile
Updated to revision 14.
Summary of conflicts:
  Tree conflicts: 1
$
```

In its output, **svn update** signifies tree conflicts using a capital C in the fourth output column. **svn status** reveals additional details of the conflict:

```
$ svn status
M       code/foo.c
A  +  C code/bar.c
      >   local edit, incoming delete upon update
Summary of conflicts:
  Tree conflicts: 1
$
```

Note how `bar.c` is automatically scheduled for re-addition in your working copy, which simplifies things in case you want to keep the file.

Because a move in Subversion is implemented as a copy operation followed by a delete operation, and these two operations cannot be easily related to one another during an update, all Subversion can warn you about is an incoming delete operation on a locally

modified file. This delete operation *may* be part of a move, or it could be a genuine delete operation. Determining exactly what semantic change was made to the repository is important—you want to know just how your own edits fit into the overall trajectory of the project. So read log messages, talk to your collaborators, study the line-based differences—do whatever you must do—to determine your best course of action.

In this case, Harry's commit log message tells you what you need to know.

```
$ svn log -r14 ^/trunk
------------------------------------------------------------------------
r14 | harry | 2011-09-06 10:38:17 -0400 (Tue, 06 Sep 2011) | 1 line
Changed paths:
   M /Makefile
   D /code/bar.c
   A /code/baz.c (from /code/bar.c:13)

Rename bar.c to baz.c, and adjust Makefile accordingly.
------------------------------------------------------------------------
$
```

svn info shows the URLs of the items involved in the conflict. The *left* URL shows the source of the local side of the conflict, while the *right* URL shows the source of the incoming side of the conflict. These URLs indicate where you should start searching the repository's history for the change which conflicts with your local change.

```
$ svn info code/bar.c
Path: code/bar.c
Name: bar.c
URL: http://svn.example.com/svn/repo/trunk/code/bar.c
…
Tree conflict: local edit, incoming delete upon update
  Source  left: (file) ^/trunk/code/bar.c@4
  Source right: (none) ^/trunk/code/bar.c@5

$
```

`bar.c` is now said to be the *victim* of a tree conflict. It cannot be committed until the conflict is resolved:

```
$ svn commit -m "Small fixes"
svn: E155015: Commit failed (details follow):
svn: E155015: Aborting commit: '/home/svn/project/code/bar.c' remains in confl
ict
$
```

To resolve this conflict, you must either agree or disagree with the move that Harry made.

If you agree with the move, your `bar.c` is superfluous. You'll want to delete it and mark the tree conflict as resolved. But wait: you made changes to that file! Before deleting `bar.c`, you need to decide if the changes you made to it need to be applied elsewhere, for example to the new `baz.c` file where all of `bar.c`'s code now lives. Let's assume that your changes do need to “follow the move”. Subversion isn't smart enough to do this work for you[5], so you need to migrate your changes manually.

[5]In some cases, Subversion 1.5 and 1.6 *would* actually handle this for you, but this somewhat hit-or-miss functionality was removed in Subversion 1.7.

In our example, you could manually re-make your change to `bar.c` pretty easily—it was, after all, a single-line change. That's not always the case, though, so we'll show a more scalable approach. We'll first use **svn diff** to create a patch file. Then we'll edit the headers of that patch file to point to the new name of our renamed file. Finally, we re-apply the modified patch to our working copy.

```
$ svn diff code/bar.c > PATCHFILE
$ cat PATCHFILE
Index: code/bar.c
===================================================================
--- code/bar.c (working copy)
+++ code/bar.c (working copy)
@@ -1,4 +1,4 @@
 const char *bar(void)
 {
-    return "Me neither!\n";
+    return "Well, I do like being moved around!\n";
 }
$ ### Edit PATCHFILE to refer to code/baz.c instead of code/bar.c
$ cat PATCHFILE
Index: code/baz.c
===================================================================
--- code/baz.c (working copy)
+++ code/baz.c (working copy)
@@ -1,4 +1,4 @@
 const char *bar(void)
 {
-    return "Me neither!\n";
+    return "Well, I do like being moved around!\n";
 }
$ svn patch PATCHFILE
U         code/baz.c
$
```

Now that the changes you originally made to `bar.c` have been successfully reproduced in `baz.c`, you can delete `bar.c` and resolve the conflict, instructing the resolution logic to accept what is currently in the working copy as the desired result.

```
$ svn delete --force code/bar.c
D         code/bar.c
$ svn resolve --accept=working code/bar.c
Resolved conflicted state of 'code/bar.c'
$ svn status
M       code/foo.c
M       code/baz.c
$ svn diff
Index: code/foo.c
===================================================================
--- code/foo.c  (revision 14)
+++ code/foo.c  (working copy)
@@ -3,5 +3,5 @@
 int main(int argc, char *argv[])
 {
     printf("I don't like being moved around!\n%s", bar());
-    return 0;
+    return 1;
 }
Index: code/baz.c
===================================================================
```

```
--- code/baz.c  (revision 14)
+++ code/baz.c  (working copy)
@@ -1,4 +1,4 @@
 const char *bar(void)
 {
-    return "Me neither!\n";
+    return "Well, I do like being moved around!\n";
 }
$
```

But what if you do not agree with the move? Well, in that case, you can delete `baz.c` instead, after making sure any changes made to it after it was renamed are either preserved or not worth keeping. (Do not forget to also revert the changes Harry made to `Makefile`.) Since `bar.c` is already scheduled for re-addition, there is nothing else left to do, and the conflict can be marked resolved:

```
$ svn delete --force code/baz.c
D         code/baz.c
$ svn resolve --accept=working code/bar.c
Resolved conflicted state of 'code/bar.c'
$ svn status
M       code/foo.c
A  +    code/bar.c
D       code/baz.c
M       Makefile
$ svn diff
Index: code/foo.c
===================================================================
--- code/foo.c (revision 14)
+++ code/foo.c (working copy)
@@ -3,5 +3,5 @@
 int main(int argc, char *argv[])
 {
     printf("I don't like being moved around!\n%s", bar());
-    return 0;
+    return 1;
 }
Index: code/bar.c
===================================================================
--- code/bar.c (revision 14)
+++ code/bar.c (working copy)
@@ -1,4 +1,4 @@
 const char *bar(void)
 {
-    return "Me neither!\n";
+    return "Well, I do like being moved around!\n";
 }
Index: code/baz.c
===================================================================
--- code/baz.c (revision 14)
+++ code/baz.c (working copy)
@@ -1,4 +0,0 @@
-const char *bar(void)
-{
-    return "Me neither!\n";
-}
Index: Makefile
===================================================================
--- Makefile (revision 14)
```

```
+++ Makefile (working copy)
@@ -1,2 +1,2 @@
 foo:
- $(CC) -o $@ code/foo.c code/baz.c
+ $(CC) -o $@ code/foo.c code/bar.c
```

You've now resolved your first tree conflict! You can commit your changes and tell Harry during tea break about all the extra work he caused for you.

Summary

Now we've covered most of the Subversion client commands. Notable exceptions are those dealing with branching and merging (see Chapter 4, *Branching and Merging*) and properties (see the section called "Properties"). However, you may want to take a moment to skim through Chapter 9, *Subversion Complete Reference* to get an idea of all the different commands that Subversion has—and how you can use them to make your work easier.

Chapter 3. Advanced Topics

If you've been reading this book chapter by chapter, from start to finish, you should by now have acquired enough knowledge to use the Subversion client to perform the most common version control operations. You understand how to check out a working copy from a Subversion repository. You are comfortable with submitting and receiving changes using the **svn commit** and **svn update** operations. You've probably even developed a reflex that causes you to run the **svn status** command almost unconsciously. For all intents and purposes, you are ready to use Subversion in a typical environment.

But the Subversion feature set doesn't stop at "common version control operations." It has other bits of functionality besides just communicating file and directory changes to and from a central repository.

This chapter highlights some of Subversion's features that, while important, may not be part of the typical user's daily routine. It assumes that you are familiar with Subversion's basic file and directory versioning capabilities. If you aren't, you'll want to first read Chapter 1, *Fundamental Concepts* and Chapter 2, *Basic Usage*. Once you've mastered those basics and consumed this chapter, you'll be a Subversion power user!

Revision Specifiers

As we described in the section called "Revisions", revision numbers in Subversion are pretty straightforward—integers that keep getting larger as you commit more changes to your versioned data. Still, it doesn't take long before you can no longer remember exactly what happened in each and every revision. Fortunately, the typical Subversion workflow doesn't often demand that you supply arbitrary revisions to the Subversion operations you perform. For operations that *do* require a revision specifier, you generally supply a revision number that you saw in a commit email, in the output of some other Subversion operation, or in some other context that would give meaning to that particular number.

Referring to revision numbers with an "`r`" prefix (`r314`, for example) is an established practice in Subversion communities, and is both supported and encouraged by many Subversion-related tools. In most places where you would specify a bare revision number on the command line, you may also use the `r`*`NNN`* syntax.

But occasionally, you need to pinpoint a moment in time for which you don't already have a revision number memorized or handy. So besides the integer revision numbers, **svn** allows as input some additional forms of revision specifiers: *revision keywords* and revision dates.

The various forms of Subversion revision specifiers can be mixed and matched when used to specify revision ranges. For example, you can use `-r` *`REV1`*`:`*`REV2`* where *`REV1`* is a revision keyword and *`REV2`* is a revision number, or where *`REV1`* is a date and *`REV2`* is a revision keyword, and so on. The individual revision specifiers are independently evaluated, so you can put whatever you want on the opposite sides of that colon.

Revision Keywords

The Subversion client understands a number of revision keywords. These keywords can be used instead of integer arguments to the `--revision` (`-r`) option, and are resolved into specific revision numbers by Subversion:

`HEAD`

The latest (or "youngest") revision in the repository.

`BASE`

The revision number of an item in a working copy. If the item has been locally modified, this refers to the way the item appears without those local modifications.

`COMMITTED`

The most recent revision prior to, or equal to, `BASE`, in which an item changed.

`PREV`

The revision immediately *before* the last revision in which an item changed. Technically, this boils down to `COMMITTED`-1.

As can be derived from their descriptions, the `PREV`, `BASE`, and `COMMITTED` revision keywords are used only when referring to a working copy path—they don't apply to repository URLs. `HEAD`, on the other hand, can be used in conjunction with both of these path types.

Here are some examples of revision keywords in action:

```
$ svn diff -r PREV:COMMITTED foo.c
# shows the last change committed to foo.c

$ svn log -r HEAD
# shows log message for the latest repository commit

$ svn diff -r HEAD
# compares your working copy (with all of its local changes) to the
# latest version of that tree in the repository

$ svn diff -r BASE:HEAD foo.c
# compares the unmodified version of foo.c with the latest version of
# foo.c in the repository

$ svn log -r BASE:HEAD
# shows all commit logs for the current versioned directory since you
# last updated

$ svn update -r PREV foo.c
# rewinds the last change on foo.c, decreasing foo.c's working revision

$ svn diff -r BASE:14 foo.c
# compares the unmodified version of foo.c with the way foo.c looked
# in revision 14
```

Revision Dates

Revision numbers reveal nothing about the world outside the version control system, but sometimes you need to correlate a moment in real time with a moment in version history. To facilitate this, the `--revision` (`-r`) option can also accept as input date specifiers wrapped in curly braces (`{` and `}`). Subversion accepts the standard ISO-8601 date and time formats, plus a few others. Here are some examples.

```
$ svn update -r {2006-02-17}
$ svn update -r {15:30}
$ svn update -r {15:30:00.200000}
$ svn update -r {"2006-02-17 15:30"}
$ svn update -r {"2006-02-17 15:30 +0230"}
$ svn update -r {2006-02-17T15:30}
$ svn update -r {2006-02-17T15:30Z}
$ svn update -r {2006-02-17T15:30-04:00}
```

```
$ svn update -r {20060217T1530}
$ svn update -r {20060217T1530Z}
$ svn update -r {20060217T1530-0500}
…
```

Keep in mind that most shells will require you to, at a minimum, quote or otherwise escape any spaces that are included as part of revision date specifiers. Certain shells may also take issue with the unescaped use of curly braces, too. Consult your shell's documentation for the requirements specific to your environment.

When you specify a date, Subversion resolves that date to the most recent revision of the repository as of that date, and then continues to operate against that resolved revision number:

```
$ svn log -r {2006-11-28}
------------------------------------------------------------------------
r12 | ira | 2006-11-27 12:31:51 -0600 (Mon, 27 Nov 2006) | 6 lines
…
```

Is Subversion a Day Early?

If you specify a single date as a revision without specifying a time of day (for example `2006-11-27`), you may think that Subversion should give you the last revision that took place on the 27th of November. Instead, you'll get back a revision from the 26th, or even earlier. Remember that Subversion will find the *most recent revision of the repository* as of the date you give. If you give a date without a timestamp, such as `2006-11-27`, Subversion assumes a time of 00:00:00, so looking for the most recent revision won't return anything on the 27th.

If you want to include the 27th in your search, you can either specify the 27th with the time (`{"2006-11-27 23:59"}`), or just specify the next day (`{2006-11-28}`).

You can also use a range of dates. Subversion will find all revisions between both dates, inclusive:

```
$ svn log -r {2006-11-20}:{2006-11-29}
…
```

Since the timestamp of a revision is stored as an unversioned, modifiable property of the revision (see the section called “Properties”), revision timestamps can be changed to represent complete falsifications of true chronology, or even removed altogether. Subversion's ability to correctly convert revision dates into real revision numbers depends on revision datestamps maintaining a sequential ordering—the younger the revision, the younger its timestamp. If this ordering isn't maintained, you will likely find that trying to use dates to specify revision ranges in your repository doesn't always return the data you might have expected.

Peg and Operative Revisions

We copy, move, rename, and completely replace files and directories on our computers all the time. And your version control sys-

tem shouldn't get in the way of your doing these things with your version-controlled files and directories, either. Subversion's file management support is quite liberating, affording almost as much flexibility for versioned files as you'd expect when manipulating your unversioned ones. But that flexibility means that across the lifetime of your repository, a given versioned object might have many paths, and a given path might represent several entirely different versioned objects. This introduces a certain level of complexity to your interactions with those paths and objects.

Subversion is pretty smart about noticing when an object's version history includes such “changes of address.” For example, if you ask for the revision history log of a particular file that was renamed last week, Subversion happily provides all those logs—the revision in which the rename itself happened, plus the logs of relevant revisions both before and after that rename. So, most of the time, you don't even have to think about such things. But occasionally, Subversion needs your help to clear up ambiguities.

The simplest example of this occurs when a directory or file is deleted from version control, and then a new directory or file is created with the same name and added to version control. The thing you deleted and the thing you later added aren't the same thing. They merely happen to have had the same path—`/trunk/object`, for example. What, then, does it mean to ask Subversion about the history of `/trunk/object`? Are you asking about the thing currently at that location, or the old thing you deleted from that location? Are you asking about the operations that have happened to *all* the objects that have ever lived at that path? Subversion needs a hint about what you really want.

And thanks to moves, versioned object history can get far more twisted than even that. For example, you might have a directory named `concept`, containing some nascent software project you've been toying with. Eventually, though, that project matures to the point that the idea seems to actually have some wings, so you do the unthinkable and decide to give the project a name.[1] Let's say you called your software Frabnaggilywort. At this point, it makes sense to rename the directory to reflect the project's new name, so `concept` is renamed to `frabnaggilywort`. Life goes on, Frabnaggilywort releases a 1.0 version and is downloaded and used daily by hordes of people aiming to improve their lives.

It's a nice story, really, but it doesn't end there. Entrepreneur that you are, you've already got another think in the tank. So you make a new directory, `concept`, and the cycle begins again. In fact, the cycle begins again many times over the years, each time starting with that old `concept` directory, then sometimes seeing that directory renamed as the idea cures, sometimes seeing it deleted when you scrap the idea. Or, to get really sick, maybe you rename `concept` to something else for a while, but later rename the thing back to `concept` for some reason.

In scenarios like these, attempting to instruct Subversion to work with these reused paths can be a little like instructing a motorist in Chicago's West Suburbs to drive east down Roosevelt Road and turn left onto Main Street. In a mere 20 minutes, you can cross “Main Street” in Wheaton, Glen Ellyn, and Lombard. And no, they aren't the same street. Our motorist—and our Subversion—need a little more detail to do the right thing.

Fortunately, Subversion allows you to tell it exactly which Main Street you meant. The mechanism used is called a *peg revision*, and you provide these to Subversion for the sole purpose of identifying unique lines of history. Because at most one versioned object may occupy a path at any given time—or, more precisely, in any one revision—the combination of a path and a peg revision is all that is needed to unambiguously identify a specific line of history. Peg revisions are specified to the Subversion command-line client using *at syntax*, so called because the syntax involves appending an “at sign” (`@`) and the peg revision to the end of the path with which the revision is associated.

But what of the `--revision` (`-r`) of which we've spoken so much in this book? That revision (or set of revisions) is called the *operative revision* (or *operative revision range*). Once a particular line of history has been identified using a path and peg revision, Subversion performs the requested operation using the operative revision(s). To map this to our Chicagoland streets analogy, if we are told to go to 606 N. Main Street in Wheaton,[2] we can think of “Main Street” as our path and “Wheaton” as our peg revision. These two pieces of information identify a unique path that can be traveled (north or south on Main Street), and they keep us from traveling up and down the wrong Main Street in search of our destination. Now we throw in “606 N.” as our operative revision of sorts, and we know *exactly* where to go.

The Peg Revision Algorithm

The Subversion command-line client performs the peg revision algorithm any time it needs to resolve possible ambiguities in

[1] “You're not supposed to name it. Once you name it, you start getting attached to it.”—Mike Wazowski
[2] 606 N. Main Street, Wheaton, Illinois, is the home of the Wheaton *History* Center. It seemed appropriate….

the paths and revisions provided to it. Here's an example of such an invocation:

```
$ svn command -r OPERATIVE-REV item@PEG-REV
```

If *OPERATIVE-REV* is older than *PEG-REV*, the algorithm is as follows:

1. Locate *item* in the revision identified by *PEG-REV*. There can be only one such object.
2. Trace the object's history backwards (through any possible renames) to its ancestor in the revision *OPERATIVE-REV*.
3. Perform the requested action on that ancestor, wherever it is located, or whatever its name might be or might have been at that time.

But what if *OPERATIVE-REV* is *younger* than *PEG-REV*? Well, that adds some complexity to the theoretical problem of locating the path in *OPERATIVE-REV*, because the path's history could have forked multiple times (thanks to copy operations) between *PEG-REV* and *OPERATIVE-REV*. And that's not all—Subversion doesn't store enough information to performantly trace an object's history forward, anyway. So the algorithm is a little different:

1. Locate *item* in the revision identified by *OPERATIVE-REV*. There can be only one such object.
2. Trace the object's history backward (through any possible renames) to its ancestor in the revision *PEG-REV*.
3. Verify that the object's location (path-wise) in *PEG-REV* is the same as it is in *OPERATIVE-REV*. If that's the case, at least the two locations are known to be directly related, so perform the requested action on the location in *OPERATIVE-REV*. Otherwise, relatedness was not established, so error out with a loud complaint that no viable location was found. (Someday, we expect that Subversion will be able to handle this usage scenario with more flexibility and grace.)

Note that even when you don't explicitly supply a peg revision or operative revision, they are still present. For your convenience, the default peg revision is `BASE` for working copy items and `HEAD` for repository URLs. And when no operative revision is provided, it defaults to being the same revision as the peg revision.

Say that long ago we created our repository, and in revision 1 we added our first `concept` directory, plus an `IDEA` file in that directory talking about the concept. After several revisions in which real code was added and tweaked, we, in revision 20, renamed this directory to `frabnaggilywort`. By revision 27, we had a new concept, a new `concept` directory to hold it, and a new `IDEA` file to describe it. And then five years and thousands of revisions flew by, just like they would in any good romance story.

Now, years later, we wonder what the `IDEA` file looked like back in revision 1. But Subversion needs to know whether we are asking about how the *current* file looked back in revision 1, or whether we are asking for the contents of whatever file lived at `concepts/IDEA` in revision 1. Certainly those questions have different answers, and because of peg revisions, you can ask those questions. To find out how the current `IDEA` file looked in that old revision, you run:

```
$ svn cat -r 1 concept/IDEA
svn: E195012: Unable to find repository location for 'concept/IDEA' in revision 1
```

Of course, in this example, the current `IDEA` file didn't exist yet in revision 1, so Subversion gives an error. The previous command is shorthand for a longer notation which explicitly lists a peg revision. The expanded notation is:

```
$ svn cat -r 1 concept/IDEA@BASE
svn: E195012: Unable to find repository location for 'concept/IDEA' in revision 1
```

And when executed, it has the expected results.

The perceptive reader is probably wondering at this point whether the peg revision syntax causes problems for working copy paths or URLs that actually have at signs in them. After all, how does **svn** know whether `news@11` is the name of a directory in my tree or just a syntax for "revision 11 of `news`"? Thankfully, while **svn** will always assume the latter, there is a trivial workaround. You need only append an at sign to the end of the path, such as `news@11@`. **svn** cares only about the last at sign in the argument, and it is not considered illegal to omit a literal peg revision specifier after that at sign. This workaround even applies to paths that end in an at sign—you would use `filename@@` to talk about a file named `filename@`.

Let's ask the other question, then—in revision 1, what were the contents of whatever file occupied the address `concepts/IDEA` at the time? We'll use an explicit peg revision to help us out.

```
$ svn cat concept/IDEA@1
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

Notice that we didn't provide an operative revision this time. That's because when no operative revision is specified, Subversion assumes a default operative revision that's the same as the peg revision.

As you can see, the output from our operation appears to be correct. The text even mentions frabbing naggily worts, so this is almost certainly the file that describes the software now called Frabnaggilywort. In fact, we can verify this using the combination of an explicit peg revision and explicit operative revision. We know that in `HEAD`, the Frabnaggilywort project is located in the `frabnaggilywort` directory. So we specify that we want to see how the line of history identified in `HEAD` as the path `frabnaggilywort/IDEA` looked in revision 1.

```
$ svn cat -r 1 frabnaggilywort/IDEA@HEAD
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

And the peg and operative revisions need not be so trivial, either. For example, say `frabnaggilywort` had been deleted from `HEAD`, but we know it existed in revision 20, and we want to see the diffs for its `IDEA` file between revisions 4 and 10. We can use peg revision 20 in conjunction with the URL that would have held Frabnaggilywort's `IDEA` file in revision 20, and then use 4 and 10 as our operative revision range.

```
$ svn diff -r 4:10 http://svn.red-bean.com/projects/frabnaggilywort/IDEA@20
Index: frabnaggilywort/IDEA
===================================================================
--- frabnaggilywort/IDEA (revision 4)
```

```
+++ frabnaggilywort/IDEA (revision 10)
@@ -1,5 +1,5 @@
-The idea behind this project is to come up with a piece of software
-that can frab a naggily wort.  Frabbing naggily worts is tricky
-business, and doing it incorrectly can have serious ramifications, so
-we need to employ over-the-top input validation and data verification
-mechanisms.
+The idea behind this project is to come up with a piece of
+client-server software that can remotely frab a naggily wort.
+Frabbing naggily worts is tricky business, and doing it incorrectly
+can have serious ramifications, so we need to employ over-the-top
+input validation and data verification mechanisms.
```

Fortunately, most folks aren't faced with such complex situations. But when you are, remember that peg revisions are that extra hint Subversion needs to clear up ambiguity.

Properties

We've already covered in detail how Subversion stores and retrieves various versions of files and directories in its repository. Whole chapters have been devoted to this most fundamental piece of functionality provided by the tool. And if the versioning support stopped there, Subversion would still be complete from a version control perspective.

But it doesn't stop there.

In addition to versioning your directories and files, Subversion provides interfaces for adding, modifying, and removing versioned metadata on each of your versioned directories and files. We refer to this metadata as *properties*, and they can be thought of as two-column tables that map property names to arbitrary values attached to each item in your working copy. Generally speaking, the names and values of the properties can be whatever you want them to be, with the constraint that the names must contain only ASCII characters. And the best part about these properties is that they, too, are versioned, just like the textual contents of your files. You can modify, commit, and revert property changes as easily as you can file content changes. And the sending and receiving of property changes occurs as part of your typical commit and update operations—you don't have to change your basic processes to accommodate them.

Subversion has reserved the set of properties whose names begin with `svn:` as its own. While there are only a handful of such properties in use today, you should avoid creating custom properties for your own needs whose names begin with this prefix. Otherwise, you run the risk that a future release of Subversion will grow support for a feature or behavior driven by a property of the same name but with perhaps an entirely different interpretation.

Properties show up elsewhere in Subversion, too. Just as files and directories may have arbitrary property names and values attached to them, each revision as a whole may have arbitrary properties attached to it. The same constraints apply—human-readable names and anything-you-want binary values. The main difference is that revision properties are not versioned. In other words, if you change the value of, or delete, a revision property, there's no way, within the scope of Subversion's functionality, to recover the previous value.

Subversion has no particular policy regarding the use of properties. It asks only that you do not use property names that begin with the prefix `svn:` as that's the namespace that it sets aside for its own use. And Subversion does, in fact, use properties—both the versioned and unversioned variety. Certain versioned properties have special meaning or effects when found on files and directories, or they house a particular bit of information about the revisions on which they are found. Certain revision properties are automatically attached to revisions by Subversion's commit process, and they carry information about the revision. Most of these properties are mentioned elsewhere in this or other chapters as part of the more general topics to which they are related. For an exhaustive list of Subversion's predefined properties, see the section called “Subversion Properties” in Chapter 9, *Subversion Complete Reference*.

While Subversion automatically attaches properties (`svn:date`, `svn:author`, `svn:log`, and so on) to revisions, it does *not* presume thereafter the existence of those properties, and neither should you or the tools you use to interact with your repository. Revision properties can be deleted programmatically or via the client (if allowed by the repository hooks) without damaging Subversion's ability to function. So, when writing scripts which operate on your Subversion repository data, do not make the mistake of assuming that any particular revision property exists on a revision.

In this section, we will examine the utility—both to users of Subversion and to Subversion itself—of property support. You'll learn about the property-related **svn** subcommands and how property modifications affect your normal Subversion workflow.

Why Properties?

Just as Subversion uses properties to store extra information about the files, directories, and revisions that it contains, you might also find properties to be of similar use. You might find it useful to have a place close to your versioned data to hang custom metadata about that data.

Say you wish to design a web site that houses many digital photos and displays them with captions and a datestamp. Now, your set of photos is constantly changing, so you'd like to have as much of this site automated as possible. These photos can be quite large, so as is common with sites of this nature, you want to provide smaller thumbnail images to your site visitors.

Now, you can get this functionality using traditional files. That is, you can have your `image123.jpg` and an `image123-thumbnail.jpg` side by side in a directory. Or if you want to keep the filenames the same, you might have your thumbnails in a different directory, such as `thumbnails/image123.jpg`. You can also store your captions and datestamps in a similar fashion, again separated from the original image file. But the problem here is that your collection of files multiplies with each new photo added to the site.

Now consider the same web site deployed in a way that makes use of Subversion's file properties. Imagine having a single image file, `image123.jpg`, with properties set on that file that are named `caption`, `datestamp`, and even `thumbnail`. Now your working copy directory looks much more manageable—in fact, it looks to the casual browser like there are nothing but image files in it. But your automation scripts know better. They know that they can use **svn** (or better yet, they can use the Subversion language bindings—see the section called “Using the APIs”) to dig out the extra information that your site needs to display without having to read an index file or play path manipulation games.

While Subversion places few restrictions on the names and values you use for properties, it has not been designed to optimally carry large property values or large sets of properties on a given file or directory. Subversion commonly holds all the property names and values associated with a single item in memory at the same time, which can cause detrimental performance or failed operations when extremely large property sets are used.

Custom revision properties are also frequently used. One common such use is a property whose value contains an issue tracker ID with which the revision is associated, perhaps because the change made in that revision fixes a bug filed in the tracker issue with that ID. Other uses include hanging more friendly names on the revision—it might be hard to remember that revision 1935 was a fully tested revision. But if there's, say, a `test-results` property on that revision with the value `all passing`, that's meaningful information to have. And Subversion allows you to easily do this via the `--with-revprop` option of the **svn commit** command:

```
$ svn commit -m "Fix up the last remaining known regression bug." \
             --with-revprop "test-results=all passing"
Sending        lib/crit_bits.c
Transmitting file data .
Committed revision 912.
$
```

Searchability (or, Why *Not* Properties)

For all their utility, Subversion properties—or, more accurately, the available interfaces to them—have a major shortcoming: while it is a simple matter to *set* a custom property, *finding* that property later is a whole different ball of wax.

Trying to locate a custom revision property generally involves performing a linear walk across all the revisions of the repository, asking of each revision, “Do you have the property I'm looking for?” Use the `--with-all-revprops` option with the **svn log** command's XML output mode to facilitate this search. Notice the presence of the custom revision property `testresults` in the following output:

```
$ svn log --with-all-revprops --xml lib/crit_bits.c
<?xml version="1.0"?>
<log>
<logentry
   revision="912">
<author>harry</author>
<date>2011-07-29T14:47:41.169894Z</date>
<msg>Fix up the last remaining known regression bug.</msg>
<revprops>
<property
   name="testresults">all passing</property>
</revprops>
</logentry>
…
$
```

Trying to find a custom versioned property is painful, too, and often involves a recursive **svn propget** across an entire working copy. In your situation, that might not be as bad as a linear walk across all revisions. But it certainly leaves much to be desired in terms of both performance and likelihood of success, especially if the scope of your search would require a working copy from the root of your repository.

For this reason, you might choose—especially in the revision property use case—to simply add your metadata to the revision's log message using some policy-driven (and perhaps programmatically enforced) formatting that is designed to be quickly parsed from the output of **svn log**. It is quite common to see the following in Subversion log messages:

```
Issue(s): IZ2376, IZ1919
Reviewed by:  sally

This fixes a nasty segfault in the wort frabbing process
…
```

But here again lies some misfortune. Subversion doesn't yet provide a log message templating mechanism, which would go a long way toward helping users be consistent with the formatting of their log-embedded revision metadata.

Manipulating Properties

The **svn** program affords a few ways to add or modify file and directory properties. For properties with short, human-readable values, perhaps the simplest way to add a new property is to specify the property name and value on the command line of the **svn propset** subcommand:

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/button.c
property 'copyright' set on 'calc/button.c'
$
```

But we've been touting the flexibility that Subversion offers for your property values. And if you are planning to have a multiline textual, or even binary, property value, you probably do not want to supply that value on the command line. So the **svn propset** subcommand takes a `--file` (`-F`) option for specifying the name of a file that contains the new property value.

```
$ svn propset license -F /path/to/LICENSE calc/button.c
property 'license' set on 'calc/button.c'
$
```

There are some restrictions on the names you can use for properties. A property name must start with a letter, a colon (`:`), or an underscore (_); after that, you can also use digits, hyphens (`-`), and periods (`.`).[3]

In addition to the **propset** command, the **svn** program supplies the **propedit** command. This command uses the configured editor program (see the section called “Config”) to add or modify properties. When you run the command, **svn** invokes your editor program on a temporary file that contains the current value of the property (or that is empty, if you are adding a new property). Then, you just modify that value in your editor program until it represents the new value you wish to store for the property, save the temporary file, and then exit the editor program. If Subversion detects that you've actually changed the existing value of the property, it will accept that as the new property value. If you exit your editor without making any changes, no property modification will occur:

```
$ svn propedit copyright calc/button.c  ### exit the editor without changes
No changes to property 'copyright' on 'calc/button.c'
$
```

We should note that, as with other **svn** subcommands, those related to properties can act on multiple paths at once. This enables you to modify properties on whole sets of files with a single command. For example, we could have done the following:

```
$ svn propset copyright '(c) 2006 Red-Bean Software' calc/*
property 'copyright' set on 'calc/Makefile'
property 'copyright' set on 'calc/button.c'
property 'copyright' set on 'calc/integer.c'
…
$
```

All of this property adding and editing isn't really very useful if you can't easily get the stored property value. So the **svn** program supplies two subcommands for displaying the names and values of properties stored on files and directories. The **svn proplist** command will list the names of properties that exist on a path. Once you know the names of the properties on the node, you can request their values individually using **svn propget**. This command will, given a property name and a path (or set of paths), print the value of the property to the standard output stream.

[3] If you're familiar with XML, this is pretty much the ASCII subset of the syntax for XML “Name”.

```
$ svn proplist calc/button.c
Properties on 'calc/button.c':
  copyright
  license
$ svn propget copyright calc/button.c
(c) 2006 Red-Bean Software
```

There's even a variation of the **proplist** command that will list both the name and the value for all of the properties. Simply supply the `--verbose` (`-v`) option.

```
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright
    (c) 2006 Red-Bean Software
  license
    ================================================================
    Copyright (c) 2006 Red-Bean Software.  All rights reserved.

    Redistribution and use in source and binary forms, with or without
    modification, are permitted provided that the following conditions
    are met:

    1. Redistributions of source code must retain the above copyright
    notice, this list of conditions, and the recipe for Fitz's famous
    red-beans-and-rice.
    …
```

The last property-related subcommand is **propdel**. Since Subversion allows you to store properties with empty values, you can't remove a property altogether using **svn propedit** or **svn propset**. For example, this command will *not* yield the desired effect:

```
$ svn propset license "" calc/button.c
property 'license' set on 'calc/button.c'
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright
    (c) 2006 Red-Bean Software
  license

$
```

You need to use the **propdel** subcommand to delete properties altogether. The syntax is similar to the other property commands:

```
$ svn propdel license calc/button.c
property 'license' deleted from 'calc/button.c'.
$ svn proplist -v calc/button.c
Properties on 'calc/button.c':
  copyright
    (c) 2006 Red-Bean Software
$
```

Remember those unversioned revision properties? You can modify those, too, using the same **svn** subcommands that we just described. Simply add the `--revprop` command-line parameter and specify the revision whose property you wish to modify. Since revisions are global, you don't need to specify a target path to these property-related commands so long as you are positioned in a working copy of the repository whose revision property you wish to modify. Otherwise, you can simply provide the URL of any path in the repository of interest (including the repository's root URL). For example, you might want to replace the commit log message of an existing revision.[4] If your current working directory is part of a working copy of your repository, you can simply run the **svn propset** command with no target path:

```
$ svn propset svn:log "* button.c: Fix a compiler warning." -r11 --revprop
property 'svn:log' set on repository revision '11'
$
```

But even if you haven't checked out a working copy from that repository, you can still effect the property change by providing the repository's root URL:

```
$ svn propset svn:log "* button.c: Fix a compiler warning." -r11 --revprop \
              http://svn.example.com/repos/project
property 'svn:log' set on repository revision '11'
$
```

Note that the ability to modify these unversioned properties must be explicitly added by the repository administrator (see the section called “Commit Log Message Correction”). That's because the properties aren't versioned, so you run the risk of losing information if you aren't careful with your edits. The repository administrator can set up methods to protect against this loss, and by default, modification of unversioned properties is disabled.

Users should, where possible, use **svn propedit** instead of **svn propset**. While the end result of the commands is identical, the former will allow them to see the current value of the property that they are about to change, which helps them to verify that they are, in fact, making the change they think they are making. This is especially true when modifying unversioned revision properties. Also, it is significantly easier to modify multiline property values in a text editor than at the command line.

Properties and the Subversion Workflow

Now that you are familiar with all of the property-related **svn** subcommands, let's see how property modifications affect the usual Subversion workflow. As we mentioned earlier, file and directory properties are versioned, just like your file contents. As a result, Subversion provides the same opportunities for merging—cleanly or with conflicts—someone else's modifications into your own.

As with file contents, your property changes are local modifications, made permanent only when you commit them to the repository with **svn commit**. Your property changes can be easily unmade, too—the **svn revert** command will restore your files and directories to their unedited states—contents, properties, and all. Also, you can receive interesting information about the state of your file and directory properties by using the **svn status** and **svn diff** commands.

[4] Fixing spelling errors, grammatical gotchas, and “just-plain-wrongness” in commit log messages is perhaps the most common use case for the `--revprop` option.

```
$ svn status calc/button.c
 M      calc/button.c
$ svn diff calc/button.c
Property changes on: calc/button.c
___________________________________________________________________
Added: copyright
## -0,0 +1 ##
+(c) 2006 Red-Bean Software
$
```

Notice how the **status** subcommand displays `M` in the second column instead of the first. That is because we have modified the properties on `calc/button.c`, but not its textual contents. Had we changed both, we would have seen `M` in the first column, too. (We cover **svn status** in the section called “See an overview of your changes”).

Property Conflicts

As with file contents, local property modifications can conflict with changes committed by someone else. If you update your working copy directory and receive property changes on a versioned object that clash with your own, Subversion will report that the object is in a conflicted state.

```
$ svn update calc
Updating 'calc':
M  calc/Makefile.in
Conflict for property 'linecount' discovered on 'calc/button.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (s) show all options: p
 C calc/button.c
Updated to revision 143.
Summary of conflicts:
  Property conflicts: 1
$
```

Subversion will also create, in the same directory as the conflicted object, a file with a `.prej` extension that contains the details of the conflict. You should examine the contents of this file so you can decide how to resolve the conflict. Until the conflict is resolved, you will see a `C` in the second column of **svn status** output for that object, and attempts to commit your local modifications will fail.

```
$ svn status calc
 C      calc/button.c
?       calc/button.c.prej
$ cat calc/button.c.prej
Trying to change property 'linecount' from '1267' to '1301',
but property has been locally changed from '1267' to '1256'.
$
```

To resolve property conflicts, simply ensure that the conflicting properties contain the values that they should, and then use the **svn resolve --accept=working** command to alert Subversion that you have manually resolved the problem.

You might also have noticed the nonstandard way that Subversion currently displays property differences. You can still use **svn diff** and redirect its output to create a usable patch file. The **patch** program will ignore property patches—as a rule, it ignores any noise it can't understand. This does, unfortunately, mean that to fully apply a patch generated by **svn diff** using **patch**, any property modifications will need to be applied by hand.

Subversion 1.7 improves this situation in two ways. First, its nonstandard display of property differences is at least machine-readable—an improvement over the display of properties in versions prior to 1.7. But Subversion 1.7 also introduces the **svn patch** subcommand, designed specifically to handle the additional information which **svn diff**'s output can carry, applying those changes to the Subversion working copy. Of specific relevance to our topic, property differences present in patch files generated by **svn diff** in Subversion 1.7 or better can be automatically applied to a working copy by the **svn patch** command. For more about **svn patch**, see svn patch in Chapter 9, *Subversion Complete Reference*.

There's one exception to how property changes are reported by **svn diff**: changes to Subversion's special `svn:mergeinfo` property—used to track information about merges which have been performed in your repository—are described in a more human-readable fashion. This is quite helpful to the humans who have to read those descriptions. But it also serves to cause patching programs (including **svn patch**) to skip those change descriptions as noise. This might sound like a bug, but it really isn't because this property is intended to be managed solely by the **svn merge** subcommand. For more about merge tracking, see Chapter 4, *Branching and Merging*.

Automatic Property Setting

Properties are a powerful feature of Subversion, acting as key components of many Subversion features discussed elsewhere in this and other chapters—textual diff and merge support, keyword substitution, newline translation, and so on. But to get the full benefit of properties, they must be set on the right files and directories. Unfortunately, that step can be easily forgotten in the routine of things, especially since failing to set a property doesn't usually result in an obvious error (at least compared to, say, failing to add a file to version control). To help your properties get applied to the places that need them, Subversion provides a couple of simple but useful features.

Whenever you introduce a file to version control using the **svn add** or **svn import** commands, Subversion tries to assist by setting some common file properties automatically. First, on operating systems whose filesystems support an execute permission bit, Subversion will automatically set the `svn:executable` property on newly added or imported files whose execute bit is enabled. (See the section called “File Executability” later in this chapter for more about this property.)

Second, Subversion tries to determine the file's MIME type. If you've configured a `mime-types-files` runtime configuration parameter, Subversion will try to find a MIME type mapping in that file for your file's extension. If it finds such a mapping, it will set your file's `svn:mime-type` property to the MIME type it found. If no mapping file is configured, or no mapping for your file's extension could be found, Subversion will fall back to heuristic algorithms to determine the file's MIME type. Depending on how it is built, Subversion 1.7 can make use of file scanning libraries[5] to detect a file's type based on its content. Failing all else, Subversion will employ its own very basic heuristic to determine whether the file contains nontextual content. If so, it automatically sets the `svn:mime-type` property on that file to `application/octet-stream` (the generic “this is a collection of bytes” MIME type). Of course, if Subversion guesses incorrectly, or if you wish to set the `svn:mime-type` property to something more precise—perhaps `image/png` or `application/x-shockwave-flash`—you can always remove or edit that property. (For more on Subversion's use of MIME types, see the section called “File Content Type” later in this chapter.)

UTF-16 is commonly used to encode files whose semantic content is textual in nature, but the encoding itself makes heavy use of bytes which are outside the typical ASCII character byte range. As such, Subversion will tend to classify such files as binary files, much to the chagrin of users who desire line-based differencing and merging, keyword substitution, and other behaviors for those files.

Subversion also provides, via its runtime configuration system (see the section called “Runtime Configuration Area”), a more flexible automatic property setting feature that allows you to create mappings of filename patterns to property names and values. Once again, these mappings affect adds and imports, and can not only override the default MIME type decision made by Subversion dur-

[5]Currently, libmagic is the support library used to accomplish this.

ing those operations, but can also set additional Subversion or custom properties, too. For example, you might create a mapping that says that anytime you add JPEG files—ones whose names match the pattern `*.jpg`—Subversion should automatically set the `svn:mime-type` property on those files to `image/jpeg`. Or perhaps any files that match `*.cpp` should have `svn:eol-style` set to `native`, and `svn:keywords` set to `Id`. Automatic property support is perhaps the handiest property-related tool in the Subversion toolbox. See the section called "Config" for more about configuring that support.

Subversion administrators commonly ask if it is possible to configure, on the server side, a set of property definitions which all connecting clients will automatically consider when operating on working copies checked out from that server. Unfortunately, Subversion doesn't offer this feature. Administrators can use hook scripts to validate that the properties added to and modified on files and directories match the administrator's preferred policies, rejecting commits which are non-compliant in this fashion. (See the section called "Implementing Repository Hooks" for more about hook scripts.) But there's no way to automatically dictate those preferences to Subversion clients beforehand.

File Portability

Fortunately for Subversion users who routinely find themselves on different computers with different operating systems, Subversion's command-line program behaves almost identically on all those systems. If you know how to wield **svn** on one platform, you know how to wield it everywhere.

However, the same is not always true of other general classes of software or of the actual files you keep in Subversion. For example, on a Windows machine, the definition of a "text file" would be similar to that used on a Linux box, but with a key difference—the character sequences used to mark the ends of the lines of those files. There are other differences, too. Unix platforms have (and Subversion supports) symbolic links; Windows does not. Unix platforms use filesystem permission to determine executability; Windows uses filename extensions.

Because Subversion is in no position to unite the whole world in common definitions and implementations of all of these things, the best it can do is to try to help make your life simpler when you need to work with your versioned files and directories on multiple computers and operating systems. This section describes some of the ways Subversion does this.

File Content Type

Subversion joins the ranks of the many applications that recognize and make use of Multipurpose Internet Mail Extensions (MIME) content types. Besides being a general-purpose storage location for a file's content type, the value of the `svn:mime-type` file property determines some behavioral characteristics of Subversion itself.

Identifying File Types

Various programs on most modern operating systems make assumptions about the type and format of the contents of a file by the file's name, specifically its file extension. For example, files whose names end in `.txt` are generally assumed to be human-readable; that is, able to be understood by simple perusal rather than requiring complex processing to decipher. Files whose names end in `.png`, on the other hand, are assumed to be of the Portable Network Graphics type—not human-readable at all, and sensible only when interpreted by software that understands the PNG format and can render the information in that format as a raster image.

Unfortunately, some of those extensions have changed their meanings over time. When personal computers first appeared, a file named `README.DOC` would have almost certainly been a plain-text file, just like today's `.txt` files. But by the mid-1990s, you could almost bet that a file of that name would not be a plain-text file at all, but instead a Microsoft Word document in a proprietary, non-human-readable format. But this change didn't occur overnight—there was certainly a period of confusion for computer users over what exactly they had in hand when they saw a `.DOC` file.[6]

The popularity of computer networking cast still more doubt on the mapping between a file's name and its content. With information being served across networks and generated dynamically by server-side scripts, there was often no real file per se, and therefore no filename. Web servers, for example, needed some other way to tell browsers what they were downloading

[6] You think that was rough? During that same era, WordPerfect also used `.DOC` for their proprietary file format's preferred extension!

> so that the browser could do something intelligent with that information, whether that was to display the data using a program registered to handle that datatype or to prompt the user for where on the client machine to store the downloaded data.
>
> Eventually, a standard emerged for, among other things, describing the contents of a data stream. In 1996, RFC 2045 was published. It was the first of five RFCs describing MIME. It describes the concept of media types and subtypes and recommends a syntax for the representation of those types. Today, MIME media types—or "MIME types"—are used almost universally across email applications, web servers, and other software as the de facto mechanism for clearing up the file content confusion.

For example, one of the benefits that Subversion typically provides is contextual, line-based merging of changes received from the server during an update into your working file. But for files containing nontextual data, there is often no concept of a "line." So, for versioned files whose `svn:mime-type` property is set to a nontextual MIME type (generally, something that doesn't begin with `text/`, though there are exceptions), Subversion does not attempt to perform contextual merges during updates. Instead, any time you have locally modified a binary working copy file that is also being updated, your file is left untouched and Subversion creates two new files. One file has a `.oldrev` extension and contains the BASE revision of the file. The other file has a `.newrev` extension and contains the contents of the updated revision of the file. This behavior is really for the protection of the user against failed attempts at performing contextual merges on files that simply cannot be contextually merged.

Additionally, since the acts of displaying line-based differences and line-based change attribution are, rather obviously, dependent on there being a meaningful definition of "line" for a given file, files with nontextual MIME types will by default trigger errors when used as the targets of **svn diff** and **svn annotate** operations. This can be especially frustrating for users with XML files whose `svn:mime-type` property is set to something such as `application/xml` which is not unambiguously human-readable and as such is treated as nontextual by Subversion. Fortunately, those subcommands offer a `--force` option for forcing Subversion to attempt the operations in spite of the apparent non-human-readability of the files.

The `svn:mime-type` property, when set to a value that does not indicate textual file contents, can cause some unexpected behaviors with respect to other properties. For example, since the idea of line endings (and therefore, line-ending conversion) makes no sense when applied to nontextual files, Subversion will prevent you from setting the `svn:eol-style` property on such files. This is obvious when attempted on a single file target—**svn propset** will error out. But it might not be as clear if you perform a recursive property set, where Subversion will silently skip over files that it deems unsuitable for a given property.

Subversion provides a number of mechanisms by which to automatically set the `svn:mime-type` property on a versioned file. See the section called "Automatic Property Setting" for details.

Also, if the `svn:mime-type` property is set, then the Subversion Apache module will use its value to populate the `Content-type:` HTTP header when responding to GET requests. This gives your web browser a crucial clue about how to display a file when you use it to peruse your Subversion repository's contents.

File Executability

On many operating systems, the ability to execute a file as a command is governed by the presence of an execute permission bit. This bit usually defaults to being disabled, and must be explicitly enabled by the user for each file that needs it. But it would be a monumental hassle to have to remember exactly which files in a freshly checked-out working copy were supposed to have their executable bits toggled on, and then to have to do that toggling. So, Subversion provides the `svn:executable` property as a way to specify that the executable bit for the file on which that property is set should be enabled, and Subversion honors that request when populating working copies with such files.

This property has no effect on filesystems that have no concept of an executable permission bit, such as FAT32 and NTFS.[7] Also, although it has no defined values, Subversion will force its value to `*` when setting this property. Finally, this property is valid only on files, not on directories.

[7] The Windows filesystems use file extensions (such as `.EXE`, `.BAT`, and `.COM`) to denote executable files.

End-of-Line Character Sequences

Unless otherwise noted using a versioned file's `svn:mime-type` property, Subversion assumes the file contains human-readable data. Generally speaking, Subversion uses this knowledge only to determine whether contextual difference reports for that file are possible. Otherwise, to Subversion, bytes are bytes.

This means that by default, Subversion doesn't pay any attention to the type of *end-of-line (EOL) markers* used in your files. Unfortunately, different operating systems have different conventions about which character sequences represent the end of a line of text in a file. For example, the usual line-ending token used by software on the Windows platform is a pair of ASCII control characters—a carriage return (`CR`) followed by a line feed (`LF`). Unix software, however, just uses the `LF` character to denote the end of a line.

Not all of the various tools on these operating systems understand files that contain line endings in a format that differs from the *native line-ending style* of the operating system on which they are running. So, typically, Unix programs treat the `CR` character present in Windows files as a regular character (usually rendered as `^M`), and Windows programs combine all of the lines of a Unix file into one giant line because no carriage return-linefeed (or `CRLF`) character combination was found to denote the ends of the lines.

This sensitivity to foreign EOL markers can be frustrating for folks who share a file across different operating systems. For example, consider a source code file, and developers who edit this file on both Windows and Unix systems. If all the developers always use tools that preserve the line-ending style of the file, no problems occur.

But in practice, many common tools either fail to properly read a file with foreign EOL markers, or convert the file's line endings to the native style when the file is saved. If the former is true for a developer, he has to use an external conversion utility (such as **dos2unix** or its companion, **unix2dos**) to prepare the file for editing. The latter case requires no extra preparation. But both cases result in a file that differs from the original quite literally on every line! Prior to committing his changes, the user has two choices. Either he can use a conversion utility to restore the modified file to the same line-ending style that it was in before his edits were made, or he can simply commit the file—new EOL markers and all.

The result of scenarios like these include wasted time and unnecessary modifications to committed files. Wasted time is painful enough. But when commits change every line in a file, this complicates the job of determining which of those lines were changed in a nontrivial way. Where was that bug really fixed? On what line was a syntax error introduced?

The solution to this problem is the `svn:eol-style` property. When this property is set to a valid value, Subversion uses it to determine what special processing to perform on the file so that the file's line-ending style isn't flip-flopping with every commit that comes from a different operating system. The valid values are:

`native`
: This causes the file to contain the EOL markers that are native to the operating system on which Subversion was run. In other words, if a user on a Windows machine checks out a working copy that contains a file with an `svn:eol-style` property set to `native`, that file will contain `CRLF` EOL markers. A Unix user checking out a working copy that contains the same file will see `LF` EOL markers in his copy of the file.

  Note that Subversion will actually store the file in the repository using normalized `LF` EOL markers regardless of the operating system. This is basically transparent to the user, though.

`CRLF`
: This causes the file to contain `CRLF` sequences for EOL markers, regardless of the operating system in use.

`LF`
: This causes the file to contain `LF` characters for EOL markers, regardless of the operating system in use.

`CR`
: This causes the file to contain `CR` characters for EOL markers, regardless of the operating system in use. This line-ending style is not very common.

Ignoring Unversioned Items

In any given working copy, there is a good chance that alongside all those versioned files and directories are other files and directories that are neither versioned nor intended to be. Text editors litter directories with backup files. Software compilers generate intermediate—or even final—files that you typically wouldn't bother to version. And users themselves drop various other files and directories wherever they see fit, often in version control working copies.

It's ludicrous to expect Subversion working copies to be somehow impervious to this kind of clutter and impurity. In fact, Subversion counts it as a *feature* that its working copies are just typical directories, just like unversioned trees. But these not-to-be-versioned files and directories can cause some annoyance for Subversion users. For example, because the **svn add** and **svn import** commands act recursively by default and don't know which files in a given tree you do and don't wish to version, it's easy to accidentally add stuff to version control that you didn't mean to. And because **svn status** reports, by default, every item of interest in a working copy—including unversioned files and directories—its output can get quite noisy where many of these things exist.

So Subversion provides two ways for telling it which files you would prefer that it simply disregard. One of the ways involves the use of Subversion's runtime configuration system (see the section called “Runtime Configuration Area”), and therefore applies to all the Subversion operations that make use of that runtime configuration—generally those performed on a particular computer or by a particular user of a computer. The other way makes use of Subversion's directory property support and is more tightly bound to the versioned tree itself, and therefore affects everyone who has a working copy of that tree. Both of the mechanisms use *file patterns* (strings of literal and special wildcard characters used to match against filenames) to decide which files to ignore.

The Subversion runtime configuration system provides an option, `global-ignores`, whose value is a whitespace-delimited collection of file patterns. The Subversion client checks these patterns against the names of the files that are candidates for addition to version control, as well as to unversioned files that the **svn status** command notices. If any file's name matches one of the patterns, Subversion will basically act as if the file didn't exist at all. This is really useful for the kinds of files that you almost never want to version, such as editor backup files such as Emacs' `*~` and `.*~` files.

File Patterns in Subversion

File patterns (also called *globs* or *shell wildcard patterns*) are strings of characters that are intended to be matched against filenames, typically for the purpose of quickly selecting some subset of similar files from a larger grouping without having to explicitly name each file. The patterns contain two types of characters: regular characters, which are compared explicitly against potential matches, and special wildcard characters, which are interpreted differently for matching purposes.

There are different types of file pattern syntaxes, but Subversion uses the one most commonly found in Unix systems implemented as the `fnmatch` system function. It supports the following wildcards, described here simply for your convenience:

`?`
: Matches any single character

`*`
: Matches any string of characters, including the empty string

`[`
: Begins a character class definition terminated by `]`, used for matching a subset of characters

You can see this same pattern matching behavior at a Unix shell prompt. The following are some examples of patterns being used for various things:

```
$ ls   ### the book sources
appa-quickstart.xml             ch06-server-configuration.xml
appb-svn-for-cvs-users.xml      ch07-customizing-svn.xml
appc-webdav.xml                 ch08-embedding-svn.xml
```

```
book.xml                          ch09-reference.xml
ch00-preface.xml                  ch10-world-peace-thru-svn.xml
ch01-fundamental-concepts.xml     copyright.xml
ch02-basic-usage.xml              foreword.xml
ch03-advanced-topics.xml          images/
ch04-branching-and-merging.xml    index.xml
ch05-repository-admin.xml         styles.css
$ ls ch*   ### the book chapters
ch00-preface.xml                  ch06-server-configuration.xml
ch01-fundamental-concepts.xml     ch07-customizing-svn.xml
ch02-basic-usage.xml              ch08-embedding-svn.xml
ch03-advanced-topics.xml          ch09-reference.xml
ch04-branching-and-merging.xml    ch10-world-peace-thru-svn.xml
ch05-repository-admin.xml
$ ls ch?0-*   ### the book chapters whose numbers end in zero
ch00-preface.xml  ch10-world-peace-thru-svn.xml
$ ls ch0[3578]-*   ### the book chapters that Mike is responsible for
ch03-advanced-topics.xml   ch07-customizing-svn.xml
ch05-repository-admin.xml  ch08-embedding-svn.xml
$
```

File pattern matching is a bit more complex than what we've described here, but this basic usage level tends to suit the majority of Subversion users.

When found on a versioned directory, the `svn:ignore` property is expected to contain a list of newline-delimited file patterns that Subversion should use to determine ignorable objects in that same directory. These patterns do not override those found in the `global-ignores` runtime configuration option, but are instead appended to that list. And it's worth noting again that, unlike the `global-ignores` option, the patterns found in the `svn:ignore` property apply only to the directory on which that property is set, and not to any of its subdirectories. The `svn:ignore` property is a good way to tell Subversion to ignore files that are likely to be present in every user's working copy of that directory, such as compiler output or—to use an example more appropriate to this book—the HTML, PDF, or PostScript files generated as the result of a conversion of some source DocBook XML files to a more legible output format.

Subversion's support for ignorable file patterns extends only to the one-time process of adding unversioned files and directories to version control. Once an object is under Subversion's control, the ignore pattern mechanisms no longer apply to it. In other words, don't expect Subversion to avoid committing changes you've made to a versioned file simply because that file's name matches an ignore pattern—Subversion *always* notices all of its versioned objects.

Ignore Patterns for CVS Users

The Subversion `svn:ignore` property is very similar in syntax and function to the CVS `.cvsignore` file. In fact, if you are migrating a CVS working copy to Subversion, you can directly migrate the ignore patterns by using the `.cvsignore` file as input file to the **svn propset** command:

```
$ svn propset svn:ignore -F .cvsignore .
property 'svn:ignore' set on '.'
$
```

There are, however, some differences in the ways that CVS and Subversion handle ignore patterns. The two systems use the

> ignore patterns at some different times, and there are slight discrepancies in what the ignore patterns apply to. Also, Subversion does not recognize the use of the `!` pattern as a reset back to having no ignore patterns at all.

The global list of ignore patterns tends to be more a matter of personal taste and ties more closely to a user's particular tool chain than to the details of any particular working copy's needs. So, the rest of this section will focus on the `svn:ignore` property and its uses.

Say you have the following output from **svn status**:

```
$ svn status calc
 M      calc/button.c
?       calc/calculator
?       calc/data.c
?       calc/debug_log
?       calc/debug_log.1
?       calc/debug_log.2.gz
?       calc/debug_log.3.gz
```

In this example, you have made some property modifications to `button.c`, but in your working copy, you also have some unversioned files: the latest `calculator` program that you've compiled from your source code, a source file named `data.c`, and a set of debugging output logfiles. Now, you know that your build system always results in the `calculator` program being generated.[8] And you know that your test suite always leaves those debugging logfiles lying around. These facts are true for all working copies of this project, not just your own. And you know that you aren't interested in seeing those things every time you run **svn status**, and you are pretty sure that nobody else is interested in them either. So you use **`svn propedit svn:ignore calc`** to add some ignore patterns to the `calc` directory.

```
$ svn propget svn:ignore calc
calculator
debug_log*
$
```

After you've added this property, you will now have a local property modification on the `calc` directory. But notice what else is different about your **svn status** output:

```
$ svn status
 M      calc
 M      calc/button.c
?       calc/data.c
```

Now, all that cruft is missing from the output! Your `calculator` compiled program and all those logfiles are still in your working copy; Subversion just isn't constantly reminding you that they are present and unversioned. And now with all the uninteresting noise removed from the display, you are left with more intriguing items—such as that source code file `data.c` that you probably forgot to add to version control.

Of course, this less-verbose report of your working copy status isn't the only one available. If you actually want to see the ignored

[8] Isn't that the whole point of a build system?

files as part of the status report, you can pass the `--no-ignore` option to Subversion:

```
$ svn status --no-ignore
 M       calc
 M       calc/button.c
I        calc/calculator
?        calc/data.c
I        calc/debug_log
I        calc/debug_log.1
I        calc/debug_log.2.gz
I        calc/debug_log.3.gz
```

As mentioned earlier, the list of file patterns to ignore is also used by **svn add** and **svn import**. Both of these operations involve asking Subversion to begin managing some set of files and directories. Rather than force the user to pick and choose which files in a tree she wishes to start versioning, Subversion uses the ignore patterns—both the global and the per-directory lists—to determine which files should not be swept into the version control system as part of a larger recursive addition or import operation. And here again, you can use the `--no-ignore` option to tell Subversion to disregard its ignores list and operate on all the files and directories present.

Even if `svn:ignore` is set, you may run into problems if you use shell wildcards in a command. Shell wildcards are expanded into an explicit list of targets before Subversion operates on them, so running **`svn SUBCOMMAND *`** is just like running **`svn SUBCOMMAND file1 file2 file3 …`**. In the case of the **svn add** command, this has an effect similar to passing the `--no-ignore` option. So instead of using a wildcard, use **`svn add --force .`** to do a bulk scheduling of unversioned things for addition. The explicit target will ensure that the current directory isn't overlooked because of being already under version control, and the `--force` option will cause Subversion to crawl through that directory, adding unversioned files while still honoring the `svn:ignore` property and `global-ignores` runtime configuration variable. Be sure to also provide the `--depth files` option to the **svn add** command if you don't want a fully recursive crawl for things to add.

Keyword Substitution

Subversion has the ability to substitute *keywords*—pieces of useful, dynamic information about a versioned file—into the contents of the file itself. Keywords generally provide information about the last modification made to the file. Because this information changes each time the file changes, and more importantly, just *after* the file changes, it is a hassle for any process except the version control system to keep the data completely up to date. Left to human authors, the information would inevitably grow stale.

For example, say you have a document in which you would like to display the last date on which it was modified. You could burden every author of that document to, just before committing their changes, also tweak the part of the document that describes when it was last changed. But sooner or later, someone would forget to do that. Instead, simply ask Subversion to perform keyword substitution on the `LastChangedDate` keyword. You control where the keyword is inserted into your document by placing a *keyword anchor* at the desired location in the file. This anchor is just a string of text formatted as `$KeywordName$`.

All keywords are case-sensitive where they appear as anchors in files: you must use the correct capitalization for the keyword to be expanded. You should consider the value of the `svn:keywords` property to be case-sensitive, too—certain keyword names will be recognized regardless of case, but this behavior is deprecated.

Subversion defines the list of keywords available for substitution. That list contains the following keywords, some of which have aliases that you can also use:

`Date`
: This keyword describes the last time the file was known to have been changed in the repository, and is of the form `$Date: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $`. It may also be specified as `LastChangedDate`. Un-

like the `Id` keyword, which uses UTC, the `Date` keyword displays dates using the local time zone.

`Revision`
: This keyword describes the last known revision in which this file changed in the repository, and looks something like `$Revision: 144 $`. It may also be specified as `LastChangedRevision` or `Rev`.

`Author`
: This keyword describes the last known user to change this file in the repository, and looks something like `$Author: harry $`. It may also be specified as `LastChangedBy`.

`HeadURL`
: This keyword describes the full URL to the latest version of the file in the repository, and looks something like `$HeadURL: http://svn.example.com/repos/trunk/calc.c $`. It may be abbreviated as `URL`.

`Id`
: This keyword is a compressed combination of the other keywords. Its substitution looks something like `$Id: calc.c 148 2006-07-28 21:30:43Z sally $`, and is interpreted to mean that the file `calc.c` was last changed in revision 148 on the evening of July 28, 2006 by the user `sally`. The date displayed by this keyword is in UTC, unlike that of the `Date` keyword (which uses the local time zone).

`Header`
: This keyword is similar to the `Id` keyword but contains the full URL of the latest revision of the item, identical to `HeadURL`. Its substitution looks something like `$Header: http://svn.example.com/repos/trunk/calc.c 148 2006-07-28 21:30:43Z sally $`.

Several of the preceding descriptions use the phrase “last known” or similar wording. Keep in mind that keyword expansion is a client-side operation, and your client “knows” only about changes that have occurred in the repository when you update your working copy to include those changes. If you never update your working copy, your keywords will never expand to different values even if those versioned files are being changed regularly in the repository.

Simply adding keyword anchor text to your file does nothing special. Subversion will never attempt to perform textual substitutions on your file contents unless explicitly asked to do so. After all, you might be writing a document[9] about how to use keywords, and you don't want Subversion to substitute your beautiful examples of unsubstituted keyword anchors!

To tell Subversion whether to substitute keywords on a particular file, we again turn to the property-related subcommands. The `svn:keywords` property, when set on a versioned file, controls which keywords will be substituted on that file. The value is a space-delimited list of keyword names or aliases.

For example, say you have a versioned file named `weather.txt` that looks like this:

```
Here is the latest report from the front lines.
$LastChangedDate$
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

With no `svn:keywords` property set on that file, Subversion will do nothing special. Now, let's enable substitution of the `LastChangedDate` keyword.

```
$ svn propset svn:keywords "Date Author" weather.txt
property 'svn:keywords' set on 'weather.txt'
$
```

[9]… or maybe even a section of a book …

Now you have made a local property modification on the `weather.txt` file. You will see no changes to the file's contents (unless you made some of your own prior to setting the property). Notice that the file contained a keyword anchor for the `Rev` keyword, yet we did not include that keyword in the property value we set. Subversion will happily ignore requests to substitute keywords that are not present in the file and will not substitute keywords that are not present in the `svn:keywords` property value.

Immediately after you commit this property change, Subversion will update your working file with the new substitute text. Instead of seeing your keyword anchor `$LastChangedDate$`, you'll see its substituted result. That result also contains the name of the keyword and continues to be delimited by the dollar sign (`$`) characters. And as we predicted, the `Rev` keyword was not substituted because we didn't ask for it to be.

Note also that we set the `svn:keywords` property to `Date Author`, yet the keyword anchor used the alias `$LastChangedDate$` and still expanded correctly:

```
Here is the latest report from the front lines.
$LastChangedDate: 2006-07-22 21:42:37 -0700 (Sat, 22 Jul 2006) $
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

If someone else now commits a change to `weather.txt`, your copy of that file will continue to display the same substituted keyword value as before—until you update your working copy. At that time, the keywords in your `weather.txt` file will be re-substituted with information that reflects the most recent known commit to that file.

Where's $GlobalRev$?

New users are often confused by how the `$Rev$` keyword works. Since the repository has a single, globally increasing revision number, many people assume that it is this number that is reflected by the `$Rev$` keyword's value. But `$Rev$` expands to show the last revision in which the file *changed*, not the last revision to which it was updated. Understanding this clears the confusion, but frustration often remains—without the support of a Subversion keyword to do so, how can you automatically get the global revision number into your files?

To do this, you need external processing. Subversion ships with a tool called **svnversion**, which was designed for just this purpose. It crawls your working copy and generates as output the revision(s) it finds. You can use this program, plus some additional tooling, to embed that revision information into your files. For more information on **svnversion**, see the section called “svnversion—Subversion Working Copy Version Info” in Chapter 9, *Subversion Complete Reference*.

You can also instruct Subversion to maintain a fixed length (in terms of the number of bytes consumed) for the substituted keyword. By using a double colon (`::`) after the keyword name, followed by a number of space characters, you define that fixed width. When Subversion goes to substitute your keyword for the keyword and its value, it will essentially replace only those space characters, leaving the overall width of the keyword field unchanged. If the substituted value is shorter than the defined field width, there will be extra padding characters (spaces) at the end of the substituted field; if it is too long, it is truncated with a special hash (#) character just before the final dollar sign terminator.

For example, say you have a document in which you have some section of tabular data reflecting the document's Subversion keywords. Using the original Subversion keyword substitution syntax, your file might look something like:

```
$Rev$:     Revision of last commit
$Author$:  Author of last commit
$Date$:    Date of last commit
```

Now, that looks nice and tabular at the start of things. But when you then commit that file (with keyword substitution enabled, of course), you see:

```
$Rev: 12 $:     Revision of last commit
$Author: harry $:  Author of last commit
$Date: 2006-03-15 02:33:03 -0500 (Wed, 15 Mar 2006) $:    Date of last commit
```

The result is not so beautiful. And you might be tempted to then adjust the file after the substitution so that it again looks tabular. But that holds only as long as the keyword values are the same width. If the last committed revision rolls into a new place value (say, from 99 to 100), or if another person with a longer username commits the file, stuff gets all crooked again. However, if you are using Subversion 1.2 or later, you can use the new fixed-length keyword syntax and define some field widths that seem sane, so your file might look like this:

```
$Rev::                $:  Revision of last commit
$Author::             $:  Author of last commit
$Date::               $:  Date of last commit
```

You commit this change to your file. This time, Subversion notices the new fixed-length keyword syntax and maintains the width of the fields as defined by the padding you placed between the double colon and the trailing dollar sign. After substitution, the width of the fields is completely unchanged—the short values for `Rev` and `Author` are padded with spaces, and the long `Date` field is truncated by a hash character:

```
$Rev:: 13             $:  Revision of last commit
$Author:: harry       $:  Author of last commit
$Date:: 2006-03-15 0#$:  Date of last commit
```

The use of fixed-length keywords is especially handy when performing substitutions into complex file formats that themselves use fixed-length fields for data, or for which the stored size of a given data field is overbearingly difficult to modify from outside the format's native application. Of course, where binary file formats are concerned, you must always take great care that any keyword substitution you introduce—fixed-length or otherwise—does not violate the integrity of that format. While it might sound easy enough, this can be an astonishingly difficult task for most of the popular binary file formats in use today, and *not* something to be undertaken by the faint of heart!

Be aware that because the width of a keyword field is measured in bytes, the potential for corruption of multibyte values exists. For example, a username that contains some multibyte UTF-8 characters might suffer truncation in the middle of the string of bytes that make up one of those characters. The result will be a mere truncation when viewed at the byte level, but will likely appear as a string with an incorrect or garbled final character when viewed as UTF-8 text. It is conceivable that certain applications, when asked to load the file, would notice the broken UTF-8 text and deem the entire file corrupt, refusing to operate on the file altogether. So, when limiting keywords to a fixed size, choose a size that allows for this type of byte-wise expansion.

Sparse Directories

By default, most Subversion operations on directories act in a recursive manner. For example, **svn checkout** creates a working copy with every file and directory in the specified area of the repository, descending recursively through the repository tree until

the entire structure is copied to your local disk. Subversion 1.5 introduces a feature called *sparse directories* (or *shallow checkouts*) that allows you to easily check out a working copy—or a portion of a working copy—more shallowly than full recursion, with the freedom to bring in previously ignored files and subdirectories at a later time.

For example, say we have a repository with a tree of files and directories with names of the members of a human family with pets. (It's an odd example, to be sure, but bear with us.) A regular **svn checkout** operation will give us a working copy of the whole tree:

```
$ svn checkout file:///var/svn/repos mom
A    mom/son
A    mom/son/grandson
A    mom/daughter
A    mom/daughter/granddaughter1
A    mom/daughter/granddaughter1/bunny1.txt
A    mom/daughter/granddaughter1/bunny2.txt
A    mom/daughter/granddaughter2
A    mom/daughter/fishie.txt
A    mom/kitty1.txt
A    mom/doggie1.txt
Checked out revision 1.
$
```

Now, let's check out the same tree again, but this time we'll ask Subversion to give us only the topmost directory with none of its children at all:

```
$ svn checkout file:///var/svn/repos mom-empty --depth empty
Checked out revision 1
$
```

Notice that we added to our original **svn checkout** command line a new `--depth` option. This option is present on many of Subversion's subcommands and is similar to the `--non-recursive` (`-N`) and `--recursive` (`-R`) options. In fact, it combines, improves upon, supercedes, and ultimately obsoletes these two older options. For starters, it expands the supported degrees of depth specification available to users, adding some previously unsupported (or inconsistently supported) depths. Here are the depth values that you can request for a given Subversion operation:

`--depth empty`
: Include only the immediate target of the operation, not any of its file or directory children.

`--depth files`
: Include the immediate target of the operation and any of its immediate file children.

`--depth immediates`
: Include the immediate target of the operation and any of its immediate file or directory children. The directory children will themselves be empty.

`--depth infinity`
: Include the immediate target, its file and directory children, its children's children, and so on to full recursion.

Of course, merely combining two existing options into one hardly constitutes a new feature worthy of a whole section in our book. Fortunately, there is more to this story. This idea of depth extends not just to the operations you perform with your Subversion client, but also as a description of a working copy citizen's *ambient depth*, which is the depth persistently recorded by the working copy for that item. Its key strength is this very persistence—the fact that it is *sticky*. The working copy remembers the depth you've selected for each item in it until you later change that depth selection; by default, Subversion commands operate on the working

copy citizens present, regardless of their selected depth settings.

You can check the recorded ambient depth of a working copy using the **svn info** command. If the ambient depth is anything other than infinite recursion, **svn info** will display a line describing that depth value:

```
$ svn info mom-immediates | grep "^Depth:"
Depth: immediates
$
```

Our previous examples demonstrated checkouts of infinite depth (the default for **svn checkout**) and empty depth. Let's look now at examples of the other depth values:

```
$ svn checkout file:///var/svn/repos mom-files --depth files
A    mom-files/kitty1.txt
A    mom-files/doggie1.txt
Checked out revision 1.
$ svn checkout file:///var/svn/repos mom-immediates --depth immediates
A    mom-immediates/son
A    mom-immediates/daughter
A    mom-immediates/kitty1.txt
A    mom-immediates/doggie1.txt
Checked out revision 1.
$
```

As described, each of these depths is something more than only the target, but something less than full recursion.

We've used **svn checkout** as an example here, but you'll find the `--depth` option present on many other Subversion commands, too. In those other commands, depth specification is a way to limit the scope of an operation to some depth, much like the way the older `--non-recursive` (`-N`) and `--recursive` (`-R`) options behave. This means that when operating on a working copy of some depth, while requesting an operation of a shallower depth, the operation is limited to that shallower depth. In fact, we can make an even more general statement: given a working copy of any arbitrary—even mixed—ambient depth, and a Subversion command with some requested operational depth, the command will maintain the ambient depth of the working copy members while still limiting the scope of the operation to the requested (or default) operational depth.

In addition to the `--depth` option, the **svn update** and **svn switch** subcommands also accept a second depth-related option: `--set-depth`. It is with this option that you can change the sticky depth of a working copy item. Watch what happens as we take our empty-depth checkout and gradually telescope it deeper using **svn update --set-depth *NEW-DEPTH TARGET***:

```
$ svn update --set-depth files mom-empty
Updating 'mom-empty':
A    mom-empty/kittie1.txt
A    mom-empty/doggie1.txt
Updated to revision 1.
$ svn update --set-depth immediates mom-empty
Updating 'mom-empty':
A    mom-empty/son
A    mom-empty/daughter
Updated to revision 1.
$ svn update --set-depth infinity mom-empty
Updating 'mom-empty':
A    mom-empty/son/grandson
```

```
A    mom-empty/daughter/granddaughter1
A    mom-empty/daughter/granddaughter1/bunny1.txt
A    mom-empty/daughter/granddaughter1/bunny2.txt
A    mom-empty/daughter/granddaughter2
A    mom-empty/daughter/fishie1.txt
Updated to revision 1.
$
```

As we gradually increased our depth selection, the repository gave us more pieces of our tree.

In our example, we operated only on the root of our working copy, changing its ambient depth value. But we can independently change the ambient depth value of *any* subdirectory inside the working copy, too. Careful use of this ability allows us to flesh out only certain portions of the working copy tree, leaving other portions absent altogether (hence the “sparse” bit of the feature's name). Here's an example of how we might build out a portion of one branch of our family's tree, enable full recursion on another branch, and keep still other pieces pruned (absent from disk).

```
$ rm -rf mom-empty
$ svn checkout file:///var/svn/repos mom-empty --depth empty
Checked out revision 1.
$ svn update --set-depth empty mom-empty/son
Updating 'mom-empty/son':
A    mom-empty/son
Updated to revision 1.
$ svn update --set-depth empty mom-empty/daughter
Updating 'mom-empty/daughter':
A    mom-empty/daughter
Updated to revision 1.
$ svn update --set-depth infinity mom-empty/daughter/granddaughter1
Updating 'mom-empty/daughter/granddaughter1':
A    mom-empty/daughter/granddaughter1
A    mom-empty/daughter/granddaughter1/bunny1.txt
A    mom-empty/daughter/granddaughter1/bunny2.txt
Updated to revision 1.
$
```

Fortunately, having a complex collection of ambient depths in a single working copy doesn't complicate the way you interact with that working copy. You can still make, revert, display, and commit local modifications in your working copy without providing any new options (including `--depth` and `--set-depth`) to the relevant subcommands. Even **svn update** works as it does elsewhere when no specific depth is provided—it updates the working copy targets that are present while honoring their sticky depths.

You might at this point be wondering, “So what? When would I use this?” One scenario where this feature finds utility is tied to a particular repository layout, specifically where you have many related or codependent projects or software modules living as siblings in a single repository location (`trunk/project1`, `trunk/project2`, `trunk/project3`, etc.). In such scenarios, it might be the case that you personally care about only a handful of those projects—maybe some primary project and a few other modules on which it depends. You can check out individual working copies of all of these things, but those working copies are disjoint and, as a result, it can be cumbersome to perform operations across several or all of them at the same time. The alternative is to use the sparse directories feature, building out a single working copy that contains only the modules you care about. You'd start with an empty-depth checkout of the common parent directory of the projects, and then update with infinite depth only the items you wish to have, like we demonstrated in the previous example. Think of it like an opt-in system for working copy citizens.

The original (Subversion 1.5) implementation of shallow checkouts was good, but didn't support de-telescoping of working copy items. Subversion 1.6 remedied this problem. For example, running **`svn update --set-depth empty`** in an infinite-depth working copy will discard everything but the topmost directory.[10] Subversion 1.6 also introduced another supported value for the `-`

`-set-depth` option: `exclude`. Using `--set-depth exclude` with **svn update** will cause the update target to be removed from the working copy entirely—a directory target won't even be left present-but-empty. This is especially handy when there are more things that you'd like to keep in a working copy than things you'd like to *not* keep.

Consider a directory with hundreds of subdirectories, one of which you would like to omit from your working copy. Using an "additive" approach to sparse directories, you might check out the directory with an empty depth, then explicitly telescope (using **`svn update --set-depth infinity`**) each and every subdirectory of the directory except the one you don't care about.

```
$ svn checkout http://svn.example.com/repos/many-dirs --depth empty
…
$ svn update --set-depth infinity many-dirs/wanted-dir-1
…
$ svn update --set-depth infinity many-dirs/wanted-dir-2
…
$ svn update --set-depth infinity many-dirs/wanted-dir-3
…
### and so on, and so on, ...
```

This could be quite tedious, especially since you don't even have stubs of these directories in your working copy to deal with. Such a working copy would also have another characteristic that you might not expect or desire: if someone else creates any new subdirectories in this top-level directory, you won't receive those when you update your working copy.

Beginning with Subversion 1.6, you can take a different approach. First, check out the directory in full. Then run **`svn update --set-depth exclude`** on the one subdirectory you don't care about.

```
$ svn checkout http://svn.example.com/repos/many-dirs
…
$ svn update --set-depth exclude many-dirs/unwanted-dir
D         many-dirs/unwanted-dir
$
```

This approach leaves your working copy with the same stuff as in the first approach, but any new subdirectories which appear in the top-level directory would also show up when you update your working copy. The downside of this approach is that you have to actually check out that whole subdirectory that you don't even want just so you can tell Subversion that you don't want it. This might not even be possible if that subdirectory is too large to fit on your disk (which might, after all, be the very reason you don't want it in your working copy).

While the functionality for excluding an existing item from a working copy was hung off of the **svn update** command, you might have noticed that the output from **`svn update --set-depth exclude`** differs from that of a normal update operation. This output betrays the fact that, under the hood, exclusion is a completely client-side operation, very much unlike a typical update.

In such a situation, you might consider a compromise approach. First, check out the top-level directory with `--depth immediates`. Then, exclude the directory you don't want using **`svn update --set-depth exclude`**. Finally, telescope all the items that remain to infinite depth, which should be fairly easy to do because they are all addressable in your shell.

```
$ svn checkout http://svn.example.com/repos/many-dirs --depth immediates
…
```

[10]Safely, of course. As in other situations, Subversion will leave on disk any files you've modified or which aren't versioned.

```
$ svn update --set-depth exclude many-dirs/unwanted-dir
D         many-dirs/unwanted-dir
$ svn update --set-depth infinity many-dirs/*
…
$
```

Once again, your working copy will have the same stuff as in the previous two scenarios. But now, any time a new file or subdirectory is committed to the top-level directory, you'll receive it—at an empty depth—when you update your working copy. You can now decide what to do with such newly appearing working copy items: expand them into infinite depth, or exclude them altogether.

Locking

Subversion's copy-modify-merge version control model lives and dies on its data merging algorithms—specifically on how well those algorithms perform when trying to resolve conflicts caused by multiple users modifying the same file concurrently. Subversion itself provides only one such algorithm: a three-way differencing algorithm that is smart enough to handle data at a granularity of a single line of text. Subversion also allows you to supplement its content merge processing with external differencing utilities (as described in the section called “External diff3” and the section called “External merge”), some of which may do an even better job, perhaps providing granularity of a word or a single character of text. But common among those algorithms is that they generally work only on text files. The landscape starts to look pretty grim when you start talking about content merges of nontextual file formats. And when you can't find a tool that can handle that type of merging, you begin to run into problems with the copy-modify-merge model.

Let's look at a real-life example of where this model runs aground. Harry and Sally are both graphic designers working on the same project, a bit of marketing collateral for an automobile mechanic. Central to the design of a particular poster is an image of a car in need of some bodywork, stored in a file using the PNG image format. The poster's layout is almost finished, and both Harry and Sally are pleased with the particular photo they chose for their damaged car—a baby blue 1967 Ford Mustang with an unfortunate bit of crumpling on the left front fender.

Now, as is common in graphic design work, there's a change in plans, which causes the car's color to be a concern. So Sally updates her working copy to `HEAD`, fires up her photo-editing software, and sets about tweaking the image so that the car is now cherry red. Meanwhile, Harry, feeling particularly inspired that day, decides that the image would have greater impact if the car also appears to have suffered greater impact. He, too, updates to `HEAD`, and then draws some cracks on the vehicle's windshield. He manages to finish his work before Sally finishes hers, and after admiring the fruits of his undeniable talent, he commits the modified image. Shortly thereafter, Sally is finished with the car's new finish and tries to commit her changes. But, as expected, Subversion fails the commit, informing Sally that her version of the image is now out of date.

Here's where the difficulty sets in. If Harry and Sally were making changes to a text file, Sally would simply update her working copy, receiving Harry's changes in the process. In the worst possible case, they would have modified the same region of the file, and Sally would have to work out by hand the proper resolution to the conflict. But these aren't text files—they are binary images. And while it's a simple matter to describe what one would expect the results of this content merge to be, there is precious little chance that any software exists that is smart enough to examine the common baseline image that each of these graphic artists worked against, the changes that Harry made, and the changes that Sally made, and then spit out an image of a busted-up red Mustang with a cracked windshield!

Of course, things would have gone more smoothly if Harry and Sally had serialized their modifications to the image—if, say, Harry had waited to draw his windshield cracks on Sally's now-red car, or if Sally had tweaked the color of a car whose windshield was already cracked. As is discussed in the section called “The copy-modify-merge solution”, most of these types of problems go away entirely where perfect communication between Harry and Sally exists.[11] But as one's version control system is, in fact, one form of communication, it follows that having that software facilitate the serialization of nonparallelizable editing efforts is no bad thing. This is where Subversion's implementation of the lock-modify-unlock model steps into the spotlight. This is where we talk about Subversion's *locking* feature, which is similar to the “reserved checkouts” mechanisms of other version control systems.

[11]Communication wouldn't have been such bad medicine for Harry and Sally's Hollywood namesakes, either, for that matter.

Subversion's locking feature exists ultimately to minimize wasted time and effort. By allowing a user to programmatically claim the exclusive right to change a file in the repository, that user can be reasonably confident that any energy he invests on unmergeable changes won't be wasted—his commit of those changes will succeed. Also, because Subversion communicates to other users that serialization is in effect for a particular versioned object, those users can reasonably expect that the object is about to be changed by someone else. They, too, can then avoid wasting their time and energy on unmergeable changes that won't be committable due to eventual out-of-dateness.

When referring to Subversion's locking feature, one is actually talking about a fairly diverse collection of behaviors, which include the ability to lock a versioned file[12] (claiming the exclusive right to modify the file), to unlock that file (yielding that exclusive right to modify), to see reports about which files are locked and by whom, to annotate files for which locking before editing is strongly advised, and so on. In this section, we'll cover all of these facets of the larger locking feature.

The Three Meanings of "Lock"

In this section, and almost everywhere in this book, the words "lock" and "locking" describe a mechanism for mutual exclusion between users to avoid clashing commits. Unfortunately, there are two other sorts of "lock" with which Subversion, and therefore this book, sometimes needs to be concerned.

The second is *working copy locks*, used internally by Subversion to prevent clashes between multiple Subversion clients operating on the same working copy. This is the sort of lock indicated by an `L` in the third column of **svn status** output, and removed by the **svn cleanup** command, as described in the section called "Sometimes You Just Need to Clean Up".

Third, there are *database locks*, used internally by the Berkeley DB backend to prevent clashes between multiple programs trying to access the database. This is the sort of lock whose unwanted persistence after an error can cause a repository to be "wedged," as described in the section called "Berkeley DB Recovery".

You can generally forget about these other kinds of locks until something goes wrong that requires you to care about them. In this book, "lock" means the first sort unless the contrary is either clear from context or explicitly stated.

Creating Locks

In the Subversion repository, a *lock* is a piece of metadata that grants exclusive access to one user to change a file. This user is said to be the *lock owner*. Each lock also has a unique identifier, typically a long string of characters, known as the *lock token*. The repository manages locks, ultimately handling their creation, enforcement, and removal. If any commit transaction attempts to modify or delete a locked file (or delete one of the parent directories of the file), the repository will demand two pieces of information—that the client performing the commit be authenticated as the lock owner, and that the lock token has been provided as part of the commit process as a form of proof that the client knows which lock it is using.

To demonstrate lock creation, let's refer back to our example of multiple graphic designers working on the same binary image files. Harry has decided to change a JPEG image. To prevent other people from committing changes to the file while he is modifying it (as well as alerting them that he is about to change it), he locks the file in the repository using the **svn lock** command.

```
$ svn lock banana.jpg -m "Editing file for tomorrow's release."
'banana.jpg' locked by user 'harry'.
$
```

The preceding example demonstrates a number of new things. First, notice that Harry passed the `--message` (`-m`) option to **svn lock**. Similar to **svn commit**, the **svn lock** command can take comments—via either `--message` (`-m`) or `--file` (`-F`)—to describe the reason for locking the file. Unlike **svn commit**, however, **svn lock** will not demand a message by launching your preferred text editor. Lock comments are optional, but still recommended to aid communication.

[12] Subversion does not currently allow locks on directories.

Second, the lock attempt succeeded. This means that the file wasn't already locked, and that Harry had the latest version of the file. If Harry's working copy of the file had been out of date, the repository would have rejected the request, forcing Harry to **svn update** and reattempt the locking command. The locking command would also have failed if the file had already been locked by someone else.

As you can see, the **svn lock** command prints confirmation of the successful lock. At this point, the fact that the file is locked becomes apparent in the output of the **svn status** and **svn info** reporting subcommands.

```
$ svn status
     K  banana.jpg

$ svn info banana.jpg
Path: banana.jpg
Name: banana.jpg
Working Copy Root Path: /home/harry/project
URL: http://svn.example.com/repos/project/banana.jpg
Repository Root: http://svn.example.com/repos/project
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 2198
Node Kind: file
Schedule: normal
Last Changed Author: frank
Last Changed Rev: 1950
Last Changed Date: 2006-03-15 12:43:04 -0600 (Wed, 15 Mar 2006)
Text Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Properties Last Updated: 2006-06-08 19:23:07 -0500 (Thu, 08 Jun 2006)
Checksum: 3b110d3b10638f5d1f4fe0f436a5a2a5
Lock Token: opaquelocktoken:0c0f600b-88f9-0310-9e48-355b44d4a58e
Lock Owner: harry
Lock Created: 2006-06-14 17:20:31 -0500 (Wed, 14 Jun 2006)
Lock Comment (1 line):
Editing file for tomorrow's release.

$
```

The fact that the **svn info** command, which does not contact the repository when run against working copy paths, can display the lock token reveals an important piece of information about those tokens: they are cached in the working copy. The presence of the lock token is critical. It gives the working copy authorization to make use of the lock later on. Also, the **svn status** command shows a `K` next to the file (short for locKed), indicating that the lock token is present.

> **Regarding Lock Tokens**
>
> A lock token isn't an authentication token, so much as an *authorization* token. The token isn't a protected secret. In fact, a lock's unique token is discoverable by anyone who runs **`svn info URL`**. A lock token is special only when it lives inside a working copy. It's proof that the lock was created in that particular working copy, and not somewhere else by some other client. Merely authenticating as the lock owner isn't enough to prevent accidents.
>
> For example, suppose you lock a file using a computer at your office, but leave work for the day before you finish your changes to that file. It should not be possible to accidentally commit changes to that same file from your home computer later that evening simply because you've authenticated as the lock's owner. In other words, the lock token prevents one piece of Subversion-related software from undermining the work of another. (In our example, if you really need to change the file from an alternative working copy, you would need to *break* the lock and relock the file.)

Now that Harry has locked `banana.jpg`, Sally is unable to change or delete that file:

```
$ svn delete banana.jpg
D         banana.jpg
$ svn commit -m "Delete useless file."
Deleting       banana.jpg
svn: E175002: Commit failed (details follow):
svn: E175002: Server sent unexpected return value (423 Locked) in response to
DELETE request for '/repos/project/!svn/wrk/64bad3a9-96f9-0310-818a-df4224ddc
35d/banana.jpg'
$
```

But Harry, after touching up the banana's shade of yellow, is able to commit his changes to the file. That's because he authenticates as the lock owner and also because his working copy holds the correct lock token:

```
$ svn status
M    K  banana.jpg
$ svn commit -m "Make banana more yellow"
Sending        banana.jpg
Transmitting file data .
Committed revision 2201.
$ svn status
$
```

Notice that after the commit is finished, **svn status** shows that the lock token is no longer present in the working copy. This is the standard behavior of **svn commit**—it searches the working copy (or list of targets, if you provide such a list) for local modifications and sends all the lock tokens it encounters during this walk to the server as part of the commit transaction. After the commit completes successfully, all of the repository locks that were mentioned are released—*even on files that weren't committed*. This is meant to discourage users from being sloppy about locking or from holding locks for too long. If Harry haphazardly locks 30 files in a directory named `images` because he's unsure of which files he needs to change, yet changes only four of those files, when he runs **`svn commit images`**, the process will still release all 30 locks.

This behavior of automatically releasing locks can be overridden with the `--no-unlock` option to **svn commit**. This is best used for those times when you want to commit changes, but still plan to make more changes and thus need to retain existing locks. You can also make this your default behavior by setting the `no-unlock` runtime configuration option (see the section called “Runtime Configuration Area”).

Of course, locking a file doesn't oblige one to commit a change to it. The lock can be released at any time with a simple **svn unlock** command:

```
$ svn unlock banana.c
'banana.c' unlocked.
```

Discovering Locks

When a commit fails due to someone else's locks, it's fairly easy to learn about them. The easiest way is to run **`svn status -u`**:

```
$ svn status -u
M               23   bar.c
```

```
M    O          32   raisin.jpg
        *       72   foo.h
Status against revision:     105
$
```

In this example, Sally can see not only that her copy of `foo.h` is out of date, but also that one of the two modified files she plans to commit is locked in the repository. The `O` symbol stands for “Other,” meaning that a lock exists on the file and was created by somebody else. If she were to attempt a commit, the lock on `raisin.jpg` would prevent it. Sally is left wondering who made the lock, when, and why. Once again, **svn info** has the answers:

```
$ svn info ^/raisin.jpg
Path: raisin.jpg
Name: raisin.jpg
URL: http://svn.example.com/repos/project/raisin.jpg
Repository Root: http://svn.example.com/repos/project
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 105
Node Kind: file
Last Changed Author: sally
Last Changed Rev: 32
Last Changed Date: 2006-01-25 12:43:04 -0600 (Sun, 25 Jan 2006)
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Lock Comment (1 line):
Need to make a quick tweak to this image.
$
```

Just as you can use **svn info** to examine objects in the working copy, you can also use it to examine objects in the repository. If the main argument to **svn info** is a working copy path, then all of the working copy's cached information is displayed; any mention of a lock means that the working copy is holding a lock token (if a file is locked by another user or in another working copy, **svn info** on a working copy path will show no lock information at all). If the main argument to **svn info** is a URL, the information reflects the latest version of an object in the repository, and any mention of a lock describes the current lock on the object.

So in this particular example, Sally can see that Harry locked the file on February 16 to “make a quick tweak.” It being June, she suspects that he probably forgot all about the lock. She might phone Harry to complain and ask him to release the lock. If he's unavailable, she might try to forcibly break the lock herself or ask an administrator to do so.

Breaking and Stealing Locks

A repository lock isn't sacred—in Subversion's default configuration state, locks can be released not only by the person who created them, but by anyone. When somebody other than the original lock creator destroys a lock, we refer to this as *breaking the lock*.

From the administrator's chair, it's simple to break locks. The **svnlook** and **svnadmin** programs have the ability to display and remove locks directly from the repository. (For more information about these tools, see the section called “An Administrator's Toolkit”.)

```
$ svnadmin lslocks /var/svn/repos
Path: /project2/images/banana.jpg
UUID Token: opaquelocktoken:c32b4d88-e8fb-2310-abb3-153ff1236923
Owner: frank
```

```
Created: 2006-06-15 13:29:18 -0500 (Thu, 15 Jun 2006)
Expires:
Comment (1 line):
Still improving the yellow color.

Path: /project/raisin.jpg
UUID Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Owner: harry
Created: 2006-02-16 13:29:18 -0500 (Thu, 16 Feb 2006)
Expires:
Comment (1 line):
Need to make a quick tweak to this image.

$ svnadmin rmlocks /var/svn/repos /project/raisin.jpg
Removed lock on '/project/raisin.jpg'.
$
```

The more interesting option is to allow users to break each other's locks over the network. To do this, Sally simply needs to pass the `--force` to the **svn unlock** command:

```
$ svn status -u
M               23   bar.c
M    O          32   raisin.jpg
        *       72   foo.h
Status against revision:     105
$ svn unlock raisin.jpg
svn: E195013: 'raisin.jpg' is not locked in this working copy
$ svn info raisin.jpg | grep URL
URL: http://svn.example.com/repos/project/raisin.jpg
$ svn unlock http://svn.example.com/repos/project/raisin.jpg
svn: warning: W160039: Unlock failed on 'raisin.jpg' (403 Forbidden)
$ svn unlock --force http://svn.example.com/repos/project/raisin.jpg
'raisin.jpg' unlocked.
$
```

Now, Sally's initial attempt to unlock failed because she ran **svn unlock** directly on her working copy of the file, and no lock token was present. To remove the lock directly from the repository, she needs to pass a URL to **svn unlock**. Her first attempt to unlock the URL fails, because she can't authenticate as the lock owner (nor does she have the lock token). But when she passes `--force`, the authentication and authorization requirements are ignored, and the remote lock is broken.

Simply breaking a lock may not be enough. In the running example, Sally may not only want to break Harry's long-forgotten lock, but relock the file for her own use. She can accomplish this by using **svn unlock** with `--force` and then **svn lock** back-to-back, but there's a small chance that somebody else might lock the file between the two commands. The simpler thing to do is to *steal* the lock, which involves breaking and relocking the file all in one atomic step. To do this, Sally passes the `--force` option to **svn lock**:

```
$ svn lock raisin.jpg
svn: warning: W160035: Path '/project/raisin.jpg' is already locked by user 'h
arry' in filesystem '/var/svn/repos/db'
$ svn lock --force raisin.jpg
'raisin.jpg' locked by user 'sally'.
$
```

In any case, whether the lock is broken or stolen, Harry may be in for a surprise. Harry's working copy still contains the original lock token, but that lock no longer exists. The lock token is said to be *defunct*. The lock represented by the lock token has either been broken (no longer in the repository) or stolen (replaced with a different lock). Either way, Harry can see this by asking **svn status** to contact the repository:

```
$ svn status
     K  raisin.jpg
$ svn status -u
     B          32   raisin.jpg
Status against revision:     105
$ svn update
Updating '.':
  B  raisin.jpg
Updated to revision 105.
$ svn status
$
```

If the repository lock was broken, then **`svn status --show-updates`** (`-u`) displays a `B` (Broken) symbol next to the file. If a new lock exists in place of the old one, then a `T` (sTolen) symbol is shown. Finally, **svn update** notices any defunct lock tokens and removes them from the working copy.

Locking Policies

Different systems have different notions of how strict a lock should be. Some folks argue that locks must be strictly enforced at all costs, releasable only by the original creator or administrator. They argue that if anyone can break a lock, chaos runs rampant and the whole point of locking is defeated. The other side argues that locks are first and foremost a communication tool. If users are constantly breaking each other's locks, it represents a cultural failure within the team and the problem falls outside the scope of software enforcement.

Subversion defaults to the “softer” approach, but still allows administrators to create stricter enforcement policies through the use of hook scripts. In particular, the `pre-lock` and `pre-unlock` hooks allow administrators to decide when lock creation and lock releases are allowed to happen. Depending on whether a lock already exists, these two hooks can decide whether to allow a certain user to break or steal a lock. The `post-lock` and `post-unlock` hooks are also available, and can be used to send email after locking actions. To learn more about repository hooks, see the section called “Implementing Repository Hooks”.

Lock Communication

We've seen how **svn lock** and **svn unlock** can be used to create, release, break, and steal locks. This satisfies the goal of serializing commit access to a file. But what about the larger problem of preventing wasted time?

For example, suppose Harry locks an image file and then begins editing it. Meanwhile, miles away, Sally wants to do the same thing. She doesn't think to run **`svn status -u`**, so she has no idea that Harry has already locked the file. She spends hours editing the file, and when she tries to commit her change, she discovers that either the file is locked or that she's out of date. Regardless, her changes aren't mergeable with Harry's. One of these two people has to throw away his or her work, and a lot of time has been wasted.

Subversion's solution to this problem is to provide a mechanism to remind users that a file ought to be locked *before* the editing begins. The mechanism is a special property: `svn:needs-lock`. If that property is attached to a file (regardless of its value, which is irrelevant), Subversion will try to use filesystem-level permissions to make the file read-only—unless, of course, the user has explicitly locked the file. When a lock token is present (as a result of using **svn lock**), the file becomes read/write. When the lock is released, the file becomes read-only again.

The theory, then, is that if the image file has this property attached, Sally would immediately notice something is strange when she opens the file for editing: many applications alert users immediately when a read-only file is opened for editing, and nearly all would prevent her from saving changes to the file. This reminds her to lock the file before editing, whereby she discovers the preexisting lock:

```
$ /usr/local/bin/gimp raisin.jpg
gimp: error: file is read-only!
$ ls -l raisin.jpg
-r--r--r--   1 sally   sally   215589 Jun  8 19:23 raisin.jpg
$ svn lock raisin.jpg
svn: warning: W160035: Path '/project/raisin.jpg' is already locked by user 'h
arry' in filesystem '/var/svn/repos/db'
$ svn info http://svn.example.com/repos/project/raisin.jpg | grep Lock
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2006-06-08 07:29:18 -0500 (Thu, 08 June 2006)
Lock Comment (1 line):
Making some tweaks.  Locking for the next two hours.
$
```

Users and administrators alike are encouraged to attach the `svn:needs-lock` property to any file that cannot be contextually merged. This is the primary technique for encouraging good locking habits and preventing wasted effort.

Note that this property is a communication tool that works independently from the locking system. In other words, any file can be locked, whether or not this property is present. And conversely, the presence of this property doesn't make the repository require a lock when committing.

Unfortunately, the system isn't flawless. It's possible that even when a file has the property, the read-only reminder won't always work. Sometimes applications misbehave and “hijack” the read-only file, silently allowing users to edit and save the file anyway. There's not much that Subversion can do in this situation—at the end of the day, there's simply no substitution for good interpersonal communication.[13]

Externals Definitions

Sometimes it is useful to construct a working copy that is made out of a number of different checkouts. For example, you may want different subdirectories to come from different locations in a repository or perhaps from different repositories altogether. You could certainly set up such a scenario by hand—using **svn checkout** to create the sort of nested working copy structure you are trying to achieve. But if this layout is important for everyone who uses your repository, every other user will need to perform the same checkout operations that you did.

Fortunately, Subversion provides support for *externals definitions*. An externals definition is a mapping of a local directory to the URL—and ideally a particular revision—of a versioned directory. In Subversion, you declare externals definitions in groups using the `svn:externals` property. You can create or modify this property using **svn propset** or **svn propedit** (see the section called “Manipulating Properties”). It can be set on any versioned directory, and its value describes both the external repository location and the client-side directory to which that location should be checked out.

The convenience of the `svn:externals` property is that once it is set on a versioned directory, everyone who checks out a working copy with that directory also gets the benefit of the externals definition. In other words, once one person has made the effort to define the nested working copy structure, no one else has to bother—Subversion will, after checking out the original work-

[13]Except, perhaps, a classic Vulcan mind-meld.

ing copy, automatically also check out the external working copies.

The relative target subdirectories of externals definitions *must not* already exist on your or other users' systems—Subversion will create them when it checks out the external working copy.

You also get in the externals definition design all the regular benefits of Subversion properties. The definitions are versioned. If you need to change an externals definition, you can do so using the regular property modification subcommands. When you commit a change to the `svn:externals` property, Subversion will synchronize the checked-out items against the changed externals definition when you next run **`svn update`**. The same thing will happen when others update their working copies and receive your changes to the externals definition.

Because the `svn:externals` property has a multiline value, we strongly recommend that you use **svn propedit** instead of **svn propset**.

Subversion releases prior to 1.5 honor an externals definition format that is a multiline table of subdirectories (relative to the versioned directory on which the property is set), optional revision flags, and fully qualified, absolute Subversion repository URLs. An example of this might look as follows:

```
$ svn propget svn:externals calc
third-party/sounds             http://svn.example.com/repos/sounds
third-party/skins -r148        http://svn.example.com/skinproj
third-party/skins/toolkit -r21 http://svn.example.com/skin-maker
```

When someone checks out a working copy of the `calc` directory referred to in the previous example, Subversion also continues to check out the items found in its externals definition.

```
$ svn checkout http://svn.example.com/repos/calc
A    calc
A    calc/Makefile
A    calc/integer.c
A    calc/button.c
Checked out revision 148.

Fetching external item into calc/third-party/sounds
A    calc/third-party/sounds/ding.ogg
A    calc/third-party/sounds/dong.ogg
A    calc/third-party/sounds/clang.ogg
…
A    calc/third-party/sounds/bang.ogg
A    calc/third-party/sounds/twang.ogg
Checked out revision 14.

Fetching external item into calc/third-party/skins
…
```

As of Subversion 1.5, though, a new format of the `svn:externals` property is supported. Externals definitions are still multiline, but the order and format of the various pieces of information have changed. The new syntax more closely mimics the order of arguments you might pass to **svn checkout**: the optional revision flags come first, then the external Subversion repository URL, and finally the relative local subdirectory. Notice, though, that this time we didn't say “fully qualified, absolute Subversion repository URLs.” That's because the new format supports relative URLs and URLs that carry peg revisions. The previous example of an

externals definition might, in Subversion 1.5, look like the following:

```
$ svn propget svn:externals calc
      http://svn.example.com/repos/sounds third-party/sounds
-r148 http://svn.example.com/skinproj third-party/skins
-r21  http://svn.example.com/skin-maker third-party/skins/toolkit
```

Or, making use of the peg revision syntax (which we describe in detail in the section called “Peg and Operative Revisions”), it might appear as:

```
$ svn propget svn:externals calc
http://svn.example.com/repos/sounds third-party/sounds
http://svn.example.com/skinproj@148 third-party/skins
http://svn.example.com/skin-maker@21 third-party/skins/toolkit
```

You should seriously consider using explicit revision numbers in all of your externals definitions. Doing so means that you get to decide when to pull down a different snapshot of external information, and exactly which snapshot to pull. Besides avoiding the surprise of getting changes to third-party repositories that you might not have any control over, using explicit revision numbers also means that as you backdate your working copy to a previous revision, your externals definitions will also revert to the way they looked in that previous revision, which in turn means that the external working copies will be updated to match the way *they* looked back when your repository was at that previous revision. For software projects, this could be the difference between a successful and a failed build of an older snapshot of your complex codebase.

For most repositories, these three ways of formatting the externals definitions have the same ultimate effect. They all bring the same benefits. Unfortunately, they all bring the same annoyances, too. Since the definitions shown use absolute URLs, moving or copying a directory to which they are attached will not affect what gets checked out as an external (though the relative local target subdirectory will, of course, move with the renamed directory). This can be confusing—even frustrating—in certain situations. For example, say you have a top-level directory named `my-project`, and you've created an externals definition on one of its subdirectories (`my-project/some-dir`) that tracks the latest revision of another of its subdirectories (`my-project/external-dir`).

```
$ svn checkout http://svn.example.com/projects .
A    my-project
A    my-project/some-dir
A    my-project/external-dir
…
Fetching external item into 'my-project/some-dir/subdir'
Checked out external at revision 11.

Checked out revision 11.
$ svn propget svn:externals my-project/some-dir
subdir http://svn.example.com/projects/my-project/external-dir

$
```

Now you use **svn move** to rename the `my-project` directory. At this point, your externals definition will still refer to a path under the `my-project` directory, even though that directory no longer exists.

```
$ svn move -q my-project renamed-project
$ svn commit -m "Rename my-project to renamed-project."
Deleting       my-project
Adding         renamed-project

Committed revision 12.
$ svn update
Updating '.':

svn: warning: W200000: Error handling externals definition for 'renamed-projec
t/some-dir/subdir':
svn: warning: W170000: URL 'http://svn.example.com/projects/my-project/externa
l-dir' at revision 12 doesn't exist
At revision 12.
svn: E205011: Failure occurred processing one or more externals definitions
$
```

Also, absolute URLs can cause problems with repositories that are available via multiple URL schemes. For example, if your Subversion server is configured to allow everyone to check out the repository over `http://` or `https://`, but only allow commits to come in via `https://`, you have an interesting problem on your hands. If your externals definitions use the `http://` form of the repository URLs, you won't be able to commit anything from the working copies created by those externals. On the other hand, if they use the `https://` form of the URLs, anyone who might be checking out via `http://` because his client doesn't support `https://` will be unable to fetch the external items. Be aware, too, that if you need to reparent your working copy (using **svn relocate**), externals definitions will *not* also be reparented.

Subversion 1.5 takes a huge step in relieving these frustrations. As mentioned earlier, the URLs used in the new externals definition format can be relative, and Subversion provides syntax magic for specifying multiple flavors of URL relativity.

`../`
: Relative to the URL of the directory on which the `svn:externals` property is set

`^/`
: Relative to the root of the repository in which the `svn:externals` property is versioned

`//`
: Relative to the scheme of the URL of the directory on which the `svn:externals` property is set

`/`
: Relative to the root URL of the server on which the `svn:externals` property is versioned

`^/../REPO-NAME`
: Relative to a sibling repository beneath the same `SVNParentPath` location as the repository in which the `svn:externals` is defined.

So, looking a fourth time at our previous externals definition example, and making use of the new absolute URL syntax in various ways, we might now see:

```
$ svn propget svn:externals calc
^/sounds third-party/sounds
/skinproj@148 third-party/skins
//svn.example.com/skin-maker@21 third-party/skins/toolkit
$
```

Subversion 1.6 brought two more improvements to externals definitions. First, it added a quoting and escape mechanism to the syntax so that the path of the external working copy may contain whitespace. This was previously problematic, of course, because whitespace is used to delimit the fields in an externals definition. Now you need only wrap such a path specification in double-quote (") characters or escape the problematic characters in the path with a backslash (\) character. Of course, if you have spaces in the *URL* portion of the external definition, you should use the standard URI-encoding mechanism to represent those.

```
$ svn propget svn:externals paint
http://svn.thirdparty.com/repos/My%20Project "My Project"
http://svn.thirdparty.com/repos/%22Quotes%20Too%22 \"Quotes\ Too\"
$
```

Subversion 1.6 also introduced support for external definitions for files. *File externals* are configured just like externals for directories and appear as a versioned file in the working copy.

For example, let's say you had the file `/trunk/bikeshed/blue.html` in your repository, and you wanted this file, as it appeared in revision 40, to appear in your working copy of `/trunk/www/` as `green.html`.

The externals definition required to achieve this should look familiar by now:

```
$ svn propget svn:externals www/
^/trunk/bikeshed/blue.html@40 green.html
$ svn update
Updating '.':

Fetching external item into 'www'
E    www/green.html
Updated external to revision 40.

Update to revision 103.
$ svn status
    X   www/green.html
$
```

As you can see in the previous output, Subversion denotes file externals with the letter `E` when they are fetched into the working copy, and with the letter `X` when showing the working copy status.

While directory externals can place the external directory at any depth, and any missing intermediate directories will be created, file externals must be placed into a working copy that is already checked out.

When examining the file external with **svn info**, you can see the URL and revision the external is coming from:

```
$ svn info www/green.html
Path: www/green.html
Name: green.html
Working Copy Root Path: /home/harry/projects/my-project
URL: http://svn.example.com/projects/my-project/trunk/bikeshed/blue.html
Repository Root: http://svn.example.com/projects/my-project
Repository UUID: b2a368dc-7564-11de-bb2b-113435390e17
Revision: 40
Node kind: file
```

```
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 40
Last Changed Date: 2009-07-20 20:38:20 +0100 (Mon, 20 Jul 2009)
Text Last Updated: 2009-07-20 23:22:36 +0100 (Mon, 20 Jul 2009)
Checksum: 01a58b04617b92492d99662c3837b33b
$
```

Because file externals appear in the working copy as versioned files, they can be modified and even committed if they reference a file at the HEAD revision. The committed changes will then appear in the external as well as the file referenced by the external. However, in our example, we pinned the external to an older revision, so attempting to commit the external fails:

```
$ svn status
M   X   www/green.html
$ svn commit -m "change the color" www/green.html
Sending        www/green.html
svn: E155011: Commit failed (details follow):
svn: E155011: File '/trunk/bikeshed/blue.html' is out of date
$
```

Keep this in mind when defining file externals. If you need the external to refer to a certain revision of a file you will not be able to modify the external. If you want to be able to modify the external, you cannot specify a revision other than the `HEAD` revision, which is implied if no revision is specified.

Unfortunately, the support which exists for externals definitions in Subversion remains less than ideal. Both file and directory externals have shortcomings. For either type of external, the local subdirectory part of the definition cannot contain `..` parent directory indicators (such as `../../skins/myskin`). File externals cannot refer to files from other repositories. A file external's URL must always be in the same repository as the URL that the file external will be inserted into. Also, file externals cannot be moved or deleted. The `svn:externals` property must be modified instead. However, file externals can be copied.

Perhaps most disappointingly, the working copies created via the externals definition support are still disconnected from the primary working copy (on whose versioned directories the `svn:externals` property was actually set). And Subversion still truly operates only on nondisjoint working copies. So, for example, if you want to commit changes that you've made in one or more of those external working copies, you must run **svn commit** explicitly on those working copies—committing on the primary working copy will not recurse into any external ones.

We've already mentioned some of the additional shortcomings of the old `svn:externals` format and how the newer Subversion 1.5 format improves upon it. But be careful when making use of the new format that you don't inadvertently introduce new problems. For example, while the latest clients will continue to recognize and support the original externals definition format, pre-1.5 clients will *not* be able to correctly parse the new format. If you change all your externals definitions to the newer format, you effectively force everyone who uses those externals to upgrade their Subversion clients to a version that can parse them. Also, be careful to avoid naively relocating the `-r`*`NNN`* portion of the definition—the older format uses that revision as a peg revision, but the newer format uses it as an operative revision (with a peg revision of `HEAD` unless otherwise specified; see the section called "Peg and Operative Revisions" for a full explanation of the distinction here).

External working copies are still completely self-sufficient working copies. You can operate directly on them as you would any other working copy. This can be a handy feature, allowing you to examine an external working copy independently of any primary working copy whose `svn:externals` property caused its instantiation. Be careful, though, that you don't inadvertently modify your external working copy in subtle ways that cause problems. For example, while an externals definition might specify that the external working copy should be held at a particular revision number, if you run **svn update** directly on the external working copy, Subversion will oblige, and now your external working copy is out of sync with its declaration in the primary working copy. Using **svn switch** to directly

switch the external working copy (or some portion thereof) to another URL could cause similar problems if the contents of the primary working copy are expecting particular contents in the external content.

Besides the **svn checkout**, **svn update**, **svn switch**, and **svn export** commands which actually manage the *disjoint* (or disconnected) subdirectories into which externals are checked out, the **svn status** command also recognizes externals definitions. It displays a status code of `X` for the disjoint external subdirectories, and then recurses into those subdirectories to display the status of the external items themselves. You can pass the `--ignore-externals` option to any of these subcommands to disable externals definition processing.

Changelists

It is commonplace for a developer to find himself working at any given time on multiple different, distinct changes to a particular bit of source code. This isn't necessarily due to poor planning or some form of digital masochism. A software engineer often spots bugs in his peripheral vision while working on some nearby chunk of source code. Or perhaps he's halfway through some large change when he realizes the solution he's working on is best committed as several smaller logical units. Often, these logical units aren't nicely contained in some module, safely separated from other changes. The units might overlap, modifying different files in the same module, or even modifying different lines in the same file.

Developers can employ various work methodologies to keep these logical changes organized. Some use separate working copies of the same repository to hold each individual change in progress. Others might choose to create short-lived feature branches in the repository and use a single working copy that is constantly switched to point to one such branch or another. Still others use **diff** and **patch** tools to back up and restore uncommitted changes to and from patch files associated with each change. Each of these methods has its pros and cons, and to a large degree, the details of the changes being made heavily influence the methodology used to distinguish them.

Subversion provides a *changelists* feature that adds yet another method to the mix. Changelists are basically arbitrary labels (currently at most one per file) applied to working copy files for the express purpose of associating multiple files together. Users of many of Google's software offerings are familiar with this concept already. For example, Gmail [http://mail.google.com/] doesn't provide the traditional folders-based email organization mechanism. In Gmail, you apply arbitrary labels to emails, and multiple emails can be said to be part of the same group if they happen to share a particular label. Viewing only a group of similarly labeled emails then becomes a simple user interface trick. Many other Web 2.0 sites have similar mechanisms—consider the “tags” used by sites such as YouTube [http://www.youtube.com/] and Flickr [http://www.flickr.com/], “categories” applied to blog posts, and so on. Folks understand today that organization of data is critical, but that how that data is organized needs to be a flexible concept. The old files-and-folders paradigm is too rigid for some applications.

Subversion's changelist support allows you to create changelists by applying labels to files you want to be associated with that changelist, remove those labels, and limit the scope of the files on which its subcommands operate to only those bearing a particular label. In this section, we'll look in detail at how to do these things.

Creating and Modifying Changelists

You can create, modify, and delete changelists using the **svn changelist** command. More accurately, you use this command to set or unset the changelist association of a particular working copy file. A changelist is effectively created the first time you label a file with that changelist; it is deleted when you remove that label from the last file that had it. Let's examine a usage scenario that demonstrates these concepts.

Harry is fixing some bugs in the calculator application's mathematics logic. His work leads him to change a couple of files:

```
$ svn status
M       integer.c
M       mathops.c
$
```

While testing his bug fix, Harry notices that his changes bring to light a tangentially related bug in the user interface logic found in `button.c`. Harry decides that he'll go ahead and fix that bug, too, as a separate commit from his math fixes. Now, in a small working copy with only a handful of files and few logical changes, Harry can probably keep his two logical change groupings mentally organized without any problem. But today he's going to use Subversion's changelists feature as a special favor to the authors of this book.

Harry first creates a changelist and associates with it the two files he's already changed. He does this by using the **svn changelist** command to assign the same arbitrary changelist name to those files:

```
$ svn changelist math-fixes integer.c mathops.c
A [math-fixes] integer.c
A [math-fixes] mathops.c
$ svn status

--- Changelist 'math-fixes':
M       integer.c
M       mathops.c
$
```

As you can see, the output of **svn status** reflects this new grouping.

Harry now sets off to fix the secondary UI problem. Since he knows which file he'll be changing, he assigns that path to a changelist, too. Unfortunately, Harry carelessly assigns this third file to the same changelist as the previous two files:

```
$ svn changelist math-fixes button.c
A [math-fixes] button.c
$ svn status

--- Changelist 'math-fixes':
        button.c
M       integer.c
M       mathops.c
$
```

Fortunately, Harry catches his mistake. At this point, he has two options. He can remove the changelist association from `button.c`, and then assign a different changelist name:

```
$ svn changelist --remove button.c
D [math-fixes] button.c
$ svn changelist ui-fix button.c
A [ui-fix] button.c
$
```

Or, he can skip the removal and just assign a new changelist name. In this case, Subversion will first warn Harry that `button.c` is being removed from the first changelist:

```
$ svn changelist ui-fix button.c
D [math-fixes] button.c
```

```
A [ui-fix] button.c
$ svn status

--- Changelist 'ui-fix':
        button.c

--- Changelist 'math-fixes':
M       integer.c
M       mathops.c
$
```

Harry now has two distinct changelists present in his working copy, and **svn status** will group its output according to these changelist determinations. Notice that even though Harry hasn't yet modified `button.c`, it still shows up in the output of **svn status** as interesting because it has a changelist assignment. Changelists can be added to and removed from files at any time, regardless of whether they contain local modifications.

Harry now fixes the user interface problem in `button.c`.

```
$ svn status

--- Changelist 'ui-fix':
M       button.c

--- Changelist 'math-fixes':
M       integer.c
M       mathops.c
$
```

Changelists As Operation Filters

The visual grouping that Harry sees in the output of **svn status** as shown in our previous section is nice, but not entirely useful. The **status** command is but one of many operations that he might wish to perform on his working copy. Fortunately, many of Subversion's other operations understand how to operate on changelists via the use of the `--changelist` option.

When provided with a `--changelist` option, Subversion commands will limit the scope of their operation to only those files to which a particular changelist name is assigned. If Harry now wants to see the actual changes he's made to the files in his `math-fixes` changelist, he *could* explicitly list only the files that make up that changelist on the **svn diff** command line.

```
$ svn diff integer.c mathops.c
Index: integer.c
===================================================================
--- integer.c (revision 1157)
+++ integer.c (working copy)
…
Index: mathops.c
===================================================================
--- mathops.c (revision 1157)
+++ mathops.c (working copy)
…
$
```

That works okay for a few files, but what if Harry's change touched 20 or 30 files? That would be an annoyingly long list of explicitly named files. Now that he's using changelists, though, Harry can avoid explicitly listing the set of files in his changelist from now on, and instead provide just the changelist name:

```
$ svn diff --changelist math-fixes
Index: integer.c
===================================================================
--- integer.c (revision 1157)
+++ integer.c (working copy)
…
Index: mathops.c
===================================================================
--- mathops.c (revision 1157)
+++ mathops.c (working copy)
…
$
```

And when it's time to commit, Harry can again use the `--changelist` option to limit the scope of the commit to files in a certain changelist. He might commit his user interface fix by doing the following:

```
$ svn commit -m "Fix a UI bug found while working on math logic." \
             --changelist ui-fix
Sending        button.c
Transmitting file data .
Committed revision 1158.
$
```

In fact, the **svn commit** command provides a second changelists-related option: `--keep-changelists`. Normally, changelist assignments are removed from files after they are committed. But if `--keep-changelists` is provided, Subversion will leave the changelist assignment on the committed (and now unmodified) files. In any case, committing files assigned to one changelist leaves other changelists undisturbed.

```
$ svn status

--- Changelist 'math-fixes':
M       integer.c
M       mathops.c
$
```

The `--changelist` option acts only as a filter for Subversion command targets, and will not add targets to an operation. For example, on a commit operation specified as **`svn commit /path/to/dir`**, the target is the directory `/path/to/dir` and its children (to infinite depth). If you then add a changelist specifier to that command, only those files in and under `/path/to/dir` that are assigned that changelist name will be considered as targets of the commit—the commit will not include files located elsewhere (such as in `/path/to/another-dir`), regardless of their changelist assignment, even if they are part of the same working copy as the operation's target(s).

Even the **svn changelist** command accepts the `--changelist` option. This allows you to quickly and easily rename or remove a changelist:

```
$ svn changelist math-bugs --changelist math-fixes --depth infinity .
D [math-fixes] integer.c
A [math-bugs] integer.c
D [math-fixes] mathops.c
A [math-bugs] mathops.c
$ svn changelist --remove --changelist math-bugs --depth infinity .
D [math-bugs] integer.c
D [math-bugs] mathops.c
$
```

Finally, you can specify multiple instances of the `--changelist` option on a single command line. Doing so limits the operation you are performing to files found in any of the specified changesets.

Changelist Limitations

Subversion's changelist feature is a handy tool for grouping working copy files, but it does have a few limitations. Changelists are artifacts of a particular working copy, which means that changelist assignments cannot be propagated to the repository or otherwise shared with other users. Changelists can be assigned only to files—Subversion doesn't currently support the use of changelists with directories. Finally, you can have at most one changelist assignment on a given working copy file. Here is where the blog post category and photo service tag analogies break down—if you find yourself needing to assign a file to multiple changelists, you're out of luck.

Network Model

At some point, you're going to need to understand how your Subversion client communicates with its server. Subversion's networking layer is abstracted, meaning that Subversion clients exhibit the same general behaviors no matter what sort of server they are operating against. Whether speaking the HTTP protocol (`http://`) with the Apache HTTP Server or speaking the custom Subversion protocol (`svn://`) with **svnserve**, the basic network model is the same. In this section, we'll explain the basics of that network model, including how Subversion manages authentication and authorization matters.

Requests and Responses

The Subversion client spends most of its time managing working copies. When it needs information from a remote repository, however, it makes a network request, and the server responds with an appropriate answer. The details of the network protocol are hidden from the user—the client attempts to access a URL, and depending on the URL scheme, a particular protocol is used to contact the server (see the section called “Addressing the Repository”).

Run **`svn --version`** to see which URL schemes and protocols the client knows how to use.

When the server process receives a client request, it often demands that the client identify itself. It issues an authentication challenge to the client, and the client responds by providing *credentials* back to the server. Once authentication is complete, the server responds with the original information that the client asked for. Notice that this system is different from systems such as CVS, where the client preemptively offers credentials (“logs in”) to the server before ever making a request. In Subversion, the server “pulls” credentials by challenging the client at the appropriate moment, rather than the client “pushing” them. This makes certain operations more elegant. For example, if a server is configured to allow anyone in the world to read a repository, the server will never issue an authentication challenge when a client attempts to **svn checkout**.

If the particular network requests issued by the client result in a new revision being created in the repository (e.g., **svn commit**), Subversion uses the authenticated username associated with those requests as the author of the revision. That is, the authenticated user's name is stored as the value of the `svn:author` property on the new revision (see the section called “Subversion Properties” in Chapter 9, *Subversion Complete Reference*). If the client was not authenticated (i.e., if the server never issued an authentic-

ation challenge), the revision's `svn:author` property is empty.

Client Credentials

Many Subversion servers are configured to require authentication. Sometimes anonymous read operations are allowed, while write operations must be authenticated. In other cases, reads and writes alike require authentication. Subversion's different server options understand different authentication protocols, but from the user's point of view, authentication typically boils down to usernames and passwords. Subversion clients offer several different ways to retrieve and store a user's authentication credentials, from interactive prompting for usernames and passwords to encrypted and non-encrypted on-disk data caches.

The security-conscious reader will suspect immediately that there is reason for concern here. “Caching passwords on disk? That's terrible! You should never do that!” Don't worry—it's not as bad as it sounds. The following sections discuss the various types of credential caches that Subversion uses, when it uses them, and how to disable that functionality in whole or in part.

Caching credentials

Subversion offers a remedy for the annoyance caused when users are forced to type their usernames and passwords over and over again. By default, whenever the command-line client successfully responds to a server's authentication challenge, credentials are cached on disk and keyed on a combination of the server's hostname, port, and authentication realm. This cache will then be automatically consulted in the future, avoiding the need for the user to re-type his or her authentication credentials. If seemingly suitable credentials are not present in the cache, or if the cached credentials ultimately fail to authenticate, the client will, by default, fall back to prompting the user for the necessary information.

The Subversion developers recognize that on-disk caches of authentication credentials can be a security risk. To offset this, Subversion works with available mechanisms provided by the operating system and environment to try to minimize the risk of leaking this information.

- On Windows, the Subversion client stores passwords in the `%APPDATA%/Subversion/auth/` directory. On Windows 2000 and later, the standard Windows cryptography services are used to encrypt the password on disk. Because the encryption key is managed by Windows and is tied to the user's own login credentials, only the user can decrypt the cached password. (Note that if the user's Windows account password is reset by an administrator, all of the cached passwords become undecipherable. The Subversion client will behave as though they don't exist, prompting for passwords when required.)

- Similarly, on Mac OS X, the Subversion client stores all repository passwords in the login keyring (managed by the Keychain service), which is protected by the user's account password. User preference settings can impose additional policies, such as requiring that the user's account password be entered each time the Subversion password is used.

- For other Unix-like operating systems, no single standard “keychain” service exists. However, the Subversion client knows how to store passwords securely using the “GNOME Keyring” and “KDE Wallet” services. Also, before storing unencrypted passwords in the `~/.subversion/auth/` caching area, the Subversion client will ask the user for permission to do so. Note that the `auth/` caching area is still permission-protected so that only the user (owner) can read data from it, not the world at large. The operating system's own file permissions protect the passwords from other non-administrative users on the same system, provided they have no direct physical access to the storage media of the home directory, or backups thereof.

Of course, for the truly paranoid, none of these mechanisms meets the test of perfection. So for those folks willing to sacrifice convenience for the ultimate in security, Subversion provides various ways of disabling its credentials caching system altogether.

Disabling password caching

When you perform a Subversion operation that requires you to authenticate, by default Subversion tries to cache your authentication credentials on disk in encrypted form. On some systems, Subversion may be unable to encrypt your authentication data. In those situations, Subversion will ask whether you want to cache your credentials to disk in plaintext:

```
$ svn checkout https://host.example.com:443/svn/private-repo
```

```
-----------------------------------------------------------------------
ATTENTION!  Your password for authentication realm:

   <https://host.example.com:443> Subversion Repository

can only be stored to disk unencrypted!  You are advised to configure
your system so that Subversion can store passwords encrypted, if
possible.  See the documentation for details.

You can avoid future appearances of this warning by setting the value
of the 'store-plaintext-passwords' option to either 'yes' or 'no' in
'/tmp/servers'.
-----------------------------------------------------------------------
Store password unencrypted (yes/no)?
```

If you want the convenience of not having to continually reenter your password for future operations, you can answer `yes` to this prompt. If you're concerned about caching your Subversion passwords in plaintext and do not want to be asked about it again and again, you can disable caching of plaintext passwords either permanently, or on a server-per-server basis.

When considering how to use Subversion's password caching system, you'll want to consult any governing policies that are in place for your client computer—many companies have strict rules about the ways that their employees' authentication credentials should be stored.

To permanently disable caching of passwords in plaintext, add the line `store-plaintext-passwords = no` to the `[global]` section in the `servers` configuration file on the local machine. To disable plaintext password caching for a particular server, use the same setting in the appropriate group section in the `servers` configuration file. (See the section called "Configuration Options" in Chapter 7, *Customizing Your Subversion Experience* for details.)

To disable password caching entirely for any single Subversion command-line operation, pass the `--no-auth-cache` option to that command line. To permanently disable caching entirely, add the line `store-passwords = no` to your local machine's Subversion configuration file.

Removing cached credentials

Sometimes users will want to remove specific credentials from the disk cache. To do this, you need to navigate into the `auth/` area and manually delete the appropriate cache file. Credentials are cached in individual files; if you look inside each file, you will see keys and values. The `svn:realmstring` key describes the particular server realm that the file is associated with:

```
$ ls ~/.subversion/auth/svn.simple/
5671adf2865e267db74f09ba6f872c28
3893ed123b39500bca8a0b382839198e
5c3c22968347b390f349ff340196ed39

$ cat ~/.subversion/auth/svn.simple/5671adf2865e267db74f09ba6f872c28

K 8
username
V 3
joe
K 8
password
V 4
blah
K 15
```

```
svn:realmstring
V 45
<https://svn.domain.com:443> Joe's repository
END
```

Once you have located the proper cache file, just delete it.

Command-line authentication

All Subversion command-line operations accept the `--username` and `--password` options, which allow you to specify your username and password, respectively, so that Subversion isn't forced to prompt you for that information. This is especially handy if you need to invoke Subversion from a script and cannot rely on Subversion being able to locate valid cached credentials for you. These options are also helpful when Subversion has already cached authentication credentials for you, but you know they aren't the ones you want it to use. Perhaps several system users share a login to the system, but each have distinct Subversion identities. You can omit the `--password` option from this pair if you wish Subversion to use only the provided username, but still prompt you for that username's password.

Authentication wrap-up

One last word about **svn**'s authentication behavior, specifically regarding the `--username` and `--password` options. Many client subcommands accept these options, but it is important to understand that using these options does *not* automatically send credentials to the server. As discussed earlier, the server “pulls” credentials from the client when it deems necessary; the client cannot “push” them at will. If a username and/or password are passed as options, they will be presented to the server only if the server requests them. These options are typically used to authenticate as a different user than Subversion would have chosen by default (such as your system login name) or when trying to avoid interactive prompting (such as when calling **svn** from a script).

A common mistake is to misconfigure a server so that it never issues an authentication challenge. When users pass `--username` and `--password` options to the client, they're surprised to see that they're never used; that is, new revisions still appear to have been committed anonymously!

Here is a final summary that describes how a Subversion client behaves when it receives an authentication challenge.

1. First, the client checks whether the user specified any credentials as command-line options (`--username` and/or `--password`). If so, the client will try to use those credentials to authenticate against the server.
2. If no command-line credentials were provided, or the provided ones were invalid, the client looks up the server's hostname, port, and realm in the runtime configuration's `auth/` area, to see whether appropriate credentials are cached there. If so, it attempts to use those credentials to authenticate.
3. Finally, if the previous mechanisms failed to successfully authenticate the user against the server, the client resorts to interactively prompting the user for valid credentials (unless instructed not to do so via the `--non-interactive` option or its client-specific equivalents).

If the client successfully authenticates by any of these methods, it will attempt to cache the credentials on disk (unless the user has disabled this behavior, as mentioned earlier).

Summary

After reading this chapter, you should have a firm grasp on some of Subversion's features that, while perhaps not used *every* time you interact with your version control system, are certainly handy to know about. But don't stop here! Read on to the following chapter, where you'll learn about branches, tags, and merging. Then you'll have nearly full mastery of the Subversion client.

Though our lawyers won't allow us to promise you anything, this additional knowledge could make you measurably more cool.[14]

[14]No purchase necessary. Certains terms and conditions apply. No guarantee of coolness—implicit or otherwise—exists. Mileage may vary.

Chapter 4. Branching and Merging

> “#### (It is upon the Trunk that a gentleman works.)”
>
> —Confucius

Branching and merging are fundamental aspects of version control, simple enough to explain conceptually but offering just enough complexity and nuance to merit their own chapter in this book. Herein, we'll introduce you to the general ideas behind these operations as well as Subversion's somewhat unique approach to them. If you've not familiarized yourself with Subversion's basic concepts (found in Chapter 1, *Fundamental Concepts*), we recommmend that you do so before reading this chapter.

What's a Branch?

Suppose it's your job to maintain a document for a division in your company—a handbook of some sort. One day a different division asks you for the same handbook, but with a few parts “tweaked” for them, since they do things slightly differently.

What do you do in this situation? You do the obvious: make a second copy of your document and begin maintaining the two copies separately. As each department asks you to make small changes, you incorporate them into one copy or the other.

You often want to make the same change to both copies. For example, if you discover a typo in the first copy, it's very likely that the same typo exists in the second copy. The two documents are almost the same, after all; they differ only in small, specific ways.

This is the basic concept of a branch—namely, a line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of something, and moves on from there, generating its own history (see Figure 4.1, “Branches of development”).

Figure 4.1. Branches of development

Subversion has commands to help you maintain parallel branches of your files and directories. It allows you to create branches by copying your data, and remembers that the copies are related to one another. It also helps you duplicate changes from one branch to another. Finally, it can make portions of your working copy reflect different branches so that you can “mix and match” different lines of development in your daily work.

Using Branches

At this point, you should understand how each commit creates a new state of the filesystem tree (called a “revision”) in the repository. If you don't, go back and read about revisions in the section called “Revisions”.

Let's revisit the example from Chapter 1, *Fundamental Concepts*. Remember that you and your collaborator, Sally, are sharing a repository that contains two projects, `paint` and `calc`. Notice that in Figure 4.2, “Starting repository layout”, however, each

project directory now contains subdirectories named `trunk` and `branches`. The reason for this will soon become clear.

Figure 4.2. Starting repository layout

As before, assume that Sally and you both have working copies of the “calc” project. Specifically, you each have a working copy of `/calc/trunk`. All the files for the project are in this subdirectory rather than in `/calc` itself, because your team has decided that `/calc/trunk` is where the “main line” of development is going to take place.

Let's say that you've been given the task of implementing a large software feature. It will take a long time to write, and will affect all the files in the project. The immediate problem is that you don't want to interfere with Sally, who is in the process of fixing small bugs here and there. She's depending on the fact that the latest version of the project (in `/calc/trunk`) is always usable. If you start committing your changes bit by bit, you'll surely break things for Sally (and other team members as well).

One strategy is to crawl into a hole: you can stop sharing information for a week or two, gutting and reorganizing all the files in your private working copy but not committing or updating until you're completely finished with your task. There are a number of problems with this, though. First, it's not very safe. Should something bad happen to your working copy or computer, you risk losing all your changes. Second, it's not very flexible. Unless you manually replicate your changes across different working copies or computers, you're stuck trying to make your changes in a single working copy. Similarly, it's difficult to share your work-in-progress with anyone else. A common software development “best practice” is to allow your peers to review your work as you go. If nobody sees your intermediate commits, you lose potential feedback and may end up going down the wrong path for weeks before another person on your team notices. Finally, when you're finished with all your changes, you might find it very difficult to merge your completed work with the rest of the company's main body of code. Sally (or others) may have made many other changes in the repository that are difficult to incorporate into your working copy when you eventually run **svn update** after weeks of isolation.

The better solution is to create your own branch, or line of development, in the repository. This allows you to save your not-yet-completed work frequently without interfering with others' changes and while still selectively sharing information with your collaborators. You'll see exactly how this works as we continue.

Creating a Branch

Creating a branch is very simple—you make a copy of your project tree in the repository using the **svn copy** command. Since your project's source code is rooted in the `/calc/trunk` directory, it's that directory that you'll copy. Where should the new copy live? Wherever you wish. The repository location in which branches are stashed is left by Subversion as a matter of project policy. Finally, your branch will need a name to distinguish it from other branches. Once again, the name you choose is unimportant to Subversion—you can use whatever name works best for you and your team.

Let's assume that your team (like most) has a policy of creating branches in the `branches` directory that is a sibling of the project's trunk (the `/calc/branches` directory in our scenario). Lacking inspiration, you settle on `my-calc-branch` as the name you wish to give your branch. This means that you'll create a new directory, `/calc/branches/my-calc-branch`, which begins its life as a copy of `/calc/trunk`.

You may already have seen **svn copy** used to copy one file to another within a working copy. But it can also be used to do a *remote copy*—a copy that immediately results in a newly committed repository revision and for which no working copy is required at all. Just copy one URL to another:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/my-calc-branch \
           -m "Creating a private branch of /calc/trunk."

Committed revision 341.
$
```

This command causes a near-instantaneous commit in the repository, creating a new directory in revision 341. The new directory is a copy of `/calc/trunk`. This is shown in Figure 4.3, “Repository with new copy”.[1] While it's also possible to create a branch by using **svn copy** to duplicate a directory within the working copy, this technique isn't recommended. It can be quite slow, in fact! Copying a directory on the client side is a linear-time operation, in that it actually has to duplicate every file and subdirectory within that working copy directory on the local disk. Copying a directory on the server, however, is a constant-time operation, and it's the way most people create branches.

Figure 4.3. Repository with new copy

[1]Subversion does not support copying between different repositories. When using URLs with **svn copy** or **svn move**, you can only copy items within the same repository.

Cheap Copies

Subversion's repository has a special design. When you copy a directory, you don't need to worry about the repository growing huge—Subversion doesn't actually duplicate any data. Instead, it creates a new directory entry that points to an *existing* tree. If you're an experienced Unix user, you'll recognize this as the same concept behind a hard link. As further changes are made to files and directories beneath the copied directory, Subversion continues to employ this hard link concept where it can. It duplicates data only when it is necessary to disambiguate different versions of objects.

This is why you'll often hear Subversion users talk about “cheap copies.” It doesn't matter how large the directory is—it takes a very tiny, constant amount of time and space to make a copy of it. In fact, this feature is the basis of how commits work in Subversion: each revision is a “cheap copy” of the previous revision, with a few items lazily changed within. (To read more about this, visit Subversion's web site and read about the “bubble up” method in Subversion's design documents.)

Of course, these internal mechanics of copying and sharing data are hidden from the user, who simply sees copies of trees. The main point here is that copies are cheap, both in time and in space. If you create a branch entirely within the repository (by running **`svn copy`** ***`URL1 URL2`***), it's a quick, constant-time operation. Make branches as often as you want.

Working with Your Branch

Now that you've created a branch of the project, you can check out a new working copy to start using it:

```
$ svn checkout http://svn.example.com/repos/calc/branches/my-calc-branch
A  my-calc-branch/Makefile
A  my-calc-branch/integer.c
A  my-calc-branch/button.c
Checked out revision 341.
$
```

There's nothing special about this working copy; it simply mirrors a different directory in the repository. When you commit changes, however, Sally won't see them when she updates, because her working copy is of `/calc/trunk`. (Be sure to read the section called "Traversing Branches" later in this chapter: the **svn switch** command is an alternative way of creating a working copy of a branch.)

Let's pretend that a week goes by, and the following commits happen:

- You make a change to `/calc/branches/my-calc-branch/button.c`, which creates revision 342.
- You make a change to `/calc/branches/my-calc-branch/integer.c`, which creates revision 343.
- Sally makes a change to `/calc/trunk/integer.c`, which creates revision 344.

Now two independent lines of development (shown in Figure 4.4, "The branching of one file's history") are happening on `integer.c`.

Figure 4.4. The branching of one file's history

Things get interesting when you look at the history of changes made to your copy of `integer.c`:

```
$ pwd
/home/user/my-calc-branch

$ svn log -v integer.c
------------------------------------------------------------------------
r343 | user | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/branches/my-calc-branch/integer.c

* integer.c:  frozzled the wazjub.
```

```
------------------------------------------------------------------------
r341 | user | 2002-11-03 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   A /calc/branches/my-calc-branch (from /calc/trunk:340)

Creating a private branch of /calc/trunk.

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   A /calc/trunk/integer.c

* integer.c:  adding this file to the project.

------------------------------------------------------------------------
```

Notice that Subversion is tracing the history of your branch's `integer.c` all the way back through time, even traversing the point where it was copied. It shows the creation of the branch as an event in the history, because `integer.c` was implicitly copied when all of `/calc/trunk/` was copied. Now look at what happens when Sally runs the same command on her copy of the file:

```
$ pwd
/home/sally/calc

$ svn log -v integer.c
------------------------------------------------------------------------
r344 | sally | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  fix a bunch of spelling errors.

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   A /calc/trunk/integer.c

* integer.c:  adding this file to the project.

------------------------------------------------------------------------
```

Sally sees her own revision 344 change, but not the change you made in revision 343. As far as Subversion is concerned, these two

commits affected different files in different repository locations. However, Subversion *does* show that the two files share a common history. Before the branch copy was made in revision 341, the files used to be the same file. That's why you and Sally both see the changes made in revisions 303 and 98.

The Key Concepts Behind Branching

You should remember two important lessons from this section. First, Subversion has no internal concept of a branch—it knows only how to make copies. When you copy a directory, the resultant directory is only a “branch” because *you* attach that meaning to it. You may think of the directory differently, or treat it differently, but to Subversion it's just an ordinary directory that happens to carry some extra historical information.

Second, because of this copy mechanism, Subversion's branches exist as *normal filesystem directories* in the repository. This is different from other version control systems, where branches are typically defined by adding extra-dimensional “labels” to collections of files. The location of your branch directory doesn't matter to Subversion. Most teams follow a convention of putting all branches into a `/branches` directory, but you're free to invent any policy you wish.

Basic Merging

Now you and Sally are working on parallel branches of the project: you're working on a private branch, and Sally is working on the trunk, or main line of development.

For projects that have a large number of contributors, it's common for most people to have working copies of the trunk. Whenever someone needs to make a long-running change that is likely to disrupt the trunk, a standard procedure is to create a private branch and commit changes there until all the work is complete.

So, the good news is that you and Sally aren't interfering with each other. The bad news is that it's very easy to drift *too* far apart. Remember that one of the problems with the “crawl in a hole” strategy is that by the time you're finished with your branch, it may be near-impossible to merge your changes back into the trunk without a huge number of conflicts.

Instead, you and Sally might continue to share changes as you work. It's up to you to decide which changes are worth sharing; Subversion gives you the ability to selectively “copy” changes between branches. And when you're completely finished with your branch, your entire set of branch changes can be copied back into the trunk. In Subversion terminology, the general act of replicating changes from one branch to another is called *merging*, and it is performed using various invocations of the **svn merge** subcommand.

In the examples that follow, we're assuming that both your Subversion client and server are running Subversion 1.7 (or later). If either client or server is older than version 1.5, things are more complicated: the system won't track changes automatically, forcing you to use painful manual methods to achieve similar results. That is, you'll always need to use the detailed merge syntax to specify specific ranges of revisions to replicate (see the section called “Merge Syntax: Full Disclosure” later in this chapter), and take special care to keep track of what's already been merged and what hasn't. For this reason, we *strongly* recommend that you make sure your client and server are at least at version 1.5.

Merge Tracking

Subversion 1.5 introduced the *merge tracking* feature to Subversion. Prior to this feature keeping track of merges required cumbersome manual procedures or the use of external tools. Subsequent releases of Subversion introduced many enhancements and bug fixes to merge tracking, which is why we recommend using the most recent versions on both your server and client. Keep in mind that even if your server is running 1.5 or 1.6, you can still use a 1.7 client. This is particularly important as regards merge tracking, because the overwhelming majority of fixes to it are on the client side.

Changesets

Before we proceed further, we should warn you that there's a lot of discussion of “changes” in the pages ahead. A lot of people experienced with version control systems use the terms “change” and “changeset” interchangeably, and we should clarify what Sub-

version understands as a *changeset*.

Everyone seems to have a slightly different definition of changeset, or at least a different expectation of what it means for a version control system to have one. For our purposes, let's say that a changeset is just a collection of changes with a unique name. The changes might include textual edits to file contents, modifications to tree structure, or tweaks to metadata. In more common speak, a changeset is just a patch with a name you can refer to.

In Subversion, a global revision number *N* names a tree in the repository: it's the way the repository looked after the *N*th commit. It's also the name of an implicit changeset: if you compare tree *N* with tree *N*-1, you can derive the exact patch that was committed. For this reason, it's easy to think of revision *N* as not just a tree, but a changeset as well. If you use an issue tracker to manage bugs, you can use the revision numbers to refer to particular patches that fix bugs—for example, “this issue was fixed by r9238.” Somebody can then run **`svn log -r 9238`** to read about the exact changeset that fixed the bug, and run **`svn diff -c 9238`** to see the patch itself. And (as you'll see shortly) Subversion's **svn merge** command is able to use revision numbers. You can merge specific changesets from one branch to another by naming them in the merge arguments: passing **`-c 9238`** to **svn merge** would merge changeset r9238 into your working copy.

Keeping a Branch in Sync

Continuing with our running example, let's suppose that a week has passed since you started working on your private branch. Your new feature isn't finished yet, but at the same time you know that other people on your team continue to make important changes in the project's `/trunk`. It's in your best interest to replicate those changes to your own branch, just to make sure they mesh well with your changes. This is done by performing a *sync merge*—a merge operation designed to bring your branch up to date with any changes made to its ancestral parent branch since your branch was created.

Frequently keeping your branch in sync with the main development line helps prevent “surprise” conflicts when the time comes for you to fold your changes back into the trunk.

Subversion is aware of the history of your branch and knows when it split away from the trunk. To perform a sync merge, first make sure your working copy of the branch is “clean”—that it has no local modifications reported by **svn status**. Then simply run:

```
$ pwd
/home/user/my-calc-branch

$ svn merge ^/calc/trunk
--- Merging r345 through r356 into '.':
U    button.c
U    integer.c
--- Recording mergeinfo for merge of r345 through r356 into '.':
 U   .
$
```

This basic syntax—**`svn merge`** ***`URL`***—tells Subversion to merge all changes which have not been previously merged from the URL to the current working directory (which is typically the root of your working copy). Notice that we're using the caret (`^`) syntax[2] to avoid having to type out the entire `/trunk` URL. Also note the “Recording mergeinfo for merge…” notification. This tells you that the merge is updating the `svn:mergeinfo` property. We'll discuss both this property and these notifications later in this chapter, in the section called “Mergeinfo and Previews”.

In this book and elsewhere (Subversion mailing lists, articles on merge tracking, etc.) you will frequently come across the term *mergeinfo*. This is simply shorthand for the `svn:mergeinfo` property.

[2]This was introduced in svn 1.6.

Keeping a Branch in Sync Without Merge Tracking

You may not always be able to use Subversion's merge tracking feature, perhaps because your server is running Subversion 1.4 or earlier. In such a scenario, you can of course still perform merges, but Subversion will need you to manually do many of the historical calculations that it automatically does on your behalf when the merge tracking feature is available.

To replicate the most recent trunk changes you need to perform sync merges the “old-fashioned” way—by specifying ranges of revisions you wish to merge.

Let's say you branched `/trunk` to `/branches/foo-feature` in revision 400:

```
$ svn log -v -r 400 ^/branches/foo-feature
------------------------------------------------------------------------
r400 | carol | 2011-11-09 10:51:27 -0500 (Wed, 09 Nov 2011) | 1 line
Changed paths:
A /branch/b2 (from /trunk:399)

Create branch for the foo feature
------------------------------------------------------------------------
```

When you are ready to syncronize your branch with the ongoing changes from trunk, you specify the starting revision as the revision of `/trunk` which the branch was copied from and the ending revision as `HEAD`:

```
$ svn merge ^/trunk -r399:HEAD
--- Merging r400 through r556 into '.':
A    include/foo.h
U    src/main.c
A    src/foo.c
…
```

After any conflicts have been resolved, you can commit the merged changed to your branch. Now, to avoid accidentally trying to merge these same changes into your branch again in the future, you'll need to record the fact that you've already merged them. But where should that record be kept? One of the simplest places to record this information is in the log message for the commit of the merge:

```
$ svn ci -m "Sync the foo-feature branch with ^/trunk through r556."
Sending        include/foo.h
…
Transmitting file data .
Committed revision 557.
```

The next time you sync `/branches/foo-branch` with `/trunk` you repeat this process, except that the starting revision is the *youngest* revision that's already been merged in from the trunk. If you've been keeping good records of your merges in the commit log messages, you should be able to determine what that youngest revision was by reading the revision logs associated with your branch. Once you know your starting revision, you can perform another sync merge:

```
$ svn merge ^/trunk -r556:HEAD
…
```

After running the prior example, your branch working copy now contains new local modifications, and these edits are duplications of all of the changes that have happened on the trunk since you first created your branch:

```
$ svn status
 M       .
M       button.c
M       integer.c
$
```

At this point, the wise thing to do is look at the changes carefully with **svn diff**, and then build and test your branch. Notice that the current working directory (“`.`”) has also been modified; **svn diff** will show that its `svn:mergeinfo` property has been either created or modified. This is important merge-related metadata that you should *not* touch, since it is needed by future **svn merge** commands. (We'll learn more about this metadata later in the chapter.)

After performing the merge, you might also need to resolve some conflicts—just as you do with **svn update**—or possibly make some small edits to get things working properly. (Remember, just because there are no *syntactic* conflicts doesn't mean there aren't any *semantic* conflicts!) If you encounter serious problems, you can always abort the local changes by running **`svn revert . -R`** (which will undo all local modifications) and starting a long “what's going on?” discussion with your collaborators. If things look good, however, you can submit these changes into the repository:

```
$ svn commit -m "Merged latest trunk changes to my-calc-branch."
Sending        .
Sending        button.c
Sending        integer.c
Transmitting file data ..
Committed revision 357.
$
```

At this point, your private branch is now “in sync” with the trunk, so you can rest easier knowing that as you continue to work in isolation, you're not drifting too far away from what everyone else is doing.

Why Not Use Patches Instead?

A question may be on your mind, especially if you're a Unix user: why bother to use **svn merge** at all? Why not simply use **svn patch** or the operating system's **patch** command to accomplish the same job? For example:

```
$ cd my-calc-branch
$ svn diff -r 341:HEAD ^/calc/trunk > my-patch-file
$ svn patch my-patch-file
U         integer.c
$
```

In this particular example, there really isn't much difference. But **svn merge** has special abilities that surpass the **patch** program. The file format used by **patch** is quite limited; it's able to tweak file contents only. There's no way to represent changes to *trees*, such as the addition, removal, or renaming of files and directories. Nor can the **patch** program notice changes to properties. If Sally's change had, say, added a new directory, the output of **svn diff** wouldn't have mentioned it at all. **svn diff** outputs only the limited patch format, so there are some ideas it simply can't express. Even Subversion's own **svn patch** subcommand, while more flexible than patch program, still has similar limitations.

The **svn merge** command, however, can express changes in tree structure and properties by directly applying them to your working copy. Even more important, this command records the changes that have been duplicated to your branch so that Subversion is aware of exactly which changes exist in each location (see the section called "Mergeinfo and Previews"). This is a critical feature that makes branch management usable; without it, users would have to manually keep notes on which sets of changes have or haven't been merged yet.

Suppose that another week has passed. You've committed more changes to your branch, and your comrades have continued to improve the trunk as well. Once again, you want to replicate the latest trunk changes to your branch and bring yourself in sync. Just run the same merge command again!

```
$ svn merge ^/calc/trunk
svn: E195020: Cannot merge into mixed-revision working copy [357:378]; try up\
dating first
$
```

Well that was unexpected! After making changes to your branch over the past week you now find yourself with a working copy that contains a mixture of revisions (see the section called "Mixed-revision working copies"). With the release of Subversion 1.7 the **svn merge** subcommand disables merges into mixed-revision working copies by default. Without going into too much detail, this is because of limitations in the way merges are tracked by the `svn:mergeinfo` property (see the section called "Mergeinfo and Previews" for details). These limitations mean that merges into mixed-revision working copies can result in unexpected text and tree conflicts.[3] We don't want any needless conflicts, so we update the working copy and then reattempt the merge.

```
$ svn up
Updating '.':
At revision 380.

$ svn merge ^/calc/trunk
--- Merging r357 through r380 into '.':
U    integer.c
U    Makefile
A    README
--- Recording mergeinfo for merge of r357 through r380 into '.':
 U   .
$
```

Subversion knows which trunk changes you previously replicated to your branch, so it carefully replicates only those changes you don't yet have. And once again, you build, test, and **svn commit** the local modifications to your branch.

Subtree Merges and Subtree Mergeinfo

[3]The **svn merge** subcommand option `--allow-mixed-revisions` allows you to override this prohibition, but you should only do so if you understand the ramifications and have a good reason for it.

In most of the examples in this chapter the merge target is the root directory of a branch (see the section called “What's a Branch?”). While this is a best practice, you may occasionally need to merge directly to some child of the branch root. This type of merge is called a *subtree merge* and the mergeinfo recorded to describe it is called *subtree mergeinfo*. There is nothing special about subtree merges or subtree mergeinfo. In fact there is really only one important point to keep in mind about these concepts: the complete record of merges to a branch may not be contained solely in the mergeinfo on the branch root. You may have to look to any subtree mergeinfo to get a full accounting. Fortunately Subversion does this for you and rarely will you need to concern yourself with it. A brief example will help explain:

```
# We need to merge r958 from trunk to branches/proj-X/doc/INSTALL,
# but that revision also affects main.c, which we don't want to merge:
$ svn log --verbose --quiet -r 958 ^/
------------------------------------------------------------------------
r958 | bruce | 2011-10-20 13:28:11 -0400 (Thu, 20 Oct 2011)
Changed paths:
   M /trunk/doc/INSTALL
   M /trunk/src/main.c
------------------------------------------------------------------------

# No problem, we'll do a subtree merge targeting the INSTALL file
# directly, but first take a note of what mergeinfo exists on the
# root of the branch:
$ cd branches/proj-X

$ svn propget svn:mergeinfo --recursive
Properties on '.':
  svn:mergeinfo
    /trunk:651-652

# Now we perform the subtree merge, note that merge source
# and target both point to INSTALL:
$ svn merge ^/trunk/doc/INSTALL doc/INSTALL -c 958
--- Merging r958 into 'doc/INSTALL':
U    doc/INSTALL
--- Recording mergeinfo for merge of r958 into 'doc/INSTALL':
 G   doc/INSTALL

# Once the merge is complete there is now subtree mergeinfo on INSTALL:
$ svn propget svn:mergeinfo --recursive
Properties on '.':
  svn:mergeinfo
    /trunk:651-652
Properties on 'doc/INSTALL':
  svn:mergeinfo
    /trunk/doc/INSTALL:651-652,958

# What if we then decide we do want all of r958? Easy, all we need do is
# repeat the merge of that revision, but this time to the root of the
# branch, Subversion notices the subtree mergeinfo on INSTALL and doesn't
# try to merge any changes to it, only the changes to main.c are merged:
$ svn merge ^/subversion/trunk . -c 958
--- Merging r958 into '.':
U    src/main.c
--- Recording mergeinfo for merge of r958 into '.':
 U   .
--- Eliding mergeinfo from 'doc/INSTALL':
 U   doc/INSTALL
```

You might be wondering why `INSTALL` in the above example has mergeinfo for r651-652, when we only merged r958. This

is due to mergeinfo inheritance, which we'll cover in the sidebar Mergeinfo Inheritance. Also note that the subtree mergeinfo on `doc/INSTALL` was removed, or "elided". This is called *mergeinfo elision* and it occurs whenever Subversion detects redundant subtree mergeinfo.

Prior to Subversion 1.7, merges unconditionally updated *all* of the subtree mergeinfo under the target to describe the merge. For users with a lot of subtree mergeinfo this meant that relatively "simple" merges (e.g. one which applied a diff to only a single file) resulted in changes to every subtree with mergeinfo, even those that were not parents of the effected path(s). This caused some level of confusion and frustration. Subversion 1.7 addresses this problem by only updating the mergeinfo on subtrees which are parents of the paths modified by the merge (i.e. paths changed, added, or deleted by application of the difference, see the section called "Merge Syntax: Full Disclosure"). The one exception to this behavior regards the actual merge target; the merge target's mergeinfo is always updated to describe the merge, even if the applied difference made no changes.

Reintegrating a Branch

What happens when you finally finish your work, though? Your new feature is done, and you're ready to merge your branch changes back to the trunk (so your team can enjoy the bounty of your labor). The process is simple. First, bring your branch into sync with the trunk again, just as you've been doing all along[4]:

```
$ svn merge ^/calc/trunk
--- Merging r381 through r385 into '.':
U    button.c
U    README
--- Recording mergeinfo for merge of r381 through r385 into '.':
 U   .

$ # build, test, ...

$ svn commit -m "Final merge of trunk changes to my-calc-branch."
Sending        .
Sending        button.c
Sending        README
Transmitting file data ..
Committed revision 390.
```

Now, use **svn merge** with the `--reintegrate` option to replicate your branch changes back into the trunk. You'll need a working copy of `/trunk`. You can get one by doing an **svn checkout**, dredging up an old trunk working copy from somewhere on your disk, or using **svn switch** (see the section called "Traversing Branches"). Your trunk working copy cannot have any local edits or contain a mixture of revisions (see the section called "Mixed-revision working copies"). While these are typically best practices for merging anyway, they are *required* when using the `--reintegrate` option.

Once you have a clean working copy of the trunk, you're ready to merge your branch back into it:

```
$ pwd
/home/user/calc-trunk

$ svn update  # (make sure the working copy is up to date)
```

[4]With Subversion 1.7 you don't absolutely have to do all your sync merges to the root of your branch as we do in this example. *If* your branch is effectively synced via a series of subtree merges then the reintegrate will work, but ask yourself, if the branch is effectively synced, then why are you doing subtree merges? Doing so is almost always needlessly complex.

```
Updating '.':
At revision 390.

$ svn merge --reintegrate ^/calc/branches/my-calc-branch
--- Merging differences between repository URLs into '.':
U    button.c
U    integer.c
U    Makefile
--- Recording mergeinfo for merge between repository URLs into '.':
 U   .

$ # build, test, verify, ...

$ svn commit -m "Merge my-calc-branch back into trunk!"
Sending        .
Sending        button.c
Sending        integer.c
Sending        Makefile
Transmitting file data ..
Committed revision 391.
```

Congratulations, your branch-specific changes have now been merged back into the main line of development. Notice our use of the `--reintegrate` option this time around. The option is critical for reintegrating changes from a branch back into its original line of development—don't forget it! It's needed because this sort of “merge back” is a different sort of work than what you've done up until now. Previously, we were asking **svn merge** to grab the “next set” of changes from one line of development (the trunk) and duplicate them to another (your branch). This is fairly straightforward, and each time Subversion knows how to pick up where it left off. In our prior examples, you can see that first it merges the ranges 345:356 from trunk to branch; later on, it continues by merging the next contiguously available range, 356:380. When doing the final sync, it merges the range 380:385.

When merging your branch back to the trunk, however, the underlying mathematics are quite different. Your feature branch is now a mishmash of both duplicated trunk changes and private branch changes, so there's no simple contiguous range of revisions to copy over. By specifying the `--reintegrate` option, you're asking Subversion to carefully replicate *only* those changes unique to your branch. (And in fact, it does this by comparing the latest trunk tree with the latest branch tree: the resulting difference is exactly your branch changes!)

Keep in mind that the `--reintegrate` option is quite specialized in contrast to the more general nature of most Subversion subcommand options. It supports the use case described above, but has little applicability outside of that. Because of this narrow focus, in addition to requiring an up-to-date working copy[5] with no mixed-revisions, it will not function in combination with most of the other **svn merge** options. You'll get an error if you use any non-global options but these: `--accept`, `--dry-run`, `--diff3-cmd`, `--extensions`, or `--quiet`.

Now that your private branch is merged to trunk, you may wish to remove it from the repository:

```
$ svn delete ^/calc/branches/my-calc-branch \
             -m "Remove my-calc-branch, reintegrated with trunk in r391."
Committed revision 392.
```

But wait! Isn't the history of that branch valuable? What if somebody wants to audit the evolution of your feature someday and look at all of your branch changes? No need to worry. Remember that even though your branch is no longer visible in the `/branches` directory, its existence is still an immutable part of the repository's history. A simple **svn log** command on the `/branches` URL will show the entire history of your branch. Your branch can even be resurrected at some point, should you de-

[5]Reintegrate merges are allowed if the target is a shallow checkout (see the section called “Sparse Directories”) but any paths affected by the diff which are “missing” due to the sparse working copy will be skipped, probably *not* what you intended!

sire (see the section called “Resurrecting Deleted Items”).

Once a `--reintegrate` merge is done from branch to trunk, the branch is no longer usable for further work. It's not able to correctly absorb new trunk changes, nor can it be properly reintegrated to trunk again. For this reason, if you want to keep working on your feature branch, we recommend destroying it and then re-creating it from the trunk:

```
$ svn delete http://svn.example.com/repos/calc/branches/my-calc-branch \
             -m "Remove my-calc-branch, reintegrated with trunk in r391."
Committed revision 392.

$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/my-calc-branch \
           -m "Recreate my-calc-branch from trunk@HEAD."
Committed revision 393.
```

There is another way of making the branch usable again after reintegration, without deleting the branch. See the section called “Keeping a Reintegrated Branch Alive”.

Mergeinfo and Previews

The basic mechanism Subversion uses to track changesets—that is, which changes have been merged to which branches—is by recording data in versioned properties. Specifically, merge data is tracked in the `svn:mergeinfo` property attached to files and directories. (If you're not familiar with Subversion properties, see the section called “Properties”.)

You can examine the property, just like any other:

```
$ cd my-calc-branch
$ svn propget svn:mergeinfo .
/trunk:341-390
$
```

While it is possible to modify `svn:mergeinfo` just as you might any other versioned property, we strongly discourage doing so unless you *really* know what you're doing.

The amount of `svn:mergeinfo` on a single path can get quite large, as can the output of a **svn propget --recursive** or **svn proplist --recursive** when dealing with large amounts of subtree mergeinfo, see Subtree Merges and Subtree Mergeinfo . The formatted output produced by the `--verbose` option with either of these subcommands is often very helpful in these cases.

The `svn:mergeinfo` property is automatically maintained by Subversion whenever you run **svn merge**. Its value indicates which changes made to a given path have been replicated into the directory in question. In our previous example, the path which is the source of the merged changes is `/trunk` and the directory which has received the changes is `/branches/my-calc-branch`. Earlier versions of Subversion maintained the `svn:mergeinfo` property silently. You could still detect the changes, after a merge completed, with the **svn diff** or **svn status** subcommands, but the merge itself gave no indication when it changed the `svn:mergeinfo` property. This is no longer true in Subversion 1.7, which has several new notifications to alert you when a merge updates the `svn:mergeinfo` property. These notifications all begin with “--- Recording mergeinfo for” and appear at the end of the merge. Unlike other merge notifications, these don't describe the application of a difference to a working copy (see the section called “Merge Syntax: Full Disclosure”), but instead describe "housekeeping" changes made to keep track of what was merged.

Subversion also provides a subcommand, **svn mergeinfo**, which is helpful in seeing not only which changesets a directory has absorbed, but also which changesets it's still eligible to receive. This gives a sort of preview of which changes a subsequent **svn merge** operation would replicate to your branch.

```
$ cd my-calc-branch

# Which changes have already been merged from trunk to branch?
$ svn mergeinfo ^/calc/trunk
r341
r342
r343
…
r388
r389
r390

# Which changes are still eligible to merge from trunk to branch?
$ svn mergeinfo ^/calc/trunk --show-revs eligible
r391
r392
r393
r394
r395
$
```

The **svn mergeinfo** command requires a “source” URL (where the changes come from), and takes an optional “target” URL (where the changes merge to). If no target URL is given, it assumes that the current working directory is the target. In the prior example, because we're querying our branch working copy, the command assumes we're interested in receiving changes to `/branches/mybranch` from the specified trunk URL.

Mergeinfo Inheritance

When a path has the `svn:mergeinfo` property set on it we say it has *explicit mergeinfo*. This explicit mergeinfo describes not only what changes were merged into that particular directory, but also all the children of that directory (because those children inherit the mergeinfo of their parent path). For example:

```
# What explicit mergeinfo exists on a branch?
$ svn propget svn:mergeinfo ^/branches/proj-X --recursive
/trunk:651-652

# What children does proj-X have?
$ svn list --recursive ^/branches/proj-X
doc/
doc/INSTALL
README
src/main.c

# Ask what revs were merged to a file with no explicit mergeinfo
$ svn mergeinfo ^/trunk/src/main.c ^/branches/proj-X/src/main.c
651
652
```

Notice from our first subcommand that only the root of `/branches/proj-X` has any explicit mergeinfo. However, when

we use **svn mergeinfo** to ask what was merged to `/branches/proj-X/src/main.c` it reports that the two revisions described in the explicit mergeinfo on `/branches/proj-X` were merged. This is because `/branches/proj-X/src/main.c`, having no explicit mergeinfo of its own, inherits the mergeinfo from its nearest parent with explicit mergeinfo, `/branches/proj-X`.

There are two cases in which mergeinfo is not inherited. First, if a path has explicit mergeinfo, then it never inherits mergeinfo. Another way to think of this is that explicit mergeinfo is always a complete record of the merges to a given path, once it exists it overrides any mergeinfo that path might otherwise inherit. The second way is when dealing with non-inheritable mergeinfo, a special type of explicit mergeinfo that applies *only* to the directory on which the `svn:mergeinfo` property is set (and it's only directories, non-inheritable mergeinfo is never set on files). For example:

```
# The '*' decorator indicates non-inheritable mergeinfo
$ svn propget svn:mergeinfo ^/branches/proj-X
/trunk:651-652,758*

# Revision 758 is non-inheritable, but still applies to the path it is
# set on. Here the '*' decorator signals that r758 is only partially
# merged from trunk.
$ svn mergeinfo ^/trunk ^/branches/proj-X
651
652
758*

# Revision 758 is not reported as merged because it is non-inheritable
# and applies only to ^/trunk
$ svn mergeinfo ^/trunk/src/main.c ^/branches/proj-X/src/main.c
651
652
```

You might never have to think about mergeinfo inheritance or encounter non-inheritable mergeinfo in your own repository. A discussion of the full ramifications of mergeinfo inheritance are beyond the scope of this book. If you have more questions check out some of the references mentioned in the section called “The Final Word on Merge Tracking”

With the release of Subversion 1.7, the **svn mergeinfo** subcommand can also account for subtree mergeinfo and non-inheritable mergeinfo. It accounts for subtree mergeinfo by use of the `--recursive` or `--depth` options, while non-inheritable mergeinfo is considered by default.

Let's say we have a branch with both subtree and non-inheritable mergeinfo:

```
$ svn propget svn:mergeinfo --recursive -v
# Non-inheritable mergeinfo
Properties on '.':
  svn:mergeinfo
    /trunk:651-652,758*
# Subtree mergeinfo
Properties on 'doc/INSTALL':
  svn:mergeinfo
    /trunk/doc/INSTALL:651-652,958,1060
```

From the above mergeinfo we see that r758 has only been merged into the root of the branch, but not any of the root's children. We also see that both r958 and r1060 have been merged only to the `doc/INSTALL` file. When we use **svn mergeinfo** with the `--recursive` option to see what has been merged from `^/trunk` to this branch, we see two revisions are flagged with the `*`

marker:

```
$ svn mergeinfo --show-revs=merged ^/trunk . --recursive
651
652
758*
958*
1060
```

The `*` indicates revisions that are only *partially* merged to the target in question (the meaning is the same if we are checking for eligible revisions). What this means in this example is that if we tried to merge r758 or r958 from `^/trunk` then more changes would result. Likewise, because r1060 is *not* flagged with a `*`, we know that it only affects `doc/INSTALL` and that trying to merge it would have no result.[6]

Another way to get a more precise preview of a merge operation is to use the `--dry-run` option:

```
$ svn merge ^/calc/trunk --dry-run
--- Merging r391 through r395 into 'branch':
U    integer.c

$ svn status
#  nothing printed, working copy is still unchanged.
```

The `--dry-run` option doesn't actually apply any local changes to the working copy. It shows only status codes that *would* be printed in a real merge. It's useful for getting a “high-level” preview of the potential merge, for those times when running **svn diff** gives too much detail.

After performing a merge operation, but before committing the results of the merge, you can use **`svn diff --depth=empty /path/to/merge/target`** to see only the changes to the immediate target of your merge. If your merge target was a directory, only property differences are displayed. This is a handy way to see the changes to the `svn:mergeinfo` property recorded by the merge operation, which will remind you about what you've just merged.

Of course, the best way to preview a merge operation is to just do it! Remember, running **svn merge** isn't an inherently risky thing (unless you've made local modifications to your working copy—but we already stressed that you shouldn't merge into such an environment). If you don't like the results of the merge, simply run **`svn revert . -R`** to revert the changes from your working copy and retry the command with different options. The merge isn't final until you actually **svn commit** the results.

Undoing Changes

An extremely common use for **svn merge** is to roll back a change that has already been committed. Suppose you're working away happily on a working copy of `/calc/trunk`, and you discover that the change made way back in revision 303, which changed `integer.c`, is completely wrong. It never should have been committed. You can use **svn merge** to “undo” the change in your working copy, and then commit the local modification to the repository. All you need to do is to specify a *reverse* difference. (You can do this by specifying `--revision 303:302`, or by an equivalent `--change -303`.)

[6] This is often termed an “inoperative” merge. Though in this example the merge of r1060 would do something: It would update the mergeinfo on the root of the branch, but it would be inoperative in the sense that no diff would be applied.

```
$ svn merge -c -303 ^/calc/trunk
--- Reverse-merging r303 into 'integer.c':
U    integer.c
--- Recording mergeinfo for reverse merge of r303 into 'integer.c':
 U   A-branch

$ svn status
 M       .
M       integer.c

$ svn diff
…
# verify that the change is removed
…

$ svn commit -m "Undoing change committed in r303."
Sending        integer.c
Transmitting file data .
Committed revision 350.
```

As we mentioned earlier, one way to think about a repository revision is as a specific changeset. By using the `-r` option, you can ask **svn merge** to apply a changeset, or a whole range of changesets, to your working copy. In our case of undoing a change, we're asking **svn merge** to apply changeset r303 to our working copy *backward*.

Keep in mind that rolling back a change like this is just like any other **svn merge** operation, so you should use **svn status** and **svn diff** to confirm that your work is in the state you want it to be in, and then use **svn commit** to send the final version to the repository. After committing, this particular changeset is no longer reflected in the `HEAD` revision.

Again, you may be thinking: well, that really didn't undo the commit, did it? The change still exists in revision 303. If somebody checks out a version of the `calc` project between revisions 303 and 349, she'll still see the bad change, right?

Yes, that's true. When we talk about “removing” a change, we're really talking about removing it from the `HEAD` revision. The original change still exists in the repository's history. For most situations, this is good enough. Most people are only interested in tracking the `HEAD` of a project anyway. There are special cases, however, where you really might want to destroy all evidence of the commit. (Perhaps somebody accidentally committed a confidential document.) This isn't so easy, it turns out, because Subversion was deliberately designed to never lose information. Revisions are immutable trees that build upon one another. Removing a revision from history would cause a domino effect, creating chaos in all subsequent revisions and possibly invalidating all working copies.[7]

Resurrecting Deleted Items

The great thing about version control systems is that information is never lost. Even when you delete a file or directory, it may be gone from the `HEAD` revision, but the object still exists in earlier revisions. One of the most common questions new users ask is, “How do I get my old file or directory back?”

The first step is to define exactly *which* item you're trying to resurrect. Here's a useful metaphor: you can think of every object in the repository as existing in a sort of two-dimensional coordinate system. The first coordinate is a particular revision tree, and the second coordinate is a path within that tree. So every version of your file or directory is defined by a specific coordinate pair. (Remember the “peg revision” syntax—foo.c@224—mentioned back in the section called “Peg and Operative Revisions”.)

First, you might need to use **svn log** to discover the exact coordinate pair you wish to resurrect. A good strategy is to run **`svn log --verbose`** in a directory that used to contain your deleted item. The `--verbose` (`-v`) option shows a list of all changed items in each revision; all you need to do is find the revision in which you deleted the file or directory. You can do this visually, or by using another tool to examine the log output (via **grep**, or perhaps via an incremental search in an editor).

[7]The Subversion project has plans, however, to someday implement a command that would accomplish the task of permanently deleting information. In the meantime, see the section called “svndumpfilter” for a possible workaround.

```
$ cd parent-dir
$ svn log -v
…
------------------------------------------------------------------------
r808 | joe | 2003-12-26 14:29:40 -0600 (Fri, 26 Dec 2003) | 3 lines
Changed paths:
   D /calc/trunk/real.c
   M /calc/trunk/integer.c

Added fast fourier transform functions to integer.c.
Removed real.c because code now in double.c.
…
```

In the example, we're assuming that you're looking for a deleted file `real.c`. By looking through the logs of a parent directory, you've spotted that this file was deleted in revision 808. Therefore, the last version of the file to exist was in the revision right before that. Conclusion: you want to resurrect the path `/calc/trunk/real.c` from revision 807.

That was the hard part—the research. Now that you know what you want to restore, you have two different choices.

One option is to use **svn merge** to apply revision 808 "in reverse." (We already discussed how to undo changes in the section called "Undoing Changes".) This would have the effect of re-adding `real.c` as a local modification. The file would be scheduled for addition, and after a commit, the file would again exist in `HEAD`.

In this particular example, however, this is probably not the best strategy. Reverse-applying revision 808 would not only schedule `real.c` for addition, but the log message indicates that it would also undo certain changes to `integer.c`, which you don't want. Certainly, you could reverse-merge revision 808 and then **svn revert** the local modifications to `integer.c`, but this technique doesn't scale well. What if 90 files were changed in revision 808?

A second, more targeted strategy is not to use **svn merge** at all, but rather to use the **svn copy** command. Simply copy the exact revision and path "coordinate pair" from the repository to your working copy:

```
$ svn copy ^/calc/trunk/real.c@807 ./real.c

$ svn status
A  +    real.c

$ svn commit -m "Resurrected real.c from revision 807, /calc/trunk/real.c."
Adding         real.c
Transmitting file data .
Committed revision 1390.
```

The plus sign in the status output indicates that the item isn't merely scheduled for addition, but scheduled for addition "with history." Subversion remembers where it was copied from. In the future, running **svn log** on this file will traverse back through the file's resurrection and through all the history it had prior to revision 807. In other words, this new `real.c` isn't really new; it's a direct descendant of the original, deleted file. This is usually considered a good and useful thing. If, however, you wanted to resurrect the file *without* maintaining a historical link to the old file, this technique works just as well:

```
$ svn cat ^/calc/trunk/real.c@807 > ./real.c

$ svn add real.c
A         real.c
```

```
$ svn commit -m "Re-created real.c from revision 807."
Adding         real.c
Transmitting file data .
Committed revision 1390.
```

Although our example shows us resurrecting a file, note that these same techniques work just as well for resurrecting deleted directories. Also note that a resurrection doesn't have to happen in your working copy—it can happen entirely in the repository:

```
$ svn copy ^/calc/trunk/real.c@807 ^/calc/trunk/ \
           -m "Resurrect real.c from revision 807."
Committed revision 1390.

$ svn update
Updating '.':
A    real.c
Updated to revision 1390.
```

Advanced Merging

Here ends the automated magic. Sooner or later, once you get the hang of branching and merging, you're going to have to ask Subversion to merge *specific* changes from one place to another. To do this, you're going to have to start passing more complicated arguments to **svn merge**. The next section describes the fully expanded syntax of the command and discusses a number of common scenarios that require it.

Cherrypicking

Just as the term “changeset” is often used in version control systems, so is the term *cherrypicking*. This word refers to the act of choosing *one* specific changeset from a branch and replicating it to another. Cherrypicking may also refer to the act of duplicating a particular set of (not necessarily contiguous!) changesets from one branch to another. This is in contrast to more typical merging scenarios, where the “next” contiguous range of revisions is duplicated automatically.

Why would people want to replicate just a single change? It comes up more often than you'd think. For example, let's go back in time and imagine that you haven't yet merged your private feature branch back to the trunk. At the water cooler, you get word that Sally made an interesting change to `integer.c` on the trunk. Looking over the history of commits to the trunk, you see that in revision 355 she fixed a critical bug that directly impacts the feature you're working on. You might not be ready to merge all the trunk changes to your branch just yet, but you certainly need that particular bug fix in order to continue your work.

```
$ svn diff -c 355 ^/calc/trunk

Index: integer.c
===================================================================
--- integer.c (revision 354)
+++ integer.c (revision 355)
@@ -147,7 +147,7 @@
     case 6:  sprintf(info->operating_system, "HPFS (OS/2 or NT)"); break;
     case 7:  sprintf(info->operating_system, "Macintosh"); break;
     case 8:  sprintf(info->operating_system, "Z-System"); break;
-    case 9:  sprintf(info->operating_system, "CP/MM");
+    case 9:  sprintf(info->operating_system, "CP/M"); break;
     case 10:  sprintf(info->operating_system, "TOPS-20"); break;
     case 11:  sprintf(info->operating_system, "NTFS (Windows NT)"); break;
```

```
     case 12:  sprintf(info->operating_system, "QDOS"); break;
```

Just as you used **svn diff** in the prior example to examine revision 355, you can pass the same option to **svn merge**:

```
$ svn merge -c 355 ^/calc/trunk
--- Merging r355 into '.':
U    integer.c
--- Recording mergeinfo for merge of r355 into '.':
 U   .

$ svn status
M       integer.c
```

You can now go through the usual testing procedures before committing this change to your branch. After the commit, Subversion marks r355 as having been merged to the branch so that future “magic” merges that synchronize your branch with the trunk know to skip over r355. (Merging the same change to the same branch almost always results in a conflict!)

```
$ cd my-calc-branch

$ svn propget svn:mergeinfo .
/trunk:341-349,355

# Notice that r355 isn't listed as "eligible" to merge, because
# it's already been merged.
$ svn mergeinfo ^/calc/trunk --show-revs eligible
r350
r351
r352
r353
r354
r356
r357
r358
r359
r360

$ svn merge ^/calc/trunk
--- Merging r350 through r354 into '.':
 U   .
U    integer.c
U    Makefile
--- Merging r356 through r360 into '.':
 U   .
U    integer.c
U    button.c
--- Recording mergeinfo for merge of r350 through r360 into '.':
 U   .
```

This use case of replicating (or *backporting*) bug fixes from one branch to another is perhaps the most popular reason for cherry-picking changes; it comes up all the time, for example, when a team is maintaining a “release branch” of software. (We discuss this pattern in the section called “Release Branches”.)

Did you notice how, in the last example, the merge invocation merged two distinct ranges? The **svn merge** command applied two independent patches to your working copy to skip over changeset 355, which your branch already contained. There's nothing inherently wrong with this, except that it has the potential to make conflict resolution trickier. If the first range of changes creates conflicts, you *must* resolve them interactively for the merge process to continue and apply the second range of changes. If you postpone a conflict from the first wave of changes, the whole merge command will bail out with an error message.[8]

A word of warning: while **svn diff** and **svn merge** are very similar in concept, they do have different syntax in many cases. Be sure to read about them in Chapter 9, *Subversion Complete Reference* for details, or ask **svn help**. For example, **svn merge** requires a working copy path as a target, that is, a place where it should apply the generated patch. If the target isn't specified, it assumes you are trying to perform one of the following common operations:

- You want to merge directory changes into your current working directory.
- You want to merge the changes in a specific file into a file by the same name that exists in your current working directory.

If you are merging a directory and haven't specified a target path, **svn merge** assumes the first case and tries to apply the changes into your current directory. If you are merging a file, and that file (or a file by the same name) exists in your current working directory, **svn merge** assumes the second case and tries to apply the changes to a local file with the same name.

Merge Syntax: Full Disclosure

You've now seen some examples of the **svn merge** command, and you're about to see several more. If you're feeling confused about exactly how merging works, you're not alone. Many users (especially those new to version control) are initially perplexed about the proper syntax of the command and about how and when the feature should be used. But fear not, this command is actually much simpler than you think! There's a very easy technique for understanding exactly how **svn merge** behaves.

The main source of confusion is the *name* of the command. The term “merge” somehow denotes that branches are combined together, or that some sort of mysterious blending of data is going on. That's not the case. A better name for the command might have been **svn diff-and-apply**, because that's all that happens: two repository trees are compared, and the differences are applied to a working copy.

If you're using **svn merge** to do basic copying of changes between branches, it will generally do the right thing automatically. For example, a command such as the following:

```
$ svn merge ^/calc/branches/some-branch
```

will attempt to duplicate any changes made on `some-branch` into your current working directory, which is presumably a working copy that shares some historical connection to the branch. The command is smart enough to only duplicate changes that your working copy doesn't yet have. If you repeat this command once a week, it will only duplicate the “newest” branch changes that happened since you last merged.

If you choose to use the **svn merge** command in all its full glory by giving it specific revision ranges to duplicate, the command takes three main arguments:

1. An initial repository tree (often called the *left side* of the comparison)
2. A final repository tree (often called the *right side* of the comparison)
3. A working copy to accept the differences as local changes (often called the *target* of the merge)

[8] At least, this is true in Subversion 1.7 at the time of this writing. This behavior may improve in future versions of Subversion.

Once these three arguments are specified, then the two trees are compared and the differences applied to the target working copy as local modifications. When the command is done, the results are no different than if you had hand-edited the files or run various **svn add** or **svn delete** commands yourself. If you like the results, you can commit them. If you don't like the results, you can simply **svn revert** all of the changes.

The syntax of **svn merge** allows you to specify the three necessary arguments rather flexibly. Here are some examples:

```
$ svn merge http://svn.example.com/repos/branch1@150 \
            http://svn.example.com/repos/branch2@212 \
            my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk my-working-copy

$ svn merge -r 100:200 http://svn.example.com/repos/trunk
```

The first syntax lays out all three arguments explicitly, naming each tree in the form *URL@REV* and naming the working copy target. The second syntax is used as a shorthand for situations when you're comparing two different revisions of the same URL. The last syntax shows how the working copy argument is optional; if omitted, it defaults to the current directory.

While the first example shows the “full” syntax of **svn merge**, use it very carefully; it can result in merges which do not record any `svn:mergeinfo` metadata at all. The next section talks a bit more about this.

Merges Without Mergeinfo

Subversion tries to generate merge metadata whenever it can, to make future invocations of **svn merge** smarter. There are still situations, however, where `svn:mergeinfo` data is not created or changed. Remember to be a bit wary of these scenarios:

Merging unrelated sources
: If you ask **svn merge** to compare two URLs that aren't related to each other, a patch is still generated and applied to your working copy, but no merging metadata is created. There's no common history between the two sources, and future “smart” merges depend on that common history.

Merging from foreign repositories
: While it's possible to run a command such as **`svn merge -r 100:200 http://svn.foreignproject.com/repos/trunk`**, the resultant patch also lacks any historical merge metadata. At the time of this writing, Subversion has no way of representing different repository URLs within the `svn:mergeinfo` property.

Using `--ignore-ancestry`
: If this option is passed to **svn merge**, it causes the merging logic to mindlessly generate differences the same way that **svn diff** does, ignoring any historical relationships. We discuss this later in this chapter in the section called “Noticing or Ignoring Ancestry”.

Applying reverse merges from a target's natural history
: Earlier in this chapter (the section called “Undoing Changes”) we discussed how to use **svn merge** to apply a “reverse patch” as a way of rolling back changes. If this technique is used to undo a change to an object's personal history (e.g., commit r5 to the trunk, then immediately roll back r5 using **`svn merge . -c -5`**), this sort of merge doesn't affect the recorded mergeinfo.[9]

Natural History and Implicit Mergeinfo

[9]Interestingly, after rolling back a revision like this, we wouldn't be able to reapply the revision using **`svn merge . -c 5`**, since the mergeinfo would already list r5 as being applied. We would have to use the `--ignore-ancestry` option to make the merge command ignore the existing mergeinfo!

As we mentioned earlier when discussing Mergeinfo Inheritance, a path that has the `svn:mergeinfo` property set on it is said to have “explicit” mergeinfo. Yes, this implies a path can have “implicit” mergeinfo, too! Implicit mergeinfo, or *natural history*, is simply a path's own history (see the section called “Examining History”) interpreted as mergeinfo. While implicit mergeinfo is largely an implementation detail, it can be a useful abstraction for understanding merge tracking behavior.

Let's say you created `^/trunk` in revision 100 and then later, in revision 201, created `^/branches/feature-branch` as a copy of `^/trunk@200`. The natural history of `^/branches/feature-branch` contains all the repository paths and revision ranges through which the history of the new branch has ever passed:

/trunk:100-200
/branches/feature-branch:201

With each new revision added to the repository, the natural history—and thus, implicit mergeinfo—of the branch continues to expand to include those revisions until the day the branch is deleted. Here's what the implicit mergeinfo of our branch would look like when the `HEAD` revision of the repository had grown to 234:

/trunk:100-200
/branches/feature-branch:201-234

Implicit mergeinfo does not actually show up in the `svn:mergeinfo` property, but Subversion acts as if it does. This is why if you check out `^/branches/feature-branch` and then run **`svn merge ^/trunk -c 58`** in the resulting working copy, nothing happens. Subversion knows that the changes committed to `^/trunk` in revision 58 are already present in the target's natural history, so there's no need to try to merge them again. After all, avoiding repeated merges of changes *is* the primary goal of Subversion's merge tracking feature!

More on Merge Conflicts

Just like the **svn update** command, **svn merge** applies changes to your working copy. And therefore it's also capable of creating conflicts. The conflicts produced by **svn merge**, however, are sometimes different, and this section explains those differences.

To begin with, assume that your working copy has no local edits. When you **svn update** to a particular revision, the changes sent by the server always apply “cleanly” to your working copy. The server produces the delta by comparing two trees: a virtual snapshot of your working copy, and the revision tree you're interested in. Because the left hand side of the comparison is exactly equal to what you already have, the delta is guaranteed to correctly convert your working copy into the right hand tree.

But **svn merge** has no such guarantees and can be much more chaotic: the advanced user can ask the server to compare *any* two trees at all, even ones that are unrelated to the working copy! This means there's large potential for human error. Users will sometimes compare the wrong two trees, creating a delta that doesn't apply cleanly. The **svn merge** subcommand does its best to apply as much of the delta as possible, but some parts may be impossible. A common sign that you merged the wrong delta is unexpected tree conflicts:

```
$ svn merge -r 1288:1351 http://svn.example.com/myrepos/branch
--- Merging r1289 through r1351 into '.':
   C bar.c
   C foo.c
   C docs
--- Recording mergeinfo for merge of r1289 through r1351 into '.':
```

```
 U   .
Summary of conflicts:
  Tree conflicts: 3

$ svn st
!     C bar.c
      >   local missing, incoming edit upon merge
!     C foo.c
      >   local missing, incoming edit upon merge
!     C docs
      >   local delete, incoming edit upon merge
```

In the previous example, it might be the case that `bar.c`, `foo.c`, and `docs` all exist in both snapshots of the branch being compared. The resultant delta wants to change the contents of the corresponding paths in your working copy, but those paths don't exist in the working copy. Whatever the case, the preponderance of tree conflicts most likely means that the user compared the wrong two trees; it's a classic sign of user error. When this happens, it's easy to recursively revert all the changes created by the merge (**`svn revert . --recursive`**), delete any unversioned files or directories left behind after the revert, and rerun **svn merge** with the correct arguments.

Also keep in mind that a merge into a working copy with no local edits can still produce text conflicts.

```
$ svn merge -c 1701 http://svn.example.com/myrepos/branchX --accept postpone
--- Merging r1701 into '.':
C     glub.c
C     sputter.c
--- Recording mergeinfo for merge of r1701 into '.':
 U   .
Summary of conflicts:
  Text conflicts: 2

C:\SVN\src-branch-1.7.x>svn st
 M       .
?       glub.c.merge-left.r1700
?       glub.c.merge-right.r1701
C       glub.c
?       glub.c.working
?       sputter.c.merge-left.r1700
?       sputter.c.merge-right.r1701
C       sputter.c
?       sputter.c.working
Summary of conflicts:
  Text conflicts: 2
```

How can a conflict possibly happen? Again, because the user can request **svn merge** to define and apply any old delta to the working copy, that delta may contain textual changes that don't cleanly apply to a working file, even if the file has no local modifications.

Another small difference between **svn update** and **svn merge** is the names of the full-text files created when a conflict happens. In the section called “Resolve Any Conflicts”, we saw that an update produces files named `filename.mine`, `filename.rOLDREV`, and `filename.rNEWREV`. When **svn merge** produces a conflict, though, it creates three files named `filename.working`, `filename.merge-left.rOLDREV`, and `filename.merge-right.rNEWREV`. In this case, the terms “merge-left” and “merge-right” are describing which side of the double-tree comparison the file came from, “rOLDREV” describes the revision of the left side, and “rNEWREV” the revision of the right side. In any case, these differing names help you distinguish between conflicts that happened as a result of an update and ones that happened as a result of a merge.

Blocking Changes

Sometimes there's a particular changeset that you don't want automatically merged. For example, perhaps your team's policy is to do new development work on `/trunk`, but is more conservative about backporting changes to a stable branch you use for releasing to the public. On one extreme, you can manually cherrypick single changesets from the trunk to the branch—just the changes that are stable enough to pass muster. Maybe things aren't quite that strict, though; perhaps most of the time you just let **svn merge** automatically merge most changes from trunk to branch. In this case, you want a way to mask a few specific changes out, that is, prevent them from ever being automatically merged.

Through Subversion 1.7, the only way to block a changeset is to make the system believe that the change has *already* been merged. To do this, invoke the merge subcommand with the `--record-only` option:

```
$ cd my-calc-branch

$ svn propget svn:mergeinfo .
/trunk:1680-3305

# Let's make the metadata list r3328 as already merged.
$ svn merge -c 3328 --record-only ^/calc/trunk
--- Recording mergeinfo for merge of r3328 into '.':
 U   .

$ svn status
M       .

$ svn propget svn:mergeinfo .
/trunk:1680-3305,3328

$ svn commit -m "Block r3328 from being merged to the branch."
…
```

Beginning with Subversion 1.7, `--record-only` merges are transitive. This means that, in addition to recording mergeinfo describing the blocked revision(s), any `svn:mergeinfo` property differences in the merge source are also applied. For example, let's say we want to block the 'frazzle' feature from ever being merged from `^/trunk` to our `^/branches/proj-X` branch. We know that all the frazzle work was done on its own branch, which was reintegrated to `trunk` in revision 1055:

```
$ svn log -v ^/trunk -r 1055
------------------------------------------------------------------------
r1055 | francesca | 2011-09-22 07:40:06 -0400 (Thu, 22 Sep 2011) | 3 lines
Changed paths:
   M /trunk
   M /trunk/src/frazzle.c

Reintegrate the frazzle-feature-branch to trunk.
```

Because revision 1055 was a reintegrate merge we know that mergeinfo was recorded describing the merge:

```
$ svn diff ^/trunk -c 1055 --depth empty
Index: .
===================================================================
--- .   (revision 1054)
+++ .   (revision 1055)
```

```
Property changes on: .
___________________________________________________________________
Modified: svn:mergeinfo
   Merged /branches/frazzle-feature-branch:r997-1003
```

Now simply blocking merges of revision 1055 from `^/trunk` isn't foolproof since someone could merge r996:1003 directly from `^/branches/frazzle-feature-branch`. Fortunately the transitive nature of `--record-only` merges in Subversion 1.7 prevents this; the `--record-only` merge applies the `svn:mergeinfo` diff from revision 1055, thus blocking merges directly from the frazzle branch *and* as it has always done prior to Subversion 1.7, it blocks merges of revision 1055 directly from `^/trunk`:

```
$ cd branches/proj-X

$ svn merge ^/trunk . -c 1055 --record-only
--- Merging r1055 into '.':
 G   .
--- Recording mergeinfo for merge of r1055 into '.':
 G   .

$ svn diff --depth empty .
Index: .
===================================================================
--- .   (revision 1070)
+++ .   (working copy)

Property changes on: .
___________________________________________________________________
Modified: svn:mergeinfo
   Merged /trunk:r1055
   Merged /branches/frazzle-feature-branch:r997-1003
```

Blocking changes with `--record-only` works, but it's also a little bit dangerous. The main problem is that we're not clearly differentiating between the ideas of “I already have this change” and “I don't have this change, but don't currently want it.” We're effectively lying to the system, making it think that the change was previously merged. This puts the responsibility on you—the user—to remember that the change wasn't actually merged, it just wasn't wanted. There's no way to ask Subversion for a list of “blocked changelists.” If you want to track them (so that you can unblock them someday) you'll need to record them in a text file somewhere, or perhaps in an invented property.

Keeping a Reintegrated Branch Alive

There is an alternative to destroying and re-creating a branch after reintegration. To understand why it works you need to understand why the branch is initially unfit for further use after it has been reintegrated.

Let's assume you created your branch in revision *A*. While working on your branch, you created one or more revisions which made changes to the branch. Before reintegrating your branch back to trunk, you made a final merge from trunk to your branch, and committed the result of this merge as revision *B*.

When reintegrating your branch into the trunk, you create a new revision *X* which changes the trunk. The changes made to trunk in this revision *X* are semantically equivalent to the changes you made to your branch between revisions *A* and *B*.

If you now try to merge outstanding changes from trunk to your branch, Subversion will consider changes made in revision *X* as eligible for merging into the branch. However, since your branch already contains all the changes made in revision *X*, merging these changes can result in spurious conflicts! These conflicts are often tree conflicts, especially if renames were made on the

branch or the trunk while the branch was in development.

So what can be done about this? We need to make sure that Subversion does not try to merge revision *X* into the branch. This is done using the `--record-only` merge option, which was introduced in the section called "Blocking Changes".

To carry out the record-only merge, get a working copy of the branch which was just reintegrated in revision *X*, and merge just revision *X* from trunk into your branch, making sure to use the `--record-only` option.

This merge uses the cherry-picking merge syntax, which was introduced in the section called "Cherrypicking". Continuing with the running example from the section called "Reintegrating a Branch", where revision *X* was revision 391:

```
$ cd my-calc-branch
$ svn update
Updating '.':
Updated to revision 393.
$ svn merge --record-only -c 391 ^/calc/trunk
--- Recording mergeinfo for merge of r391 into '.':
 U   .
$ svn commit -m "Block revision 391 from being merged into my-calc-branch."
Sending        .

Committed revision 394.
```

Now your branch is ready to soak up changes from the trunk again. After another sync of your branch to the trunk, you can even reintegrate the branch a second time. If necessary, you can do another record-only merge to keep the branch alive. Rinse and repeat.

It should now also be apparent why deleting the branch and re-creating it has the same effect as doing the above record-only merge. Because revision *X* is part of the natural history (see the sidebar Natural History and Implicit Mergeinfo) of the newly created branch, Subversion will never attempt to merge revision *X* into the branch, avoiding spurious conflicts.

Merge-Sensitive Logs and Annotations

One of the main features of any version control system is to keep track of who changed what, and when they did it. The **svn log** and **svn blame** subcommands are just the tools for this: when invoked on individual files, they show not only the history of changesets that affected the file, but also exactly which user wrote which line of code, and when she did it.

When changes start getting replicated between branches, however, things start to get complicated. For example, if you were to ask **svn log** about the history of your feature branch, it would show exactly every revision that ever affected the branch:

```
$ cd my-calc-branch
$ svn log -q
------------------------------------------------------------------------
r390 | user | 2002-11-22 11:01:57 -0600 (Fri, 22 Nov 2002)
------------------------------------------------------------------------
r388 | user | 2002-11-21 05:20:00 -0600 (Thu, 21 Nov 2002)
------------------------------------------------------------------------
r381 | user | 2002-11-20 15:07:06 -0600 (Wed, 20 Nov 2002)
------------------------------------------------------------------------
r359 | user | 2002-11-19 19:19:20 -0600 (Tue, 19 Nov 2002)
------------------------------------------------------------------------
r357 | user | 2002-11-15 14:29:52 -0600 (Fri, 15 Nov 2002)
------------------------------------------------------------------------
r343 | user | 2002-11-07 13:50:10 -0600 (Thu, 07 Nov 2002)
------------------------------------------------------------------------
r341 | user | 2002-11-03 07:17:16 -0600 (Sun, 03 Nov 2002)
```

```
------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002)
------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002)
------------------------------------------------------------------------
```

But is this really an accurate picture of all the changes that happened on the branch? What's left out here is the fact that revisions 390, 381, and 357 were actually the results of merging changes from the trunk. If you look at one of these logs in detail, the multiple trunk changesets that comprised the branch change are nowhere to be seen:

```
$ svn log -v -r 390
------------------------------------------------------------------------
r390 | user | 2002-11-22 11:01:57 -0600 (Fri, 22 Nov 2002) | 1 line
Changed paths:
   M /branches/my-calc-branch/button.c
   M /branches/my-calc-branch/README

Final merge of trunk changes to my-calc-branch.
```

We happen to know that this merge to the branch was nothing but a merge of trunk changes. How can we see those trunk changes as well? The answer is to use the `--use-merge-history` (`-g`) option. This option expands those “child” changes that were part of the merge.

```
$ svn log -v -r 390 -g
------------------------------------------------------------------------
r390 | user | 2002-11-22 11:01:57 -0600 (Fri, 22 Nov 2002) | 1 line
Changed paths:
   M /branches/my-calc-branch/button.c
   M /branches/my-calc-branch/README

Final merge of trunk changes to my-calc-branch.
------------------------------------------------------------------------
r383 | sally | 2002-11-21 03:19:00 -0600 (Thu, 21 Nov 2002) | 2 lines
Changed paths:
   M /branches/my-calc-branch/button.c
Merged via: r390

Fix inverse graphic error on button.
------------------------------------------------------------------------
r382 | sally | 2002-11-20 16:57:06 -0600 (Wed, 20 Nov 2002) | 2 lines
Changed paths:
   M /branches/my-calc-branch/README
Merged via: r390

Document my last fix in README.
```

By making the log operation use merge history, we see not just the revision we queried (r390), but also the two revisions that came along on the ride with it—a couple of changes made by Sally to the trunk. This is a much more complete picture of history!

The **svn blame** command also takes the `--use-merge-history` (`-g`) option. If this option is neglected, somebody looking at a line-by-line annotation of `button.c` may get the mistaken impression that you were responsible for the lines that fixed a certain error:

```
$ svn blame button.c
…
   390    user    retval = inverse_func(button, path);
   390    user    return retval;
   390    user    }
…
```

And while it's true that you did actually commit those three lines in revision 390, two of them were actually written by Sally back in revision 383:

```
$ svn blame button.c -g
…
G    383    sally   retval = inverse_func(button, path);
G    383    sally   return retval;
     390    user    }
…
```

Now we know who to *really* blame for those two lines of code!

Noticing or Ignoring Ancestry

When conversing with a Subversion developer, you might very likely hear reference to the term *ancestry*. This word is used to describe the relationship between two objects in a repository: if they're related to each other, one object is said to be an ancestor of the other.

For example, suppose you commit revision 100, which includes a change to a file `foo.c`. Then `foo.c@99` is an “ancestor” of `foo.c@100`. On the other hand, suppose you commit the deletion of `foo.c` in revision 101, and then add a new file by the same name in revision 102. In this case, `foo.c@99` and `foo.c@102` may appear to be related (they have the same path), but in fact are completely different objects in the repository. They share no history or “ancestry.”

The reason for bringing this up is to point out an important difference between **svn diff** and **svn merge**. The former command ignores ancestry, while the latter command is quite sensitive to it. For example, if you asked **svn diff** to compare revisions 99 and 102 of `foo.c`, you would see line-based diffs; the **diff** command is blindly comparing two paths. But if you asked **svn merge** to compare the same two objects, it would notice that they're unrelated and first attempt to delete the old file, then add the new file; the output would indicate a deletion followed by an add:

```
D    foo.c
A    foo.c
```

Most merges involve comparing trees that are ancestrally related to one another; therefore, **svn merge** defaults to this behavior. Occasionally, however, you may want the **merge** command to compare two unrelated trees. For example, you may have imported two source-code trees representing different vendor releases of a software project (see the section called “Vendor Branches”). If you ask **svn merge** to compare the two trees, you'd see the entire first tree being deleted, followed by an add of the entire second tree! In these situations, you'll want **svn merge** to do a path-based comparison only, ignoring any relations between files and directories. Add the `--ignore-ancestry` option to your **merge** command, and it will behave just like **svn diff**. (And conversely, the `--notice-ancestry` option will cause **svn diff** to behave like the **svn merge** command.)

The `--ignore-ancestry` option also disables Merge Tracking. This means that `svn:mergeinfo` is not considered when **svn merge** is determining what revisions to merge, nor is `svn:mergeinfo` recorded to describe the merge.

Merges and Moves

A common desire is to refactor source code, especially in Java-based software projects. Files and directories are shuffled around and renamed, often causing great disruption to everyone working on the project. Sounds like a perfect case to use a branch, doesn't it? Just create a branch, shuffle things around, and then merge the branch back to the trunk, right?

Alas, this scenario doesn't work so well right now and is considered one of Subversion's current weak spots. The problem is that Subversion's **svn update** command isn't as robust as it should be, particularly when dealing with copy and move operations.

When you use **svn copy** to duplicate a file, the repository remembers where the new file came from, but it fails to transmit that information to the client which is running **svn update** or **svn merge**. Instead of telling the client, “Copy that file you already have to this new location,” it sends down an entirely new file. This can lead to problems, especially because the same thing happens with renamed files. A lesser-known fact about Subversion is that it lacks “true renames”—the **svn move** command is nothing more than an aggregation of **svn copy** and **svn delete**.

For example, suppose that while working on your private branch, you rename `integer.c` to `whole.c`. Effectively you've created a new file in your branch that is a copy of the original file, and deleted the original file. Meanwhile, back on `trunk`, Sally has committed some improvements to `integer.c`. Now you decide to merge your branch to the trunk:

```
$ cd calc/trunk

$ svn merge --reintegrate ^/calc/branches/my-calc-branch
--- Merging differences between repository URLs into '.':
D    integer.c
A    whole.c
U    .
--- Recording mergeinfo for merge between repository URLs into '.':
 U   .
```

This doesn't look so bad at first glance, but it's also probably not what you or Sally expected. The merge operation has deleted the latest version of the `integer.c` file (the one containing Sally's latest changes), and blindly added your new `whole.c` file—which is a duplicate of the *older* version of `integer.c`. The net effect is that merging your “rename” to the trunk has removed Sally's recent changes from the latest revision!

This isn't true data loss. Sally's changes are still in the repository's history, but it may not be immediately obvious that this has happened. The moral of this story is that until Subversion improves, be very careful about merging copies and renames from one branch to another.

Blocking Merge-Unaware Clients

If you've just upgraded your server to Subversion 1.5 or later, there's a risk that pre-1.5 Subversion clients can cause problems with Merge Tracking. This is because pre-1.5 clients don't support this feature; when one of these older clients performs **svn merge**, it doesn't modify the value of the `svn:mergeinfo` property at all. So the subsequent commit, despite being the result of a merge, doesn't tell the repository about the duplicated changes—that information is lost. Later on, when “merge-aware” clients attempt automatic merging, they're likely to run into all sorts of conflicts resulting from repeated merges.

If you and your team are relying on the merge-tracking features of Subversion, you may want to configure your repository to prevent older clients from committing changes. The easy way to do this is by inspecting the “capabilities” parameter in the `start-`

`commit` hook script. If the client reports itself as having `mergeinfo` capabilities, the hook script can allow the commit to start. If the client doesn't report that capability, have the hook deny the commit. Example 4.1, “Merge-tracking gatekeeper start-commit hook script” gives an example of such a hook script:

Example 4.1. Merge-tracking gatekeeper start-commit hook script

```
#!/usr/bin/env python
import sys

# The start-commit hook is invoked before a Subversion txn is created
# in the process of doing a commit.  Subversion runs this hook
# by invoking a program (script, executable, binary, etc.) named
# 'start-commit' (for which this file is a template)
# with the following ordered arguments:
#
#   [1] REPOS-PATH   (the path to this repository)
#   [2] USER         (the authenticated user attempting to commit)
#   [3] CAPABILITIES (a colon-separated list of capabilities reported
#                     by the client; see note below)

capabilities = sys.argv[3].split(':')
if "mergeinfo" not in capabilities:
  sys.stderr.write("Commits from merge-tracking-unaware clients are "
                   "not permitted.  Please upgrade to Subversion 1.5 "
                   "or newer.\n")
  sys.exit(1)
sys.exit(0)
```

For more information about hook scripts, see the section called “Implementing Repository Hooks”.

The Final Word on Merge Tracking

The bottom line is that Subversion's merge-tracking feature has an extremely complex internal implementation, and the `svn:mergeinfo` property is the only window the user has into the machinery.

Sometimes mergeinfo will appear on paths that you didn't expect to be touched by an operation. Sometimes mergeinfo won't be generated at all, when you expect it to. Furthermore, the management of mergeinfo metadata has a whole set of taxonomies and behaviors around it, such as “explicit” versus “implicit” mergeinfo, “operative” versus “inoperative” revisions, specific mechanisms of mergeinfo “elision,” and even “inheritance” from parent to child directories.

We've chosen to only briefly cover, if at all, these detailed topics for a couple of reasons. First, the level of detail is absolutely overwhelming for a typical user. Second, and more importantly, the typical user *shouldn't* have to understand these concepts; they should typically remain in the background as pesky implementation details. All that said, if you enjoy this sort of thing, you can get a fantastic overview in a paper posted at CollabNet's website: http://www.collab.net/community/subversion/articles/merge-info.html.

For now, if you want to steer clear of the complexities of merge tracking, we recommend that you follow these simple best practices:

- For short-term feature branches, follow the simple procedure described throughout the section called “Basic Merging”.
- Avoid subtree merges and subtree mergeinfo, perform merges only on the root of your branches, not on subdirectories or files (see Subtree Merges and Subtree Mergeinfo) .
- Don't ever edit the `svn:mergeinfo` property directly; use **svn merge** with the `--record-only` option to effect a desired

change to the metadata (as demonstrated in the section called “Blocking Changes”).

- Your merge target should be a working copy which represents the root of a *complete* tree representing a *single* location in the repository at a single point in time:
  - Don't use the `--allow-mixed-revisions` option to merge into mixed-revision working copies.
  - Don't merge to targets with “switched” subdirectories (as described next in the section called “Traversing Branches”).
  - Avoid merges to targets with sparse directories. Likewise, don't merge to depths other than `--depth=infinity`
  - Be sure you have read access to all of the merge source and read/write access to all of the merge target.

Traversing Branches

The **svn switch** command transforms an existing working copy to reflect a different branch. While this command isn't strictly necessary for working with branches, it provides a nice shortcut. In one of our earlier examples, after creating your private branch, you checked out a fresh working copy of the new repository directory. Instead, you can simply ask Subversion to change your working copy of `/calc/trunk` to mirror the new branch location:

```
$ cd calc

$ svn info | grep URL
URL: http://svn.example.com/repos/calc/trunk

$ svn switch ^/calc/branches/my-calc-branch
U    integer.c
U    button.c
U    Makefile
Updated to revision 341.

$ svn info | grep URL
URL: http://svn.example.com/repos/calc/branches/my-calc-branch
```

“Switching” a working copy that has no local modifications to a different branch results in the working copy looking just as it would if you'd done a fresh checkout of the directory. It's usually more efficient to use this command, because often branches differ by only a small degree. The server sends only the minimal set of changes necessary to make your working copy reflect the branch directory.

The **svn switch** command also takes a `--revision` (`-r`) option, so you need not always move your working copy to the `HEAD` of the branch.

Of course, most projects are more complicated than our `calc` example, and contain multiple subdirectories. Subversion users often follow a specific algorithm when using branches:

1. Copy the project's entire “trunk” to a new branch directory.
2. Switch only *part* of the trunk working copy to mirror the branch.

In other words, if a user knows that the branch work needs to happen on only a specific subdirectory, she uses **svn switch** to move only that subdirectory to the branch. (Or sometimes users will switch just a single working file to the branch!) That way, the user can continue to receive normal “trunk” updates to most of her working copy, but the switched portions will remain immune (unless someone commits a change to her branch). This feature adds a whole new dimension to the concept of a “mixed working

—not only can working copies contain a mixture of working revisions, but they can also contain a mixture of repository locations as well.

Typically switched subdirectories share common ancestry with the location which is switched “away” from. However **svn switch** can switch a subdirectory to mirror a repository location which it shares no common ancestry with. To do this you need to use the `--ignore-ancestry` option.

If your working copy contains a number of switched subtrees from different repository locations, it continues to function as normal. When you update, you'll receive patches to each subtree as appropriate. When you commit, your local changes are still applied as a single, atomic change to the repository.

Note that while it's okay for your working copy to reflect a mixture of repository locations, these locations must all be within the *same* repository. Subversion repositories aren't yet able to communicate with one another; that feature is planned for the future.[10]

Administrators who need to change the URL of a repository which is accessed via HTTP are encouraged to add to their `httpd.conf` configuration file a permanent redirect from the old URL location to the new one (via the `RedirectPermanent` directive). Subversion clients will generally display the new repository URL in error messages generated when the user attempts to use working copies which still reflect the old URL location. In fact, Subversion 1.7 clients will go a step further, automatically relocating the working copy to the new URL.

Switches and Updates

Have you noticed that the output of **svn switch** and **svn update** looks the same? The switch command is actually a superset of the update command.

When you run **svn update**, you're asking the repository to compare two trees. The repository does so, and then sends a description of the differences back to the client. The only difference between **svn switch** and **svn update** is that the latter command always compares two identical repository paths.

That is, if your working copy is a mirror of `/calc/trunk`, **svn update** will automatically compare your working copy of `/calc/trunk` to `/calc/trunk` in the `HEAD` revision. If you're switching your working copy to a branch, **svn switch** will compare your working copy of `/calc/trunk` to some *other* branch directory in the `HEAD` revision.

In other words, an update moves your working copy through time. A switch moves your working copy through time *and* space.

Because **svn switch** is essentially a variant of **svn update**, it shares the same behaviors; any local modifications in your working copy are preserved when new data arrives from the repository.

Have you ever found yourself making some complex edits (in your `/trunk` working copy) and suddenly realized, “Hey, these changes ought to be in their own branch?” There is a great two step technique to do this:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/newbranch \
           -m "Create branch 'newbranch'."
Committed revision 353.
$ svn switch ^/calc/branches/newbranch
At revision 353.
```

[10] You *can*, however, use **svn relocate** if the URL of your server changes and you don't want to abandon an existing working copy. See svn relocate in Chapter 9, *Subversion Complete Reference* for more information and an example.

The **svn switch** command, like **svn update**, preserves your local edits. At this point, your working copy is now a reflection of the newly created branch, and your next **svn commit** invocation will send your changes there.

Tags

Another common version control concept is a tag. A tag is just a “snapshot” of a project in time. In Subversion, this idea already seems to be everywhere. Each repository revision is exactly that—a snapshot of the filesystem after each commit.

However, people often want to give more human-friendly names to tags, such as `release-1.0`. And they want to make snapshots of smaller subdirectories of the filesystem. After all, it's not so easy to remember that release 1.0 of a piece of software is a particular subdirectory of revision 4822.

Creating a Simple Tag

Once again, **svn copy** comes to the rescue. If you want to create a snapshot of `/calc/trunk` exactly as it looks in the `HEAD` revision, make a copy of it:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/tags/release-1.0 \
           -m "Tagging the 1.0 release of the 'calc' project."

Committed revision 902.
```

This example assumes that a `/calc/tags` directory already exists. (If it doesn't, you can create it using **svn mkdir**.) After the copy completes, the new `release-1.0` directory is forever a snapshot of how the `/trunk` directory looked in the `HEAD` revision at the time you made the copy. Of course, you might want to be more precise about exactly which revision you copy, in case somebody else may have committed changes to the project when you weren't looking. So if you know that revision 901 of `/calc/trunk` is exactly the snapshot you want, you can specify it by passing `-r 901` to the **svn copy** command.

But wait a moment: isn't this tag creation procedure the same procedure we used to create a branch? Yes, in fact, it is. In Subversion, there's no difference between a tag and a branch. Both are just ordinary directories that are created by copying. Just as with branches, the only reason a copied directory is a “tag” is because *humans* have decided to treat it that way: as long as nobody ever commits to the directory, it forever remains a snapshot. If people start committing to it, it becomes a branch.

If you are administering a repository, there are two approaches you can take to managing tags. The first approach is “hands off”: as a matter of project policy, decide where your tags will live, and make sure all users know how to treat the directories they copy. (That is, make sure they know not to commit to them.) The second approach is more paranoid: you can use one of the access control scripts provided with Subversion to prevent anyone from doing anything but creating new copies in the tags area (see Chapter 6, *Server Configuration*). The paranoid approach, however, isn't usually necessary. If a user accidentally commits a change to a tag directory, you can simply undo the change as discussed in the previous section. This is version control, after all!

Creating a Complex Tag

Sometimes you may want a “snapshot” that is more complicated than a single directory at a single revision.

For example, pretend your project is much larger than our `calc` example: suppose it contains a number of subdirectories and many more files. In the course of your work, you may decide that you need to create a working copy that is designed to have specific features and bug fixes. You can accomplish this by selectively backdating files or directories to particular revisions (using **svn update** with the `-r` option liberally), by switching files and directories to particular branches (making use of **svn switch**), or even just by making a bunch of local changes. When you're done, your working copy is a hodgepodge of repository locations from different revisions. But after testing, you know it's the precise combination of data you need to tag.

Time to make a snapshot. Copying one URL to another won't work here. In this case, you want to make a snapshot of your exact working copy arrangement and store it in the repository. Luckily, **svn copy** actually has four different uses (which you can read about in Chapter 9, *Subversion Complete Reference*), including the ability to copy a working copy tree to the repository:

```
$ ls
my-working-copy/

$ svn copy my-working-copy \
           http://svn.example.com/repos/calc/tags/mytag \
           -m "Tag my existing working copy state."

Committed revision 940.
```

Now there is a new directory in the repository, `/calc/tags/mytag`, which is an exact snapshot of your working copy—mixed revisions, URLs, local changes, and all.

Other users have found interesting uses for this feature. Sometimes there are situations where you have a bunch of local changes made to your working copy, and you'd like a collaborator to see them. Instead of running **svn diff** and sending a patch file (which won't capture directory or symlink changes), you can use **svn copy** to “upload” your working copy to a private area of the repository. Your collaborator can then either check out a verbatim copy of your working copy or use **svn merge** to receive your exact changes.

While this is a nice method for uploading a quick snapshot of your working copy, note that this is *not* a good way to initially create a branch. Branch creation should be an event unto itself, and this method conflates the creation of a branch with extra changes to files, all within a single revision. This makes it very difficult (later on) to identify a single revision number as a branch point.

Branch Maintenance

You may have noticed by now that Subversion is extremely flexible. Because it implements branches and tags with the same underlying mechanism (directory copies), and because branches and tags appear in normal filesystem space, many people find Subversion intimidating. It's almost *too* flexible. In this section, we'll offer some suggestions for arranging and managing your data over time.

Repository Layout

There are some standard, recommended ways to organize the contents of a repository. Most people create a `trunk` directory to hold the “main line” of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies. If a repository holds only one project, often people create these top-level directories:

```
/
  trunk/
  branches/
  tags/
```

If a repository contains multiple projects, admins typically index their layout by project. See the section called “Planning Your Repository Organization” to read more about “project roots”, but here's an example of such a layout:

```
/
```

```
paint/
  trunk/
  branches/
  tags/
calc/
  trunk/
  branches/
  tags/
```

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at any time. Because branches and tags are ordinary directories, the **svn move** command can move or rename them however you wish. Switching from one layout to another is just a matter of issuing a series of server-side moves; if you don't like the way things are organized in the repository, just juggle the directories around.

Remember, though, that while moving directories is easy to do, you need to be considerate of other users as well. Your juggling can disorient users with existing working copies. If a user has a working copy of a particular repository directory and your **svn move** subcommand removes the path from the latest revision, then when the user next runs **svn update**, she is told that her working copy represents a path that no longer exists. She is then forced to **svn switch** to the new location.

Data Lifetimes

Another nice feature of Subversion's model is that branches and tags can have finite lifetimes, just like any other versioned item. For example, suppose you eventually finish all your work on your personal branch of the `calc` project. After merging all of your changes back into `/calc/trunk`, there's no need for your private branch directory to stick around anymore:

```
$ svn delete http://svn.example.com/repos/calc/branches/my-calc-branch \
             -m "Removing obsolete branch of calc project."

Committed revision 375.
```

And now your branch is gone. Of course, it's not really gone: the directory is simply missing from the `HEAD` revision, no longer distracting anyone. If you use **svn checkout**, **svn switch**, or **svn list** to examine an earlier revision, you can still see your old branch.

If browsing your deleted directory isn't enough, you can always bring it back. Resurrecting data is very easy in Subversion. If there's a deleted directory (or file) that you'd like to bring back into `HEAD`, simply use **svn copy** to copy it from the old revision:

```
$ svn copy http://svn.example.com/repos/calc/branches/my-calc-branch@374 \
           http://svn.example.com/repos/calc/branches/my-calc-branch \
           -m "Restore my-calc-branch."

Committed revision 376.
```

In our example, your personal branch had a relatively short lifetime: you may have created it to fix a bug or implement a new feature. When your task is done, so is the branch. In software development, though, it's also common to have two “main” branches running side by side for very long periods. For example, suppose it's time to release a stable version of the `calc` project to the public, and you know it's going to take a couple of months to shake bugs out of the software. You don't want people to add new

features to the project, but you don't want to tell all developers to stop programming either. So instead, you create a “stable” branch of the software that won't change much:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/branches/stable-1.0 \
           -m "Creating stable branch of calc project."

Committed revision 377.
```

And now developers are free to continue adding cutting-edge (or experimental) features to `/calc/trunk`, and you can declare a project policy that only bug fixes are to be committed to `/calc/branches/stable-1.0`. That is, as people continue to work on the trunk, a human selectively ports bug fixes over to the stable branch. Even after the stable branch has shipped, you'll probably continue to maintain the branch for a long time—that is, as long as you continue to support that release for customers. We'll discuss this more in the next section.

Common Branching Patterns

There are many different uses for branching and **svn merge**, and this section describes the most common.

Version control is most often used for software development, so here's a quick peek at two of the most common branching/merging patterns used by teams of programmers. If you're not using Subversion for software development, feel free to skip this section. If you're a software developer using version control for the first time, pay close attention, as these patterns are often considered best practices by experienced folk. These processes aren't specific to Subversion; they're applicable to any version control system. Still, it may help to see them described in Subversion terms.

Release Branches

Most software has a typical life cycle: code, test, release, repeat. There are two problems with this process. First, developers need to keep writing new features while quality assurance teams take time to test supposedly stable versions of the software. New work cannot halt while the software is tested. Second, the team almost always needs to support older, released versions of software; if a bug is discovered in the latest code, it most likely exists in released versions as well, and customers will want to get that bug fix without having to wait for a major new release.

Here's where version control can help. The typical procedure looks like this:

1. *Developers commit all new work to the trunk.* Day-to-day changes are committed to `/trunk`: new features, bug fixes, and so on.

2. *The trunk is copied to a “release” branch.* When the team thinks the software is ready for release (say, a 1.0 release), `/trunk` might be copied to `/branches/1.0`.

3. *Teams continue to work in parallel.* One team begins rigorous testing of the release branch, while another team continues new work (say, for version 2.0) on `/trunk`. If bugs are discovered in either location, fixes are ported back and forth as necessary. At some point, however, even that process stops. The branch is “frozen” for final testing right before a release.

4. *The branch is tagged and released.* When testing is complete, `/branches/1.0` is copied to `/tags/1.0.0` as a reference snapshot. The tag is packaged and released to customers.

5. *The branch is maintained over time.* While work continues on `/trunk` for version 2.0, bug fixes continue to be ported from `/trunk` to `/branches/1.0`. When enough bug fixes have accumulated, management may decide to do a 1.0.1 release: `/branches/1.0` is copied to `/tags/1.0.1`, and the tag is packaged and released.

This entire process repeats as the software matures: when the 2.0 work is complete, a new 2.0 release branch is created, tested,

tagged, and eventually released. After some years, the repository ends up with a number of release branches in “maintenance” mode, and a number of tags representing final shipped versions.

Feature Branches

A *feature branch* is the sort of branch that's been the dominant example in this chapter (the one you've been working on while Sally continues to work on `/trunk`). It's a temporary branch created to work on a complex change without interfering with the stability of `/trunk`. Unlike release branches (which may need to be supported forever), feature branches are born, used for a while, merged back to the trunk, and then ultimately deleted. They have a finite span of usefulness.

Again, project policies vary widely concerning exactly when it's appropriate to create a feature branch. Some projects never use feature branches at all: commits to `/trunk` are a free-for-all. The advantage to this system is that it's simple—nobody needs to learn about branching or merging. The disadvantage is that the trunk code is often unstable or unusable. Other projects use branches to an extreme: no change is *ever* committed to the trunk directly. Even the most trivial changes are created on a short-lived branch, carefully reviewed, and merged to the trunk. Then the branch is deleted. This system guarantees an exceptionally stable and usable trunk at all times, but at the cost of tremendous process overhead.

Most projects take a middle-of-the-road approach. They commonly insist that `/trunk` compile and pass regression tests at all times. A feature branch is required only when a change requires a large number of destabilizing commits. A good rule of thumb is to ask this question: if the developer worked for days in isolation and then committed the large change all at once (so that `/trunk` were never destabilized), would it be too large a change to review? If the answer to that question is “yes,” the change should be developed on a feature branch. As the developer commits incremental changes to the branch, they can be easily reviewed by peers.

Finally, there's the issue of how to best keep a feature branch in “sync” with the trunk as work progresses. As we mentioned earlier, there's a great risk to working on a branch for weeks or months; trunk changes may continue to pour in, to the point where the two lines of development differ so greatly that it may become a nightmare trying to merge the branch back to the trunk.

This situation is best avoided by regularly merging trunk changes to the branch. Make up a policy: once a week, merge the last week's worth of trunk changes to the branch.

When you are eventually ready to merge the “synchronized” feature branch back to the trunk, begin by doing a final merge of the latest trunk changes to the branch. When that's done, the latest versions of branch and trunk are absolutely identical except for your branch changes. You then merge back with the `--reintegrate` option:

```
$ cd trunk-working-copy

$ svn update
Updating '.':
At revision 1910.

$ svn merge --reintegrate ^/calc/branches/mybranch
--- Merging differences between repository URLs into '.':
U    real.c
U    integer.c
A    newdirectory
A    newdirectory/newfile
 U   .
…
```

Another way of thinking about this pattern is that your weekly sync of trunk to branch is analogous to running **svn update** in a working copy, while the final merge step is analogous to running **svn commit** from a working copy. After all, what else *is* a working copy but a very shallow private branch? It's a branch that's capable of storing only one change at a time.

Vendor Branches

As is especially the case when developing software, the data that you maintain under version control is often closely related to, or perhaps dependent upon, someone else's data. Generally, the needs of your project will dictate that you stay as up to date as possible with the data provided by that external entity without sacrificing the stability of your own project. This scenario plays itself out all the time—anywhere that the information generated by one group of people has a direct effect on that which is generated by another group.

For example, software developers might be working on an application that makes use of a third-party library. Subversion has just such a relationship with the Apache Portable Runtime (APR) library (see the section called “The Apache Portable Runtime Library”). The Subversion source code depends on the APR library for all its portability needs. In earlier stages of Subversion's development, the project closely tracked APR's changing API, always sticking to the “bleeding edge” of the library's code churn. Now that both APR and Subversion have matured, Subversion attempts to synchronize with APR's library API only at well-tested, stable release points.

Now, if your project depends on someone else's information, you could attempt to synchronize that information with your own in several ways. Most painfully, you could issue oral or written instructions to all the contributors of your project, telling them to make sure they have the specific versions of that third-party information that your project needs. If the third-party information is maintained in a Subversion repository, you could also use Subversion's externals definitions to effectively “pin down” specific versions of that information to some location in your own working copy (see the section called “Externals Definitions”).

But sometimes you want to maintain custom modifications to third-party code in your own version control system. Returning to the software development example, programmers might need to make modifications to that third-party library for their own purposes. These modifications might include new functionality or bug fixes, maintained internally only until they become part of an official release of the third-party library. Or the changes might never be relayed back to the library maintainers, existing solely as custom tweaks to make the library further suit the needs of the software developers.

Now you face an interesting situation. Your project could house its custom modifications to the third-party data in some disjointed fashion, such as using patch files or full-fledged alternative versions of files and directories. But these quickly become maintenance headaches, requiring some mechanism by which to apply your custom changes to the third-party code and necessitating regeneration of those changes with each successive version of the third-party code that you track.

The solution to this problem is to use *vendor branches*. A vendor branch is a directory tree in your own version control system that contains information provided by a third-party entity, or vendor. Each version of the vendor's data that you decide to absorb into your project is called a *vendor drop*.

Vendor branches provide two benefits. First, by storing the currently supported vendor drop in your own version control system, you ensure that the members of your project never need to question whether they have the right version of the vendor's data. They simply receive that correct version as part of their regular working copy updates. Second, because the data lives in your own Subversion repository, you can store your custom changes to it in-place—you have no more need of an automated (or worse, manual) method for swapping in your customizations.

General Vendor Branch Management Procedure

Managing vendor branches generally works like this: first, you create a top-level directory (such as `/vendor`) to hold the vendor branches. Then you import the third-party code into a subdirectory of that top-level directory. You then copy that subdirectory into your main development branch (e.g., `/trunk`) at the appropriate location. You always make your local changes in the main development branch. With each new release of the code you are tracking, you bring it into the vendor branch and merge the changes into `/trunk`, resolving whatever conflicts occur between your local changes and the upstream changes.

An example will help to clarify this algorithm. We'll use a scenario where your development team is creating a calculator program that links against a third-party complex number arithmetic library, libcomplex. We'll begin with the initial creation of the vendor branch and the import of the first vendor drop. We'll call our vendor branch directory `libcomplex`, and our code drops will go into a subdirectory of our vendor branch called `current`. And since **svn import** creates all the intermediate parent directories it needs, we can actually accomplish both of these steps with a single command:

```
$ svn import /path/to/libcomplex-1.0 \
             http://svn.example.com/repos/vendor/libcomplex/current \
```

```
             -m "importing initial 1.0 vendor drop"
…
```

We now have the current version of the libcomplex source code in `/vendor/libcomplex/current`. Now, we tag that version (see the section called "Tags") and then copy it into the main development branch. Our copy will create a new directory called `libcomplex` in our existing `calc` project directory. It is in this copied version of the vendor data that we will make our customizations:

```
$ svn copy http://svn.example.com/repos/vendor/libcomplex/current  \
           http://svn.example.com/repos/vendor/libcomplex/1.0      \
           -m "tagging libcomplex-1.0"
…
$ svn copy http://svn.example.com/repos/vendor/libcomplex/1.0  \
           http://svn.example.com/repos/calc/libcomplex        \
           -m "bringing libcomplex-1.0 into the main branch"
…
```

We check out our project's main branch—which now includes a copy of the first vendor drop—and we get to work customizing the libcomplex code. Before we know it, our modified version of libcomplex is now completely integrated into our calculator program.[11]

A few weeks later, the developers of libcomplex release a new version of their library—version 1.1—which contains some features and functionality that we really want. We'd like to upgrade to this new version, but without losing the customizations we made to the existing version. What we essentially would like to do is to replace our current baseline version of libcomplex 1.0 with a copy of libcomplex 1.1, and then re-apply the custom modifications we previously made to that library to the new version. But we actually approach the problem from the other direction, applying the changes made to libcomplex between versions 1.0 and 1.1 to our modified copy of it.

To perform this upgrade, we check out a copy of our vendor branch and replace the code in the `current` directory with the new libcomplex 1.1 source code. We quite literally copy new files on top of existing files, perhaps exploding the libcomplex 1.1 release tarball atop our existing files and directories. The goal here is to make our `current` directory contain only the libcomplex 1.1 code and to ensure that all that code is under version control. Oh, and we want to do this with as little version control history disturbance as possible.

After replacing the 1.0 code with 1.1 code, **svn status** will show files with local modifications as well as, perhaps, some unversioned files. If we did what we were supposed to do, the unversioned files are only those new files introduced in the 1.1 release of libcomplex—we run **svn add** on those to get them under version control. If the 1.1 code no longer has certain files that were in the 1.0 tree, it may be hard to notice them; you'd have to compare the two trees with some external tool and then **svn delete** any files present in 1.0 but not in 1.1. (Although it might also be just fine to let these same files live on in unused obscurity!) Finally, once our `current` working copy contains only the libcomplex 1.1 code, we commit the changes we made to get it looking that way.

Our `current` branch now contains the new vendor drop. We tag the new version as 1.1 (in the same way we previously tagged the version 1.0 vendor drop), and then merge the differences between the tag of the previous version and the new current version into our main development branch:

```
$ cd working-copies/calc
$ svn merge ^/vendor/libcomplex/1.0      \
            ^/vendor/libcomplex/current  \
            libcomplex
… # resolve all the conflicts between their changes and our changes
```

[11] And is entirely bug-free, of course!

```
$ svn commit -m "merging libcomplex-1.1 into the main branch"
…
```

In the trivial use case, the new version of our third-party tool would look, from a files-and-directories point of view, just like the previous version. None of the libcomplex source files would have been deleted, renamed, or moved to different locations—the new version would contain only textual modifications against the previous one. In a perfect world, our modifications would apply cleanly to the new version of the library, with absolutely no complications or conflicts.

But things aren't always that simple, and in fact it is quite common for source files to get moved around between releases of software. This complicates the process of ensuring that our modifications are still valid for the new version of code, and things can quickly degrade into a situation where we have to manually re-create our customizations in the new version. Once Subversion knows about the history of a given source file—including all its previous locations—the process of merging in the new version of the library is pretty simple. But we are responsible for telling Subversion how the source file layout changed from vendor drop to vendor drop.

svn_load_dirs.pl

Vendor drops that contain more than a few deletes, additions, and moves complicate the process of upgrading to each successive version of the third-party data. So Subversion supplies the **svn_load_dirs.pl** script to assist with this process. This script automates the importing steps we mentioned in the general vendor branch management procedure to make sure mistakes are minimized. You will still be responsible for using the merge commands to merge the new versions of the third-party data into your main development branch, but **svn_load_dirs.pl** can help you more quickly and easily arrive at that stage.

In short, **svn_load_dirs.pl** is an enhancement to **svn import** that has several important characteristics:

- It can be run at any point in time to bring an existing directory in the repository to exactly match an external directory, performing all the necessary adds and deletes, and optionally performing moves, too.
- It takes care of complicated series of operations between which Subversion requires an intermediate commit—such as before renaming a file or directory twice.
- It will optionally tag the newly imported directory.
- It will optionally add arbitrary properties to files and directories that match a regular expression.

svn_load_dirs.pl takes three mandatory arguments. The first argument is the URL to the base Subversion directory to work in. This argument is followed by the URL—relative to the first argument—into which the current vendor drop will be imported. Finally, the third argument is the local directory to import. Using our previous example, a typical run of **svn_load_dirs.pl** might look like this:

```
$ svn_load_dirs.pl http://svn.example.com/repos/vendor/libcomplex \
                   current                                         \
                   /path/to/libcomplex-1.1
…
```

You can indicate that you'd like **svn_load_dirs.pl** to tag the new vendor drop by passing the `-t` command-line option and specifying a tag name. This tag is another URL relative to the first program argument.

```
$ svn_load_dirs.pl -t libcomplex-1.1                              \
                   http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

When you run **svn_load_dirs.pl**, it examines the contents of your existing “current” vendor drop and compares them with the proposed new vendor drop. In the trivial case, no files will be in one version and not the other, and the script will perform the new import without incident. If, however, there are discrepancies in the file layouts between versions, **svn_load_dirs.pl** will ask you how to resolve those differences. For example, you will have the opportunity to tell the script that you know that the file `math.c` in version 1.0 of libcomplex was renamed to `arithmetic.c` in libcomplex 1.1. Any discrepancies not explained by moves are treated as regular additions and deletions.

The script also accepts a separate configuration file for setting properties on *added* files and directories which match a regular expression. This configuration file is specified to **svn_load_dirs.pl** using the `-p` command-line option. Each line of the configuration file is a whitespace-delimited set of two or four values: a Perl-style regular expression against which to match the added path, a control keyword (either `break` or `cont`), and then optionally a property name and value.

```
\.png$              break   svn:mime-type   image/png
\.jpe?g$            break   svn:mime-type   image/jpeg
\.m3u$              cont    svn:mime-type   audio/x-mpegurl
\.m3u$              break   svn:eol-style   LF
.*                  break   svn:eol-style   native
```

For each added path, the configured property changes whose regular expression matches the path are applied in order, unless the control specification is `break` (which means that no more property changes should be applied to that path). If the control specification is `cont`—an abbreviation for `continue`—matching will continue with the next line of the configuration file.

Any whitespace in the regular expression, property name, or property value must be surrounded by either single or double quotes. You can escape quotes that are not used for wrapping whitespace by preceding them with a backslash (\) character. The backslash escapes only quotes when parsing the configuration file, so do not protect any other characters beyond what is necessary for the regular expression.

To Branch or Not to Branch?

To branch or not to branch—that is an interesting question. This chapter has provided thus far a pretty deep dive into the waters of branching and merging, topics which have historically been the premier source of Subversion user confusion. As if the rote actions involved in branching and branch management aren't sometimes tricky enough, some users get hung up on deciding whether they need to branch at all. As you've learned, Subversion can handle common branching and branch management scenarios. So, the decision of whether or not to branch a project's history is rarely a technical one. Rather, the social impact of the decision often carries more weight. Let's examine some of the benefits and costs of using branches in a software project.

The most obvious benefit of working on a branch is isolation. Changes made to the branch don't affect the other lines of development in the project; changes made to those other lines don't affect the branch. In this way, a branch can serve as a great place to experiment with new features, complex bug fixes, major code rewrites, and so on. No matter how much stuff Sally breaks on her branch, Harry and the rest of the team can continue with their work unhindered outside the branch.

Branches also provide a great way to organize related changes into readily identifiable collections. For example, the changes which comprise the complete solution to a particular bug might be a list of non-sequential revision numbers. You might describe them in human language as “revisions 1534, 1543, 1587 and 1588”. You'd probably reproduce those numbers manually (or otherwise) in the issue tracker artifact which tracks the bug. When porting the bug fix to other product versions, you'd need to make sure to port all those revisions. But had those changes been made on a unique branch, you'd find yourself referring only to that branch by its

name in conversation, in issue tracker comments, and when porting changes.

The unfortunate downside of branches, though, is that the very isolation that makes them so useful *can* be at odds with the collaborative needs of the project team. Depending on the work habits of your project peers, changes made to branches might not get the kind of constructive review, criticism, and testing that changes made to the main line of development do. The isolation of a branch can encourage users to forsake certain version control “best practices”, leading to version history which is difficult to review *post facto*. Developers on long-lived branches sometimes need to work extra hard to ensure that the evolutionary direction of their isolated copy of the codebase is in harmony with the direction their peers are steering the main code lines. Now, these drawbacks might be less of an issue for true exploratory branches aimed at experimenting with the future of a codebase with no expectation of reintegrating the results back into the main development lines—mere policy needn't be a vision-killer! But the simple fact remains that projects generally benefit from an orderly approach to version control where code and code changes enjoy the review and comprehension of more than one team member.

That's not to say that there are no technical penalties to branching. Pardon us while we “go meta” for a bit here. If you think about it, every time you checkout a Subversion working copy, you're creating a branch of sorts of your project. It's a special sort of branch. It lives only on your client machine; not in the repository. You synchronize this branch with changes made in the repository using **svn update**—which acts almost like a special-cased, simplified form of an **svn merge** command.[12] You effectively reintegrate your branch each time you run **svn commit**. So, in that special sense, Subversion users deal with branches and merges all the time. Given the similarities between updating and merging, it's no surprise, then, that the areas in which Subversion seems to have the most shortcomings—namely, handling file and directory renames and dealing with tree conflicts in general—are problematic for both the **svn update** and **svn merge** operations. Unfortunately, **svn merge** has a harder time of it precisely because of the fact that, for every way in which **svn update** is a special-cased, simplified kind of generic merge operation, a true Subversion merge is neither special-cased nor simplified. For this reason, merges perform much more slowly than updates, require explicit tracking (via the `svn:mergeinfo` property we've discussed in this chapter) and history-crunching arithmetic, and generally offer more opportunities for something to go awry.

To branch or not to branch? Ultimately, that depends on what your team needs in order to find that sweet balance of collaboration and isolation.

Summary

We covered a lot of ground in this chapter. We discussed the concepts of tags and branches and demonstrated how Subversion implements these concepts by copying directories with the **svn copy** command. We showed how to use **svn merge** to copy changes from one branch to another or roll back bad changes. We went over the use of **svn switch** to create mixed-location working copies. And we talked about how one might manage the organization and lifetimes of branches in a repository.

Remember the Subversion mantra: branches and tags are cheap. So don't be afraid to use them when needed!

As a helpful reminder of all the operations we discussed, here is handy reference table you can consult as you begin to make use of branches.

Table 4.1. Branching and merging commands

Action	Command
Create a branch or tag	**`svn copy URL1 URL2`**
Switch a working copy to a branch or tag	**`svn switch URL`**
Synchronize a branch with trunk	**`svn merge trunkURL; svn commit`**
See merge history or eligible changesets	**`svn mergeinfo SOURCE TARGET`**
Merge a branch back into trunk	**`svn merge --reintegrate branchURL; svn commit`**

[12] Actually, you *could* use **`svn merge -rLAST_UPDATED_REV:HEAD .`** in your working copy to quite literally merge in all the repository changes since your last update if really wanted to!

Action	Command
Merge one specific change	`svn merge -c REV URL; svn commit`
Merge a range of changes	`svn merge -r REV1:REV2 URL; svn commit`
Block a change from automatic merging	`svn merge -c REV --record-only URL; svn commit`
Preview a merge	`svn merge URL --dry-run`
Abandon merge results	`svn revert -R .`
Resurrect something from history	`svn copy URL@REV localPATH`
Undo a committed change	`svn merge -c -REV URL; svn commit`
Examine merge-sensitive history	`svn log -g; svn blame -g`
Create a tag from a working copy	`svn copy . tagURL`
Rearrange a branch or tag	`svn move URL1 URL2`
Remove a branch or tag	`svn delete URL`

Chapter 5. Repository Administration

The Subversion repository is the central storehouse of all your versioned data. As such, it becomes an obvious candidate for all the love and attention an administrator can offer. While the repository is generally a low-maintenance item, it is important to understand how to properly configure and care for it so that potential problems are avoided, and so actual problems are safely resolved.

In this chapter, we'll discuss how to create and configure a Subversion repository. We'll also talk about repository maintenance, providing examples of how and when to use various related tools provided with Subversion. We'll address some common questions and mistakes and give some suggestions on how to arrange the data in the repository.

If you plan to access a Subversion repository only in the role of a user whose data is under version control (i.e., via a Subversion client), you can skip this chapter altogether. However, if you are, or wish to become, a Subversion repository administrator,[1] this chapter is for you.

The Subversion Repository, Defined

Before jumping into the broader topic of repository administration, let's further define what a repository is. How does it look? How does it feel? Does it take its tea hot or iced, sweetened, and with lemon? As an administrator, you'll be expected to understand the composition of a repository both from a literal, OS-level perspective—how a repository looks and acts with respect to non-Subversion tools—and from a logical perspective—dealing with how data is represented *inside* the repository.

Seen through the eyes of a typical file browser application (such as Windows Explorer) or command-line based filesystem navigation tools, the Subversion repository is just another directory full of stuff. There are some subdirectories with human-readable configuration files in them, some subdirectories with some not-so-human-readable data files, and so on. As in other areas of the Subversion design, modularity is given high regard, and hierarchical organization is preferred to cluttered chaos. So a shallow glance into a typical repository from a nuts-and-bolts perspective is sufficient to reveal the basic components of the repository:

```
$ ls repos
conf/  db/  format  hooks/  locks/  README.txt
```

Here's a quick fly-by overview of what exactly you're seeing in this directory listing. (Don't get bogged down in the terminology—detailed coverage of these components exists elsewhere in this and other chapters.)

conf/
: This directory is a container for configuration files.

db/
: This directory contains the data store for all of your versioned data.[2]

format
: This file describes the repository's internal organizational scheme. (As it turns out, the `db/` subdirectory sometimes also contains a `format` file which describes only the contents of that subdirectory and which is not to be confused with this file.)

hooks/
: This directory contains hook script templates and hook scripts, if any have been installed.

[1] This may sound really prestigious and lofty, but we're just talking about anyone who is interested in that mysterious realm beyond the working copy where everyone's data hangs out.
[2] Strictly speaking, Subversion doesn't dictate that the versioned data live here, and there are known (albeit proprietary) alternative backend storage implementations which do not, in fact, store data in this directory.

locks/
: Subversion uses this directory to house repository lock files, used for managing concurrent access to the repository.

README.txt
: This is a brief text file containing merely a notice to readers that the directory they are looking in is a Subversion repository.

Prior to Subversion 1.5, the on-disk repository structure also always contained a `dav` subdirectory. `mod_dav_svn` used this directory to store information about WebDAV *activities*—mappings of high-level WebDAV protocol concepts to Subversion commit transactions. Subversion 1.5 changed that behavior, moving ownership of the activities directory, and the ability to configure its location, into `mod_dav_svn` itself. Now, new repositories will not necessarily have a `dav` subdirectory unless `mod_dav_svn` is in use and hasn't been configured to store its activities database elsewhere. See the section called "Directives" in Chapter 9, *Subversion Complete Reference* for more information.

Of course, when accessed via the Subversion libraries, this otherwise unremarkable collection of files and directories suddenly becomes an implementation of a virtual, versioned filesystem, complete with customizable event triggers. This filesystem has its own notions of directories and files, very similar to the notions of such things held by real filesystems (such as NTFS, FAT32, ext3, etc.). But this is a special filesystem—it hangs these directories and files from revisions, keeping all the changes you've ever made to them safely stored and forever accessible. This is where the entirety of your versioned data lives.

Strategies for Repository Deployment

Due largely to the simplicity of the overall design of the Subversion repository and the technologies on which it relies, creating and configuring a repository are fairly straightforward tasks. There are a few preliminary decisions you'll want to make, but the actual work involved in any given setup of a Subversion repository is pretty basic, tending toward mindless repetition if you find yourself setting up multiples of these things.

Some things you'll want to consider beforehand, though, are:

- What data do you expect to live in your repository (or repositories), and how will that data be organized?
- Where will your repository live, and how will it be accessed?
- What types of access control and repository event reporting do you need?
- Which of the available types of data store do you want to use?

In this section, we'll try to help you answer those questions.

Planning Your Repository Organization

While Subversion allows you to move around versioned files and directories without any loss of information, and even provides ways of moving whole sets of versioned history from one repository to another, doing so can greatly disrupt the workflow of those who access the repository often and come to expect things to be at certain locations. So before creating a new repository, try to peer into the future a bit; plan ahead before placing your data under version control. By conscientiously "laying out" your repository or repositories and their versioned contents ahead of time, you can prevent many future headaches.

Let's assume that as repository administrator, you will be responsible for supporting the version control system for several projects. Your first decision is whether to use a single repository for multiple projects, or to give each project its own repository, or some compromise of these two.

There are benefits to using a single repository for multiple projects, most obviously the lack of duplicated maintenance. A single

repository means that there is one set of hook programs, one thing to routinely back up, one thing to dump and load if Subversion releases an incompatible new version, and so on. Also, you can move data between projects easily, without losing any historical versioning information.

The downside of using a single repository is that different projects may have different requirements in terms of the repository event triggers, such as needing to send commit notification emails to different mailing lists, or having different definitions about what does and does not constitute a legitimate commit. These aren't insurmountable problems, of course—it just means that all of your hook scripts have to be sensitive to the layout of your repository rather than assuming that the whole repository is associated with a single group of people. Also, remember that Subversion uses repository-global revision numbers. While those numbers don't have any particular magical powers, some folks still don't like the fact that even though no changes have been made to their project lately, the youngest revision number for the repository keeps climbing because other projects are actively adding new revisions.[3]

A middle-ground approach can be taken, too. For example, projects can be grouped by how well they relate to each other. You might have a few repositories with a handful of projects in each repository. That way, projects that are likely to want to share data can do so easily, and as new revisions are added to the repository, at least the developers know that those new revisions are at least remotely related to everyone who uses that repository.

After deciding how to organize your projects with respect to repositories, you'll probably want to think about directory hierarchies within the repositories themselves. Because Subversion uses regular directory copies for branching and tagging (see Chapter 4, *Branching and Merging*), the Subversion community recommends that you choose a repository location for each project root—the "topmost" directory that contains data related to that project—and then create three subdirectories beneath that root: `trunk`, meaning the directory under which the main project development occurs; `branches`, which is a directory in which to create various named branches of the main development line; and `tags`, which is a collection of tree snapshots that are created, and perhaps destroyed, but never changed.[4]

For example, your repository might look like this:

```
/
  calc/
    trunk/
    tags/
    branches/
  calendar/
    trunk/
    tags/
    branches/
  spreadsheet/
    trunk/
    tags/
    branches/
  …
```

Note that it doesn't matter where in your repository each project root is. If you have only one project per repository, the logical place to put each project root is at the root of that project's respective repository. If you have multiple projects, you might want to arrange them in groups inside the repository, perhaps putting projects with similar goals or shared code in the same subdirectory, or maybe just grouping them alphabetically. Such an arrangement might look like this:

```
/
  utils/
```

[3]Whether founded in ignorance or in poorly considered concepts about how to derive legitimate software development metrics, global revision numbers are a silly thing to fear, and *not* the kind of thing you should weigh when deciding how to arrange your projects and repositories.
[4]The `trunk`, `tags`, and `branches` trio is sometimes referred to as "the TTB directories."

```
    calc/
      trunk/
      tags/
      branches/
    calendar/
      trunk/
      tags/
      branches/
    …
  office/
    spreadsheet/
      trunk/
      tags/
      branches/
    …
```

Lay out your repository in whatever way you see fit. Subversion does not expect or enforce a particular layout—in its eyes, a directory is a directory is a directory. Ultimately, you should choose the repository arrangement that meets the needs of the people who work on the projects that live there.

In the name of full disclosure, though, we'll mention another very common layout. In this layout, the `trunk`, `tags`, and `branches` directories live in the root directory of your repository, and your projects are in subdirectories beneath those, like so:

```
/
  trunk/
    calc/
    calendar/
    spreadsheet/
    …
  tags/
    calc/
    calendar/
    spreadsheet/
    …
  branches/
    calc/
    calendar/
    spreadsheet/
    …
```

There's nothing particularly incorrect about such a layout, but it may or may not seem as intuitive for your users. Especially in large, multiproject situations with many users, those users may tend to be familiar with only one or two of the projects in the repository. But the projects-as-branch-siblings approach tends to deemphasize project individuality and focus on the entire set of projects as a single entity. That's a social issue, though. We like our originally suggested arrangement for purely practical reasons—it's easier to ask about (or modify, or migrate elsewhere) the entire history of a single project when there's a single repository path that holds the entire history—past, present, tagged, and branched—for that project and that project alone.

Deciding Where and How to Host Your Repository

Before creating your Subversion repository, an obvious question you'll need to answer is where the thing is going to live. This is strongly connected to myriad other questions involving how the repository will be accessed (via a Subversion server or directly),

by whom (users behind your corporate firewall or the whole world out on the open Internet), what other services you'll be providing around Subversion (repository browsing interfaces, email-based commit notification, etc.), your data backup strategy, and so on.

We cover server choice and configuration in Chapter 6, *Server Configuration*, but the point we'd like to briefly make here is simply that the answers to some of these other questions might have implications that force your hand when deciding where your repository will live. For example, certain deployment scenarios might require accessing the repository via a remote filesystem from multiple computers, in which case (as you'll read in the next section) your choice of a repository backend data store turns out not to be a choice at all because only one of the available backends will work in this scenario.

Addressing each possible way to deploy Subversion is both impossible and outside the scope of this book. We simply encourage you to evaluate your options using these pages and other sources as your reference material and to plan ahead.

Choosing a Data Store

Subversion provides two options for the type of underlying data store—often referred to as “the backend” or, somewhat confusingly, “the (versioned) filesystem”—that each repository uses. One type of data store keeps everything in a Berkeley DB (or BDB) database environment; repositories that use this type are often referred to as being “BDB-backed.” The other type stores data in ordinary flat files, using a custom format. Subversion developers have adopted the habit of referring to this latter data storage mechanism as *FSFS*[5]—a versioned filesystem implementation that uses the native OS filesystem directly—rather than via a database library or some other abstraction layer—to store data.

Table 5.1, “Repository data store comparison” gives a comparative overview of Berkeley DB and FSFS repositories.

Table 5.1. Repository data store comparison

Category	Feature	Berkeley DB	FSFS
Reliability	Data integrity	When properly deployed, extremely reliable; Berkeley DB 4.4 brings auto-recovery	Older versions had some rarely demonstrated, but data-destroying bugs
	Sensitivity to interruptions	Very; crashes and permission problems can leave the database “wedged,” requiring journaled recovery procedures	Quite insensitive
Accessibility	Usable from a read-only mount	No	Yes
	Platform-independent storage	No	Yes
	Usable over network filesystems	Generally, no	Yes
	Group permissions handling	Sensitive to user umask problems; best if accessed by only one user	Works around umask problems
Scalability	Repository disk usage	Larger (especially if logfiles aren't purged)	Smaller
	Number of revision trees	Database; no problems	Some older native filesystems don't scale well with thousands of entries in a single directory
	Directories with many files	Slower	Faster

[5]Often pronounced “fuzz-fuzz,” if Jack Repenning has anything to say about it. (This book, however, assumes that the reader is thinking “eff-ess-eff-ess.”)

Category	Feature	Berkeley DB	FSFS
Performance	Checking out latest revision	No meaningful difference	No meaningful difference
	Large commits	Slower overall, but cost is amortized across the lifetime of the commit	Faster overall, but finalization delay may cause client timeouts

There are advantages and disadvantages to each of these two backend types. Neither of them is more "official" than the other, though the newer FSFS is the default data store as of Subversion 1.2. Both are reliable enough to trust with your versioned data. But as you can see in Table 5.1, "Repository data store comparison", the FSFS backend provides quite a bit more flexibility in terms of its supported deployment scenarios. More flexibility means you have to work a little harder to find ways to deploy it incorrectly. Those reasons—plus the fact that not using Berkeley DB means there's one fewer component in the system—largely explain why today almost everyone uses the FSFS backend when creating new repositories.

Fortunately, most programs that access Subversion repositories are blissfully ignorant of which backend data store is in use. And you aren't even necessarily stuck with your first choice of a data store—in the event that you change your mind later, Subversion provides ways of migrating your repository's data into another repository that uses a different backend data store. We talk more about that later in this chapter.

The following subsections provide a more detailed look at the available backend data store types.

Berkeley DB

When the initial design phase of Subversion was in progress, the developers decided to use Berkeley DB for a variety of reasons, including its open source license, transaction support, reliability, performance, API simplicity, thread safety, support for cursors, and so on.

Berkeley DB provides real transaction support—perhaps its most powerful feature. Multiple processes accessing your Subversion repositories don't have to worry about accidentally clobbering each other's data. The isolation provided by the transaction system is such that for any given operation, the Subversion repository code sees a static view of the database—not a database that is constantly changing at the hand of some other process—and can make decisions based on that view. If the decision made happens to conflict with what another process is doing, the entire operation is rolled back as though it never happened, and Subversion gracefully retries the operation against a new, updated (and yet still static) view of the database.

Another great feature of Berkeley DB is *hot backups*—the ability to back up the database environment without taking it "offline." We'll discuss how to back up your repository later in this chapter (in the section called "Repository Backup"), but the benefits of being able to make fully functional copies of your repositories without any downtime should be obvious.

Berkeley DB is also a very reliable database system when properly used. Subversion uses Berkeley DB's logging facilities, which means that the database first writes to on-disk logfiles a description of any modifications it is about to make, and then makes the modification itself. This is to ensure that if anything goes wrong, the database system can back up to a previous *checkpoint*—a location in the logfiles known not to be corrupt—and replay transactions until the data is restored to a usable state. See the section called "Managing Disk Space" later in this chapter for more about Berkeley DB logfiles.

But every rose has its thorn, and so we must note some known limitations of Berkeley DB. First, Berkeley DB environments are not portable. You cannot simply copy a Subversion repository that was created on a Unix system onto a Windows system and expect it to work. While much of the Berkeley DB database format is architecture-independent, other aspects of the environment are not. Second, Subversion uses Berkeley DB in a way that will not operate on Windows 95/98 systems—if you need to house a BDB-backed repository on a Windows machine, stick with Windows 2000 or later.

While Berkeley DB promises to behave correctly on network shares that meet a particular set of specifications,[6] most networked filesystem types and appliances do *not* actually meet those requirements. And in no case can you allow a BDB-backed repository that resides on a network share to be accessed by multiple clients of that share at once (which quite often is the whole point of having the repository live on a network share in the first place).

[6]Berkeley DB requires that the underlying filesystem implement strict POSIX locking semantics, and more importantly, the ability to map files directly into process memory.

If you attempt to use Berkeley DB on a noncompliant remote filesystem, the results are unpredictable—you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted. You should strongly consider using the FSFS data store for repositories that need to live on a network share.

Finally, because Berkeley DB is a library linked directly into Subversion, it's more sensitive to interruptions than a typical relational database system. Most SQL systems, for example, have a dedicated server process that mediates all access to tables. If a program accessing the database crashes for some reason, the database daemon notices the lost connection and cleans up any mess left behind. And because the database daemon is the only process accessing the tables, applications don't need to worry about permission conflicts. These things are not the case with Berkeley DB, however. Subversion (and programs using Subversion libraries) access the database tables directly, which means that a program crash can leave the database in a temporarily inconsistent, inaccessible state. When this happens, an administrator needs to ask Berkeley DB to restore to a checkpoint, which is a bit of an annoyance. Other things can cause a repository to "wedge" besides crashed processes, such as programs conflicting over ownership and permissions on the database files.

Berkeley DB 4.4 brings (to Subversion 1.4 and later) the ability for Subversion to automatically and transparently recover Berkeley DB environments in need of such recovery. When a Subversion process attaches to a repository's Berkeley DB environment, it uses some process accounting mechanisms to detect any unclean disconnections by previous processes, performs any necessary recovery, and then continues on as though nothing happened. This doesn't completely eliminate instances of repository wedging, but it does drastically reduce the amount of human interaction required to recover from them.

So while a Berkeley DB repository is quite fast and scalable, it's best used by a single server process running as one user—such as Apache's **httpd** or **svnserve** (see Chapter 6, *Server Configuration*)—rather than accessing it as many different users via `file://` or `svn+ssh://` URLs. If you're accessing a Berkeley DB repository directly as multiple users, be sure to read the section called "Supporting Multiple Repository Access Methods" later in this chapter.

FSFS

In mid-2004, a second type of repository storage system—one that doesn't use a database at all—came into being. An FSFS repository stores the changes associated with a revision in a single file, and so all of a repository's revisions can be found in a single subdirectory full of numbered files. Transactions are created in separate subdirectories as individual files. When complete, the transaction file is renamed and moved into the revisions directory, thus guaranteeing that commits are atomic. And because a revision file is permanent and unchanging, the repository also can be backed up while "hot," just like a BDB-backed repository.

Revision files and shards

FSFS repositories contain files that describe the changes made in a single revision, and files that contain the revision properties associated with a single revision. Repositories created in versions of Subversion prior to 1.5 keep these files in two directories—one for each type of file. As new revisions are committed to the repository, Subversion drops more files into these two directories—over time, the number of these files in each directory can grow to be quite large. This has been observed to cause performance problems on certain network-based filesystems.

Subversion 1.5 creates FSFS-backed repositories using a slightly modified layout in which the contents of these two directories are *sharded*, or scattered across several subdirectories. This can greatly reduce the time it takes the system to locate any one of these files, and therefore increases the overall performance of Subversion when reading from the repository.

Subversion 1.6 and later takes the sharded layout one step further, allowing administrators to optionally *pack* each of their repository shards up into a single multi-revision file. This can have both performance and disk usage benefits. See the section called "Packing FSFS filesystems" for more information.

The FSFS revision files describe a revision's directory structure, file contents, and deltas against files in other revision trees. Unlike a Berkeley DB database, this storage format is portable across different operating systems and isn't sensitive to CPU architecture.

Because no journaling or shared-memory files are being used, the repository can be safely accessed over a network filesystem and examined in a read-only environment. The lack of database overhead also means the overall repository size is a bit smaller.

FSFS has different performance characteristics, too. When committing a directory with a huge number of files, FSFS is able to more quickly append directory entries. On the other hand, FSFS has a longer delay when finalizing a commit while it performs tasks that the BDB backend amortizes across the lifetime of the commit, which could in extreme cases cause clients to time out while waiting for a response.

The most important distinction, however, is FSFS's imperviousness to wedging when something goes wrong. If a process using a Berkeley DB database runs into a permissions problem or suddenly crashes, the database can be left in an unusable state until an administrator recovers it. If the same scenarios happen to a process using an FSFS repository, the repository isn't affected at all. At worst, some transaction data is left behind.

Creating and Configuring Your Repository

Earlier in this chapter (in the section called “Strategies for Repository Deployment”), we looked at some of the important decisions that should be made before creating and configuring your Subversion repository. Now, we finally get to get our hands dirty! In this section, we'll see how to actually create a Subversion repository and configure it to perform custom actions when special repository events occur.

Creating the Repository

Subversion repository creation is an incredibly simple task. The **svnadmin** utility that comes with Subversion provides a subcommand (**svnadmin create**) for doing just that.

```
$ # Create a repository
$ svnadmin create /var/svn/repos
$
```

Assuming that the parent directory `/var/svn` exists and that you have sufficient permissions to modify that directory, the previous command creates a new repository in the directory `/var/svn/repos`, and with the default filesystem data store (FSFS). You can explicitly choose the filesystem type using the `--fs-type` argument, which accepts as a parameter either `fsfs` or `bdb`.

```
$ # Create an FSFS-backed repository
$ svnadmin create --fs-type fsfs /var/svn/repos
$
```

```
# Create a Berkeley-DB-backed repository
$ svnadmin create --fs-type bdb /var/svn/repos
$
```

After running this simple command, you have a Subversion repository. Depending on how users will access this new repository, you might need to fiddle with its filesystem permissions. But since basic system administration is rather outside the scope of this text, we'll leave further exploration of that topic as an exercise to the reader.

The path argument to **svnadmin** is just a regular filesystem path and not a URL like the **svn** client program uses when referring to repositories. Both **svnadmin** and **svnlook** are considered server-side utilities—they are used on the machine where the repository resides to examine or modify aspects of the repository, and are in fact unable to perform tasks across a network. A common mistake made by Subversion newcomers is trying to pass URLs (even "local" `file://` ones) to these two programs.

Present in the `db/` subdirectory of your repository is the implementation of the versioned filesystem. Your new repository's versioned filesystem begins life at revision 0, which is defined to consist of nothing but the top-level root (`/`) directory. Initially, revision 0 also has a single revision property, `svn:date`, set to the time at which the repository was created.

Now that you have a repository, it's time to customize it.

While some parts of a Subversion repository—such as the configuration files and hook scripts—are meant to be examined and modified manually, you shouldn't (and shouldn't need to) tamper with the other parts of the repository "by hand." The **svnadmin** tool should be sufficient for any changes necessary to your repository, or you can look to third-party tools (such as Berkeley DB's tool suite) for tweaking relevant subsections of the repository. Do *not* attempt manual manipulation of your version control history by poking and prodding around in your repository's data store files!

Implementing Repository Hooks

A *hook* is a program triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property. Some hooks (the so-called "pre hooks") run in advance of a repository operation and provide a means by which to both report what is about to happen and prevent it from happening at all. Other hooks (the "post hooks") run after the completion of a repository event and are useful for performing tasks that examine—but don't modify—the repository. Each hook is handed enough information to tell what that event is (or was), the specific repository changes proposed (or completed), and the username of the person who triggered the event.

The `hooks` subdirectory is, by default, filled with templates for various repository hooks:

```
$ ls repos/hooks/
post-commit.tmpl          post-unlock.tmpl  pre-revprop-change.tmpl
post-lock.tmpl            pre-commit.tmpl   pre-unlock.tmpl
post-revprop-change.tmpl  pre-lock.tmpl     start-commit.tmpl
$
```

There is one template for each hook that the Subversion repository supports; by examining the contents of those template scripts, you can see what triggers each script to run and what data is passed to that script. Also present in many of these templates are examples of how one might use that script, in conjunction with other Subversion-supplied programs, to perform common useful tasks. To actually install a working hook, you need only place some executable program or script into the `repos/hooks` directory, which can be executed as the name (such as **start-commit** or **post-commit**) of the hook.

On Unix platforms, this means supplying a script or program (which could be a shell script, a Python program, a compiled C binary, or any number of other things) named exactly like the name of the hook. Of course, the template files are present for more than just informational purposes—the easiest way to install a hook on Unix platforms is to simply copy the appropriate template file to a new file that lacks the `.tmpl` extension, customize the hook's contents, and ensure that the script is executable. Windows, however, uses file extensions to determine whether a program is executable, so you would need to supply a program whose basename is the name of the hook and whose extension is one of the special extensions recognized by Windows for executable programs, such as `.exe` for programs and `.bat` for batch files.

For security reasons, the Subversion repository executes hook programs with an empty environment—that is, no environment variables are set at all, not even `$PATH` (or `%PATH%`, under Windows). Because of this, many administrators are baffled when their hook program runs fine by hand, but doesn't work when run by Subversion. Be sure to explicitly set any necessary environment variables in your hook program and/or use absolute paths to programs.

Subversion executes hooks as the same user who owns the process that is accessing the Subversion repository. In most cases, the repository is being accessed via a Subversion server, so this user is the same user as whom the server runs on the system. The hooks themselves will need to be configured with OS-level permissions that allow that user to execute them. Also, this means that any programs or files (including the Subversion repository) accessed directly or indirectly by the hook will be accessed as the same user. In other words, be alert to potential permission-related problems that could prevent the hook from performing the tasks it is designed to perform.

There are several hooks implemented by the Subversion repository, and you can get details about each of them in the section called "Repository Hooks" in Chapter 9, *Subversion Complete Reference*. As a repository administrator, you'll need to decide which hooks you wish to implement (by way of providing an appropriately named and permissioned hook program), and how. When you make this decision, keep in mind the big picture of how your repository is deployed. For example, if you are using server configuration to determine which users are permitted to commit changes to your repository, you don't need to do this sort of access control via the hook system.

There is no shortage of Subversion hook programs and scripts that are freely available either from the Subversion community itself or elsewhere. These scripts cover a wide range of utility—basic access control, policy adherence checking, issue tracker integration, email- or syndication-based commit notification, and beyond. Or, if you wish to write your own, see Chapter 8, *Embedding Subversion*.

While hook scripts can do almost anything, there is one dimension in which hook script authors should show restraint: do *not* modify a commit transaction using hook scripts. While it might be tempting to use hook scripts to automatically correct errors, shortcomings, or policy violations present in the files being committed, doing so can cause problems. Subversion keeps client-side caches of certain bits of repository data, and if you change a commit transaction in this way, those caches become indetectably stale. This inconsistency can lead to surprising and unexpected behavior. Instead of modifying the transaction, you should simply *validate* the transaction in the `pre-commit` hook and reject the commit if it does not meet the desired requirements. As a bonus, your users will learn the value of careful, compliance-minded work habits.

Berkeley DB Configuration

A Berkeley DB environment is an encapsulation of one or more databases, logfiles, region files, and configuration files. The Berkeley DB environment has its own set of default configuration values for things such as the number of database locks allowed to be taken out at any given time, the maximum size of the journaling logfiles, and so on. Subversion's filesystem logic additionally chooses default values for some of the Berkeley DB configuration options. However, sometimes your particular repository, with its unique collection of data and access patterns, might require a different set of configuration option values.

The producers of Berkeley DB understand that different applications and database environments have different requirements, so they have provided a mechanism for overriding at runtime many of the configuration values for the Berkeley DB environment. BDB checks for the presence of a file named `DB_CONFIG` in the environment directory (namely, the repository's `db` subdirectory), and parses the options found in that file. Subversion itself creates this file when it creates the rest of the repository. The file initially contains some default options, as well as pointers to the Berkeley DB online documentation so that you can read about what those options do. Of course, you are free to add any of the supported Berkeley DB options to your `DB_CONFIG` file. Just be aware that while Subversion never attempts to read or interpret the contents of the file and makes no direct use of the option settings in it, you'll want to avoid any configuration changes that may cause Berkeley DB to behave in a fashion that is at odds with what Subversion might expect. Also, changes made to `DB_CONFIG` won't take effect until you recover the database environment (using **svnadmin recover**).

FSFS Configuration

As of Subversion 1.6, FSFS filesystems have several configurable parameters which an administrator can use to fine-tune the performance or disk usage of their repositories. You can find these options—and the documentation for them—in the `db/fsfs.conf` file in the repository.

Repository Maintenance

Maintaining a Subversion repository can be daunting, mostly due to the complexities inherent in systems that have a database backend. Doing the task well is all about knowing the tools—what they are, when to use them, and how. This section will introduce you to the repository administration tools provided by Subversion and discuss how to wield them to accomplish tasks such as repository data migration, upgrades, backups, and cleanups.

An Administrator's Toolkit

Subversion provides a handful of utilities useful for creating, inspecting, modifying, and repairing your repository. Let's look more closely at each of those tools. Afterward, we'll briefly examine some of the utilities included in the Berkeley DB distribution that provide functionality specific to your repository's database backend not otherwise provided by Subversion's own tools.

svnadmin

The **svnadmin** program is the repository administrator's best friend. Besides providing the ability to create Subversion repositories, this program allows you to perform several maintenance operations on those repositories. The syntax of **svnadmin** is similar to that of other Subversion command-line programs:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]
Type 'svnadmin help <subcommand>' for help on a specific subcommand.
Type 'svnadmin --version' to see the program version and FS modules.

Available subcommands:
   crashtest
   create
   deltify
…
```

Previously in this chapter (in the section called “Creating the Repository”), we were introduced to the **svnadmin create** subcommand. Most of the other **svnadmin** subcommands we will cover later in this chapter. And you can consult the section called “svnadmin—Subversion Repository Administration” in Chapter 9, *Subversion Complete Reference* for a full rundown of subcommands and what each of them offers.

svnlook

svnlook is a tool provided by Subversion for examining the various revisions and *transactions* (which are revisions in the making) in a repository. No part of this program attempts to change the repository. **svnlook** is typically used by the repository hooks for reporting the changes that are about to be committed (in the case of the **pre-commit** hook) or that were just committed (in the case of the **post-commit** hook) to the repository. A repository administrator may use this tool for diagnostic purposes.

svnlook has a straightforward syntax:

```
$ svnlook help
general usage: svnlook SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]
Note: any subcommand which takes the '--revision' and '--transaction'
      options will, if invoked without one of those options, act on
```

```
      the repository's youngest revision.
Type 'svnlook help <subcommand>' for help on a specific subcommand.
Type 'svnlook --version' to see the program version and FS modules.
…
```

Most of **svnlook**'s subcommands can operate on either a revision or a transaction tree, printing information about the tree itself, or how it differs from the previous revision of the repository. You use the `--revision` (`-r`) and `--transaction` (`-t`) options to specify which revision or transaction, respectively, to examine. In the absence of both the `--revision` (`-r`) and `--transaction` (`-t`) options, **svnlook** will examine the youngest (or `HEAD`) revision in the repository. So the following two commands do exactly the same thing when 19 is the youngest revision in the repository located at `/var/svn/repos`:

```
$ svnlook info /var/svn/repos
$ svnlook info /var/svn/repos -r 19
```

One exception to these rules about subcommands is the **svnlook youngest** subcommand, which takes no options and simply prints out the repository's youngest revision number:

```
$ svnlook youngest /var/svn/repos
19
$
```

Keep in mind that the only transactions you can browse are uncommitted ones. Most repositories will have no such transactions because transactions are usually either committed (in which case, you should access them as revision with the `--revision` (`-r`) option) or aborted and removed.

Output from **svnlook** is designed to be both human- and machine-parsable. Take, as an example, the output of the **svnlook info** subcommand:

```
$ svnlook info /var/svn/repos
sally
2002-11-04 09:29:13 -0600 (Mon, 04 Nov 2002)
27
Added the usual
Greek tree.
$
```

The output of **svnlook info** consists of the following, in the order given:

1. The author, followed by a newline
2. The date, followed by a newline
3. The number of characters in the log message, followed by a newline

4. The log message itself, followed by a newline

This output is human-readable, meaning items such as the datestamp are displayed using a textual representation instead of something more obscure (such as the number of nanoseconds since the Tastee Freez guy drove by). But the output is also machine-parsable—because the log message can contain multiple lines and be unbounded in length, **svnlook** provides the length of that message before the message itself. This allows scripts and other wrappers around this command to make intelligent decisions about the log message, such as how much memory to allocate for the message, or at least how many bytes to skip in the event that this output is not the last bit of data in the stream.

svnlook can perform a variety of other queries: displaying subsets of bits of information we've mentioned previously, recursively listing versioned directory trees, reporting which paths were modified in a given revision or transaction, showing textual and property differences made to files and directories, and so on. See the section called “svnlook—Subversion Repository Examination” in Chapter 9, *Subversion Complete Reference* for a full reference of **svnlook**'s features.

svndumpfilter

While it won't be the most commonly used tool at the administrator's disposal, **svndumpfilter** provides a very particular brand of useful functionality—the ability to quickly and easily modify streams of Subversion repository history data by acting as a path-based filter.

The syntax of **svndumpfilter** is as follows:

```
$ svndumpfilter help
general usage: svndumpfilter SUBCOMMAND [ARGS & OPTIONS ...]
Type 'svndumpfilter help <subcommand>' for help on a specific subcommand.
Type 'svndumpfilter --version' to see the program version.

Available subcommands:
   exclude
   include
   help (?, h)
```

There are only two interesting subcommands: **svndumpfilter exclude** and **svndumpfilter include**. They allow you to make the choice between implicit or explicit inclusion of paths in the stream. You can learn more about these subcommands and **svndumpfilter**'s unique purpose later in this chapter, in the section called “Filtering Repository History”.

svnrdump

The **svnrdump** program is, to put it simply, essentially just network-aware flavors of the **svnadmin dump** and **svnadmin load** subcommands, rolled up into a separate program.

```
$ svnrdump help
general usage: svnrdump SUBCOMMAND URL [-r LOWER[:UPPER]]
Type 'svnrdump help <subcommand>' for help on a specific subcommand.
Type 'svnrdump --version' to see the program version and RA modules.

Available subcommands:
   dump
   load
   help (?, h)

$
```

We discuss the use of **svnrdump** and the aforementioned **svnadmin** commands later in this chapter (see the section called "Migrating Repository Data Elsewhere").

svnsync

The **svnsync** program provides all the functionality required for maintaining a read-only mirror of a Subversion repository. The program really has one job—to transfer one repository's versioned history into another repository. And while there are few ways to do that, its primary strength is that it can operate remotely—the "source" and "sink"[7] repositories may be on different computers from each other and from **svnsync** itself.

As you might expect, **svnsync** has a syntax that looks very much like every other program we've mentioned in this chapter:

```
$ svnsync help
general usage: svnsync SUBCOMMAND DEST_URL  [ARGS & OPTIONS ...]
Type 'svnsync help <subcommand>' for help on a specific subcommand.
Type 'svnsync --version' to see the program version and RA modules.

Available subcommands:
   initialize (init)
   synchronize (sync)
   copy-revprops
   info
   help (?, h)
$
```

We talk more about replicating repositories with **svnsync** later in this chapter (see the section called "Repository Replication").

fsfs-reshard.py

While not an official member of the Subversion toolchain, the **fsfs-reshard.py** script (found in the `tools/server-side` directory of the Subversion source distribution) is a useful performance tuning tool for administrators of FSFS-backed Subversion repositories. As described in the sidebar Revision files and shards, FSFS repositories use individual files to house information about each revision. Sometimes these files all live in a single directory; sometimes they are sharded across many directories. But the neat thing is that the number of directories used to house these files is configurable. That's where **fsfs-reshard.py** comes in.

fsfs-reshard.py reshuffles the repository's file structure into a new arrangement that reflects the requested number of sharding subdirectories and updates the repository configuration to preserve this change. When used in conjunction with the **svnadmin upgrade** command, this is especially useful for upgrading a pre-1.5 Subversion (unsharded) repository to the latest filesystem format and sharding its data files (which Subversion will not automatically do for you). This script can also be used for fine-tuning an already sharded repository.

Berkeley DB utilities

If you're using a Berkeley DB repository, all of your versioned filesystem's structure and data live in a set of database tables within the `db/` subdirectory of your repository. This subdirectory is a regular Berkeley DB environment directory and can therefore be used in conjunction with any of the Berkeley database tools, typically provided as part of the Berkeley DB distribution.

For day-to-day Subversion use, these tools are unnecessary. Most of the functionality typically needed for Subversion repositories has been duplicated in the **svnadmin** tool. For example, **svnadmin list-unused-dblogs** and **svnadmin list-dblogs** perform a subset of what is provided by the Berkeley **db_archive** utility, and **svnadmin recover** reflects the common use cases of the **db_recover** utility.

However, there are still a few Berkeley DB utilities that you might find useful. The **db_dump** and **db_load** programs write and

[7] Or is that, the "sync"?

read, respectively, a custom file format that describes the keys and values in a Berkeley DB database. Since Berkeley databases are not portable across machine architectures, this format is a useful way to transfer those databases from machine to machine, irrespective of architecture or operating system. As we describe later in this chapter, you can also use **svnadmin dump** and **svnadmin load** for similar purposes, but **db_dump** and **db_load** can do certain jobs just as well and much faster. They can also be useful if the experienced Berkeley DB hacker needs to do in-place tweaking of the data in a BDB-backed repository for some reason, which is something Subversion's utilities won't allow. Also, the **db_stat** utility can provide useful information about the status of your Berkeley DB environment, including detailed statistics about the locking and storage subsystems.

For more information on the Berkeley DB tool chain, visit the documentation section of the Berkeley DB section of Oracle's web site, located at http://www.oracle.com/technology/documentation/berkeley-db/db/.

Commit Log Message Correction

Sometimes a user will have an error in her log message (a misspelling or some misinformation, perhaps). If the repository is configured (using the `pre-revprop-change` hook; see the section called "Implementing Repository Hooks") to accept changes to this log message after the commit is finished, the user can "fix" her log message remotely using **svn propset** (see svn propset (pset, ps) in Chapter 9, *Subversion Complete Reference*). However, because of the potential to lose information forever, Subversion repositories are not, by default, configured to allow changes to unversioned properties—except by an administrator.

If a log message needs to be changed by an administrator, this can be done using **svnadmin setlog**. This command changes the log message (the `svn:log` property) on a given revision of a repository, reading the new value from a provided file.

```
$ echo "Here is the new, correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt -r 388
```

The **svnadmin setlog** command, by default, is still bound by the same protections against modifying unversioned properties as a remote client is—the `pre-revprop-change` and `post-revprop-change` hooks are still triggered, and therefore must be set up to accept changes of this nature. But an administrator can get around these protections by passing the `--bypass-hooks` option to the **svnadmin setlog** command.

Remember, though, that by bypassing the hooks, you are likely avoiding such things as email notifications of property changes, backup systems that track unversioned property changes, and so on. In other words, be very careful about what you are changing, and how you change it.

Managing Disk Space

While the cost of storage has dropped incredibly in the past few years, disk usage is still a valid concern for administrators seeking to version large amounts of data. Every bit of version history information stored in the live repository needs to be backed up elsewhere, perhaps multiple times as part of rotating backup schedules. It is useful to know what pieces of Subversion's repository data need to remain on the live site, which need to be backed up, and which can be safely removed.

How Subversion saves disk space

To keep the repository small, Subversion uses *deltification* (or delta-based storage) within the repository itself. Deltification involves encoding the representation of a chunk of data as a collection of differences against some other chunk of data. If the two pieces of data are very similar, this deltification results in storage savings for the deltified chunk—rather than taking up space equal to the size of the original data, it takes up only enough space to say, "I look just like this other piece of data over here, except for the following couple of changes." The result is that most of the repository data that tends to be bulky—namely, the contents of versioned files—is stored at a much smaller size than the original full-text representation of that data.

While deltified storage has been a part of Subversion's design since the very beginning, there have been additional improvements made over the years. Subversion repositories created with Subversion 1.4 or later benefit from compression of the full-text repres-

entations of file contents. Repositories created with Subversion 1.6 or later further enjoy the disk space savings afforded by *representation sharing*, a feature which allows multiple files or file revisions with identical file content to refer to a single shared instance of that data rather than each having their own distinct copy thereof.

Because all of the data that is subject to deltification in a BDB-backed repository is stored in a single Berkeley DB database file, reducing the size of the stored values will not immediately reduce the size of the database file itself. Berkeley DB will, however, keep internal records of unused areas of the database file and consume those areas first before growing the size of the database file. So while deltification doesn't produce immediate space savings, it can drastically slow future growth of the database.

Removing dead transactions

Though they are uncommon, there are circumstances in which a Subversion commit process might fail, leaving behind in the repository the remnants of the revision-to-be that wasn't—an uncommitted transaction and all the file and directory changes associated with it. This could happen for several reasons: perhaps the client operation was inelegantly terminated by the user, or a network failure occurred in the middle of an operation. Regardless of the reason, dead transactions can happen. They don't do any real harm, other than consuming disk space. A fastidious administrator may nonetheless wish to remove them.

You can use the **svnadmin lstxns** command to list the names of the currently outstanding transactions:

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

Each item in the resultant output can then be used with **svnlook** (and its `--transaction` (`-t`) option) to determine who created the transaction, when it was created, what types of changes were made in the transaction—information that is helpful in determining whether the transaction is a safe candidate for removal! If you do indeed want to remove a transaction, its name can be passed to **svnadmin rmtxns**, which will perform the cleanup of the transaction. In fact, **svnadmin rmtxns** can take its input directly from the output of **svnadmin lstxns**!

```
$ svnadmin rmtxns myrepos `svnadmin lstxns myrepos`
$
```

If you use these two subcommands like this, you should consider making your repository temporarily inaccessible to clients. That way, no one can begin a legitimate transaction before you start your cleanup. Example 5.1, “txn-info.sh (reporting outstanding transactions)” contains a bit of shell-scripting that can quickly generate information about each outstanding transaction in your repository.

Example 5.1. txn-info.sh (reporting outstanding transactions)

```
#!/bin/sh

### Generate informational output for all outstanding transactions in
### a Subversion repository.
```

```
REPOS="${1}"
if [ "x$REPOS" = x ] ; then
  echo "usage: $0 REPOS_PATH"
  exit
fi

for TXN in `svnadmin lstxns ${REPOS}`; do
  echo "---[ Transaction ${TXN} ]-------------------------------------------"
  svnlook info "${REPOS}" -t "${TXN}"
done
```

The output of the script is basically a concatenation of several chunks of **svnlook info** output (see the section called “svnlook”) and will look something like this:

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]------------------------------------------
harry
2001-09-10 16:50:30 -0500 (Mon, 10 Sep 2001)
39
Trying to commit over a faulty network.
---[ Transaction a45 ]------------------------------------------
sally
2001-09-12 11:09:28 -0500 (Wed, 12 Sep 2001)
0
$
```

A long-abandoned transaction usually represents some sort of failed or interrupted commit. A transaction's datestamp can provide interesting information—for example, how likely is it that an operation begun nine months ago is still active?

In short, transaction cleanup decisions need not be made unwisely. Various sources of information—including Apache's error and access logs, Subversion's operational logs, Subversion revision history, and so on—can be employed in the decision-making process. And of course, an administrator can often simply communicate with a seemingly dead transaction's owner (via email, e.g.) to verify that the transaction is, in fact, in a zombie state.

Purging unused Berkeley DB logfiles

Until recently, the largest offender of disk space usage with respect to BDB-backed Subversion repositories were the logfiles in which Berkeley DB performs its prewrites before modifying the actual database files. These files capture all the actions taken along the route of changing the database from one state to another—while the database files, at any given time, reflect a particular state, the logfiles contain all of the many changes along the way *between* states. Thus, they can grow and accumulate quite rapidly.

Fortunately, beginning with the 4.2 release of Berkeley DB, the database environment has the ability to remove its own unused logfiles automatically. Any repositories created using **svnadmin** when compiled against Berkeley DB version 4.2 or later will be configured for this automatic logfile removal. If you don't want this feature enabled, simply pass the `--bdb-log-keep` option to the **svnadmin create** command. If you forget to do this or change your mind at a later time, simply edit the `DB_CONFIG` file found in your repository's `db` directory, comment out the line that contains the `set_flags DB_LOG_AUTOREMOVE` directive, and then run **svnadmin recover** on your repository to force the configuration changes to take effect. See the section called “Berkeley DB Configuration” for more information about database configuration.

Without some sort of automatic logfile removal in place, logfiles will accumulate as you use your repository. This is actually somewhat of a feature of the database system—you should be able to recreate your entire database using nothing but the logfiles, so

these files can be useful for catastrophic database recovery. But typically, you'll want to archive the logfiles that are no longer in use by Berkeley DB, and then remove them from disk to conserve space. Use the **svnadmin list-unused-dblogs** command to list the unused logfiles:

```
$ svnadmin list-unused-dblogs /var/svn/repos
/var/svn/repos/log.0000000031
/var/svn/repos/log.0000000032
/var/svn/repos/log.0000000033
…
$ rm `svnadmin list-unused-dblogs /var/svn/repos`
## disk space reclaimed!
```

BDB-backed repositories whose logfiles are used as part of a backup or disaster recovery plan should *not* make use of the logfile autoremoval feature. Reconstruction of a repository's data from logfiles can only be accomplished only when *all* the logfiles are available. If some of the logfiles are removed from disk before the backup system has a chance to copy them elsewhere, the incomplete set of backed-up logfiles is essentially useless.

Packing FSFS filesystems

As described in the sidebar Revision files and shards, FSFS-backed Subversion repositories create, by default, a new on-disk file for each revision added to the repository. Having thousands of these files present on your Subversion server—even when housed in separate shard directories—can lead to inefficiencies.

The first problem is that the operating system has to reference many different files over a short period of time. This leads to inefficient use of disk caches and, as a result, more time spent seeking across large disks. Because of this, Subversion pays a performance penalty when accessing your versioned data.

The second problem is a bit more subtle. Because of the ways that most filesystems allocate disk space, each file claims more space on the disk than it actually uses. The amount of extra space required to house a single file can average anywhere from 2 to 16 kilobytes *per file*, depending on the underlying filesystem in use. This translates directly into a per-revision disk usage penalty for FSFS-backed repositories. The effect is most pronounced in repositories which have many small revisions, since the overhead involved in storing the revision file quickly outgrows the size of the actual data being stored.

To solve these problems, Subversion 1.6 introduced the **svnadmin pack** command. By concatenating all the files of a completed shard into a single “pack” file and then removing the original per-revision files, **svnadmin pack** reduces the file count within a given shard down to just a single file. In doing so, it aids filesystem caches and reduces (to one) the number of times a file storage overhead penalty is paid.

Subversion can pack existing sharded repositories which have been upgraded to the 1.6 filesystem format or later (see svnadmin upgrade) in Chapter 9, *Subversion Complete Reference*. To do so, just run **svnadmin pack** on the repository:

```
$ svnadmin pack /var/svn/repos
Packing shard 0...done.
Packing shard 1...done.
Packing shard 2...done.
…
Packing shard 34...done.
Packing shard 35...done.
Packing shard 36...done.
$
```

Because the packing process obtains the required locks before doing its work, you can run it on live repositories, or even as part of a post-commit hook. Repacking packed shards is legal, but will have no effect on the disk usage of the repository.

svnadmin pack has no effect on BDB-backed Subversion repositories.

Berkeley DB Recovery

As mentioned in the section called "Berkeley DB", a Berkeley DB repository can sometimes be left in a frozen state if not closed properly. When this happens, an administrator needs to rewind the database back into a consistent state. This is unique to BDB-backed repositories, though—if you are using FSFS-backed ones instead, this won't apply to you. And for those of you using Subversion 1.4 with Berkeley DB 4.4 or later, you should find that Subversion has become much more resilient in these types of situations. Still, wedged Berkeley DB repositories do occur, and an administrator needs to know how to safely deal with this circumstance.

To protect the data in your repository, Berkeley DB uses a locking mechanism. This mechanism ensures that portions of the database are not simultaneously modified by multiple database accessors, and that each process sees the data in the correct state when that data is being read from the database. When a process needs to change something in the database, it first checks for the existence of a lock on the target data. If the data is not locked, the process locks the data, makes the change it wants to make, and then unlocks the data. Other processes are forced to wait until that lock is removed before they are permitted to continue accessing that section of the database. (This has nothing to do with the locks that you, as a user, can apply to versioned files within the repository; we try to clear up the confusion caused by this terminology collision in the sidebar The Three Meanings of "Lock".)

In the course of using your Subversion repository, fatal errors or interruptions can prevent a process from having the chance to remove the locks it has placed in the database. The result is that the backend database system gets "wedged." When this happens, any attempts to access the repository hang indefinitely (since each new accessor is waiting for a lock to go away—which isn't going to happen).

If this happens to your repository, don't panic. The Berkeley DB filesystem takes advantage of database transactions, checkpoints, and prewrite journaling to ensure that only the most catastrophic of events[8] can permanently destroy a database environment. A sufficiently paranoid repository administrator will have made off-site backups of the repository data in some fashion, but don't head off to the tape backup storage closet just yet.

Instead, use the following recipe to attempt to "unwedge" your repository:

1. Make sure no processes are accessing (or attempting to access) the repository. For networked repositories, this also means shutting down the Apache HTTP Server or svnserve daemon.

2. Become the user who owns and manages the repository. This is important, as recovering a repository while running as the wrong user can tweak the permissions of the repository's files in such a way that your repository will still be inaccessible even after it is "unwedged."

3. Run the command **`svnadmin recover /var/svn/repos`**. You should see output such as this:

   ```
   Repository lock acquired.
   Please wait; recovering the repository may take some time...

   Recovery completed.
   The latest repos revision is 19.
   ```

   This command may take many minutes to complete.

4. Restart the server process.

[8] For example, hard drive + huge electromagnet = disaster.

This procedure fixes almost every case of repository wedging. Make sure that you run this command as the user that owns and manages the database, not just as `root`. Part of the recovery process might involve re-creating from scratch various database files (shared memory regions, e.g.). Recovering as `root` will create those files such that they are owned by `root`, which means that even after you restore connectivity to your repository, regular users will be unable to access it.

If the previous procedure, for some reason, does not successfully unwedge your repository, you should do two things. First, move your broken repository directory aside (perhaps by renaming it to something like `repos.BROKEN`) and then restore your latest backup of it. Then, send an email to the Subversion users mailing list (at `<users@subversion.apache.org>`) describing your problem in detail. Data integrity is an extremely high priority to the Subversion developers.

Migrating Repository Data Elsewhere

A Subversion filesystem has its data spread throughout files in the repository, in a fashion generally understood by (and of interest to) only the Subversion developers themselves. However, circumstances may arise that call for all, or some subset, of that data to be copied or moved into another repository.

Subversion provides such functionality by way of *repository dump streams*. A repository dump stream (often referred to as a "dump file" when stored as a file on disk) is a portable, flat file format that describes the various revisions in your repository—what was changed, by whom, when, and so on. This dump stream is the primary mechanism used to marshal versioned history—in whole or in part, with or without modification—between repositories. And Subversion provides the tools necessary for creating and loading these dump streams: the **svnadmin dump** and **svnadmin load** subcommands, respectively, and the **svnrdump** program.

While the Subversion repository dump format contains human-readable portions and a familiar structure (it resembles an RFC 822 format, the same type of format used for most email), it is *not* a plain-text file format. It is a binary file format, highly sensitive to meddling. For example, many text editors will corrupt the file by automatically converting line endings.

There are many reasons for dumping and loading Subversion repository data. Early in Subversion's life, the most common reason was due to the evolution of Subversion itself. As Subversion matured, there were times when changes made to the backend database schema caused compatibility issues with previous versions of the repository, so users had to dump their repository data using the previous version of Subversion and load it into a freshly created repository with the new version of Subversion. Now, these types of schema changes haven't occurred since Subversion's 1.0 release, and the Subversion developers promise not to force users to dump and load their repositories when upgrading between minor versions (such as from 1.3 to 1.4) of Subversion. But there are still other reasons for dumping and loading, including re-deploying a Berkeley DB repository on a new OS or CPU architecture, switching between the Berkeley DB and FSFS backends, or (as we'll cover later in this chapter in the section called "Filtering Repository History") purging versioned data from repository history.

The Subversion repository dump format describes versioned repository changes only. It will not carry any information about uncommitted transactions, user locks on filesystem paths, repository or server configuration customizations (including hook scripts), and so on.

The Subversion repository dump format also enables conversion from a different storage mechanism or version control system altogether. Because the dump file format is, for the most part, human-readable, it should be relatively easy to describe generic sets of changes—each of which should be treated as a new revision—using this file format. In fact, the **cvs2svn** utility (see the section called "Converting a Repository from CVS to Subversion") uses the dump format to represent the contents of a CVS repository so that those contents can be copied into a Subversion repository.

For now, we'll concern ourselves only with migration of repository data between Subversion repositories, which we'll describe in detail in the sections which follow.

Repository data migration using svnadmin

Whatever your reason for migrating repository history, using the **svnadmin dump** and **svnadmin load** subcommands is straight-

forward. **svnadmin dump** will output a range of repository revisions that are formatted using Subversion's custom filesystem dump format. The dump format is printed to the standard output stream, while informative messages are printed to the standard error stream. This allows you to redirect the output stream to a file while watching the status output in your terminal window. For example:

```
$ svnlook youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
* Dumped revision 25.
* Dumped revision 26.
```

At the end of the process, you will have a single file (`dumpfile` in the previous example) that contains all the data stored in your repository in the requested range of revisions. Note that **svnadmin dump** is reading revision trees from the repository just like any other “reader” process would (e.g., **svn checkout**), so it's safe to run this command at any time.

The other subcommand in the pair, **svnadmin load**, parses the standard input stream as a Subversion repository dump file and effectively replays those dumped revisions into the target repository for that operation. It also gives informative feedback, this time using the standard output stream:

```
$ svnadmin load newrepos < dumpfile
<<< Started new txn, based on original revision 1
     * adding path : A ... done.
     * adding path : A/B ... done.
     …
------- Committed new rev 1 (loaded from original rev 1) >>>

<<< Started new txn, based on original revision 2
     * editing path : A/mu ... done.
     * editing path : A/D/G/rho ... done.

------- Committed new rev 2 (loaded from original rev 2) >>>

…

<<< Started new txn, based on original revision 25
     * editing path : A/D/gamma ... done.

------- Committed new rev 25 (loaded from original rev 25) >>>

<<< Started new txn, based on original revision 26
     * adding path : A/Z/zeta ... done.
     * editing path : A/mu ... done.

------- Committed new rev 26 (loaded from original rev 26) >>>
```

The result of a load is new revisions added to a repository—the same thing you get by making commits against that repository from a regular Subversion client. Just as in a commit, you can use hook programs to perform actions before and after each of the commits made during a load process. By passing the `--use-pre-commit-hook` and `--use-post-commit-hook` options to **svnadmin load**, you can instruct Subversion to execute the pre-commit and post-commit hook programs, respectively, for each loaded revision. You might use these, for example, to ensure that loaded revisions pass through the same validation steps that regu-

lar commits pass through. Of course, you should use these options with care—if your post-commit hook sends emails to a mailing list for each new commit, you might not want to spew hundreds or thousands of commit emails in rapid succession at that list! You can read more about the use of hook scripts in the section called “Implementing Repository Hooks”.

Note that because **svnadmin** uses standard input and output streams for the repository dump and load processes, people who are feeling especially saucy can try things such as this (perhaps even using different versions of **svnadmin** on each side of the pipe):

```
$ svnadmin create newrepos
$ svnadmin dump oldrepos | svnadmin load newrepos
```

By default, the dump file will be quite large—much larger than the repository itself. That's because by default every version of every file is expressed as a full text in the dump file. This is the fastest and simplest behavior, and it's nice if you're piping the dump data directly into some other process (such as a compression program, filtering program, or loading process). But if you're creating a dump file for longer-term storage, you'll likely want to save disk space by using the `--deltas` option. With this option, successive revisions of files will be output as compressed, binary differences—just as file revisions are stored in a repository. This option is slower, but it results in a dump file much closer in size to the original repository.

We mentioned previously that **svnadmin dump** outputs a range of revisions. Use the `--revision` (`-r`) option to specify a single revision, or a range of revisions, to dump. If you omit this option, all the existing repository revisions will be dumped.

```
$ svnadmin dump myrepos -r 23 > rev-23.dumpfile
$ svnadmin dump myrepos -r 100:200 > revs-100-200.dumpfile
```

As Subversion dumps each new revision, it outputs only enough information to allow a future loader to re-create that revision based on the previous one. In other words, for any given revision in the dump file, only the items that were changed in that revision will appear in the dump. The only exception to this rule is the first revision that is dumped with the current **svnadmin dump** command.

By default, Subversion will not express the first dumped revision as merely differences to be applied to the previous revision. For one thing, there is no previous revision in the dump file! And second, Subversion cannot know the state of the repository into which the dump data will be loaded (if it ever is). To ensure that the output of each execution of **svnadmin dump** is self-sufficient, the first dumped revision is, by default, a full representation of every directory, file, and property in that revision of the repository.

However, you can change this default behavior. If you add the `--incremental` option when you dump your repository, **svnadmin** will compare the first dumped revision against the previous revision in the repository—the same way it treats every other revision that gets dumped. It will then output the first revision exactly as it does the rest of the revisions in the dump range—mentioning only the changes that occurred in that revision. The benefit of this is that you can create several small dump files that can be loaded in succession, instead of one large one, like so:

```
$ svnadmin dump myrepos -r 0:1000 > dumpfile1
$ svnadmin dump myrepos -r 1001:2000 --incremental > dumpfile2
$ svnadmin dump myrepos -r 2001:3000 --incremental > dumpfile3
```

These dump files could be loaded into a new repository with the following command sequence:

```
$ svnadmin load newrepos < dumpfile1
$ svnadmin load newrepos < dumpfile2
$ svnadmin load newrepos < dumpfile3
```

Another neat trick you can perform with this `--incremental` option involves appending to an existing dump file a new range of dumped revisions. For example, you might have a `post-commit` hook that simply appends the repository dump of the single revision that triggered the hook. Or you might have a script that runs nightly to append dump file data for all the revisions that were added to the repository since the last time the script ran. Used like this, **svnadmin dump** can be one way to back up changes to your repository over time in case of a system crash or some other catastrophic event.

The dump format can also be used to merge the contents of several different repositories into a single repository. By using the `--parent-dir` option of **svnadmin load**, you can specify a new virtual root directory for the load process. That means if you have dump files for three repositories—say `calc-dumpfile`, `cal-dumpfile`, and `ss-dumpfile`—you can first create a new repository to hold them all:

```
$ svnadmin create /var/svn/projects
$
```

Then, make new directories in the repository that will encapsulate the contents of each of the three previous repositories:

```
$ svn mkdir -m "Initial project roots" \
            file:///var/svn/projects/calc \
            file:///var/svn/projects/calendar \
            file:///var/svn/projects/spreadsheet
Committed revision 1.
$
```

Lastly, load the individual dump files into their respective locations in the new repository:

```
$ svnadmin load /var/svn/projects --parent-dir calc < calc-dumpfile
…
$ svnadmin load /var/svn/projects --parent-dir calendar < cal-dumpfile
…
$ svnadmin load /var/svn/projects --parent-dir spreadsheet < ss-dumpfile
…
$
```

Repository data migration using svnrdump

In Subversion 1.7, **svnrdump** joined the set of stock Subversion tools. It offers fairly specialized functionality, essentially as a network-aware version of the **svnadmin dump** and **svnadmin load** commands which we discuss in depth in the section called “Repository data migration using svnadmin”. **svnrdump dump** will generate a dump stream from a remote repository, spewing it to standard output; **svnrdump load** will read a dump stream from standard input and load it into a remote repository. Using **svnrdump**, you can generate incremental dumps just as you might with **svnadmin dump**. You can even dump a subtree of the repository—something that **svnadmin dump** cannot do.

The primary difference is that instead of requiring direct access to the repository, **svnrdump** operates remotely, using the very same Repository Access (RA) protocols that the Subversion client does. As such, you might need to provide authentication credentials. Also, your remote interations are subject to any authorization limitations configured on the Subversion server.

svnrdump dump requires that the remote server be running Subversion 1.4 or newer. It currently generates dump streams only of the sort which are created when you pass the `--deltas` option to **svnadmin dump**. This isn't interesting in the typical use-cases, but might impact specific types of custom transformations you might wish to apply to the resulting dump stream.

Because it modifies revision properties after committing new revisions, **svnrdump load** requires that the target repository have revision property changes enabled via the pre-revprop-change hook. See pre-revprop-change in Chapter 9, *Subversion Complete Reference* for details.

As you might expect, you can use **svnadmin** and **svnrdump** in concert. You can, for example, use **svnrdump dump** to generate a dump stream from a remote repository, and pipe the results thereof through **svnadmin load** to copy all that repository history into a local repository. Or you can do the reverse, copying history from a local repository into a remote one.

By using `file://` URLs, **svnrdump** can also access local repositories, but it will be doing so via Subversion's Repository Access (RA) abstraction layer—you'll get better performance out of **svnadmin** in such situations.

Filtering Repository History

Since Subversion stores your versioned history using, at the very least, binary differencing algorithms and data compression (optionally in a completely opaque database system), attempting manual tweaks is unwise if not quite difficult, and at any rate strongly discouraged. And once data has been stored in your repository, Subversion generally doesn't provide an easy way to remove that data.[9] But inevitably, there will be times when you would like to manipulate the history of your repository. You might need to strip out all instances of a file that was accidentally added to the repository (and shouldn't be there for whatever reason).[10] Or, perhaps you have multiple projects sharing a single repository, and you decide to split them up into their own repositories. To accomplish tasks such as these, administrators need a more manageable and malleable representation of the data in their repositories—the Subversion repository dump format.

As we described earlier in the section called "Migrating Repository Data Elsewhere", the Subversion repository dump format is a human-readable representation of the changes that you've made to your versioned data over time. Use the **svnadmin dump** or **svnrdump dump** command to generate the dump data, and **svnadmin load** or **svnrdump load** to populate a new repository with it. The great thing about the human-readability aspect of the dump format is that, if you aren't careless about it, you can manually inspect and modify it. Of course, the downside is that if you have three years' worth of repository activity encapsulated in what is likely to be a very large dump file, it could take you a long, long time to manually inspect and modify it.

That's where **svndumpfilter** becomes useful. This program acts as a path-based filter for repository dump streams. Simply give it either a list of paths you wish to keep or a list of paths you wish to not keep, and then pipe your repository dump data through this filter. The result will be a modified stream of dump data that contains only the versioned paths you (explicitly or implicitly) requested.

Let's look at a realistic example of how you might use this program. Earlier in this chapter (see the section called "Planning Your Repository Organization"), we discussed the process of deciding how to choose a layout for the data in your repositories—using one repository per project or combining them, arranging stuff within your repository, and so on. But sometimes after new revisions start flying in, you rethink your layout and would like to make some changes. A common change is the decision to move multiple projects that are sharing a single repository into separate repositories for each project.

Our imaginary repository contains three projects: `calc`, `calendar`, and `spreadsheet`. They have been living side-by-side in a layout like this:

[9] That's rather the reason you use version control at all, right?

[10] Conscious, cautious removal of certain bits of versioned data is actually supported by real use cases. That's why an "obliterate" feature has been one of the most highly requested Subversion features, and one which the Subversion developers hope to soon provide.

```
/
  calc/
    trunk/
    branches/
    tags/
  calendar/
    trunk/
    branches/
    tags/
  spreadsheet/
    trunk/
    branches/
    tags/
```

To get these three projects into their own repositories, we first dump the whole repository:

```
$ svnadmin dump /var/svn/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

Next, run that dump file through the filter, each time including only one of our top-level directories. This results in three new dump files:

```
$ svndumpfilter include calc < repos-dumpfile > calc-dumpfile
…
$ svndumpfilter include calendar < repos-dumpfile > cal-dumpfile
…
$ svndumpfilter include spreadsheet < repos-dumpfile > ss-dumpfile
…
$
```

At this point, you have to make a decision. Each of your dump files will create a valid repository, but will preserve the paths exactly as they were in the original repository. This means that even though you would have a repository solely for your `calc` project, that repository would still have a top-level directory named `calc`. If you want your `trunk`, `tags`, and `branches` directories to live in the root of your repository, you might wish to edit your dump files, tweaking the `Node-path` and `Node-copyfrom-path` headers so that they no longer have that first `calc/` path component. Also, you'll want to remove the section of dump data that creates the `calc` directory. It will look something like the following:

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0
```

If you do plan on manually editing the dump file to remove a top-level directory, make sure your editor is not set to automatically convert end-of-line characters to the native format (e.g., `\r\n` to `\n`), as the content will then not agree with the metadata. This will render the dump file useless.

All that remains now is to create your three new repositories, and load each dump file into the right repository, ignoring the UUID found in the dump stream:

```
$ svnadmin create calc
$ svnadmin load --ignore-uuid calc < calc-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : button.c ... done.
…
$ svnadmin create calendar
$ svnadmin load --ignore-uuid calendar < cal-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : cal.c ... done.
…
$ svnadmin create spreadsheet
$ svnadmin load --ignore-uuid spreadsheet < ss-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : ss.c ... done.
…
$
```

Both of **svndumpfilter**'s subcommands accept options for deciding how to deal with “empty” revisions. If a given revision contains only changes to paths that were filtered out, that now-empty revision could be considered uninteresting or even unwanted. So to give the user control over what to do with those revisions, **svndumpfilter** provides the following command-line options:

`--drop-empty-revs`
: Do not generate empty revisions at all—just omit them.

`--renumber-revs`
: If empty revisions are dropped (using the `--drop-empty-revs` option), change the revision numbers of the remaining revisions so that there are no gaps in the numeric sequence.

`--preserve-revprops`
: If empty revisions are not dropped, preserve the revision properties (log message, author, date, custom properties, etc.) for those empty revisions. Otherwise, empty revisions will contain only the original datestamp, and a generated log message that indicates that this revision was emptied by **svndumpfilter**.

While **svndumpfilter** can be very useful and a huge timesaver, there are unfortunately a couple of gotchas. First, this utility is overly sensitive to path semantics. Pay attention to whether paths in your dump file are specified with or without leading slashes. You'll want to look at the `Node-path` and `Node-copyfrom-path` headers.

```
…
Node-path: spreadsheet/Makefile
…
```

If the paths have leading slashes, you should include leading slashes in the paths you pass to **svndumpfilter include** and **svndumpfilter exclude** (and if they don't, you shouldn't). Further, if your dump file has an inconsistent usage of leading slashes for some reason,[11] you should probably normalize those paths so that they all have, or all lack, leading slashes.

Also, copied paths can give you some trouble. Subversion supports copy operations in the repository, where a new path is created by copying some already existing path. It is possible that at some point in the lifetime of your repository, you might have copied a file or directory from some location that **svndumpfilter** is excluding, to a location that it is including. To make the dump data self-sufficient, **svndumpfilter** needs to still show the addition of the new path—including the contents of any files created by the copy—and not represent that addition as a copy from a source that won't exist in your filtered dump data stream. But because the Subversion repository dump format shows only what was changed in each revision, the contents of the copy source might not be readily available. If you suspect that you have any copies of this sort in your repository, you might want to rethink your set of included/excluded paths, perhaps including the paths that served as sources of your troublesome copy operations, too.

Finally, **svndumpfilter** takes path filtering quite literally. If you are trying to copy the history of a project rooted at `trunk/my-project` and move it into a repository of its own, you would, of course, use the **svndumpfilter include** command to keep all the changes in and under `trunk/my-project`. But the resultant dump file makes no assumptions about the repository into which you plan to load this data. Specifically, the dump data might begin with the revision that added the `trunk/my-project` directory, but it will *not* contain directives that would create the `trunk` directory itself (because `trunk` doesn't match the include filter). You'll need to make sure that any directories that the new dump stream expects to exist actually do exist in the target repository before trying to load the stream into that repository.

Repository Replication

There are several scenarios in which it is quite handy to have a Subversion repository whose version history is exactly the same as some other repository's. Perhaps the most obvious one is the maintenance of a simple backup repository, used when the primary repository has become inaccessible due to a hardware failure, network outage, or other such annoyance. Other scenarios include deploying mirror repositories to distribute heavy Subversion load across multiple servers, use as a soft-upgrade mechanism, and so on.

Subversion provides a program for managing scenarios such as these. **svnsync** works by essentially asking the Subversion server to "replay" revisions, one at a time. It then uses that revision information to mimic a commit of the same to another repository. Neither repository needs to be locally accessible to the machine on which **svnsync** is running—its parameters are repository URLs, and it does all its work through Subversion's Repository Access (RA) interfaces. All it requires is read access to the source repository and read/write access to the destination repository.

When using **svnsync** against a remote source repository, the Subversion server for that repository must be running Subversion version 1.4 or later.

Replication with svnsync

Assuming you already have a source repository that you'd like to mirror, the next thing you need is a target repository that will actually serve as that mirror. This target repository can use either of the available filesystem data-store backends (see the section called "Choosing a Data Store")—Subversion's abstraction layers ensure that such details don't matter. But by default, it must not yet have any version history in it. (We'll discuss an exception to this later in this section.)

The protocol that **svnsync** uses to communicate revision information is highly sensitive to mismatches between the versioned histories contained in the source and target repositories. For this reason, while **svnsync** cannot *demand* that the target repository be

[11] While **svnadmin dump** has a consistent leading slash policy (to not include them), other programs that generate dump data might not be so consistent.

read-only,[12] allowing the revision history in the target repository to change by any mechanism other than the mirroring process is a recipe for disaster.

Do *not* modify a mirror repository in such a way as to cause its version history to deviate from that of the repository it mirrors. The only commits and revision property modifications that ever occur on that mirror repository should be those performed by the **svnsync** tool.

Another requirement of the target repository is that the **svnsync** process be allowed to modify revision properties. Because **svnsync** works within the framework of that repository's hook system, the default state of the repository (which is to disallow revision property changes; see pre-revprop-change in Chapter 9, *Subversion Complete Reference*) is insufficient. You'll need to explicitly implement the pre-revprop-change hook, and your script must allow **svnsync** to set and change revision properties. With those provisions in place, you are ready to start mirroring repository revisions.

It's a good idea to implement authorization measures that allow your repository replication process to perform its tasks while preventing other users from modifying the contents of your mirror repository at all.

Let's walk through the use of **svnsync** in a somewhat typical mirroring scenario. We'll pepper this discourse with practical recommendations, which you are free to disregard if they aren't required by or suitable for your environment.

We will be mirroring the public Subversion repository which houses the source code for this very book and exposing that mirror publicly on the Internet, hosted on a different machine than the one on which the original Subversion source code repository lives. This remote host has a global configuration that permits anonymous users to read the contents of repositories on the host, but requires users to authenticate to modify those repositories. (Please forgive us for glossing over the details of Subversion server configuration for the moment—those are covered thoroughly in Chapter 6, *Server Configuration*.) And for no other reason than that it makes for a more interesting example, we'll be driving the replication process from a third machine—the one that we currently find ourselves using.

First, we'll create the repository which will be our mirror. This and the next couple of steps do require shell access to the machine on which the mirror repository will live. Once the repository is all configured, though, we shouldn't need to touch it directly again.

```
$ ssh admin@svn.example.com "svnadmin create /var/svn/svn-mirror"
admin@svn.example.com's password: ********
$
```

At this point, we have our repository, and due to our server's configuration, that repository is now “live” on the Internet. Now, because we don't want anything modifying the repository except our replication process, we need a way to distinguish that process from other would-be committers. To do so, we use a dedicated username for our process. Only commits and revision property modifications performed by the special username `syncuser` will be allowed.

We'll use the repository's hook system both to allow the replication process to do what it needs to do and to enforce that only it is doing those things. We accomplish this by implementing two of the repository event hooks—pre-revprop-change and start-commit. Our `pre-revprop-change` hook script is found in Example 5.2, “Mirror repository's pre-revprop-change hook script”, and basically verifies that the user attempting the property changes is our `syncuser` user. If so, the change is allowed; otherwise, it is denied.

Example 5.2. Mirror repository's pre-revprop-change hook script

```
#!/bin/sh
```

[12] In fact, it can't truly be read-only, or **svnsync** itself would have a tough time copying revision history into it.

```
USER="$3"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may change revision properties" >&2
exit 1
```

That covers revision property changes. Now we need to ensure that only the `syncuser` user is permitted to commit new revisions to the repository. We do this using a `start-commit` hook script such as the one in Example 5.3, “Mirror repository's start-commit hook script”.

Example 5.3. Mirror repository's start-commit hook script

```
#!/bin/sh

USER="$2"

if [ "$USER" = "syncuser" ]; then exit 0; fi

echo "Only the syncuser user may commit new revisions" >&2
exit 1
```

After installing our hook scripts and ensuring that they are executable by the Subversion server, we're finished with the setup of the mirror repository. Now, we get to actually do the mirroring.

The first thing we need to do with **svnsync** is to register in our target repository the fact that it will be a mirror of the source repository. We do this using the **svnsync initialize** subcommand. The URLs we provide point to the root directories of the target and source repositories, respectively. In Subversion 1.4, this is required—only full mirroring of repositories is permitted. Beginning with Subversion 1.5, though, you can use **svnsync** to mirror only some subtree of the repository, too.

```
$ svnsync help init
initialize (init): usage: svnsync initialize DEST_URL SOURCE_URL

Initialize a destination repository for synchronization from
another repository.
…
$ svnsync initialize http://svn.example.com/svn-mirror \
                     http://svnbook.googlecode.com/svn \
                     --sync-username syncuser --sync-password syncpass
Copied properties for revision 0 (svn:sync-* properties skipped).
NOTE: Normalized svn:* properties to LF line endings (1 rev-props, 0 node-props).
$
```

Our target repository will now remember that it is a mirror of the public Subversion source code repository. Notice that we provided a username and password as arguments to **svnsync**—that was required by the pre-revprop-change hook on our mirror repository.

In Subversion 1.4, the values given to **svnsync**'s `--username` and `--password` command-line options were used for authentication against both the source and destination repositories. This caused problems when a user's credentials

weren't exactly the same for both repositories, especially when running in noninteractive mode (with the `--non-interactive` option). This was fixed in Subversion 1.5 with the introduction of two new pairs of options. Use `--source-username` and `--source-password` to provide authentication credentials for the source repository; use `--sync-username` and `--sync-password` to provide credentials for the destination repository. (The old `--username` and `--password` options still exist for compatibility, but we advise against using them.)

And now comes the fun part. With a single subcommand, we can tell **svnsync** to copy all the as-yet-unmirrored revisions from the source repository to the target.[13] The **svnsync synchronize** subcommand will peek into the special revision properties previously stored on the target repository and determine how much of the source repository has been previously mirrored—in this case, the most recently mirrored revision is r0. Then it will query the source repository and determine what the latest revision in that repository is. Finally, it asks the source repository's server to start replaying all the revisions between 0 and that latest revision. As **svnsync** gets the resultant response from the source repository's server, it begins forwarding those revisions to the target repository's server as new commits.

```
$ svnsync help synchronize
synchronize (sync): usage: svnsync synchronize DEST_URL [SOURCE_URL]

Transfer all pending revisions to the destination from the source
with which it was initialized.
…
$ svnsync synchronize http://svn.example.com/svn-mirror \
                      http://svnbook.googlecode.com/svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Transmitting file data .
Committed revision 3.
Copied properties for revision 3.
…
Transmitting file data .
Committed revision 4063.
Copied properties for revision 4063.
Transmitting file data .
Committed revision 4064.
Copied properties for revision 4064.
Transmitting file data ....
Committed revision 4065.
Copied properties for revision 4065.
$
```

Of particular interest here is that for each mirrored revision, there is first a commit of that revision to the target repository, and then property changes follow. This two-phase replication is required because the initial commit is performed by (and attributed to) the user `syncuser` and is datestamped with the time as of that revision's creation. **svnsync** has to follow up with an immediate series of property modifications that copy into the target repository all the original revision properties found for that revision in the source repository, which also has the effect of fixing the author and datestamp of the revision to match that of the source repository.

Also noteworthy is that **svnsync** performs careful bookkeeping that allows it to be safely interrupted and restarted without ruining the integrity of the mirrored data. If a network glitch occurs while mirroring a repository, simply repeat the **svnsync synchronize** command, and it will happily pick up right where it left off. In fact, as new revisions appear in the source repository, this is exactly what you do to keep your mirror up to date.

[13] Be forewarned that while it will take only a few seconds for the average reader to parse this paragraph and the sample output that follows it, the actual time required to complete such a mirroring operation is, shall we say, quite a bit longer.

As part of its bookkeeping, **svnsync** records in the mirror repository the URL with which the mirror was initialized. Because of this, invocations of **svnsync** which follow the initialization step do not *require* that you provide the source URL on the command line again. However, for security purposes, we recommend that you continue to do so. Depending on how it is deployed, it may not be safe for **svnsync** to trust the source URL which it retrieves from the mirror repository, and from which it pulls versioned data.

svnsync Bookkeeping

svnsync needs to be able to set and modify revision properties on the mirror repository because those properties are part of the data it is tasked with mirroring. As those properties change in the source repository, those changes need to be reflected in the mirror repository, too. But **svnsync** also uses a set of custom revision properties—stored in revision 0 of the mirror repository—for its own internal bookkeeping. These properties contain information such as the URL and UUID of the source repository, plus some additional state-tracking information.

One of those pieces of state-tracking information is a flag that essentially just means “there's a synchronization in progress right now.” This is used to prevent multiple **svnsync** processes from colliding with each other while trying to mirror data to the same destination repository. Now, generally you won't need to pay any attention whatsoever to *any* of these special properties (all of which begin with the prefix `svn:sync-`). Occasionally, though, if a synchronization fails unexpectedly, Subversion never has a chance to remove this particular state flag. This causes all future synchronization attempts to fail because it appears that a synchronization is still in progress when, in fact, none is. Fortunately, recovering from this situation is easy to do. In Subversion 1.7, you can use the newly introduced `--steal-lock` option with **svnsync**'s commands. In previous Subversion versions, you need only to remove the `svn:sync-lock` property which serves as this flag from revision 0 of the mirror repository:

```
$ svn propdel --revprop -r0 svn:sync-lock http://svn.example.com/svn-mirror
property 'svn:sync-lock' deleted from repository revision 0
$
```

Also, **svnsync** stores the source repository URL provided at mirror initialization time in a bookkeeping property on the mirror repository. Future synchronization operations against that mirror which omit the source URL at the command line will consult the special `svn:sync-from-url` property stored on the mirror itself to know where to synchronize from. This value is used literally by the synchronization process, though. Be wary of using non-fully-qualified domain names (such as referring to `svnbook.red-bean.com` as simply `svnbook` because that happens to work when you are connected directly to the `red-bean.com` network), domain names which don't resolve or resolve differently depending on where you happen to be operating from, or IP addresses (which can change over time). But here again, if you need an existing mirror to start referring to a different URL for the same source repository, you can change the bookkeeping property which houses that information. Users of Subversion 1.7 or better can use **svnsync init --allow-non-empty** to reinitialize their mirrors with new source URL:

```
$ svnsync initialize --allow-non-empty http://svn.example.com/svn-mirror \
                                       NEW-SOURCE-URL
Copied properties for revision 4065.
$
```

If you are running an older version of Subversion, you'll need to manually tweak the `svn:sync-from-url` bookkeeping property:

```
$ svn propset --revprop -r0 svn:sync-from-url NEW-SOURCE-URL \
      http://svn.example.com/svn-mirror
property 'svn:sync-from-url' set on repository revision 0
$
```

Another interesting thing about these special bookkeeping properties is that **svnsync** will not attempt to mirror any of those properties when they are found in the source repository. The reason is probably obvious, but basically boils down to **svnsync** not being able to distinguish the special properties it has merely copied from the source repository from those it needs to consult and maintain for its own bookkeeping needs. This situation could occur if, for example, you were maintaining a mirror of a mirror of a third repository. When **svnsync** sees its own special properties in revision 0 of the source repository, it simply ignores them.

An **svnsync info** subcommand was added in Subversion 1.6 to easily display the special bookkeeping properties in the destination repository.

```
$ svnsync help info
info: usage: svnsync info DEST_URL

Print information about the synchronization destination repository
located at DEST_URL.
…
$ svnsync info http://svn.example.com/svn-mirror
Source URL: http://svnbook.googlecode.com/svn
Source Repository UUID: 931749d0-5854-0410-9456-f14be4d6b398
Last Merged Revision: 4065
$
```

There is, however, one bit of inelegance in the process. Because Subversion revision properties can be changed at any time throughout the lifetime of the repository, and because they don't leave an audit trail that indicates when they were changed, replication processes have to pay special attention to them. If you've already mirrored the first 15 revisions of a repository and someone then changes a revision property on revision 12, **svnsync** won't know to go back and patch up its copy of revision 12. You'll need to tell it to do so manually by using (or with some additional tooling around) the **svnsync copy-revprops** subcommand, which simply rereplicates all the revision properties for a particular revision or range thereof.

```
$ svnsync help copy-revprops
copy-revprops: usage:

    1. svnsync copy-revprops DEST_URL [SOURCE_URL]
    2. svnsync copy-revprops DEST_URL REV[:REV2]

…
$ svnsync copy-revprops http://svn.example.com/svn-mirror 12
Copied properties for revision 12.
$
```

That's repository replication via **svnsync** in a nutshell. You'll likely want some automation around such a process. For example, while our example was a pull-and-push setup, you might wish to have your primary repository push changes to one or more blessed mirrors as part of its post-commit and post-revprop-change hook implementations. This would enable the mirror to be up to date in as near to real time as is likely possible.

Partial replication with svnsync

svnsync isn't limited to full copies of everything which lives in a repository. It can handle various shades of partial replication, too. For example, while it isn't very commonplace to do so, **svnsync** does gracefully mirror repositories in which the user as whom it authenticates has only partial read access. It simply copies only the bits of the repository that it is permitted to see. Obviously, such a mirror is not useful as a backup solution.

As of Subversion 1.5, **svnsync** also has the ability to mirror a subset of a repository rather than the whole thing. The process of setting up and maintaining such a mirror is exactly the same as when mirroring a whole repository, except that instead of specifying the source repository's root URL when running **svnsync init**, you specify the URL of some subdirectory within that repository. Synchronization to that mirror will now copy only the bits that changed under that source repository subdirectory. There are some limitations to this support, though. First, you can't mirror multiple disjoint subdirectories of the source repository into a single mirror repository—you'd need to instead mirror some parent directory that is common to both. Second, the filtering logic is entirely path-based, so if the subdirectory you are mirroring was renamed at some point in the past, your mirror would contain only the revisions since the directory appeared at the URL you specified. And likewise, if the source subdirectory is renamed in the future, your synchronization processes will stop mirroring data at the point that the source URL you specified is no longer valid.

A quick trick for mirror creation

We mentioned previously the cost of setting up an initial mirror of an existing repository. For many folks, the sheer cost of transmitting thousands—or millions—of revisions of history to a new mirror repository via **svnsync** is a show-stopper. Fortunately, Subversion 1.7 provides a workaround by way of a new `--allow-non-empty` option to **svnsync initialize**. This option allows you to initialize one repository as a mirror of another while bypassing the verification that the to-be-initialized mirror has no version history present in it. Per our previous warnings about the sensitivity of this whole replication process, you should rightly discern that this is an option to be used only with great caution. But it's wonderfully handy when you have administrative access to the source repository, where you can simply make a physical copy of the repository and then initialize that copy as a new mirror:

```
$ svnadmin hotcopy /path/to/repos /path/to/mirror-repos
$ ### create /path/to/mirror-repos/hooks/pre-revprop-change
$ svnsync initialize file:///path/to/mirror-repos \
                     file:///path/to/repos
svnsync: E000022: Destination repository already contains revision history; co
nsider using --allow-non-empty if the repository's revisions are known to mirr
or their respective revisions in the source repository
$ svnsync initialize --allow-non-empty file:///path/to/mirror-repos \
                                       file:///path/to/repos
Copied properties for revision 32042.
$
```

Admins who are running a version of Subversion prior to 1.7 (and thus do not have access to **svnsync initialize**'s `--allow-non-empty` feature) can accomplish effectively the same thing that that feature does through *careful* manipulation of the r0 revision properties on the copy of the repository which is slated to become a mirror of the original. Use **svnadmin setrevprop** to create the same bookkeeping properties that **svnsync** would have created there.

Replication wrap-up

We've discussed a couple of ways to replicate revision history from one repository to another. So let's look now at the user end of these operations. How does replication and the various situations which call for it affect Subversion clients?

As far as user interaction with repositories and mirrors goes, it *is* possible to have a single working copy that interacts with both, but you'll have to jump through some hoops to make it happen. First, you need to ensure that both the primary and mirror repositories have the same repository UUID (which is not the case by default). See the section called “Managing Repository UUIDs” later in this chapter for more about this.

Once the two repositories have the same UUID, you can use **svn relocate** to point your working copy to whichever of the repositories you wish to operate against, a process that is described in svn relocate in Chapter 9, *Subversion Complete Reference*. There is a possible danger here, though, in that if the primary and mirror repositories aren't in close synchronization, a working copy up to date with, and pointing to, the primary repository will, if relocated to point to an out-of-date mirror, become confused about the apparent sudden loss of revisions it fully expects to be present, and it will throw errors to that effect. If this occurs, you can relocate your working copy back to the primary repository and then either wait until the mirror repository is up to date, or backdate your working copy to a revision you know is present in the sync repository, and then retry the relocation.

Finally, be aware that the revision-based replication provided by **svnsync** is only that—replication of revisions. Only the kinds of information carried by the Subversion repository dump file format are available for replication. As such, tools such as **svnsync** (and **svnrdump**, which we discuss in the section called "Repository data migration using svnrdump") are limited in ways similar to that of the repository dump stream. They do not include in their replicated information such things as the hook implementations, repository or server configuration data, uncommitted transactions, or information about user locks on repository paths.

Repository Backup

Despite numerous advances in technology since the birth of the modern computer, one thing unfortunately rings true with crystalline clarity—sometimes things go very, very awry. Power outages, network connectivity dropouts, corrupt RAM, and crashed hard drives are but a taste of the evil that Fate is poised to unleash on even the most conscientious administrator. And so we arrive at a very important topic—how to make backup copies of your repository data.

There are two types of backup methods available for Subversion repository administrators—full and incremental. A full backup of the repository involves squirreling away in one sweeping action all the information required to fully reconstruct that repository in the event of a catastrophe. Usually, it means, quite literally, the duplication of the entire repository directory (which includes either a Berkeley DB or FSFS environment). Incremental backups are lesser things: backups of only the portion of the repository data that has changed since the previous backup.

As far as full backups go, the naïve approach might seem like a sane one, but unless you temporarily disable all other access to your repository, simply doing a recursive directory copy runs the risk of generating a faulty backup. In the case of Berkeley DB, the documentation describes a certain order in which database files can be copied that will guarantee a valid backup copy. A similar ordering exists for FSFS data. But you don't have to implement these algorithms yourself, because the Subversion development team has already done so. The **svnadmin hotcopy** command takes care of the minutia involved in making a hot backup of your repository. And its invocation is as trivial as the Unix **cp** or Windows **copy** operations:

```
$ svnadmin hotcopy /var/svn/repos /var/svn/repos-backup
```

The resultant backup is a fully functional Subversion repository, able to be dropped in as a replacement for your live repository should something go horribly wrong.

When making copies of a Berkeley DB repository, you can even instruct **svnadmin hotcopy** to purge any unused Berkeley DB logfiles (see the section called "Purging unused Berkeley DB logfiles") from the original repository upon completion of the copy. Simply provide the `--clean-logs` option on the command line.

```
$ svnadmin hotcopy --clean-logs /var/svn/bdb-repos /var/svn/bdb-repos-backup
```

Additional tooling around this command is available, too. The `tools/backup/` directory of the Subversion source distribution holds the **hot-backup.py** script. This script adds a bit of backup management atop **svnadmin hotcopy**, allowing you to keep only the most recent configured number of backups of each repository. It will automatically manage the names of the backed-up repository directories to avoid collisions with previous backups and will "rotate off" older backups, deleting them so that only the most recent ones remain. Even if you also have an incremental backup, you might want to run this program on a regular basis. For example, you might consider using **hot-backup.py** from a program scheduler (such as **cron** on Unix systems), which can cause it to

run nightly (or at whatever granularity of time you deem safe).

Some administrators use a different backup mechanism built around generating and storing repository dump data. We described in the section called "Migrating Repository Data Elsewhere" how to use **svnadmin dump** with the `--incremental` option to perform an incremental backup of a given revision or range of revisions. And of course, you can achieve a full backup variation of this by omitting the `--incremental` option to that command. There is some value in these methods, in that the format of your backed-up information is flexible—it's not tied to a particular platform, versioned filesystem type, or release of Subversion or Berkeley DB. But that flexibility comes at a cost, namely that restoring that data can take a long time—longer with each new revision committed to your repository. Also, as is the case with so many of the various backup methods, revision property changes that are made to already backed-up revisions won't get picked up by a nonoverlapping, incremental dump generation. For these reasons, we recommend against relying solely on dump-based backup approaches.

As you can see, each of the various backup types and methods has its advantages and disadvantages. The easiest is by far the full hot backup, which will always result in a perfect working replica of your repository. Should something bad happen to your live repository, you can restore from the backup with a simple recursive directory copy. Unfortunately, if you are maintaining multiple backups of your repository, these full copies will each eat up just as much disk space as your live repository. Incremental backups, by contrast, tend to be quicker to generate and smaller to store. But the restoration process can be a pain, often involving applying multiple incremental backups. And other methods have their own peculiarities. Administrators need to find the balance between the cost of making the backup and the cost of restoring it.

The **svnsync** program (see the section called "Repository Replication") actually provides a rather handy middle-ground approach. If you are regularly synchronizing a read-only mirror with your main repository, in a pinch your read-only mirror is probably a good candidate for replacing that main repository if it falls over. The primary disadvantage of this method is that only the versioned repository data gets synchronized—repository configuration files, user-specified repository path locks, and other items that might live in the physical repository directory but not *inside* the repository's virtual versioned filesystem are not handled by **svnsync**.

In any backup scenario, repository administrators need to be aware of how modifications to unversioned revision properties affect their backups. Since these changes do not themselves generate new revisions, they will not trigger post-commit hooks, and may not even trigger the pre-revprop-change and post-revprop-change hooks.[14] And since you can change revision properties without respect to chronological order—you can change any revision's properties at any time—an incremental backup of the latest few revisions might not catch a property modification to a revision that was included as part of a previous backup.

Generally speaking, only the truly paranoid would need to back up their entire repository, say, every time a commit occurred. However, assuming that a given repository has some other redundancy mechanism in place with relatively fine granularity (such as per-commit emails or incremental dumps), a hot backup of the database might be something that a repository administrator would want to include as part of a system-wide nightly backup. It's your data—protect it as much as you'd like.

Often, the best approach to repository backups is a diversified one that leverages combinations of the methods described here. The Subversion developers, for example, back up the Subversion source code repository nightly using **hot-backup.py** and an off-site **rsync** of those full backups; keep multiple archives of all the commit and property change notification emails; and have repository mirrors maintained by various volunteers using **svnsync**. Your solution might be similar, but should be catered to your needs and that delicate balance of convenience with paranoia. And whatever you do, validate your backups from time to time—what good is a spare tire that has a hole in it? While all of this might not save your hardware from the iron fist of Fate,[15] it should certainly help you recover from those trying times.

Managing Repository UUIDs

Subversion repositories have a universally unique identifier (UUID) associated with them. This is used by Subversion clients to verify the identity of a repository when other forms of verification aren't good enough (such as checking the repository URL, which can change over time). Most Subversion repository administrators rarely, if ever, need to think about repository UUIDs as anything more than a trivial implementation detail of Subversion. Sometimes, however, there is cause for attention to this detail.

As a general rule, you want the UUIDs of your live repositories to be unique. That is, after all, the point of having UUIDs. But there are times when you want the repository UUIDs of two repositories to be exactly the same. For example, if you make a copy of a repository for backup purposes, you want the backup to be a perfect replica of the original so that, in the event that you have to

[14] **svnadmin setlog** can be called in a way that bypasses the hook interface altogether.
[15] You know—the collective term for all of her "fickle fingers."

restore that backup and replace the live repository, users don't suddenly see what looks like a different repository. When dumping and loading repository history (as described earlier in the section called “Migrating Repository Data Elsewhere”), you get to decide whether to apply the UUID encapsulated in the data dump stream to the repository in which you are loading the data. The particular circumstance will dictate the correct behavior.

There are a couple of ways to set (or reset) a repository's UUID, should you need to. As of Subversion 1.5, this is as simple as using the **svnadmin setuuid** command. If you provide this subcommand with an explicit UUID, it will validate that the UUID is well-formed and then set the repository UUID to that value. If you omit the UUID, a brand-new UUID will be generated for your repository.

```
$ svnlook uuid /var/svn/repos
cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
$ svnadmin setuuid /var/svn/repos   # generate a new UUID
$ svnlook uuid /var/svn/repos
3c3c38fe-acc0-11dc-acbc-1b37ff1c8e7c
$ svnadmin setuuid /var/svn/repos \
           cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec  # restore the old UUID
$ svnlook uuid /var/svn/repos
cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
$
```

For folks using versions of Subversion earlier than 1.5, these tasks are a little more complicated. You can explicitly set a repository's UUID by piping a repository dump file stub that carries the new UUID specification through **svnadmin load --force-uuid *REPOS-PATH***.

```
$ svnadmin load --force-uuid /var/svn/repos <<EOF
SVN-fs-dump-format-version: 2

UUID: cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
EOF
$ svnlook uuid /var/svn/repos
cf2b9d22-acb5-11dc-bc8c-05e83ce5dbec
$
```

Having older versions of Subversion generate a brand-new UUID is not quite as simple to do, though. Your best bet here is to find some other way to generate a UUID, and then explicitly set the repository's UUID to that value.

Moving and Removing Repositories

Subversion repository data is wholly contained within the repository directory. As such, you can move a Subversion repository to some other location on disk, rename a repository, copy a repository, or delete a repository altogether using the tools provided by your operating system for manipulating directories—**mv**, **cp -a**, and **rm -r** on Unix platforms; **copy**, **move**, and **rmdir /s /q** on Windows; vast numbers of mouse and menu gyrations in various graphical file explorer applications, and so on.

Of course, there's often still more to be done when trying to cleanly affect changes such as this. For example, you might need to update your Subversion server configuration to point to the new location of a relocated repository or to remove configuration bits for a now-deleted repository. If you have automated processes that publish information from or about your repositories, they may need to be updated. Hook scripts might need to be reconfigured. Users may need to be notified. The list can go on indefinitely, or at least to the extent that you've built processes and procedures around your Subversion repository.

In the case of a copied repository, you should also consider the fact that Subversion uses repository UUIDs to distinguish repositories. If you copy a Subversion repository using a typical shell recursive copy command, you'll wind up with two repositories that are

identical in every way—including their UUIDs. In some circumstances, this might be desirable. But in the instances where it is not, you'll need to generate a new UUID for one of these identical repositories. See the section called “Managing Repository UUIDs” for more about managing repository UUIDs.

Summary

By now you should have a basic understanding of how to create, configure, and maintain Subversion repositories. We introduced you to the various tools that will assist you with this task. Throughout the chapter, we noted common administration pitfalls and offered suggestions for avoiding them.

All that remains is for you to decide what exciting data to store in your repository, and finally, how to make it available over a network. The next chapter is all about networking.

Chapter 6. Server Configuration

A Subversion repository can be accessed simultaneously by clients running on the same machine on which the repository resides using URLs carrying the `file://` scheme. But the typical Subversion setup involves a single server machine being accessed from clients on computers all over the office—or, perhaps, all over the world.

This chapter describes how to get your Subversion repository exposed outside its host machine for use by remote clients. We will cover Subversion's currently available server mechanisms, discussing the configuration and use of each. After reading this chapter, you should be able to decide which networking setup is right for your needs, as well as understand how to enable such a setup on your host computer.

Overview

Subversion was designed with an abstract repository access layer. This means that a repository can be programmatically accessed by any sort of server process, and the client "repository access" API allows programmers to write plug-ins that speak relevant network protocols. In theory, Subversion can use an infinite number of network implementations. In practice, there are only two Subversion servers in widespread use today.

Apache is an extremely popular web server; using the **mod_dav_svn** module, Apache can access a repository and make it available to clients via the WebDAV/DeltaV protocol, which is an extension of HTTP. Because Apache is an extremely extensible server, it provides a number of features "for free," such as encrypted SSL communication, logging, integration with a number of third-party authentication systems, and limited built-in web browsing of repositories.

In the other corner is **svnserve**: a small, lightweight server program that speaks a custom protocol with clients. Because its protocol is explicitly designed for Subversion and is stateful (unlike HTTP), it provides significantly faster network operations—but at the cost of some features as well. While it can use SASL to provide a variety of authentication and encryption options, it has no logging or built-in web browsing. It is, however, extremely easy to set up and is often the best option for small teams just starting out with Subversion.

The network protocol which **svnserve** speaks may also be tunneled over an SSH connection. This deployment option for **svnserve** differs quite a bit in features from a traditional **svnserve** deployment. SSH is used to encrypt all communication. SSH is also used exclusively to authenticate, so real system accounts are required on the server host (unlike vanilla **svnserve**, which has its own private user accounts). Finally, because this setup requires that each user spawn a private, temporary **svnserve** process, it's equivalent (from a permissions point of view) to allowing a group of local users to all access the repository via `file://` URLs. Path-based access control has no meaning, since each user is accessing the repository database files directly.

Table 6.1, "Comparison of subversion server options" provides a quick summary of the three typical server deployments.

Table 6.1. Comparison of subversion server options

Feature	Apache + mod_dav_svn	svnserve	svnserve over SSH
Authentication options	HTTP Basic or Digest auth, X.509 certificates, LDAP, NTLM, or any other mechanism available to Apache httpd	CRAM-MD5 by default; LDAP, NTLM, or any other mechanism available to SASL	SSH
User account options	Private "users" file, or other mechanisms available to Apache httpd (LDAP, SQL, etc.)	Private "users" file, or other mechanisms available to SASL (LDAP, SQL, etc.)	System accounts
Authorization options	Read/write access can be granted over the whole repository, or specified per path	Read/write access can be granted over the whole repository, or specified per path	Read/write access only grantable over the whole repository
Encryption	Available via optional SSL	Available via optional SASL	Inherent in SSH connection

Feature	Apache + mod_dav_svn	svnserve	svnserve over SSH
	(https)	features	
Logging	High-level operational logging of Subversion operations plus detailed logging at the per-HTTP-request level	High-level operational logging only	High-level operational logging only
Interoperability	Accessible by other WebDAV clients	Talks only to svn clients	Talks only to svn clients
Web viewing	Limited built-in support, or via third-party tools such as ViewVC	Only via third-party tools such as ViewVC	Only via third-party tools such as ViewVC
Master-slave server replication	Transparent write-proxying available from slave to master	Can only create read-only slave servers	Can only create read-only slave servers
Speed	Somewhat slower	Somewhat faster	Somewhat faster
Initial setup	Somewhat complex	Extremely simple	Moderately simple

Choosing a Server Configuration

So, which server should you use? Which is best?

Obviously, there's no right answer to that question. Every team has different needs, and the different servers all represent different sets of trade-offs. The Subversion project itself doesn't endorse one server or another, or consider either server more “official” than another.

Here are some reasons why you might choose one deployment over another, as well as reasons you might *not* choose one.

The svnserve Server

Why you might want to use it:

- Quick and easy to set up.
- Network protocol is stateful and noticeably faster than WebDAV.
- No need to create system accounts on server.
- Password is not passed over the network.

Why you might want to avoid it:

- By default, only one authentication method is available, the network protocol is not encrypted, and the server stores clear text passwords. (All these things can be changed by configuring SASL, but it's a bit more work to do.)
- No advanced logging facilities.
- No built-in web browsing. (You'd have to install a separate web server and repository browsing software to add this.)

svnserve over SSH

Why you might want to use it:

- The network protocol is stateful and noticeably faster than WebDAV.
- You can take advantage of existing SSH accounts and user infrastructure.
- All network traffic is encrypted.

Why you might want to avoid it:

- Only one choice of authentication method is available.
- No advanced logging facilities.
- It requires users to be in the same system group, or use a shared SSH key.
- If used improperly, it can lead to file permission problems.

The Apache HTTP Server

Why you might want to use it:

- It allows Subversion to use any of the numerous authentication systems already integrated with Apache.
- There is no need to create system accounts on the server.
- Full Apache logging is available.
- Network traffic can be encrypted via SSL.
- HTTP(S) can usually go through corporate firewalls.
- Built-in repository browsing is available via web browser.
- The repository can be mounted as a network drive for transparent version control (see the section called “Autoversioning”).

Why you might want to avoid it:

- Noticeably slower than **svnserve**, because HTTP is a stateless protocol and requires more network turnarounds.
- Initial setup can be complex.

Recommendations

In general, the authors of this book recommend a vanilla **svnserve** installation for small teams just trying to get started with a Subversion server; it's the simplest to set up and has the fewest maintenance issues. You can always switch to a more complex server deployment as your needs change.

Here are some general recommendations and tips, based on years of supporting users:

- If you're trying to set up the simplest possible server for your group, a vanilla **svnserve** installation is the easiest, fastest route. Note, however, that your repository data will be transmitted in the clear over the network. If your deployment is entirely within your company's LAN or VPN, this isn't an issue. If the repository is exposed to the wide-open Internet, you might want to make sure that either the repository's contents aren't sensitive (e.g., it contains only open source code), or that you go the extra mile in configuring SASL to encrypt network communications.

- If you need to integrate with existing legacy identity systems (LDAP, Active Directory, NTLM, X.509, etc.), you must use either the Apache-based server or **svnserve** configured with SASL.
- If you've decided to use either Apache or stock **svnserve**, create a single **svn** user on your system and run the server process as that user. Be sure to make the repository directory wholly owned by the **svn** user as well. From a security point of view, this keeps the repository data nicely siloed and protected by operating system filesystem permissions, changeable by only the Subversion server process itself.
- If you have an existing infrastructure that is heavily based on SSH accounts, and if your users already have system accounts on your server machine, it makes sense to deploy an **svnserve**-over-SSH solution. Otherwise, we don't widely recommend this option to the public. It's generally considered safer to have your users access the repository via (imaginary) accounts managed by **svnserve** or Apache, rather than by full-blown system accounts. If your deep desire for encrypted communication still draws you to this option, we recommend using Apache with SSL or **svnserve** with SASL encryption instead.
- Do *not* be seduced by the simple idea of having all of your users access a repository directly via `file://` URLs. Even if the repository is readily available to everyone via a network share, this is a bad idea. It removes any layers of protection between the users and the repository: users can accidentally (or intentionally) corrupt the repository database, it becomes hard to take the repository offline for inspection or upgrade, and it can lead to a mess of file permission problems (see the section called "Supporting Multiple Repository Access Methods"). Note that this is also one of the reasons we warn against accessing repositories via `svn+ssh://` URLs—from a security standpoint, it's effectively the same as local users accessing via `file://`, and it can entail all the same problems if the administrator isn't careful.

svnserve, a Custom Server

The **svnserve** program is a lightweight server, capable of speaking to clients over TCP/IP using a custom, stateful protocol. Clients contact an **svnserve** server by using URLs that begin with the `svn://` or `svn+ssh://` scheme. This section will explain the different ways of running **svnserve**, how clients authenticate themselves to the server, and how to configure appropriate access control to your repositories.

Invoking the Server

There are a few different ways to run the **svnserve** program:

- Run **svnserve** as a standalone daemon, listening for requests.
- Have the Unix **inetd** daemon temporarily spawn **svnserve** whenever a request comes in on a certain port.
- Have SSH invoke a temporary **svnserve** over an encrypted tunnel.
- Run **svnserve** as a Microsoft Windows service.
- Run **svnserve** as a launchd job.

The following sections will cover in detail these various deployment options for **svnserve**.

svnserve as daemon

The easiest option is to run **svnserve** as a standalone "daemon" process. Use the `-d` option for this:

```
$ svnserve -d
$               # svnserve is now running, listening on port 3690
```

When running **svnserve** in daemon mode, you can use the `--listen-port` and `--listen-host` options to customize the exact port and hostname to “bind” to.

Once we successfully start **svnserve** as explained previously, it makes every repository on your system available to the network. A client needs to specify an *absolute* path in the repository URL. For example, if a repository is located at `/var/svn/project1`, a client would reach it via `svn://host.example.com/var/svn/project1`. To increase security, you can pass the `-r` option to **svnserve**, which restricts it to exporting only repositories below that path. For example:

```
$ svnserve -d -r /var/svn
…
```

Using the `-r` option effectively modifies the location that the program treats as the root of the remote filesystem space. Clients then use URLs that have that path portion removed from them, leaving much shorter (and much less revealing) URLs:

```
$ svn checkout svn://host.example.com/project1
…
```

svnserve via inetd

If you want **inetd** to launch the process, you need to pass the `-i` (`--inetd`) option. In the following example, we've shown the output from running `svnserve -i` at the command line, but note that this isn't how you actually start the daemon; see the paragraphs following the example for how to configure **inetd** to start **svnserve**.

```
$ svnserve -i
( success ( 2 2 ( ) ( edit-pipeline svndiff1 absent-entries commit-revprops d\
epth log-revprops atomic-revprops partial-replay ) ) )
```

When invoked with the `--inetd` option, **svnserve** attempts to speak with a Subversion client via `stdin` and `stdout` using a custom protocol. This is the standard behavior for a program being run via **inetd**. The IANA has reserved port 3690 for the Subversion protocol, so on a Unix-like system you can add lines to `/etc/services` such as these (if they don't already exist):

```
svn             3690/tcp   # Subversion
svn             3690/udp   # Subversion
```

If your system is using a classic Unix-like **inetd** daemon, you can add this line to `/etc/inetd.conf`:

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

Make sure “svnowner” is a user that has appropriate permissions to access your repositories. Now, when a client connection comes into your server on port 3690, **inetd** will spawn an **svnserve** process to service it. Of course, you may also want to add `-r` to the configuration line as well, to restrict which repositories are exported.

svnserve over a tunnel

Another way to invoke **svnserve** is in tunnel mode, using the `-t` option. This mode assumes that a remote-service program such as **rsh** or **ssh** has successfully authenticated a user and is now invoking a private **svnserve** process *as that user*. (Note that you, the user, will rarely, if ever, have reason to invoke **svnserve** with the `-t` at the command line; instead, the SSH daemon does so for you.) The **svnserve** program behaves normally (communicating via `stdin` and `stdout`) and assumes that the traffic is being automatically redirected over some sort of tunnel back to the client. When **svnserve** is invoked by a tunnel agent like this, be sure that the authenticated user has full read and write access to the repository database files. It's essentially the same as a local user accessing the repository via `file://` URLs.

This option is described in much more detail later in this chapter in the section called “Tunneling over SSH”.

svnserve as a Windows service

If your Windows system is a descendant of Windows NT (Windows 2000 or newer), you can run **svnserve** as a standard Windows service. This is typically a much nicer experience than running it as a standalone daemon with the `--daemon` (`-d`) option. Using daemon mode requires launching a console, typing a command, and then leaving the console window running indefinitely. A Windows service, however, runs in the background, can start at boot time automatically, and can be started and stopped using the same consistent administration interface as other Windows services.

You'll need to define the new service using the command-line tool **SC.EXE**. Much like the **inetd** configuration line, you must specify an exact invocation of **svnserve** for Windows to run at startup time:

```
C:\> sc create svn
        binpath= "C:\svn\bin\svnserve.exe --service -r C:\repos"
        displayname= "Subversion Server"
        depend= Tcpip
        start= auto
```

This defines a new Windows service named `svn` which executes a particular **svnserve.exe** command when started (in this case, rooted at `C:\repos`). There are a number of caveats in the prior example, however.

First, notice that the **svnserve.exe** program must always be invoked with the `--service` option. Any other options to **svnserve** must then be specified on the same line, but you cannot add conflicting options such as `--daemon` (`-d`), `--tunnel`, or `--inetd` (`-i`). Options such as `-r` or `--listen-port` are fine, though. Second, be careful about spaces when invoking the **SC.EXE** command: the `key= value` patterns must have no spaces between `key=` and must have exactly one space before the `value`. Lastly, be careful about spaces in your command line to be invoked. If a directory name contains spaces (or other characters that need escaping), place the entire inner value of `binpath` in double quotes, by escaping them:

```
C:\> sc create svn
        binpath= "\"C:\program files\svn\bin\svnserve.exe\" --service -r C:\repos"
        displayname= "Subversion Server"
        depend= Tcpip
        start= auto
```

Also note that the word `binpath` is misleading—its value is a *command line*, not the path to an executable. That's why you need to surround it with quotes if it contains embedded spaces.

Once the service is defined, it can be stopped, started, or queried using standard GUI tools (the Services administrative control panel), or at the command line:

```
C:\> net stop svn
C:\> net start svn
```

The service can also be uninstalled (i.e., undefined) by deleting its definition: **`sc delete svn`**. Just be sure to stop the service first! The **SC.EXE** program has many other subcommands and options; run **`sc /?`** to learn more about it.

svnserve as a launchd job

Mac OS X (10.4 and higher) uses **launchd** to manage processes (including daemons) both system-wide and per-user. A **launchd** job is specified by parameters in an XML property list file, and the **launchctl** command is used to manage the lifecycle of those jobs.

When configured to run as a **launchd** job, **svnserve** is automatically launched on demand whenever incoming Subversion `svn://` network traffic needs to be handled. This is far more convenient than a configuration which requires you to manually invoke **svnserve** as a long-running background process.

To configure **svnserve** as a **launchd** job, first create a job definition file named `/Library/LaunchDaemons/org.apache.subversion.svnserve.plist`. Example 6.1, "A sample svnserve launchd job definition" provides an example of such a file.

Example 6.1. A sample svnserve launchd job definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>Label</key>
        <string>org.apache.subversion.svnserve</string>
        <key>ServiceDescription</key>
        <string>Host Subversion repositories using svn:// scheme</string>
        <key>ProgramArguments</key>
        <array>
            <string>/usr/bin/svnserve</string>
            <string>--inetd</string>
            <string>--root=/var/svn</string>
        </array>
        <key>UserName</key>
        <string>svn</string>
        <key>GroupName</key>
        <string>svn</string>
        <key>inetdCompatibility</key>
        <dict>
            <key>Wait</key>
            <false/>
        </dict>
        <key>Sockets</key>
        <dict>
            <key>Listeners</key>
            <array>
                <dict>
                    <key>SockServiceName</key>
                    <string>svn</string>
                    <key>Bonjour</key>
                    <true/>
```

```
                </dict>
            </array>
        </dict>
    </dict>
</plist>
```

The **launchd** system can be somewhat challenging to learn. Fortunately, documentation exists for the commands described in this section. For example, run **`man launchd`** from the command line to see the manual page for **launchd** itself, **`man launchd.plist`** to read about the job definition format, etc.

Once your job definition file is created, you can activate the job using **launchctl load**:

```
$ sudo launchctl load \
       -w /Library/LaunchDaemons/org.apache.subversion.svnserve.plist
```

To be clear, this action doesn't actually launch **svnserve** yet. It simply tells **launchd** how to fire up **svnserve** when incoming networking traffic arrives on the `svn` network port; it will be terminated it after the traffic has been handled.

Because we want **svnserve** to be a system-wide daemon process, we need to use **sudo** to manage this job as an administrator. Note also that the `UserName` and `GroupName` keys in the definition file are optional—if omitted, the job will run as the user who loaded the job.

Deactivating the job is just as easy to do—use **launchctl unload**:

```
$ sudo launchctl unload \
       -w /Library/LaunchDaemons/org.apache.subversion.svnserve.plist
```

launchctl also provides a way for you to query the status of jobs. If the job is loaded, there will be line which matches the `Label` specified in the job definition file:

```
$ sudo launchctl list | grep org.apache.subversion.svnserve
-       0       org.apache.subversion.svnserve
$
```

Built-in Authentication and Authorization

When a client connects to an **svnserve** process, the following things happen:

- The client selects a specific repository.
- The server processes the repository's `conf/svnserve.conf` file and begins to enforce any authentication and authorization policies it describes.

- Depending on the defined policies, one of the following may occur:
  - The client may be allowed to make requests anonymously, without ever receiving an authentication challenge.
  - The client may be challenged for authentication at any time.
  - If operating in tunnel mode, the client will declare itself to be already externally authenticated (typically by SSH).

The **svnserve** server, by default, knows only how to send a CRAM-MD5[1] authentication challenge. In essence, the server sends a small amount of data to the client. The client uses the MD5 hash algorithm to create a fingerprint of the data and password combined, and then sends the fingerprint as a response. The server performs the same computation with the stored password to verify that the result is identical. *At no point does the actual password travel over the network.*

If your **svnserve** server was built with SASL support, it not only knows how to send CRAM-MD5 challenges, but also likely knows a whole host of other authentication mechanisms. See the section called "Using **svnserve** with SASL" later in this chapter to learn how to configure SASL authentication and encryption.

It's also possible, of course, for the client to be externally authenticated via a tunnel agent, such as **ssh**. In that case, the server simply examines the user it's running as, and uses this name as the authenticated username. For more on this, see the later section, the section called "Tunneling over SSH".

As you've already guessed, a repository's `svnserve.conf` file is the central mechanism for controlling authentication and authorization policies. The file has the same format as other configuration files (see the section called "Runtime Configuration Area"): section names are marked by square brackets (`[` and `]`), comments begin with hashes (#), and each section contains specific variables that can be set (`variable = value`). Let's walk through these files and learn how to use them.

Create a users file and realm

For now, the `[general]` section of `svnserve.conf` has all the variables you need. Begin by changing the values of those variables: choose a name for a file that will contain your usernames and passwords and choose an authentication realm:

```
[general]
password-db = userfile
realm = example realm
```

The `realm` is a name that you define. It tells clients which sort of "authentication namespace" they're connecting to; the Subversion client displays it in the authentication prompt and uses it as a key (along with the server's hostname and port) for caching credentials on disk (see the section called "Caching credentials"). The `password-db` variable points to a separate file that contains a list of usernames and passwords, using the same familiar format. For example:

```
[users]
harry = foopassword
sally = barpassword
```

The value of `password-db` can be an absolute or relative path to the users file. For many admins, it's easy to keep the file right in the `conf/` area of the repository, alongside `svnserve.conf`. On the other hand, it's possible you may want to have two or more repositories share the same users file; in that case, the file should probably live in a more public place. The repositories sharing the users file should also be configured to have the same realm, since the list of users essentially defines an authentication realm. Wherever the file lives, be sure to set the file's read and write permissions appropriately. If you know which user(s) **svn-**

[1] See RFC 2195.

serve will run as, restrict read access to the users file as necessary.

Set access controls

There are two more variables to set in the `svnserve.conf` file: they determine what unauthenticated (anonymous) and authenticated users are allowed to do. The variables `anon-access` and `auth-access` can be set to the value `none`, `read`, or `write`. Setting the value to `none` prohibits both reading and writing; `read` allows read-only access to the repository, and `write` allows complete read/write access to the repository. For example:

```
[general]
password-db = userfile
realm = example realm

# anonymous users can only read the repository
anon-access = read

# authenticated users can both read and write
auth-access = write
```

The example settings are, in fact, the default values of the variables, should you forget to define them. If you want to be even more conservative, you can block anonymous access completely:

```
[general]
password-db = userfile
realm = example realm

# anonymous users aren't allowed
anon-access = none

# authenticated users can both read and write
auth-access = write
```

The server process understands not only these “blanket” access controls to the repository, but also finer-grained access restrictions placed on specific files and directories within the repository. To make use of this feature, you need to define a file containing more detailed rules, and then set the `authz-db` variable to point to it:

```
[general]
password-db = userfile
realm = example realm

# Specific access rules for specific locations
authz-db = authzfile
```

We discuss the syntax of the `authzfile` file in detail later in this chapter, in the section called “Path-Based Authorization”. Note that the `authz-db` variable isn't mutually exclusive with the `anon-access` and `auth-access` variables; if all the variables are defined at once, *all* of the rules must be satisfied before access is allowed.

Using svnserve with SASL

For many teams, the built-in CRAM-MD5 authentication is all they need from **svnserve**. However, if your server (and your Sub-

version clients) were built with the Cyrus Simple Authentication and Security Layer (SASL) library, you have a number of authentication and encryption options available to you.

What Is SASL?

The Cyrus Simple Authentication and Security Layer is open source software written by Carnegie Mellon University. It adds generic authentication and encryption capabilities to any network protocol, and as of Subversion 1.5 and later, both the **svnserve** server and **svn** client know how to make use of this library. It may or may not be available to you: if you're building Subversion yourself, you'll need to have at least version 2.1 of SASL installed on your system, and you'll need to make sure that it's detected during Subversion's build process. The Subversion command-line client will report the availability of Cyrus SASL when you run **`svn --version`**; if you're using some other Subversion client, you might need to check with the package maintainer as to whether SASL support was compiled in.

SASL comes with a number of pluggable modules that represent different authentication systems: Kerberos (GSSAPI), NTLM, One-Time-Passwords (OTP), DIGEST-MD5, LDAP, Secure-Remote-Password (SRP), and others. Certain mechanisms may or may not be available to you; be sure to check which modules are provided.

You can download Cyrus SASL (both code and documentation) from http://asg.web.cmu.edu/sasl/sasl-library.html.

Normally, when a subversion client connects to **svnserve**, the server sends a greeting that advertises a list of the capabilities it supports, and the client responds with a similar list of capabilities. If the server is configured to require authentication, it then sends a challenge that lists the authentication mechanisms available; the client responds by choosing one of the mechanisms, and then authentication is carried out in some number of round-trip messages. Even when SASL capabilities aren't present, the client and server inherently know how to use the CRAM-MD5 and ANONYMOUS mechanisms (see the section called “Built-in Authentication and Authorization”). If server and client were linked against SASL, a number of other authentication mechanisms may also be available. However, you'll need to explicitly configure SASL on the server side to advertise them.

Authenticating with SASL

To activate specific SASL mechanisms on the server, you'll need to do two things. First, create a `[sasl]` section in your repository's `svnserve.conf` file with an initial key-value pair:

```
[sasl]
use-sasl = true
```

Second, create a main SASL configuration file called `svn.conf` in a place where the SASL library can find it—typically in the directory where SASL plug-ins are located. You'll have to locate the plug-in directory on your particular system, such as `/usr/lib/sasl2/` or `/etc/sasl2/`. (Note that this is *not* the `svnserve.conf` file that lives within a repository!)

On a Windows server, you'll also have to edit the system registry (using a tool such as **regedit**) to tell SASL where to find things. Create a registry key named `[HKEY_LOCAL_MACHINE\SOFTWARE\Carnegie Mellon\Project Cyrus\SASL Library]`, and place two keys inside it: a key called `SearchPath` (whose value is a path to the directory containing the SASL `sasl*.dll` plug-in libraries), and a key called `ConfFile` (whose value is a path to the parent directory containing the `svn.conf` file you created).

Because SASL provides so many different kinds of authentication mechanisms, it would be foolish (and far beyond the scope of this book) to try to describe every possible server-side configuration. Instead, we recommend that you read the documentation supplied in the `doc/` subdirectory of the SASL source code. It goes into great detail about every mechanism and how to configure the server appropriately for each. For the purposes of this discussion, we'll just demonstrate a simple example of configuring the DIGEST-MD5 mechanism. For example, if your `subversion.conf` (or `svn.conf`) file contains the following:

```
pwcheck_method: auxprop
```

```
auxprop_plugin: sasldb
sasldb_path: /etc/my_sasldb
mech_list: DIGEST-MD5
```

you've told SASL to advertise the DIGEST-MD5 mechanism to clients and to check user passwords against a private password database located at `/etc/my_sasldb`. A system administrator can then use the **saslpasswd2** program to add or modify usernames and passwords in the database:

```
$ saslpasswd2 -c -f /etc/my_sasldb -u realm username
```

A few words of warning: first, make sure the “realm” argument to **saslpasswd2** matches the same realm you've defined in your repository's `svnserve.conf` file; if they don't match, authentication will fail. Also, due to a shortcoming in SASL, the common realm must be a string with no space characters. Finally, if you decide to go with the standard SASL password database, make sure the **svnserve** program has read access to the file (and possibly write access as well, if you're using a mechanism such as OTP).

This is just one simple way of configuring SASL. Many other authentication mechanisms are available, and passwords can be stored in other places such as in LDAP or a SQL database. Consult the full SASL documentation for details.

Remember that if you configure your server to only allow certain SASL authentication mechanisms, this forces all connecting clients to have SASL support as well. Any Subversion client built without SASL support (which includes all pre-1.5 clients) will be unable to authenticate. On the one hand, this sort of restriction may be exactly what you want (“My clients must all use Kerberos!”). However, if you still want non-SASL clients to be able to authenticate, be sure to advertise the CRAM-MD5 mechanism as an option. All clients are able to use CRAM-MD5, whether they have SASL capabilities or not.

SASL encryption

SASL is also able to perform data encryption if a particular mechanism supports it. The built-in CRAM-MD5 mechanism doesn't support encryption, but DIGEST-MD5 does, and mechanisms such as SRP actually require use of the OpenSSL library. To enable or disable different levels of encryption, you can set two values in your repository's `svnserve.conf` file:

```
[sasl]
use-sasl = true
min-encryption = 128
max-encryption = 256
```

The `min-encryption` and `max-encryption` variables control the level of encryption demanded by the server. To disable encryption completely, set both values to 0. To enable simple checksumming of data (i.e., prevent tampering and guarantee data integrity without encryption), set both values to 1. If you wish to allow—but not require—encryption, set the minimum value to 0, and the maximum value to some bit length. To require encryption unconditionally, set both values to numbers greater than 1. In our previous example, we require clients to do at least 128-bit encryption, but no more than 256-bit encryption.

Tunneling over SSH

svnserve's built-in authentication (and SASL support) can be very handy, because it avoids the need to create real system accounts. On the other hand, some administrators already have well-established SSH authentication frameworks in place. In these situations, all of the project's users already have system accounts and the ability to “SSH into” the server machine.

It's easy to use SSH in conjunction with **svnserve**. The client simply uses the `svn+ssh://` URL scheme to connect:

```
$ whoami
harry

$ svn list svn+ssh://host.example.com/repos/project
harryssh@host.example.com's password:  *****

foo
bar
baz
…
```

In this example, the Subversion client is invoking a local **ssh** process, connecting to `host.example.com`, authenticating as the user `harryssh` (according to SSH user configuration), then spawning a private **svnserve** process on the remote machine running as the user `harryssh`. The **svnserve** command is being invoked in tunnel mode (`-t`), and its network protocol is being "tunneled" over the encrypted connection by **ssh**, the tunnel agent. If the client performs a commit, the authenticated username `harryssh` will be used as the author of the new revision.

The important thing to understand here is that the Subversion client is *not* connecting to a running **svnserve** daemon. This method of access doesn't require a daemon, nor does it notice one if present. It relies wholly on the ability of **ssh** to spawn a temporary **svnserve** process, which then terminates when the network connection is closed.

When using `svn+ssh://` URLs to access a repository, remember that it's the **ssh** program prompting for authentication, and *not* the **svn** client program. That means there's no automatic password-caching going on (see the section called "Caching credentials"). The Subversion client often makes multiple connections to the repository, though users don't normally notice this due to the password caching feature. When using `svn+ssh://` URLs, however, users may be annoyed by **ssh** repeatedly asking for a password for every outbound connection. The solution is to use a separate SSH password-caching tool such as **ssh-agent** on a Unix-like system, or **pageant** on Windows.

When running over a tunnel, authorization is primarily controlled by operating system permissions to the repository's database files; it's very much the same as if Harry were accessing the repository directly via a `file://` URL. If multiple system users are going to be accessing the repository directly, you may want to place them into a common group, and you'll need to be careful about umasks (be sure to read the section called "Supporting Multiple Repository Access Methods" later in this chapter). But even in the case of tunneling, you can still use the `svnserve.conf` file to block access, by simply setting `auth-access = read` or `auth-access = none`.[2]

You'd think that the story of SSH tunneling would end here, but it doesn't. Subversion allows you to create custom tunnel behaviors in your runtime `config` file (see the section called "Runtime Configuration Area"). For example, suppose you want to use RSH instead of SSH.[3] In the `[tunnels]` section of your `config` file, simply define it like this:

```
[tunnels]
rsh = rsh
```

And now, you can use this new tunnel definition by using a URL scheme that matches the name of your new variable: `svn+rsh://host/path`. When using the new URL scheme, the Subversion client will actually be running the command **`rsh host svnserve -t`** behind the scenes. If you include a username in the URL (e.g., `svn+rsh://username@host/path`), the client will also include that in its command (**`rsh username@host svnserve -t`**). But you can define new tunneling schemes to be much more clever than that:

[2]Note that using any sort of **svnserve**-enforced access control at all is a bit pointless; the user already has direct access to the repository database.
[3]We don't actually recommend this, since RSH is notably less secure than SSH.

```
[tunnels]
joessh = $JOESSH /opt/alternate/ssh -p 29934
```

This example demonstrates a couple of things. First, it shows how to make the Subversion client launch a very specific tunneling binary (the one located at `/opt/alternate/ssh`) with specific options. In this case, accessing an `svn+joessh://` URL would invoke the particular SSH binary with `-p 29934` as arguments—useful if you want the tunnel program to connect to a nonstandard port.

Second, it shows how to define a custom environment variable that can override the name of the tunneling program. Setting the `SVN_SSH` environment variable is a convenient way to override the default SSH tunnel agent. But if you need to have several different overrides for different servers, each perhaps contacting a different port or passing a different set of options to SSH, you can use the mechanism demonstrated in this example. Now if we were to set the `JOESSH` environment variable, its value would override the entire value of the tunnel variable—**$JOESSH** would be executed instead of **`/opt/alternate/ssh -p 29934`**.

SSH Configuration Tricks

It's possible to control not only the way in which the client invokes **ssh**, but also to control the behavior of **sshd** on your server machine. In this section, we'll show how to control the exact **svnserve** command executed by **sshd**, as well as how to have multiple users share a single system account.

Initial setup

To begin, locate the home directory of the account you'll be using to launch **svnserve**. Make sure the account has an SSH public/private keypair installed, and that the user can log in via public-key authentication. Password authentication will not work, since all of the following SSH tricks revolve around using the SSH `authorized_keys` file.

If it doesn't already exist, create the `authorized_keys` file (on Unix, typically `~/.ssh/authorized_keys`). Each line in this file describes a public key that is allowed to connect. The lines are typically of the form:

```
ssh-dsa AAAABtce9euch… user@example.com
```

The first field describes the type of key, the second field is the base64-encoded key itself, and the third field is a comment. However, it's a lesser known fact that the entire line can be preceded by a `command` field:

```
command="program" ssh-dsa AAAABtce9euch… user@example.com
```

When the `command` field is set, the SSH daemon will run the named program instead of the typical tunnel-mode **svnserve** invocation that the Subversion client asks for. This opens the door to a number of server-side tricks. In the following examples, we abbreviate the lines of the file as:

```
command="program" TYPE KEY COMMENT
```

Controlling the invoked command

Because we can specify the executed server-side command, it's easy to name a specific **svnserve** binary to run and to pass it extra arguments:

```
command="/path/to/svnserve -t -r /virtual/root" TYPE KEY COMMENT
```

In this example, `/path/to/svnserve` might be a custom wrapper script around **svnserve** which sets the umask (see the section called "Supporting Multiple Repository Access Methods"). It also shows how to anchor **svnserve** in a virtual root directory, just as one often does when running **svnserve** as a daemon process. This might be done either to restrict access to parts of the system, or simply to relieve the user of having to type an absolute path in the `svn+ssh://` URL.

It's also possible to have multiple users share a single account. Instead of creating a separate system account for each user, generate a public/private key pair for each person. Then place each public key into the `authorized_keys` file, one per line, and use the `--tunnel-user` option:

```
command="svnserve -t --tunnel-user=harry" TYPE1 KEY1 harry@example.com
command="svnserve -t --tunnel-user=sally" TYPE2 KEY2 sally@example.com
```

This example allows both Harry and Sally to connect to the same account via public key authentication. Each of them has a custom command that will be executed; the `--tunnel-user` option tells **svnserve** to assume that the named argument is the authenticated user. Without `--tunnel-user`, it would appear as though all commits were coming from the one shared system account.

A final word of caution: giving a user access to the server via public-key in a shared account might still allow other forms of SSH access, even if you've set the `command` value in `authorized_keys`. For example, the user may still get shell access through SSH or be able to perform X11 or general port forwarding through your server. To give the user as little permission as possible, you may want to specify a number of restrictive options immediately after the `command`:

```
  command="svnserve -t --tunnel-user=harry",no-port-forwarding,no-agent-forw
arding,no-X11-forwarding,no-pty TYPE1 KEY1 harry@example.com
```

Note that this all must be on one line—truly on one line—since SSH `authorized_keys` files do not even allow the conventional backslash character (\) for line continuation. The only reason we've shown it with a line break is to fit it on the physical page of a book.

httpd, the Apache HTTP Server

The Apache HTTP Server is a "heavy-duty" network server that Subversion can leverage. Via a custom module, **httpd** makes Subversion repositories available to clients via the WebDAV/DeltaV[4] protocol, which is an extension to HTTP 1.1. This protocol takes the ubiquitous HTTP protocol that is the core of the World Wide Web, and adds writing—specifically, versioned writing—capabilities. The result is a standardized, robust system that is conveniently packaged as part of the Apache 2.0 software, supported by numerous operating systems and third-party products, and doesn't require network administrators to open up yet another custom port.[5] While an Apache-Subversion server has more features than **svnserve**, it's also a bit more difficult to set up. With flexibility often comes more complexity.

Much of the following discussion includes references to Apache configuration directives. While some examples are given of the

[4]See http://www.webdav.org/.
[5]They really hate doing that.

use of these directives, describing them in full is outside the scope of this chapter. The Apache team maintains excellent documentation, publicly available on their web site at http://httpd.apache.org. For example, a general reference for the configuration directives is located at http://httpd.apache.org/docs-2.0/mod/directives.html.

Also, as you make changes to your Apache setup, it is likely that somewhere along the way a mistake will be made. If you are not already familiar with Apache's logging subsystem, you should become aware of it. In your `httpd.conf` file are directives that specify the on-disk locations of the access and error logs generated by Apache (the `CustomLog` and `ErrorLog` directives, respectively). Subversion's **mod_dav_svn** uses Apache's error logging interface as well. You can always browse the contents of those files for information that might reveal the source of a problem that is not clearly noticeable otherwise.

Prerequisites

To network your repository over HTTP, you basically need four components, available in two packages. You'll need Apache **httpd** 2.0 or newer, the **mod_dav** DAV module that comes with it, Subversion, and the **mod_dav_svn** filesystem provider module distributed with Subversion. Once you have all of those components, the process of networking your repository is as simple as:

- Getting httpd up and running with the **mod_dav** module
- Installing the **mod_dav_svn** backend to **mod_dav**, which uses Subversion's libraries to access the repository
- Configuring your `httpd.conf` file to export (or expose) the repository

You can accomplish the first two items either by compiling **httpd** and Subversion from source code or by installing prebuilt binary packages of them on your system. For the most up-to-date information on how to compile Subversion for use with the Apache HTTP Server, as well as how to compile and configure Apache itself for this purpose, see the `INSTALL` file in the top level of the Subversion source code tree.

Basic Apache Configuration

Once you have all the necessary components installed on your system, all that remains is the configuration of Apache via its `httpd.conf` file. Instruct Apache to load the **mod_dav_svn** module using the `LoadModule` directive. This directive must precede any other Subversion-related configuration items. If your Apache was installed using the default layout, your **mod_dav_svn** module should have been installed in the `modules` subdirectory of the Apache install location (often `/usr/local/apache2`). The `LoadModule` directive has a simple syntax, mapping a named module to the location of a shared library on disk:

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Apache interprets the `LoadModule` configuration item's library path as relative to its own server root. If configured as previously shown, Apache will look for the Subversion DAV module shared library in its own `modules/` subdirectory. Depending on how Subversion was installed on your system, you might need to specify a different path for this library altogether, perhaps even an absolute path such as in the following example:

```
LoadModule dav_svn_module     C:/Subversion/lib/mod_dav_svn.so
```

Note that if **mod_dav** was compiled as a shared object (instead of statically linked directly to the **httpd** binary), you'll need a similar `LoadModule` statement for it, too. Be sure that it comes before the **mod_dav_svn** line:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

At a later location in your configuration file, you now need to tell Apache where you keep your Subversion repository (or repositories). The `Location` directive has an XML-like notation, starting with an opening tag and ending with a closing tag, with various other configuration directives in the middle. The purpose of the `Location` directive is to instruct Apache to do something special when handling requests that are directed at a given URL or one of its children. In the case of Subversion, you want Apache to simply hand off support for URLs that point at versioned resources to the DAV layer. You can instruct Apache to delegate the handling of all URLs whose path portions (the part of the URL that follows the server's name and the optional port number) begin with `/repos/` to a DAV provider whose repository is located at `/var/svn/repository` using the following `httpd.conf` syntax:

```
<Location /repos>
  DAV svn
  SVNPath /var/svn/repository
</Location>
```

If you plan to support multiple Subversion repositories that will reside in the same parent directory on your local disk, you can use an alternative directive—`SVNParentPath`—to indicate that common parent directory. For example, if you know you will be creating multiple Subversion repositories in a directory `/var/svn` that would be accessed via URLs such as `http://my.server.com/svn/repos1`, `http://my.server.com/svn/repos2`, and so on, you could use the `httpd.conf` configuration syntax in the following example:

```
<Location /svn>
  DAV svn

  # Automatically map any "/svn/foo" URL to repository /var/svn/foo
  SVNParentPath /var/svn
</Location>
```

Using this syntax, Apache will delegate the handling of all URLs whose path portions begin with `/svn/` to the Subversion DAV provider, which will then assume that any items in the directory specified by the `SVNParentPath` directive are actually Subversion repositories. This is a particularly convenient syntax in that, unlike the use of the `SVNPath` directive, you don't have to restart Apache to add or remove hosted repositories.

Be sure that when you define your new `Location`, it doesn't overlap with other exported locations. For example, if your main `DocumentRoot` is exported to `/www`, do not export a Subversion repository in `<Location /www/repos>`. If a request comes in for the URI `/www/repos/foo.c`, Apache won't know whether to look for a file `repos/foo.c` in the `DocumentRoot`, or whether to delegate **mod_dav_svn** to return `foo.c` from the Subversion repository. The result is often an error from the server of the form `301 Moved Permanently`.

Server Names and the COPY Request

Subversion makes use of the `COPY` request type to perform server-side copies of files and directories. As part of the sanity checking done by the Apache modules, the source of the copy is expected to be located on the same machine as the destination of the copy. To satisfy this requirement, you might need to tell **mod_dav** the name you use as the hostname of your server. Generally, you can use the `ServerName` directive in `httpd.conf` to accomplish this.

```
ServerName svn.example.com
```

If you are using Apache's virtual hosting support via the `NameVirtualHost` directive, you may need to use the `ServerAlias` directive to specify additional names by which your server is known. Again, refer to the Apache documentation for full details.

At this stage, you should strongly consider the question of permissions. If you've been running Apache for some time now as your regular web server, you probably already have a collection of content—web pages, scripts, and such. These items have already been configured with a set of permissions that allows them to work with Apache, or more appropriately, that allows Apache to work with those files. Apache, when used as a Subversion server, will also need the correct permissions to read and write to your Subversion repository.

You will need to determine a permission system setup that satisfies Subversion's requirements without messing up any previously existing web page or script installations. This might mean changing the permissions on your Subversion repository to match those in use by other things that Apache serves for you, or it could mean using the `User` and `Group` directives in `httpd.conf` to specify that Apache should run as the user and group that owns your Subversion repository. There is no single correct way to set up your permissions, and each administrator will have different reasons for doing things a certain way. Just be aware that permission-related problems are perhaps the most common oversight when configuring a Subversion repository for use with Apache.

Authentication Options

At this point, if you configured `httpd.conf` to contain something such as the following:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn
</Location>
```

your repository is “anonymously” accessible to the world. Until you configure some authentication and authorization policies, the Subversion repositories that you make available via the `Location` directive will be generally accessible to everyone. In other words:

- Anyone can use a Subversion client to check out a working copy of a repository URL (or any of its subdirectories).
- Anyone can interactively browse the repository's latest revision simply by pointing a web browser to the repository URL.
- Anyone can commit to the repository.

Of course, you might have already set up a `pre-commit` hook script to prevent commits (see the section called “Implementing Repository Hooks”). But as you read on, you'll see that it's also possible to use Apache's built-in methods to restrict access in specific ways.

Requiring authentication defends against invalid users directly accessing the repository, but does not guard the privacy of valid users' network activity. See the section called “Protecting network traffic with SSL” for how to configure your server to support SSL encryption, which can provide that extra layer of protection.

Basic authentication

The easiest way to authenticate a client is via the HTTP Basic authentication mechanism, which simply uses a username and password to verify a user's identity. Apache provides the **htpasswd** utility[6] for managing files containing usernames and passwords.

Basic authentication is *extremely* insecure, because it sends passwords over the network in very nearly plain text. See the section called "Digest authentication" for details on using the much safer Digest mechanism.

First, create a password file and grant access to users Harry and Sally:

```
$ ### First time: use -c to create the file
$ ### Use -m to use MD5 encryption of the password, which is more secure
$ htpasswd -c -m /etc/svn-auth.htpasswd harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd -m /etc/svn-auth.htpasswd sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

Next, add some more directives inside the `<Location>` block to tell Apache how to use the password file:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Basic
  AuthName "Subversion repository"
  AuthType Basic
  AuthUserFile /etc/svn-auth.htpasswd
</Location>
```

These directives work as follows:

- `AuthName` is an arbitrary name that you choose for the authentication domain. Most browsers display this name in the dialog box when prompting for username and password.
- `AuthType` specifies the type of authentication to use.
- `AuthUserFile` specifies the location of the password file to use.

However, this `<Location>` block doesn't yet do anything useful. It merely tells Apache that *if* authorization were required, it should challenge the Subversion client for a username and password. (When authorization is required, Apache requires authentication as well.) What's missing here, however, are directives that tell Apache *which sorts* of client requests require authorization; currently, none do. The simplest thing is to specify that *all* requests require authorization by adding `Require valid-user` to the

[6]See http://httpd.apache.org/docs/current/programs/htpasswd.html.

block:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Basic
  AuthName "Subversion repository"
  AuthType Basic
  AuthUserFile /etc/svn-auth.htpasswd

  # Authorization: Authenticated users only
  Require valid-user
</Location>
```

Refer to the section called “Authorization Options” for more detail on the `Require` directive and other ways to set authorization policies.

Digest authentication

Digest authentication is an improvement on Basic authentication which allows the server to verify a client's identity without sending the password over the network unprotected. Both client and server create a non-reversible MD5 hash of the username, password, requested URI, and a *nonce* (number used once) provided by the server and changed each time authentication is required. The client sends its hash to the server, and the server then verifies that the hashes match.

Configuring Apache to use Digest authentication is straightforward, with only small variations on our prior example:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Authenticated users only
  Require valid-user
</Location>
```

Notice that `AuthType` is now set to `Digest`, and we specify a different path for `AuthUserFile`. Digest authentication uses a different file format than Basic authentication; it is created using Apache's **htdigest** utility[7] rather than **htpasswd**. Digest authentication also has the additional concept of a “realm”, which must match the value of the `AuthName` directive. The password file can be created as follows:

```
$ ### First time: use -c to create the file
$ htdigest -c /etc/svn-auth.htdigest "Subversion repository" harry
Adding password for harry in realm Subversion repository.
New password: *****
Re-type new password: *****
$ htdigest /etc/svn-auth.htdigest "Subversion repository" sally
```

[7]See http://httpd.apache.org/docs/current/programs/htdigest.html.

```
Adding user sally in realm Subversion repository
New password: *******
Re-type new password: *******
$
```

Authorization Options

At this point, you've configured authentication, but not authorization. Apache is able to challenge clients and confirm identities, but it has not been told how to allow or restrict access to the clients bearing those identities. This section describes two strategies for controlling access to your repositories.

Blanket access control

The simplest form of access control is to authorize certain users for either read-only access to a repository or read/write access to a repository.

You can restrict access on all repository operations by adding `Require valid-user` directly inside the `<Location>` block. The example from the section called "Digest authentication" allows only clients that successfully authenticate to do anything with the Subversion repository:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Authenticated users only
  Require valid-user
</Location>
```

Sometimes you don't need to run such a tight ship. For example, Subversion's own source code repository at http://svn.collab.net/repos/svn allows anyone in the world to perform read-only repository tasks (such as checking out working copies and browsing the repository), but restricts write operations to authenticated users. The `Limit` and `LimitExcept` directives allow for this type of selective restriction. Like the `Location` directive, these blocks have starting and ending tags, and you would nest them inside your `<Location>` block.

The parameters present on the `Limit` and `LimitExcept` directives are HTTP request types that are affected by that block. For example, to allow anonymous read-only operations, you would use the `LimitExcept` directive (passing the `GET`, `PROPFIND`, `OPTIONS`, and `REPORT` request type parameters) and place the previously mentioned `Require valid-user` directive inside the `<LimitExcept>` block instead of just inside the `<Location>` block.

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
```

```
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Authenticated users only for non-read-only
  #                (write) operations; allow anonymous reads
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

These are only a few simple examples. For more in-depth information about Apache access control and the `Require` directive, take a look at the `Security` section of the Apache documentation's tutorials collection at http://httpd.apache.org/docs-2.0/misc/tutorials.html.

Per-directory access control

It's possible to set up finer-grained permissions using **mod_authz_svn**. This Apache module grabs the various opaque URLs passing from client to server, asks **mod_dav_svn** to decode them, and then possibly vetoes requests based on access policies defined in a configuration file.

If you've built Subversion from source code, **mod_authz_svn** is automatically built and installed alongside **mod_dav_svn**. Many binary distributions install it automatically as well. To verify that it's installed correctly, make sure it comes right after **mod_dav_svn**'s `LoadModule` directive in `httpd.conf`:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
LoadModule authz_svn_module   modules/mod_authz_svn.so
```

To activate this module, you need to configure your `<Location>` block to use the `AuthzSVNAccessFile` directive, which specifies a file containing the permissions policy for paths within your repositories. (In a moment, we'll discuss the format of that file.)

Apache is flexible, so you have the option to configure your block in one of three general patterns. To begin, choose one of these basic configuration patterns. (The following examples are very simple; look at Apache's own documentation for much more detail on Apache authentication and authorization options.)

The most open approach is to allow access to everyone. This means Apache never sends authentication challenges, and all users are treated as “anonymous”. (See Example 6.2, “A sample configuration for anonymous access”.)

Example 6.2. A sample configuration for anonymous access

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: None

  # Authorization: Path-based access control
  AuthzSVNAccessFile /path/to/access/file
</Location>
```

On the opposite end of the paranoia scale, you can configure Apache to authenticate all clients. This block unconditionally requires authentication via the `Require valid-user` directive, and defines a means to authenticate valid users. (See Example 6.3, “A sample configuration for authenticated access”.)

Example 6.3. A sample configuration for authenticated access

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Path-based access control; authenticated users only
  AuthzSVNAccessFile /path/to/access/file
  Require valid-user
</Location>
```

A third very popular pattern is to allow a combination of authenticated and anonymous access. For example, many administrators want to allow anonymous users to read certain repository directories, but restrict access to more sensitive areas to authenticated users. In this setup, all users start out accessing the repository anonymously. If your access control policy demands a real username at any point, Apache will demand authentication from the client. To do this, use both the `Satisfy Any` and `Require valid-user` directives. (See Example 6.4, “A sample configuration for mixed authenticated/anonymous access”.)

Example 6.4. A sample configuration for mixed authenticated/anonymous access

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn

  # Authentication: Digest
  AuthName "Subversion repository"
  AuthType Digest
  AuthUserFile /etc/svn-auth.htdigest

  # Authorization: Path-based access control; try anonymous access
  #                first, but authenticate if necessary
  AuthzSVNAccessFile /path/to/access/file
  Satisfy Any
  Require valid-user
</Location>
```

The next step is to create the authorization file containing access rules for particular paths within the repository. We describe how later in this chapter, in the section called “Path-Based Authorization”.

Disabling path-based checks

The **mod_dav_svn** module goes through a lot of work to make sure that data you've marked “unreadable” doesn't get accidentally leaked. This means it needs to closely monitor all of the paths and file-contents returned by commands such as **svn checkout** and

svn update. If these commands encounter a path that isn't readable according to some authorization policy, the path is typically omitted altogether. In the case of history or rename tracing—for example, running a command such as **`svn cat -r OLD foo.c`** on a file that was renamed long ago—the rename tracking will simply halt if one of the object's former names is determined to be read-restricted.

All of this path checking can sometimes be quite expensive, especially in the case of **svn log**. When retrieving a list of revisions, the server looks at every changed path in each revision and checks it for readability. If an unreadable path is discovered, it's omitted from the list of the revision's changed paths (normally seen with the `--verbose` (`-v`) option), and the whole log message is suppressed. Needless to say, this can be time-consuming on revisions that affect a large number of files. This is the cost of security: even if you haven't configured a module such as **mod_authz_svn** at all, the **mod_dav_svn** module is still asking Apache **httpd** to run authorization checks on every path. The **mod_dav_svn** module has no idea what authorization modules have been installed, so all it can do is ask Apache to invoke whatever might be present.

On the other hand, there's also an escape hatch of sorts, which allows you to trade security features for speed. If you're not enforcing any sort of per-directory authorization (i.e., not using **mod_authz_svn** or similar module), you can disable all of this path checking. In your `httpd.conf` file, use the `SVNPathAuthz` directive as shown in Example 6.5, “Disabling path checks altogether”.

Example 6.5. Disabling path checks altogether

```
<Location /repos>
  DAV svn
  SVNParentPath /var/svn

  SVNPathAuthz off
</Location>
```

The `SVNPathAuthz` directive is “on” by default. When set to “off,” all path-based authorization checking is disabled; **mod_dav_svn** stops invoking authorization checks on every path it discovers.

Protecting network traffic with SSL

Connecting to a repository via `http://` means that all Subversion activity is sent across the network in the clear. This means that actions such as checkouts, commits, and updates could potentially be intercepted by an unauthorized party “sniffing” network traffic. Encrypting traffic using SSL is a good way to protect potentially sensitive information over the network.

If a Subversion client is compiled to use OpenSSL, it gains the ability to speak to an Apache server via `https://` URLs, so all traffic is encrypted with a per-connection session key. The WebDAV library used by the Subversion client is not only able to verify server certificates, but can also supply client certificates when challenged by the server.

Subversion server SSL certificate configuration

It's beyond the scope of this book to describe how to generate client and server SSL certificates and how to configure Apache to use them. Many other references, including Apache's own documentation, describe the process.

SSL certificates from well-known entities generally cost money, but at a bare minimum, you can configure Apache to use a self-signed certificate generated with a tool such as OpenSSL (http://openssl.org).[8]

Subversion client SSL certificate management

When connecting to Apache via `https://`, a Subversion client can receive two different types of responses:

[8] While self-signed certificates are still vulnerable to a “man-in-the-middle” attack, such an attack is much more difficult for a casual observer to pull off, compared to sniffing unprotected passwords.

- A server certificate
- A challenge for a client certificate

Server certificate

When the client receives a server certificate, it needs to verify that the server is who it claims to be. OpenSSL does this by examining the signer of the server certificate, or *certificate authority* (CA). If OpenSSL is unable to automatically trust the CA, or if some other problem occurs (such as an expired certificate or hostname mismatch), the Subversion command-line client will ask you whether you want to trust the server certificate anyway:

```
$ svn list https://host.example.com/repos/project

Error validating server certificate for 'https://host.example.com:443':
 - The certificate is not issued by a trusted authority.  Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: host.example.com
 - Valid: from Jan 30 19:23:56 2004 GMT until Jan 30 19:23:56 2006 GMT
 - Issuer: CA, example.com, Sometown, California, US
 - Fingerprint: 7d:e1:a9:34:33:39:ba:6a:e9:a5:c4:22:98:7b:76:5c:92:a0:9c:7b

(R)eject, accept (t)emporarily or accept (p)ermanently?
```

This dialogue is essentially the same question you may have seen coming from your web browser (which is just another HTTP client like Subversion). If you choose the `(p)`ermanent option, Subversion will cache the server certificate in your private runtime `auth/` area, just as your username and password are cached (see the section called “Caching credentials”), and will automatically trust the certificate in the future.

Your runtime `servers` file also gives you the ability to make your Subversion client automatically trust specific CAs, either globally or on a per-host basis. Simply set the `ssl-authority-files` variable to a semicolon-separated list of PEM-encoded CA certificates:

```
[global]
ssl-authority-files = /path/to/CAcert1.pem;/path/to/CAcert2.pem
```

Many OpenSSL installations also have a predefined set of “default” CAs that are nearly universally trusted. To make the Subversion client automatically trust these standard authorities, set the `ssl-trust-default-ca` variable to `true`.

Client certificate challenge

If the client receives a challenge for a certificate, the server is asking the client to prove its identity. The client must send back a certificate signed by a CA that the server trusts, along with a *challenge response* which proves that the client owns the private key associated with the certificate. The private key and certificate are usually stored in an encrypted format on disk, protected by a passphrase. When Subversion receives this challenge, it will ask you for the path to the encrypted file and the passphrase that protects it:

```
$ svn list https://host.example.com/repos/project

Authentication realm: https://host.example.com:443
```

```
Client certificate filename: /path/to/my/cert.p12
Passphrase for '/path/to/my/cert.p12':  ********
```

Notice that the client credentials are stored in a `.p12` file. To use a client certificate with Subversion, it must be in PKCS#12 format, which is a portable standard. Most web browsers are able to import and export certificates in that format. Another option is to use the OpenSSL command-line tools to convert existing certificates into PKCS#12.

The runtime `servers` file also allows you to automate this challenge on a per-host basis. If you set the `ssl-client-cert-file` and `ssl-client-cert-password` variables, Subversion can automatically respond to a client certificate challenge without prompting you:

```
[groups]
examplehost = host.example.com

[examplehost]
ssl-client-cert-file = /path/to/my/cert.p12
ssl-client-cert-password = somepassword
```

More security-conscious folk might want to exclude `ssl-client-cert-password` to avoid storing the passphrase in the clear on disk.

Extra Goodies

We've covered most of the authentication and authorization options for Apache and **mod_dav_svn**. But there are a few other nice features that Apache provides.

Repository browsing

One of the most useful benefits of an Apache/WebDAV configuration for your Subversion repository is that your versioned files and directories are immediately available for viewing via a regular web browser. Since Subversion uses URLs to identify versioned resources, those URLs used for HTTP-based repository access can be typed directly into a web browser. Your browser will issue an HTTP `GET` request for that URL; based on whether that URL represents a versioned directory or file, **mod_dav_svn** will respond with a directory listing or with file contents.

URL syntax

If the URLs do not contain any information about which version of the resource you wish to see, **mod_dav_svn** will answer with the youngest version. This functionality has the wonderful side effect that you can pass around Subversion URLs to your peers as references to documents, and those URLs will always point at the latest manifestation of that document. Of course, you can even use the URLs as hyperlinks from other web sites, too.

As of Subversion 1.6, **mod_dav_svn** supports a public URI syntax for examining older revisions of both files and directories. The syntax uses the query string portion of the URL to specify either or both of a peg revision and operative revision, which Subversion will then use to determine which version of the file or directory to display to your web browser. Add the query string name/value pair `p=`*`PEGREV`*, where *`PEGREV`* is a revision number, to specify the peg revision you wish to apply to the request. Use `r=`*`REV`*, where *`REV`* is a revision number, to specify an operative revision.

For example, if you wish to see the latest version of a `README.txt` file located in your project's `/trunk`, point your web browser to that file's repository URL, which might look something like the following:

```
http://host.example.com/repos/project/trunk/README.txt
```

If you now wish to see some older version of that file, add an operative revision to the URL's query string:

```
http://host.example.com/repos/project/trunk/README.txt?r=1234
```

What if the thing you're trying to view no longer exists in the youngest revision of the repository? That's where a peg revision is handy:

```
http://host.example.com/repos/project/trunk/deleted-thing.txt?p=321
```

And of course, you can combine peg revision and operative revision specifiers to fine-tune the exact item you wish to view:

```
http://host.example.com/repos/project/trunk/renamed-thing.txt?p=123&r=21
```

The previous URL would display revision 21 of the object which, in revision 123, was located at `/trunk/renamed-thing.txt` in the repository. See the section called “Peg and Operative Revisions” for a detailed explanation of these “peg revision” and “operative revision” concepts. They can be a bit tricky to wrap your head around.

As a reminder, this feature of **mod_dav_svn** offers only a limited repository browsing experience. You can see directory listings and file contents, but no revision properties (such as commit log messages) or file/directory properties. For folks who require more extensive browsing of repositories and their history, there are several third-party software packages which offer this. Some examples include ViewVC (http://viewvc.tigris.org), Trac (http://trac.edgewall.org) and WebSVN (http://websvn.info). These third-party tools don't affect **mod_dav_svn**'s built-in “browseability”, and generally offer a much wider set of features, including the display of the aforementioned property sets, display of content differences between file revisions, and so on.

Proper MIME type

When browsing a Subversion repository, the web browser gets a clue about how to render a file's contents by looking at the `Content-Type:` header returned in Apache's response to the HTTP `GET` request. The value of this header is some sort of MIME type. By default, Apache will tell the web browsers that all repository files are of the “default” MIME type, typically `text/plain`. This can be frustrating, however, if a user wishes repository files to render as something more meaningful—for example, it might be nice to have a `foo.html` file in the repository actually render as HTML when browsing.

To make this happen, you need only to make sure that your files have the proper `svn:mime-type` set. We discuss this in more detail in the section called “File Content Type”, and you can even configure your client to automatically attach proper `svn:mime-type` properties to files entering the repository for the first time; see the section called “Automatic Property Setting”.

Continuing our example, if one were to set the `svn:mime-type` property to `text/html` on file `foo.html`, Apache would properly tell your web browser to render the file as HTML. One could also attach proper `image/*` MIME-type properties to image files and ultimately get an entire web site to be viewable directly from a repository! There's generally no problem with this, as long as the web site doesn't contain any dynamically generated content.

Customizing the look

You generally will get more use out of URLs to versioned files—after all, that's where the interesting content tends to lie. But you might have occasion to browse a Subversion directory listing, where you'll quickly note that the generated HTML used to display that listing is very basic, and certainly not intended to be aesthetically pleasing (or even interesting). To enable customization of these directory displays, Subversion provides an XML index feature. A single `SVNIndexXSLT` directive in your repository's `Location` block of `httpd.conf` will instruct **mod_dav_svn** to generate XML output when displaying a directory listing, and to reference the XSLT stylesheet of your choice:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn
  SVNIndexXSLT "/svnindex.xsl"
  …
</Location>
```

Using the `SVNIndexXSLT` directive and a creative XSLT stylesheet, you can make your directory listings match the color schemes and imagery used in other parts of your web site. Or, if you'd prefer, you can use the sample stylesheets provided in the Subversion source distribution's `tools/xslt/` directory. Keep in mind that the path provided to the `SVNIndexXSLT` directory is actually a URL path—browsers need to be able to read your stylesheets to make use of them!

Listing repositories

If you're serving a collection of repositories from a single URL via the `SVNParentPath` directive, then it's also possible to have Apache display all available repositories to a web browser. Just activate the `SVNListParentPath` directive:

```
<Location /svn>
  DAV svn
  SVNParentPath /var/svn
  SVNListParentPath on
  …
</Location>
```

If a user now points her web browser to the URL `http://host.example.com/svn/`, she'll see a list of all Subversion repositories sitting in `/var/svn`. Obviously, this can be a security problem, so this feature is turned off by default.

Apache logging

Because Apache is an HTTP server at heart, it contains fantastically flexible logging features. It's beyond the scope of this book to discuss all of the ways logging can be configured, but we should point out that even the most generic `httpd.conf` file will cause Apache to produce two logs: `error_log` and `access_log`. These logs may appear in different places, but are typically created in the logging area of your Apache installation. (On Unix, they often live in `/usr/local/apache2/logs/`.)

The `error_log` describes any internal errors that Apache runs into as it works. The `access_log` file records every incoming HTTP request received by Apache. This makes it easy to see, for example, which IP addresses Subversion clients are coming from, how often particular clients use the server, which users are authenticating properly, and which requests succeed or fail.

Unfortunately, because HTTP is a stateless protocol, even the simplest Subversion client operation generates multiple network requests. It's very difficult to look at the `access_log` and deduce what the client was doing—most operations look like a series of cryptic `PROPPATCH`, `GET`, `PUT`, and `REPORT` requests. To make things worse, many client operations send nearly identical series of requests, so it's even harder to tell them apart.

mod_dav_svn, however, can come to your aid. By activating an “operational logging” feature, you can ask **mod_dav_svn** to create a separate log file describing what sort of high-level operations your clients are performing.

To do this, you need to make use of Apache's `CustomLog` directive (which is explained in more detail in Apache's own documentation). Be sure to invoke this directive *outside* your Subversion `Location` block:

```
<Location /svn>
  DAV svn
  …
</Location>

CustomLog logs/svn_logfile "%t %u %{SVN-ACTION}e" env=SVN-ACTION
```

In this example, we're asking Apache to create a special logfile, `svn_logfile`, in the standard Apache `logs` directory. The `%t` and `%u` variables are replaced by the time and username of the request, respectively. The really important parts are the two instances of `SVN-ACTION`. When Apache sees that variable, it substitutes the value of the `SVN-ACTION` environment variable, which is automatically set by **mod_dav_svn** whenever it detects a high-level client action.

So, instead of having to interpret a traditional `access_log` like this:

```
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/vcc/default HTTP/1.1" 207 398
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc/!svn/bln/59 HTTP/1.1" 207 449
[26/Jan/2007:22:25:29 -0600] "PROPFIND /svn/calc HTTP/1.1" 207 647
[26/Jan/2007:22:25:29 -0600] "REPORT /svn/calc/!svn/vcc/default HTTP/1.1" 200 607
[26/Jan/2007:22:25:31 -0600] "OPTIONS /svn/calc HTTP/1.1" 200 188
[26/Jan/2007:22:25:31 -0600] "MKACTIVITY
/svn/calc/!svn/act/e6035ef7-5df0-4ac0-b811-4be7c823f998 HTTP/1.1" 201 227
…
```

you can peruse a much more intelligible `svn_logfile` like this:

```
[26/Jan/2007:22:24:20 -0600] - get-dir /tags r1729 props
[26/Jan/2007:22:24:27 -0600] - update /trunk r1729 depth=infinity
[26/Jan/2007:22:25:29 -0600] - status /trunk/foo r1729 depth=infinity
[26/Jan/2007:22:25:31 -0600] sally commit r1730
```

In addition to the `SVN-ACTION` environment variable, **mod_dav_svn** also populates the `SVN-REPOS` and `SVN-REPOS-NAME` variables, which carry the filesystem path to the repository and the basename thereof, respectively. You might wish to include references to one or both of these variables in your `CustomLog` format string, too, especially if you are combining usage information from multiple repositories into a single log file.

For an exhaustive list of all actions logged, see the section called “High-level Logging”.

Write-through proxying

One of the nice advantages of using Apache as a Subversion server is that it can be set up for simple replication. For example, suppose that your team is distributed across four offices around the globe. The Subversion repository can exist only in one of those offices, which means the other three offices will not enjoy accessing it—they're likely to experience significantly slower traffic and response times when updating and committing code. A powerful solution is to set up a system consisting of one *master* Apache server and several *slave* Apache servers. If you place a slave server in each office, users can check out a working copy from whichever slave is closest to them. All read requests go to their local slave. Write requests get automatically routed to the single master server. When the commit completes, the master then automatically “pushes” the new revision to each slave server using the

svnsync replication tool.

This configuration creates a huge perceptual speed increase for your users, because Subversion client traffic is typically 80–90% read requests. And if those requests are coming from a *local* server, it's a huge win.

In this section, we'll walk you through a standard setup of this single-master/multiple-slave system. However, keep in mind that your servers must be running at least Apache 2.2.0 (with **mod_proxy** loaded) and Subversion 1.5 (**mod_dav_svn**).

Ours is just one example of how you might setup a Subversion write-through proxy configuration. There are other approaches. For example, rather than having the master server push changes out to every slave server, the slaves could periodically pull those changes from the master. Or perhaps the master could push changes to a single slave, which then pushes the same change to the next slave, and so on down the line. Administrators are encouraged to use this section for basic understanding of what happens in a Subversion WebDAV proxy deployment scenario, and then implement the specific approach that works best for their organization.

Configure the servers

First, configure your master server's `httpd.conf` file in the usual way. Make the repository available at a certain URI location, and configure authentication and authorization however you'd like. After that's done, configure each of your “slave” servers in the exact same way, but add the special `SVNMasterURI` directive to the block:

```
<Location /svn>
  DAV svn
  SVNPath /var/svn/repos
  SVNMasterURI http://master.example.com/svn
  …
</Location>
```

This new directive tells a slave server to redirect all write requests to the master. (This is done automatically via Apache's **mod_proxy** module.) Ordinary read requests, however, are still serviced by the slaves. Be sure that your master and slave servers all have matching authentication and authorization configurations; if they fall out of sync, it can lead to big headaches.

Next, we need to deal with the problem of infinite recursion. With the current configuration, imagine what will happen when a Subversion client performs a commit to the master server. After the commit completes, the server uses **svnsync** to replicate the new revision to each slave. But because **svnsync** appears to be just another Subversion client performing a commit, the slave will immediately attempt to proxy the incoming write request back to the master! Hilarity ensues.

The solution to this problem is to have the master push revisions to a different `<Location>` on the slaves. This location is configured to *not* proxy write requests at all, but to accept normal commits from (and only from) the master's IP address:

```
<Location /svn-proxy-sync>
  DAV svn
  SVNPath /var/svn/repos
  Order deny,allow
  Deny from all
  # Only let the server's IP address access this Location:
  Allow from 10.20.30.40
  …
</Location>
```

Set up replication

Now that you've configured your `Location` blocks on master and slaves, you need to configure the master to replicate to the slaves. Our walkthough uses **svnsync**, which is covered in more detail in the section called “Replication with svnsync”.

First, make sure that each slave repository has a `pre-revprop-change` hook script which allows remote revision property changes. (This is standard procedure for being on the receiving end of **svnsync**.) Then log into the master server and configure each of the slave repository URIs to receive data from the master repository on the local disk:

```
$ svnsync init http://slave1.example.com/svn-proxy-sync \
               file:///var/svn/repos
Copied properties for revision 0.
$ svnsync init http://slave2.example.com/svn-proxy-sync \
               file:///var/svn/repos
Copied properties for revision 0.
$ svnsync init http://slave3.example.com/svn-proxy-sync \
               file:///var/svn/repos
Copied properties for revision 0.

# Perform the initial replication

$ svnsync sync http://slave1.example.com/svn-proxy-sync \
               file:///var/svn/repos
Transmitting file data ....
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .......
Committed revision 2.
Copied properties for revision 2.
…

$ svnsync sync http://slave2.example.com/svn-proxy-sync \
               file:///var/svn/repos
Transmitting file data ....
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .......
Committed revision 2.
Copied properties for revision 2.
…

$ svnsync sync http://slave3.example.com/svn-proxy-sync \
               file:///var/svn/repos
Transmitting file data ....
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .......
Committed revision 2.
Copied properties for revision 2.
…
```

After this is done, we configure the master server's `post-commit` hook script to invoke **svnsync** on each slave server:

```
#!/bin/sh
# Post-commit script to replicate newly committed revision to slaves

svnsync sync http://slave1.example.com/svn-proxy-sync \
             file:///var/svn/repos > /dev/null 2>&1 &
svnsync sync http://slave2.example.com/svn-proxy-sync \
```

```
             file:///var/svn/repos > /dev/null 2>&1 &
svnsync sync http://slave3.example.com/svn-proxy-sync \
             file:///var/svn/repos > /dev/null 2>&1 &
```

The extra bits on the end of each line aren't necessary, but they're a sneaky way to allow the sync commands to run in the background so that the Subversion client isn't left waiting forever for the commit to finish. In addition to this `post-commit` hook, you'll need a `post-revprop-change` hook as well so that when a user, say, modifies a log message, the slave servers get that change also:

```
#!/bin/sh
# Post-revprop-change script to replicate revprop-changes to slaves

REV=${2}
svnsync copy-revprops http://slave1.example.com/svn-proxy-sync \
                      file:///var/svn/repos \
                      -r ${REV} > /dev/null 2>&1 &
svnsync copy-revprops http://slave2.example.com/svn-proxy-sync \
                      file:///var/svn/repos \
                      -r ${REV} > /dev/null 2>&1 &
svnsync copy-revprops http://slave3.example.com/svn-proxy-sync \
                      file:///var/svn/repos \
                      -r ${REV} > /dev/null 2>&1 &
```

The only thing we've left out here is what to do about user-level locks (of the **svn lock** variety). Locks are enforced by the master server during commit operations; but all the information about locks is distributed during read operations such as **svn update** and **svn status** by the slave server. As such, a fully functioning proxy setup needs to perfectly replicate lock information from the master server to the slave servers. Unfortunately, most of the mechanisms that one might employ to accomplish this replication fall short in one way or another[9]. Many teams don't use Subversion's locking features at all, so this may be a nonissue for you. Sadly, for those teams which do use locks, we have no recommendations on how to gracefully work around this shortcoming.

Caveats

Your master/slave replication system should now be ready to use. A couple of words of warning are in order, however. Remember that this replication isn't entirely robust in the face of computer or network crashes. For example, if one of the automated **svnsync** commands fails to complete for some reason, the slaves will begin to fall behind. For example, your remote users will see that they've committed revision 100, but then when they run **svn update**, their local server will tell them that revision 100 doesn't yet exist! Of course, the problem will be automatically fixed the next time another commit happens and the subsequent **svnsync** is successful—the sync will replicate all waiting revisions. But still, you may want to set up some sort of out-of-band monitoring to notice synchronization failures and force **svnsync** to run when things go wrong.

Another limitation of the write-through proxy deployment model involves version mismatches—of the version of Subversion which is installed, that is—between the master and slave servers. Each new release of Subversion may (and often does) add new features to the network protocol used between the clients and servers. Since feature negotiation happens against the slave, it is the slave's protocol version and feature set which is used. But write operations are passed through to the master server quite literally. Therefore, there is always a risk that the slave server will answer a feature negotiation request from the client in way that is true for the slave, but untrue for the master if the master is running an older version of Subversion. This could result in the client trying to use a new feature that the master doesn't understand, and failing. There are a couple of known problems of this sort in Subversion 1.7, which introduced a major revision of its HTTP protocol. If you are deploying a Subversion 1.7 slave server in front of a pre-1.7 master, you'll want to configure your slave server's Subversion `<Location>` block with the `SVNAdvertiseV2ProtocolOff` directive.

[9] http://subversion.tigris.org/issues/show_bug.cgi?id=3457 tracks these problems.

For the best results possible, try to run the same version of Subversion on your master and slave servers.

Can We Set Up Replication with svnserve?

If you're using **svnserve** instead of Apache as your server, you can certainly configure your repository's hook scripts to invoke **svnsync** as we've shown here, thereby causing automatic replication from master to slaves. Unfortunately, at the time of this writing there is no way to make slave **svnserve** servers automatically proxy write requests back to the master server. This means your users would only be able to check out read-only working copies from the slave servers. You'd have to configure your slave servers to disallow write access completely. This might be useful for creating read-only “mirrors” of popular open source projects, but it's not a transparent proxying system.

Other Apache features

Several of the features already provided by Apache in its role as a robust web server can be leveraged for increased functionality or security in Subversion as well. The Subversion client is able to use SSL (the Secure Sockets Layer, discussed earlier). If your Subversion client is built to support SSL, it can access your Apache server using `https://` and enjoy a high-quality encrypted network session.

Equally useful are other features of the Apache and Subversion relationship, such as the ability to specify a custom port (instead of the default HTTP port 80) or a virtual domain name by which the Subversion repository should be accessed, or the ability to access the repository through an HTTP proxy.

Finally, because **mod_dav_svn** is speaking a subset of the WebDAV/DeltaV protocol, it's possible to access the repository via third-party DAV clients. Most modern operating systems (Win32, OS X, and Linux) have the built-in ability to mount a DAV server as a standard network “shared folder.” This is a complicated topic, but also wondrous when implemented. For details, read Appendix C, *WebDAV and Autoversioning*.

Note that there are a number of other small tweaks one can make to **mod_dav_svn** that are too obscure to mention in this chapter. For a complete list of all `httpd.conf` directives that **mod_dav_svn** responds to, see the section called “Directives” in Chapter 9, *Subversion Complete Reference*.

Path-Based Authorization

Both Apache and **svnserve** are capable of granting (or denying) permissions to users. Typically this is done over the entire repository: a user can read the repository (or not), and she can write to the repository (or not). It's also possible, however, to define finer-grained access rules. One set of users may have permission to write to a certain directory in the repository, but not others; another directory might not even be readable by all but a few special people. As files are paths, too, it's even possible to restrict access on a per file basis.

Both servers use a common file format to describe these path-based access rules. In the case of Apache, one needs to load the **mod_authz_svn** module and then add the `AuthzSVNAccessFile` directive (within the `httpd.conf` file) pointing to your own access rules file. (For a full explanation, see the section called “Per-directory access control”.) If you're using **svnserve**, you need to make the `authz-db` variable (within `svnserve.conf`) point to your access rules file.

Do You Really Need Path-Based Access Control?

A lot of administrators setting up Subversion for the first time tend to jump into path-based access control without giving it a lot of thought. The administrator usually knows which teams of people are working on which projects, so it's easy to jump in and grant certain teams access to certain directories and not others. It seems like a natural thing, and it appeases the administrator's desire to maintain tight control of the repository.

Note, though, that there are often invisible (and visible!) costs associated with this feature. In the visible category, the server

needs to do a lot more work to ensure that the user has the right to read or write each specific path; in certain situations, there's very noticeable performance loss. In the invisible category, consider the culture you're creating. Most of the time, while certain users *shouldn't* be committing changes to certain parts of the repository, that social contract doesn't need to be technologically enforced. Teams can sometimes spontaneously collaborate with each other; someone may want to help someone else out by committing to an area she doesn't normally work on. By preventing this sort of thing at the server level, you're setting up barriers to unexpected collaboration. You're also creating a bunch of rules that need to be maintained as projects develop, new users are added, and so on. It's a bunch of extra work to maintain.

Remember that this is a version control system! Even if somebody accidentally commits a change to something she shouldn't, it's easy to undo the change. And if a user commits to the wrong place with deliberate malice, it's a social problem anyway, and that the problem needs to be dealt with outside Subversion.

So, before you begin restricting users' access rights, ask yourself whether there's a real, honest need for this, or whether it's just something that “sounds good” to an administrator. Decide whether it's worth sacrificing some server speed, and remember that there's very little risk involved; it's bad to become dependent on technology as a crutch for social problems.[10]

As an example to ponder, consider that the Subversion project itself has always had a notion of who is allowed to commit where, but it's always been enforced socially. This is a good model of community trust, especially for open source projects. Of course, sometimes there *are* truly legitimate needs for path-based access control; within corporations, for example, certain types of data really can be sensitive, and access needs to be genuinely restricted to small groups of people.

Once your server knows where to find your access file, it's time to define the rules.

The syntax of the file is the same familiar one used by `svnserve.conf` and the runtime configuration files. Lines that start with a hash (#) are ignored. In its simplest form, each section names a repository and path within it, as well as the authenticated usernames are the option names within each section. The value of each option describes the user's level of access to the repository path: either `r` (read-only) or `rw` (read/write). If the user is not mentioned at all, no access is allowed.

To be more specific: the value of the section names is either of the form `[repos-name:path]` or of the form `[path]`.

Prior to version 1.7, Subversion treated repository names and paths in a case-insensitive fashion for the purposes of access control, converting them to lower case internally before comparing them against the contents of your access file. It now does these comparisons case-sensitively. If you upgraded to Subversion 1.7 from an older version, you should review your access files for case correctness.

If you're using the `SVNParentPath` directive, it's important to specify the repository names in your sections. If you omit them, a section such as `[/some/dir]` will match the path `/some/dir` in *every* repository. If you're using the `SVNPath` directive, however, it's fine to only define paths in your sections—after all, there's only one repository.

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r
```

In this first example, the user `harry` has full read and write access on the `/branches/calc/bug-142` directory in the `calc` repository, but the user `sally` has read-only access. Any other users are blocked from accessing this directory.

mod_dav_svn offers a directive, `SVNReposName`, which allows administrators to define a more human-friendly name, of sorts, for a repository:

[10] A common theme in this book!

```
<Location /svn/calc>
  SVNPath /var/svn/calc
  SVNReposName "Calculator Application"
…
```

This allows **mod_dav_svn** to identify the repository by something other than merely its server directory basename—`calc`, in the previous example—when providing directory listings of repository content. Be aware, however, that when consulting the access file for authorization rules, Subversion uses this repository basename for comparison, *not* any configured human-friendly name.

Of course, permissions are inherited from parent to child directory. That means we can specify a subdirectory with a different access policy for Sally:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

# give sally write access only to the 'testing' subdir
[calc:/branches/calc/bug-142/testing]
sally = rw
```

Now Sally can write to the `testing` subdirectory of the branch, but can still only read other parts. Harry, meanwhile, continues to have complete read/write access to the whole branch.

It's also possible to explicitly deny permission to someone via inheritance rules, by setting the username variable to nothing:

```
[calc:/branches/calc/bug-142]
harry = rw
sally = r

[calc:/branches/calc/bug-142/secret]
harry =
```

In this example, Harry has read/write access to the entire `bug-142` tree, but has absolutely no access at all to the `secret` subdirectory within it.

The thing to remember is that the most specific path always matches first. The server tries to match the path itself, and then the parent of the path, then the parent of that, and so on. The net effect is that mentioning a specific path in the access file will always override any permissions inherited from parent directories.

By default, nobody has any access to the repository at all. That means that if you're starting with an empty file, you'll probably want to give at least read permission to all users at the root of the repository. You can do this by using the asterisk variable (`*`), which means “all users”:

```
[/]
* = r
```

This is a common setup; notice that no repository name is mentioned in the section name. This makes all repositories world-readable to all users. Once all users have read access to the repositories, you can give explicit `rw` permission to certain users on specific subdirectories within specific repositories.

The access file also allows you to define whole groups of users, much like the Unix `/etc/group` file:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = harry, sally, joe, frank, jane
```

Groups can be granted access control just like users. Distinguish them with an “at” (@) prefix:

```
[calc:/projects/calc]
@calc-developers = rw

[paint:/projects/paint]
jane = r
@paint-developers = rw
```

Another important fact is that group permissions are not overridden by individual user permissions. Rather, the *combination* of all matching permissions is granted. In the prior example, Jane is a member of the `paint-developers` group, which has read/write access. Combined with the `jane = r` rule, this still gives Jane read/write access. Permissions for group members can only be extended beyond the permissions the group already has. Restricting users who are part of a group to less than their group's permissions is impossible.

Groups can also be defined to contain other groups:

```
[groups]
calc-developers = harry, sally, joe
paint-developers = frank, sally, jane
everyone = @calc-developers, @paint-developers
```

Subversion 1.5 brought several useful features to the access file syntax—username aliases, authentication class tokens, and a new rule exclusion mechanism—all of which further simplify the maintenance of the access file. We'll describe first the username aliases feature.

Some authentication systems expect and carry relatively short usernames of the sorts we've been describing here—`harry`, `sally`, `joe`, and so on. But other authentication systems—such as those which use LDAP stores or SSL client certificates—may carry much more complex usernames. For example, Harry's username in an LDAP-protected system might be `CN=Harold Hacker,OU=Engineers,DC=red-bean,DC=com`. With usernames like that, the access file can become quite bloated with long or obscure usernames that are easy to mistype. Fortunately, username aliases allow you to have to type the correct complex username only once, in a statement which assigns to it a more easily digestable alias.

```
[aliases]
harry = CN=Harold Hacker,OU=Engineers,DC=red-bean,DC=com
sally = CN=Sally Swatterbug,OU=Engineers,DC=red-bean,DC=com
joe = CN=Gerald I. Joseph,OU=Engineers,DC=red-bean,DC=com
…
```

Once you've defined a set of aliases, you can refer to the users elsewhere in the access file via their aliases in all the same places you could have instead used their actual usernames. Simply prepend an ampersand to the alias to distinguish it from a regular username:

```
[groups]
calc-developers = &harry, &sally, &joe
paint-developers = &frank, &sally, &jane
everyone = @calc-developers, @paint-developers
```

You might also choose to use aliases if your users' usernames change frequently. Doing so allows you to need to update only the aliases table when these username changes occur, instead of doing global-search-and-replace operations on the whole access file.

Subversion also supports some “magic” tokens for helping you to make rule assignments based on the user's authentication class. One such token is the `$authenticated` token. Use this token where you would otherwise specify a username, alias, or group name in your authorization rules to declare the permissions granted to any user who has authenticated with any username at all. Similarly employed is the `$anonymous` token, except that it matches everyone who has *not* authenticated with a username.

```
[calendar:/projects/calendar]
$anonymous = r
$authenticated = rw
```

Finally, another handy bit of access file syntax magic is the use of the tilde (~) character as an exclusion marker. In your authorization rules, prefixing a username, alias, group name, or authentication class token with a tilde character will cause Subversion to apply the rule to users who do *not* match the rule. Though somewhat unnecessarily obfuscated, the following block is equivalent to the one in the previous example:

```
[calendar:/projects/calendar]
~$authenticated = r
~$anonymous = rw
```

A less obvious example might be as follows:

```
[groups]
calc-developers = &harry, &sally, &joe
calc-owners = &hewlett, &packard
calc = @calc-developers, @calc-owners

# Any calc participant has read-write access...
```

```
[calc:/projects/calc]
@calc = rw

# ...but only allow the owners to make and modify release tags.
[calc:/projects/calc/tags]
~@calc-owners = r
```

All of the above examples use directories, because defining access rules on them is the most common case. But is similarly able to restrict access on file paths, too.

```
[calendar:/projects/calendar/manager.ics]
harry = rw
sally = r
```

Partial Readability and Checkouts

If you're using Apache as your Subversion server and have made certain subdirectories of your repository unreadable to certain users, you need to be aware of a possible nonoptimal behavior with **svn checkout**.

When the client requests a checkout or update over HTTP, it makes a single server request and receives a single (often large) server response. When the server receives the request, that is the *only* opportunity Apache has to demand user authentication. This has some odd side effects. For example, if a certain subdirectory of the repository is readable only by user Sally, and user Harry checks out a parent directory, his client will respond to the initial authentication challenge as Harry. As the server generates the large response, there's no way it can resend an authentication challenge when it reaches the special subdirectory; thus the subdirectory is skipped altogether, rather than asking the user to reauthenticate as Sally at the right moment. In a similar way, if the root of the repository is anonymously world-readable, the entire checkout will be done without authentication—again, skipping the unreadable directory, rather than asking for authentication partway through.

High-level Logging

Both the Apache **httpd** and **svnserve** Subversion servers provide support for high-level logging of Subversion operations. Configuring each of the server options to provide this level of logging is done differently, of course, but the output from each is designed to conform to a uniform syntax.

To enable high-level logging in **svnserve**, you need only use the `--log-file` command-line option when starting the server, passing as the value to the option the file to which **svnserve** should write its log output.

```
$ svnserve -d -r /path/to/repositories --log-file /var/log/svn.log
```

Enabling the same in Apache is a bit more involved, but is essentially an extension of Apache's stock log output configuration mechanisms—see the section called “Apache logging” for details.

The following is a list of Subversion action log messages produced by its high-level logging mechanism, followed by one or more examples of the log message as it appears in the log output.

Checkout or export

```
checkout-or-export /path r62 depth=infinity
```

Commit

```
commit harry r100
```

Diffs

```
diff /path r15:20 depth=infinity ignore-ancestry
diff /path1@15 /path2@20 depth=infinity ignore-ancestry
```

Fetch a directory

```
get-dir /trunk r17 text
```

Fetch a file

```
get-file /path r20 props
```

Fetch a file revision

```
get-file-revs /path r12:15 include-merged-revisions
```

Fetch merge information

```
get-mergeinfo (/path1 /path2)
```

Lock

```
lock /path steal
```

Log

```
log (/path1,/path2,/path3) r20:90 discover-changed-paths revprops=()
```

Replay revisions (svnsync)

```
replay /path r19
```

Revision property change

```
change-rev-prop r50 propertyname
```

Revision property list

```
rev-proplist r34
```

Status

```
status /path r62 depth=infinity
```

Switch

```
switch /pathA /pathB@50 depth=infinity
```

Unlock

```
unlock /path break
```

Update

```
update /path r17 send-copyfrom-args
```

As a convenience to administrators who wish to post-process their Subversion high-level logging output (perhaps for reporting or analysis purposes), Subversion source code distributions provide a Python module (located at `tools/server-side/svn_server_log_parse.py`) which can be used to parse Subversion's log output.

Server Optimization

Part of the due diligence when offering a service such as a Subversion server involves capacity planning and performance tuning. Subversion doesn't tend to be particularly greedy in terms of server resources such as CPU cycles and memory, but any service can benefit from optimizations, especially when usage of the service skyrockets[11]. In this section, we'll discuss some ways you can tweak your Subversion server configuration to offer even better performance and scalability.

Data Caching

Generally speaking, the most expensive part of a Subversion server's job is fetching data from the repository. Subversion 1.6 attempted to offset this cost by introducing some in-memory caching of certain classes of data read from the repository. But Subversion 1.7 takes this a step further, not only caching the results of some of the more costly operations, but also by providing in each of the available servers the means by which fine-tune the size and some behaviors of the cache.

For **svnserve**, you can specify the size of the cache using the `--memory-cache-size` (`-M`) command-line option. You can also dictate whether **svnserve** should attempt to cache content fulltexts and deltas via the boolean `--cache-fulltexts` and `-`

[11]In Subversion's case, the skyrocketing affect is, of course, due to its cool name. Well, that and its popularity, reliability, ease of use….

`-cache-txdeltas` options, respectively.

```
$ svnserve -d -r /path/to/repositories \
           --memory-cache-size 1024 \
           --cache-txdeltas yes \
           --cache-fulltexts yes
…
$
```

mod_dav_svn provides the same degree of cache configurability via `httpd.conf` directives. The `SVNInMemoryCacheSize`, `SVNCacheFullTexts`, and `SVNCacheTextDeltas` directives may be used at the server configuration level to control Subversion's data cache characteristics:

```
<IfModule dav_svn_module>
  # Enable a 1 Gb Subversion data cache for both fulltext and deltas.
  SVNInMemoryCacheSize 1048576
  SVNCacheTextDeltas On
  SVNCacheFullTexts On
</IfModule>
```

So what settings should you use? Certainly you need to consider what resources are available on your server. To get any benefit out of the cache at all, you'll probably want to let the cache be at least large enough to hold all the files which are most commonly accessed in your repository (for example, your project's `trunk` directory tree).

Setting the memory cache size to `0` will disable this enhanced caching mechanism and cause Subversion to fall back to using the older cache mechanisms introduced in Subversion 1.6.

Currently, only repositories which make use of the FSFS backend data store make use of this data caching functionality.

Network Compression of Data

Compressing the data transmitted across the wire can greatly reduce the size of those network transmissions, but comes at the cost of server (and client) CPU cycles. Depending on your server's CPU capacity, the typical access patterns of the clients who use your servers, and the bandwidth of the networks between them, you might wish to fine tune just how hard your server will work to compress the data it sends across the wire. To assist with this fine tuning process, Subversion 1.7 offers the `--compression` (`-c`) option to **svnserve** and the `SVNCompressionLevel` directive for **mod_dav_svn**. Both accept a value which is an integer between 0 and 9 (inclusive), where 9 offers the best compression of wire data, and 0 disables compression altogether.

For example, on a local area network (LAN) with 1-Gigabit connections, it might not make sense to have the server compress its network transmissions (which also forces the clients to decompress them), as the network itself is so fast that users won't really benefit from the smaller overall network payload. On the other hand, servers which are accessed primarily by clients with low-bandwidth connections would be doing those clients a favor by minimizing the overall size of its network communications.

Supporting Multiple Repository Access Methods

You've seen how a repository can be accessed in many different ways. But is it possible—or safe—for your repository to be accessed by multiple methods simultaneously? The answer is yes, provided you use a bit of foresight.

At any given time, these processes may require read and write access to your repository:

- Regular system users using a Subversion client (as themselves) to access the repository directly via `file://` URLs
- Regular system users connecting to SSH-spawned private **svnserve** processes (running as themselves), which access the repository
- An **svnserve** process—either a daemon or one launched by **inetd**—running as a particular fixed user
- An Apache **httpd** process, running as a particular fixed user

The most common problem administrators run into is repository ownership and permissions. Does every process (or user) in the preceding list have the rights to read and write the repository's underlying data files? Assuming you have a Unix-like operating system, a straightforward approach might be to place every potential repository user into a new `svn` group, and make the repository wholly owned by that group. But even that's not enough, because a process may write to the database files using an unfriendly umask—one that prevents access by other users.

So the next step beyond setting up a common group for repository users is to force every repository-accessing process to use a sane umask. For users accessing the repository directly, you can make the **svn** program into a wrapper script that first runs **`umask 002`** and then runs the real **svn** client program. You can write a similar wrapper script for the **svnserve** program, and add a **`umask 002`** command to Apache's own startup script, `apachectl`. For example:

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real "$@"
```

Another common problem is often encountered on Unix-like systems. If your repository is backed by Berkeley DB, for example, it occasionally creates new log files to journal its actions. Even if the Berkeley DB repository is wholly owned by the **svn** group, these newly created log files won't necessarily be owned by that same group, which then creates more permissions problems for your users. A good workaround is to set the group SUID bit on the repository's `db` directory. This causes all newly created log files to have the same group owner as the parent directory.

Once you've jumped through these hoops, your repository should be accessible by all the necessary processes. It may seem a bit messy and complicated, but the problems of having multiple users sharing write access to common files are classic ones that are not often elegantly solved.

Fortunately, most repository administrators will never *need* to have such a complex configuration. Users who wish to access repositories that live on the same machine are not limited to using `file://` access URLs—they can typically contact the Apache HTTP server or **svnserve** using `localhost` for the server name in their `http://` or `svn://` URL. And maintaining multiple server processes for your Subversion repositories is likely to be more of a headache than necessary. We recommend that you choose a single server that best meets your needs and stick with it!

The svn+ssh:// Server Checklist

It can be quite tricky to get a bunch of users with existing SSH accounts to share a repository without permissions problems. If you're confused about all the things that you (as an administrator) need to do on a Unix-like system, here's a quick checklist that resummarizes some of the topics discussed in this section:

- All of your SSH users need to be able to read and write to the repository, so put all the SSH users into a single group.

- Make the repository wholly owned by that group.
- Set the group permissions to read/write.
- Your users need to use a sane umask when accessing the repository, so make sure **svnserve** (`/usr/bin/svnserve`, or wherever it lives in `$PATH`) is actually a wrapper script that runs **`umask 002`** and executes the real **svnserve** binary.
- Take similar measures when using **svnlook** and **svnadmin**. Either run them with a sane umask or wrap them as just described.

Chapter 7. Customizing Your Subversion Experience

Version control can be a complex subject, as much art as science, that offers myriad ways of getting stuff done. Throughout this book, you've read of the various Subversion command-line client subcommands and the options that modify their behavior. In this chapter, we'll look into still more ways to customize the way Subversion works for you—setting up the Subversion runtime configuration, using external helper applications, Subversion's interaction with the operating system's configured locale, and so on.

Runtime Configuration Area

Subversion provides many optional behaviors that the user can control. Many of these options are of the kind that a user would wish to apply to all Subversion operations. So, rather than forcing users to remember command-line arguments for specifying these options and to use them for every operation they perform, Subversion uses configuration files, segregated into a Subversion configuration area.

The Subversion *configuration area* is a two-tiered hierarchy of option names and their values. Usually, this boils down to a special directory that contains *configuration files* (the first tier), which are just text files in standard INI format where “sections” provide the second tier. You can easily edit these files using your favorite text editor (such as Emacs or vi), and they contain directives read by the client to determine which of several optional behaviors the user prefers.

Configuration Area Layout

The first time the **svn** command-line client is executed, it creates a per-user configuration area. On Unix-like systems, this area appears as a directory named `.subversion` in the user's home directory. On Win32 systems, Subversion creates a folder named `Subversion`, typically inside the `Application Data` area of the user's profile directory (which, by the way, is usually a hidden directory). However, on this platform, the exact location differs from system to system and is dictated by the Windows Registry.[1] We will refer to the per-user configuration area using its Unix name, `.subversion`.

In addition to the per-user configuration area, Subversion also recognizes the existence of a system-wide configuration area. This gives system administrators the ability to establish defaults for all users on a given machine. Note that the system-wide configuration area alone does not dictate mandatory policy—the settings in the per-user configuration area override those in the system-wide one, and command-line arguments supplied to the **svn** program have the final word on behavior. On Unix-like platforms, the system-wide configuration area is expected to be the `/etc/subversion` directory; on Windows machines, it looks for a `Subversion` directory inside the common `Application Data` location (again, as specified by the Windows Registry). Unlike the per-user case, the **svn** program does not attempt to create the system-wide configuration area.

The per-user configuration area currently contains three files—two configuration files (`config` and `servers`), and a `README.txt` file, which describes the INI format. At the time of their creation, the files contain default values for each of the supported Subversion options, mostly commented out and grouped with textual descriptions about how the values for the key affect Subversion's behavior. To change a certain behavior, you need only to load the appropriate configuration file into a text editor, and to modify the desired option's value. If at any time you wish to have the default configuration settings restored, you can simply remove (or rename) your configuration directory and then run some innocuous **svn** command, such as **`svn --version`**. A new configuration directory with the default contents will be created.

Subversion also allows you to override individual configuration option values at the command line via the `--config-option` option, which is especially useful if you need to make a (very) temporary change in behavior. For more about this option's proper usage, see the section called “svn Options”.

The per-user configuration area also contains a cache of authentication data. The `auth` directory holds a set of subdirectories that contain pieces of cached information used by Subversion's various supported authentication methods. This directory is created in such a way that only the user herself has permission to read its contents.

[1] The `APPDATA` environment variable points to the `Application Data` area, so you can always refer to this folder as `%APPDATA%\Subversion`.

Configuration and the Windows Registry

In addition to the usual INI-based configuration area, Subversion clients running on Windows platforms may also use the Windows Registry to hold the configuration data. The option names and their values are the same as in the INI files. The “file/section” hierarchy is preserved as well, though addressed in a slightly different fashion—in this schema, files and sections are just levels in the Registry key tree.

Subversion looks for system-wide configuration values under the `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion` key. For example, the `global-ignores` option, which is in the `miscellany` section of the `config` file, would be found at `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores`. Per-user configuration values should be stored under `HKEY_CURRENT_USER\Software\Tigris.org\Subversion`.

Registry-based configuration options are parsed *before* their file-based counterparts, so they are overridden by values found in the configuration files. In other words, Subversion looks for configuration information in the following locations on a Windows system; lower-numbered locations take precedence over higher-numbered locations:

1. Command-line options
2. The per-user INI files
3. The per-user Registry values
4. The system-wide INI files
5. The system-wide Registry values

Also, the Windows Registry doesn't really support the notion of something being “commented out.” However, Subversion will ignore any option key whose name begins with a hash (#) character. This allows you to effectively comment out a Subversion option without deleting the entire key from the Registry, obviously simplifying the process of restoring that option.

The **svn** command-line client never attempts to write to the Windows Registry and will not attempt to create a default configuration area there. You can create the keys you need using the **REGEDIT** program. Alternatively, you can create a `.reg` file (such as the one in Example 7.1, “Sample registration entries (.reg) file”), and then double-click on that file's icon in the Explorer shell, which will cause the data to be merged into your Registry.

Example 7.1. Sample registration entries (.reg) file

```
REGEDIT4

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\groups]

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-auth-types"="basic;digest;negotiate"
"#http-compression"="yes"
"#http-library"=""
"#http-proxy-exceptions"=""
"#http-proxy-host"=""
"#http-proxy-password"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
"#http-timeout"="0"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""
```

```
"#ssl-pkcs11-provider"=""
"#ssl-trust-default-ca"=""
"#store-auth-creds"="yes"
"#store-passwords"="yes"
"#store-plaintext-passwords"="ask"
"#store-ssl-client-cert-pp"="yes"
"#store-ssl-client-cert-pp-plaintext"="ask"
"#username"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auth]
"#password-stores"="windows-cryptoapi"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\helpers]
"#diff-cmd"=""
"#diff-extensions"="-u"
"#diff3-cmd"=""
"#diff3-has-program-arg"=""
"#editor-cmd"="notepad"
"#merge-tool-cmd"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\tunnels]

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\miscellany]
"#enable-auto-props"="no"
"#global-ignores"="*.o *.lo *.la *.al .libs *.so *.so.[0-9]* *.a *.pyc *.pyo *.rej *~
#*# .#* .*.swp .DS_Store"
"#interactive-conflicts"="yes"
"#log-encoding"=""
"#mime-types-file"=""
"#no-unlock"="no"
"#preserved-conflict-file-exts"="doc ppt xls od?"
"#use-commit-times"="no"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auto-props]
```

Example 7.1, “Sample registration entries (.reg) file” shows the contents of a `.reg` file, which contains some of the most commonly used configuration options and their default values. Note the presence of both system-wide (for network proxy-related options) and per-user settings (editor programs and password storage, among others). Also note that all the options are effectively commented out. You need only to remove the hash (#) character from the beginning of the option names and set the values as you desire.

Configuration Options

In this section, we will discuss the specific runtime configuration options that Subversion currently supports.

Servers

The `servers` file contains Subversion configuration options related to the network layers. There are two special sections in this file—`[groups]` and `[global]`. The `[groups]` section is essentially a cross-reference table. The keys in this section are the names of other sections in the file; their values are *globs*—textual tokens that possibly contain wildcard characters—that are compared against the hostnames of the machine to which Subversion requests are sent.

```
[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
```

…

```
[collabnet]
…
```

When Subversion is used over a network, it attempts to match the name of the server it is trying to reach with a group name under the `[groups]` section. If a match is made, Subversion then looks for a section in the `servers` file whose name is the matched group's name. From that section, it reads the actual network configuration settings.

The `[global]` section contains the settings that are meant for all of the servers not matched by one of the globs under the `[groups]` section. The options available in this section are exactly the same as those that are valid for the other server sections in the file (except, of course, the special `[groups]` section), and are as follows:

`http-auth-types`
: This is a semicolon-delimited list of HTTP authentication types which the client will deem acceptable. Valid types are `basic`, `digest`, and `negotiate`, with the default behavior being acceptance of any these authentication types. A client which insists on not transmitting authentication credentials in cleartext might, for example, be configured such that the value of this option is `digest;negotiate`—omitting `basic` from the list. (Note that this setting is only honored by Subversion's Neon-based HTTP provider module.)

`http-compression`
: This specifies whether Subversion should attempt to compress network requests made to DAV-ready servers. The default value is `yes` (though compression will occur only if that capability is compiled into the network layer). Set this to `no` to disable compression, such as when debugging network transmissions.

`http-library`
: Subversion provides a pair of repository access modules that understand its WebDAV network protocol. The original one, which shipped with Subversion 1.0, is `libsvn_ra_neon` (though back then it was called `libsvn_ra_dav`). Newer Subversion versions also provide `libsvn_ra_serf`, which uses a different underlying implementation and aims to support some of the newer HTTP concepts.

  At this point, `libsvn_ra_serf` is still considered experimental, though it appears to work in the common cases quite well. To encourage experimentation, Subversion provides the `http-library` runtime configuration option to allow users to specify (generally, or in a per-server-group fashion) which WebDAV access module they'd prefer to use—`neon` or `serf`.

`http-proxy-exceptions`
: This specifies a comma-separated list of patterns for repository hostnames that should be accessed directly, without using the proxy machine. The pattern syntax is the same as is used in the Unix shell for filenames. A repository hostname matching any of these patterns will not be proxied.

`http-proxy-host`
: This specifies the hostname of the proxy computer through which your HTTP-based Subversion requests must pass. It defaults to an empty value, which means that Subversion will not attempt to route HTTP requests through a proxy computer, and will instead attempt to contact the destination machine directly.

`http-proxy-password`
: This specifies the password to supply to the proxy machine. It defaults to an empty value.

`http-proxy-port`
: This specifies the port number on the proxy host to use. It defaults to an empty value.

`http-proxy-username`
: This specifies the username to supply to the proxy machine. It defaults to an empty value.

`http-timeout`

This specifies the amount of time, in seconds, to wait for a server response. If you experience problems with a slow network connection causing Subversion operations to time out, you should increase the value of this option. The default value is `0`, which instructs the underlying HTTP library, Neon, to use its default timeout setting.

`neon-debug-mask`
This is an integer mask that the underlying HTTP library, Neon, uses for choosing what type of debugging output to yield. The default value is `0`, which will silence all debugging output. For more information about how Subversion makes use of Neon, see Chapter 8, *Embedding Subversion*.

`ssl-authority-files`
This is a semicolon-delimited list of paths to files containing certificates of the certificate authorities (or CAs) that are accepted by the Subversion client when accessing the repository over HTTPS.

`ssl-client-cert-file`
If a host (or set of hosts) requires an SSL client certificate, you'll normally be prompted for a path to your certificate. By setting this variable to that same path, Subversion will be able to find your client certificate automatically without prompting you. There's no standard place to store your certificate on disk; Subversion will grab it from any path you specify.

`ssl-client-cert-password`
If your SSL client certificate file is encrypted by a passphrase, Subversion will prompt you for the passphrase whenever the certificate is used. If you find this annoying (and don't mind storing the password in the `servers` file), you can set this variable to the certificate's passphrase. You won't be prompted anymore.

`ssl-pkcs11-provider`
The value of this option is the name of the PKCS#11 provider from which an SSL client certificate will be drawn (if the server asks for one). This setting is only honored by Subversion's Neon-based HTTP provider module.

`ssl-trust-default-ca`
Set this variable to `yes` if you want Subversion to automatically trust the set of default CAs that ship with OpenSSL.

`store-auth-creds`
This setting is the same as `store-passwords`, except that it enables or disables on-disk caching of *all* authentication information: usernames, passwords, server certificates, and any other types of cacheable credentials.

`store-passwords`
This instructs Subversion to cache, or not to cache, passwords that are supplied by the user in response to server authentication challenges. The default value is `yes`. Set this to `no` to disable this on-disk password caching. You can override this option for a single instance of the **svn** command using the `--no-auth-cache` command-line parameter (for those subcommands that support it). For more information regarding that, see the section called "Caching credentials". Note that regardless of how this option is configured, Subversion will not store passwords in plaintext unless the `store-plaintext-passwords` option is also set to `yes`.

`store-plaintext-passwords`
This variable is only important on UNIX-like systems. It controls what the Subversion client does in case the password for the current authentication realm can only be cached on disk in unencrypted form, in the `~/.subversion/auth/` caching area. You can set it to `yes` or `no` to enable or disable caching of passwords in unencrypted form, respectively. The default setting is `ask`, which causes the Subversion client to ask you each time a *new* password is about to be added to the `~/.subversion/auth/` caching area.

`store-ssl-client-cert-pp`
This option controls whether Subversion will cache SSL client certificate passphrases provided by the user. Its value defaults to `yes`. Set this to `no` to disable this passphrase caching.

`store-ssl-client-cert-pp-plaintext`
This option controls whether Subversion, when attempting to cache an SSL client certificate passphrase, will be allowed to do so using its on-disk plaintext storage mechanism. The default value of this option is `ask`, which causes the Subversion client to ask you each time a *new* client certificate passphrase is about to be added to the `~/.subversion/auth/` caching area. Set this option's value to `yes` or `no` to indicate your preference and avoid related prompts.

Config

The `config` file contains the rest of the currently available Subversion runtime options—those not related to networking. There are only a few options in use as of this writing, but they are again grouped into sections in expectation of future additions.

The `[auth]` section contains settings related to Subversion's authentication and authorization against the repository. It contains the following:

`password-stores`
: This comma-delimited list specifies which (if any) system-provided password stores Subversion should attempt to use when saving and retrieving cached authentication credentials, and in what order Subversion should prefer them. The default value is `gnome-keyring, kwallet, keychain, windows-crypto-api`, representing the GNOME Keyring, KDE Wallet, Mac OS X Keychain, and Microsoft Windows cryptography API, respectively. Listed stores which are not available on the system are ignored.

`store-passwords`
: This option has been deprecated from the `config` file. It now lives as a per-server configuration item in the `servers` configuration area. See the section called "Servers" for details.

`store-auth-creds`
: This option has been deprecated from the `config` file. It now lives as a per-server configuration item in the `servers` configuration area. See the section called "Servers" for details.

The `[helpers]` section controls which external applications Subversion uses to accomplish its tasks. Valid options in this section are:

`diff-cmd`
: This specifies the absolute path of a differencing program, used when Subversion generates "diff" output (such as when using the **svn diff** command). By default, Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See the section called "Using External Differencing and Merge Tools" for more details on using such programs.

`diff-extensions`
: Like the `--extensions` (`-x`) command-line option, this specifies additional options passed to the file content differencing engine. The set of meaningful extension options differs depending on whether the client is using Subversion's internal differencing engine or an external mechanism. See the output of **`svn help diff`** for details. The default value for this option is `-u`.

`diff3-cmd`
: This specifies the absolute path of a three-way differencing program. Subversion uses this program to merge changes made by the user with those received from the repository. By default, Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See the section called "Using External Differencing and Merge Tools" for more details on using such programs.

`diff3-has-program-arg`
: This flag should be set to `true` if the program specified by the `diff3-cmd` option accepts a `--diff-program` command-line parameter.

`editor-cmd`
: This specifies the program Subversion will use to query the user for certain types of textual metadata or when interactively resolving conflicts. See the section called "Using External Editors" for more details on using external text editors with Subversion.

`merge-tool-cmd`
: This specifies the program that Subversion will use to perform three-way merge operations on your versioned files. See the section called "Using External Differencing and Merge Tools" for more details on using such programs.

The `[tunnels]` section allows you to define new tunnel schemes for use with **svnserve** and `svn://` client connections. For more details, see the section called "Tunneling over SSH".

The `miscellany` section is where everything that doesn't belong elsewhere winds up.[2] In this section, you can find:

`enable-auto-props`
: This instructs Subversion to automatically set properties on newly added or imported files. The default value is `no`, so set this to `yes` to enable this feature. The `[auto-props]` section of this file specifies which properties are to be set on which files.

`global-ignores`
: When running the **svn status** command, Subversion lists unversioned files and directories along with the versioned ones, annotating them with a `?` character (see the section called "See an overview of your changes"). Sometimes it can be annoying to see uninteresting, unversioned items—for example, object files that result from a program's compilation—in this display. The `global-ignores` option is a list of whitespace-delimited globs that describe the names of files and directories that Subversion should not display unless they are versioned. The default value is `*.o *.lo *.la *.al .libs *.so *.so.[0-9]* *.a *.pyc *.pyo *.rej *~ #*# .#* .*.swp .DS_Store`.

  As well as **svn status**, the **svn add** and **svn import** commands also ignore files that match the list when they are scanning a directory. You can override this behavior for a single instance of any of these commands by explicitly specifying the filename, or by using the `--no-ignore` command-line flag.

  For information on finer-grained control of ignored items, see the section called "Ignoring Unversioned Items".

`interactive-conflicts`
: This is a Boolean option that specifies whether Subversion should try to resolve conflicts interactively. If its value is `yes` (which is the default value), Subversion will prompt the user for how to handle conflicts in the manner demonstrated in the section called "Resolve Any Conflicts". Otherwise, it will simply flag the conflict and continue its operation, postponing resolution to a later time.

`log-encoding`
: This variable sets the default character set encoding for commit log messages. It's a permanent form of the `--encoding` option (see the section called "svn Options"). The Subversion repository stores log messages in UTF-8 and assumes that your log message is written using your operating system's native locale. You should specify a different encoding if your commit messages are written in any other encoding.

`mime-types-file`
: This option, new to Subversion 1.5, specifies the path of a MIME types mapping file, such as the `mime.types` file provided by the Apache HTTP Server. Subversion uses this file to assign MIME types to newly added or imported files. See the section called "Automatic Property Setting" and the section called "File Content Type" for more about Subversion's detection and use of file content types.

`no-unlock`
: This Boolean option corresponds to **svn commit**'s `--no-unlock` option, which tells Subversion not to release locks on files you've just committed. If this runtime option is set to `yes`, Subversion will never release locks automatically, leaving you to run **svn unlock** explicitly. It defaults to `no`.

`preserved-conflict-file-exts`
: The value of this option is a space-delimited list of file extensions that Subversion should preserve when generating conflict filenames. By default, the list is empty. This option is new to Subversion 1.5.

  When Subversion detects conflicting file content changes, it defers resolution of those conflicts to the user. To assist in the resolution, Subversion keeps pristine copies of the various competing versions of the file in the working copy. By default, those conflict files have names constructed by appending to the original filename a custom extension such as `.mine` or `.`*`REV`* (where *`REV`* is a revision number). A mild annoyance with this naming scheme is that on operating systems where a file's extension determines the default application used to open and edit that file, appending a custom extension prevents the file from being easily opened by its native application. For example, if the file `ReleaseNotes.pdf` was conflicted, the conflict files

[2] Anyone for potluck dinner?

might be named `ReleaseNotes.pdf.mine` or `ReleaseNotes.pdf.r4231`. While your system might be configured to use Adobe's Acrobat Reader to open files whose extensions are `.pdf`, there probably isn't an application configured on your system to open all files whose extensions are `.r4231`.

You can fix this annoyance by using this configuration option, though. For files with one of the specified extensions, Subversion will append to the conflict file names the custom extension just as before, but then also reappend the file's original extension. Using the previous example, and assuming that `pdf` is one of the extensions configured in this list thereof, the conflict files generated for `ReleaseNotes.pdf` would instead be named `ReleaseNotes.pdf.mine.pdf` and `ReleaseNotes.pdf.r4231.pdf`. Because each file ends in `.pdf`, the correct default application will be used to view them.

`use-commit-times`

Normally your working copy files have timestamps that reflect the last time they were touched by any process, whether your own editor or some **svn** subcommand. This is generally convenient for people developing software, because build systems often look at timestamps as a way of deciding which files need to be recompiled.

In other situations, however, it's sometimes nice for the working copy files to have timestamps that reflect the last time they were changed in the repository. The **svn export** command always places these “last-commit timestamps” on trees that it produces. By setting this config variable to `yes`, the **svn checkout**, **svn update**, **svn switch**, and **svn revert** commands will also set last-commit timestamps on files that they touch.

The `[auto-props]` section controls the Subversion client's ability to automatically set properties on files when they are added or imported. It contains any number of key-value pairs in the format *`PATTERN = PROPNAME=VALUE`*`[;`*`PROPNAME=VALUE`*`...]`, where *`PATTERN`* is a file pattern that matches one or more filenames and the rest of the line is a semicolon-delimited set of property assignments. (If you need to use a semicolon in your property's name or value, you can escape it by doubling it.)

```
$ cat ~/.subversion/config
…
[auto-props]
*.c = svn:eol-style=native
*.html = svn:eol-style=native;svn:mime-type=text/html;; charset=UTF8
*.sh = svn:eol-style=native;svn:executable
…
$ cd projects/myproject
$ svn status
?       www/index.html
$ svn add www/index.html
A         www/index.html
$ svn diff www/index.html
…

Property changes on: www/index.html
___________________________________________________________________
Added: svn:mime-type
## -0,0 +1 ##
+text/html; charset=UTF8
Added: svn:eol-style
## -0,0 +1 ##
+native
$
```

Multiple matches on a file will result in multiple propsets for that file; however, there is no guarantee that auto-props will be applied in the order in which they are listed in the config file, so you can't have one rule “override” another. You can find several examples of auto-props usage in the `config` file. Lastly, don't forget to set `enable-auto-props` to `yes` in the `miscellany` section if you want to enable auto-props.

Localization

Localization is the act of making programs behave in a region-specific way. When a program formats numbers or dates in a way specific to your part of the world or prints messages (or accepts input) in your native language, the program is said to be *localized*. This section describes steps Subversion has made toward localization.

Understanding Locales

Most modern operating systems have a notion of the “current locale”—that is, the region or country whose localization conventions are honored. These conventions—typically chosen by some runtime configuration mechanism on the computer—affect the way in which programs present data to the user, as well as the way in which they accept user input.

On most Unix-like systems, you can check the values of the locale-related runtime configuration options by running the **locale** command:

```
$ locale
LANG=
LC_COLLATE="C"
LC_CTYPE="C"
LC_MESSAGES="C"
LC_MONETARY="C"
LC_NUMERIC="C"
LC_TIME="C"
LC_ALL="C"
$
```

The output is a list of locale-related environment variables and their current values. In this example, the variables are all set to the default `C` locale, but users can set these variables to specific country/language code combinations. For example, if one were to set the `LC_TIME` variable to `fr_CA`, programs would know to present time and date information formatted according to a French-speaking Canadian's expectations. And if one were to set the `LC_MESSAGES` variable to `zh_TW`, programs would know to present human-readable messages in Traditional Chinese. Setting the `LC_ALL` variable has the effect of changing every locale variable to the same value. The value of `LANG` is used as a default value for any locale variable that is unset. To see the list of available locales on a Unix system, run the command **`locale -a`**.

On Windows, locale configuration is done via the “Regional and Language Options” control panel item. There you can view and select the values of individual settings from the available locales, and even customize (at a sickening level of detail) several of the display formatting conventions.

Subversion's Use of Locales

The Subversion client, **svn**, honors the current locale configuration in two ways. First, it notices the value of the `LC_MESSAGES` variable and attempts to print all messages in the specified language. For example:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

This behavior works identically on both Unix and Windows systems. Note, though, that while your operating system might have support for a certain locale, the Subversion client still may not be able to speak the particular language. In order to produce localized messages, human volunteers must provide translations for each language. The translations are written using the GNU gettext package, which results in translation modules that end with the `.mo` filename extension. For example, the German translation file

is named `de.mo`. These translation files are installed somewhere on your system. On Unix, they typically live in `/usr/share/locale/`, while on Windows they're often found in the `share\locale\` folder in Subversion's installation area. Once installed, a module is named after the program for which it provides translations. For example, the `de.mo` file may ultimately end up installed as `/usr/share/locale/de/LC_MESSAGES/subversion.mo`. By browsing the installed `.mo` files, you can see which languages the Subversion client is able to speak.

The second way in which the locale is honored involves how **svn** interprets your input. The repository stores all paths, filenames, and log messages in Unicode, encoded as UTF-8. In that sense, the repository is *internationalized*—that is, the repository is ready to accept input in any human language. This means, however, that the Subversion client is responsible for sending only UTF-8 filenames and log messages into the repository. To do this, it must convert the data from the native locale into UTF-8.

For example, suppose you create a file named `caffè.txt`, and then when committing the file, you write the log message as “Adesso il caffè è più forte.” Both the filename and the log message contain non-ASCII characters, but because your locale is set to `it_IT`, the Subversion client knows to interpret them as Italian. It uses an Italian character set to convert the data to UTF-8 before sending it off to the repository.

Note that while the repository demands UTF-8 filenames and log messages, it *does not* pay attention to file contents. Subversion treats file contents as opaque strings of bytes, and neither client nor server makes an attempt to understand the character set or encoding of the contents.

Character Set Conversion Errors

While using Subversion, you might get hit with an error related to character set conversions:

```
svn: E000022: Can't convert string from native encoding to 'UTF-8':
…
svn: E000022: Can't convert string from 'UTF-8' to native encoding:
…
```

Errors such as this typically occur when the Subversion client has received a UTF-8 string from the repository, but not all of the characters in that string can be represented using the encoding of the current locale. For example, if your locale is `en_US` but a collaborator has committed a Japanese filename, you're likely to see this error when you receive the file during an **svn update**.

The solution is either to set your locale to something that *can* represent the incoming UTF-8 data, or to change the filename or log message in the repository. (And don't forget to slap your collaborator's hand—projects should decide on common languages ahead of time so that all participants are using the same locale.)

Using External Editors

The most obvious way to get data into Subversion is through the addition of files to version control, committing changes to those files, and so on. But other pieces of information besides merely versioned file data live in your Subversion repository. Some of these bits of information—commit log messages, lock comments, and some property values—tend to be textual in nature and are provided explicitly by users. Most of this information can be provided to the Subversion command-line client using the `--message` (`-m`) and `--file` (`-F`) options with the appropriate subcommands.

Each of these options has its pros and cons. For example, when performing a commit, `--file` (`-F`) works well if you've already prepared a text file that holds your commit log message. If you didn't, though, you can use `--message` (`-m`) to provide a log message on the command line. Unfortunately, it can be tricky to compose anything more than a simple one-line message on the command line. Users want more flexibility—multiline, free-form log message editing on demand.

Subversion supports this by allowing you to specify an external text editor that it will launch as necessary to give you a more powerful input mechanism for this textual metadata. There are several ways to tell Subversion which editor you'd like use. Subver-

sion checks the following things, in the order specified, when it wants to launch such an editor:

1. `--editor-cmd` command-line option
2. `SVN_EDITOR` environment variable
3. `editor-cmd` runtime configuration option
4. `VISUAL` environment variable
5. `EDITOR` environment variable
6. Possibly, a fallback value built into the Subversion libraries (not present in the official builds)

The value of any of these options or variables is the beginning of a command line to be executed by the shell. Subversion appends to that command line a space and the pathname of a temporary file to be edited. So, to be used with Subversion, the configured or specified editor needs to support an invocation in which its last command-line parameter is a file to be edited, and it should be able to save the file in place and return a zero exit code to indicate success.

As noted, external editors can be used to provide commit log messages to any of the committing subcommands (such as **svn commit** or **import**, **svn mkdir** or **delete** when provided a URL target, etc.), and Subversion will try to launch the editor automatically if you don't specify either of the `--message` (`-m`) or `--file` (`-F`) options. The **svn propedit** command is built almost entirely around the use of an external editor. And beginning in version 1.5, Subversion will also use the configured external text editor when the user asks it to launch an editor during interactive conflict resolution. Oddly, there doesn't appear to be a way to use external editors to interactively provide lock comments.

Using External Differencing and Merge Tools

The interface between Subversion and external two- and three-way differencing tools harkens back to a time when Subversion's only contextual differencing capabilities were built around invocations of the GNU diffutils toolchain, specifically the **diff** and **diff3** utilities. To get the kind of behavior Subversion needed, it called these utilities with more than a handful of options and parameters, most of which were quite specific to the utilities. Some time later, Subversion grew its own internal differencing library, and as a failover mechanism, the `--diff-cmd` and `--diff3-cmd` options were added to the Subversion command-line client so that users could more easily indicate that they preferred to use the GNU diff and diff3 utilities instead of the newfangled internal diff library. If those options were used, Subversion would simply ignore the internal diff library, and fall back to running those external programs, lengthy argument lists and all. And that's where things remain today.

It didn't take long for folks to realize that having such easy configuration mechanisms for specifying that Subversion should use the external GNU diff and diff3 utilities located at a particular place on the system could be applied toward the use of other differencing tools, too. After all, Subversion didn't actually verify that the things it was being told to run were members of the GNU diffutils toolchain. But the only configurable aspect of using those external tools is their location on the system—not the option set, parameter order, and so on. Subversion continues to throw all those GNU utility options at your external diff tool regardless of whether that program can understand those options. And that's where things get unintuitive for most users.

The decision on when to fire off a contextual two- or three-way diff as part of a larger Subversion operation is made entirely by Subversion and is affected by, among other things, whether the files being operated on are human-readable as determined by their `svn:mime-type` property. This means, for example, that even if you had the niftiest Microsoft Word-aware differencing or merging tool in the universe, it would never be invoked by Subversion as long as your versioned Word documents had a configured MIME type that denoted that they were not human-readable (such as `application/msword`). For more about MIME type settings, see the section called “File Content Type”

Much later, Subversion 1.5 introduced interactive resolution of conflicts (described in the section called “Resolve Any Conflicts”). One of the options that this feature provides to users is the ability to interactively launch a third-party merge tool. If this action is taken, Subversion will check to see if the user has specified such a tool for use in this way. Subversion will first check the

`SVN_MERGE` environment variable for the name of an external merge tool. If that variable is not set, it will look for the same information in the value of the `merge-tool-cmd` runtime configuration option. Upon finding a configured external merge tool, it will invoke that tool.

While the general purposes of the three-way differencing and merge tools are roughly the same (finding a way to make separate-but-overlapping file changes live in harmony), Subversion exercises each of these options at different times and for different reasons. The internal three-way differencing engine and its optional external replacement are used when interaction with the user is *not* expected. In fact, significant delay introduced by such a tool can actually result in the failure of some time-sensitive Subversion operations. It's the external merge tool that is intended to be invoked interactively.

Now, while the interface between Subversion and an external merge tool is significantly less convoluted than that between Subversion and the diff and diff3 tools, the likelihood of finding such a tool whose calling conventions exactly match what Subversion expects is still quite low. The key to using external differencing and merge tools with Subversion is to use wrapper scripts, which convert the input from Subversion into something that your specific differencing tool can understand, and then convert the output of your tool back into a format that Subversion expects. The following sections cover the specifics of those expectations.

External diff

Subversion calls external diff programs with parameters suitable for the GNU diff utility, and expects only that the external program will return with a successful error code per the GNU diff definition thereof. For most alternative diff programs, only the sixth and seventh arguments—the paths of the files that represent the left and right sides of the diff, respectively—are of interest. Note that Subversion runs the diff program once per modified file covered by the Subversion operation, so if your program runs in an asynchronous fashion (or is “backgrounded”), you might have several instances of it all running simultaneously. Finally, Subversion expects that your program return an error code of 1 if your program detected differences, or 0 if it did not—any other error code is considered a fatal error.[3]

Example 7.2, “diffwrap.py” and Example 7.3, “diffwrap.bat” are templates for external diff tool wrappers in the Python and Windows batch scripting languages, respectively.

Example 7.2. diffwrap.py

```
#!/usr/bin/env python
import sys
import os

# Configure your favorite diff program here.
DIFF = "/usr/local/bin/my-diff-tool"

# Subversion provides the paths we need as the last two parameters.
LEFT  = sys.argv[-2]
RIGHT = sys.argv[-1]

# Call the diff command (change the following line to make sense for
# your diff program).
cmd = [DIFF, '--left', LEFT, '--right', RIGHT]
os.execv(cmd[0], cmd)

# Return an errorcode of 0 if no differences were detected, 1 if some were.
# Any other errorcode will be treated as fatal.
```

[3]The GNU diff manual page puts it this way: “An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble.”

Example 7.3. diffwrap.bat

```
@ECHO OFF

REM Configure your favorite diff program here.
SET DIFF="C:\Program Files\Funky Stuff\My Diff Tool.exe"

REM Subversion provides the paths we need as the last two parameters.
REM These are parameters 6 and 7 (unless you use svn diff -x, in
REM which case, all bets are off).
SET LEFT=%6
SET RIGHT=%7

REM Call the diff command (change the following line to make sense for
REM your diff program).
%DIFF% --left %LEFT% --right %RIGHT%

REM Return an errorcode of 0 if no differences were detected, 1 if some were.
REM Any other errorcode will be treated as fatal.
```

External diff3

Subversion invokes three-way differencing programs to perform non-interactive merges. When configured to use an external three-way differencing program, it executes that program with parameters suitable for the GNU diff3 utility, expecting that the external program will return with a successful error code and that the full file contents that result from the completed merge operation are printed on the standard output stream (so that Subversion can redirect them into the appropriate version-controlled file). For most alternative merge programs, only the ninth, tenth, and eleventh arguments, the paths of the files which represent the “mine”, “older”, and “yours” inputs, respectively, are of interest. Note that because Subversion depends on the output of your merge program, your wrapper script must not exit before that output has been delivered to Subversion. When it finally does exit, it should return an error code of 0 if the merge was successful, or 1 if unresolved conflicts remain in the output—any other error code is considered a fatal error.

Example 7.4, “diff3wrap.py” and Example 7.5, “diff3wrap.bat” are templates for external three-way differencing tool wrappers in the Python and Windows batch scripting languages, respectively.

Example 7.4. diff3wrap.py

```
#!/usr/bin/env python
import sys
import os

# Configure your favorite three-way diff program here.
DIFF3 = "/usr/local/bin/my-diff3-tool"

# Subversion provides the paths we need as the last three parameters.
MINE  = sys.argv[-3]
OLDER = sys.argv[-2]
YOURS = sys.argv[-1]

# Call the three-way diff command (change the following line to make
# sense for your three-way diff program).
cmd = [DIFF3, '--older', OLDER, '--mine', MINE, '--yours', YOURS]
os.execv(cmd[0], cmd)
```

```
# After performing the merge, this script needs to print the contents
# of the merged file to stdout.  Do that in whatever way you see fit.
# Return an errorcode of 0 on successful merge, 1 if unresolved conflicts
# remain in the result.  Any other errorcode will be treated as fatal.
```

Example 7.5. diff3wrap.bat

```
@ECHO OFF

REM Configure your favorite three-way diff program here.
SET DIFF3="C:\Program Files\Funky Stuff\My Diff3 Tool.exe"

REM Subversion provides the paths we need as the last three parameters.
REM These are parameters 9, 10, and 11.  But we have access to only
REM nine parameters at a time, so we shift our nine-parameter window
REM twice to let us get to what we need.
SHIFT
SHIFT
SET MINE=%7
SET OLDER=%8
SET YOURS=%9

REM Call the three-way diff command (change the following line to make
REM sense for your three-way diff program).
%DIFF3% --older %OLDER% --mine %MINE% --yours %YOURS%

REM After performing the merge, this script needs to print the contents
REM of the merged file to stdout.  Do that in whatever way you see fit.
REM Return an errorcode of 0 on successful merge, 1 if unresolved conflicts
REM remain in the result.  Any other errorcode will be treated as fatal.
```

External merge

Subversion optionally invokes an external merge tool as part of its support for interactive conflict resolution. It provides as arguments to the merge tool the following: the path of the unmodified base file, the path of the “theirs” file (which contains upstream changes), the path of the “mine” file (which contains local modifications), the path of the file into which the final resolved contents should be stored by the merge tool, and the working copy path of the conflicted file (relative to the original target of the merge operation). The merge tool is expected to return an error code of 0 to indicate success, or 1 to indicate failure.

Example 7.6, “mergewrap.py” and Example 7.7, “mergewrap.bat” are templates for external merge tool wrappers in the Python and Windows batch scripting languages, respectively.

Example 7.6. mergewrap.py

```
#!/usr/bin/env python
import sys
import os

# Configure your favorite merge program here.
MERGE = "/usr/local/bin/my-merge-tool"
```

```
# Get the paths provided by Subversion.
BASE   = sys.argv[1]
THEIRS = sys.argv[2]
MINE   = sys.argv[3]
MERGED = sys.argv[4]
WCPATH = sys.argv[5]

# Call the merge command (change the following line to make sense for
# your merge program).
cmd = [DIFF3, '--base', BASE, '--mine', MINE, '--theirs', THEIRS,
              '--outfile', MERGED]
os.execv(cmd[0], cmd)

# Return an errorcode of 0 if the conflict was resolved; 1 otherwise.
# Any other errorcode will be treated as fatal.
```

Example 7.7. mergewrap.bat

```
@ECHO OFF

REM Configure your favorite merge program here.
SET DIFF3="C:\Program Files\Funky Stuff\My Merge Tool.exe"

REM Get the paths provided by Subversion.
SET BASE=%1
SET THEIRS=%2
SET MINE=%3
SET MERGED=%4
SET WCPATH=%5

REM Call the merge command (change the following line to make sense for
REM your merge program).
%DIFF3% --base %BASE% --mine %MINE% --theirs %THEIRS% --outfile %MERGED%

REM Return an errorcode of 0 if the conflict was resolved; 1 otherwise.
REM Any other errorcode will be treated as fatal.
```

Summary

Sometimes there's a single right way to do things; sometimes there are many. Subversion's developers understand that while the majority of its exact behaviors are acceptable to most of its users, there are some corners of its functionality where such a universally pleasing approach doesn't exist. In those places, Subversion offers users the opportunity to tell it how *they* want it to behave.

In this chapter, we explored Subversion's runtime configuration system and other mechanisms by which users can control those configurable behaviors. If you are a developer, though, the next chapter will take you one step further. It describes how you can further customize your Subversion experience by writing your own software against Subversion's libraries.

Chapter 8. Embedding Subversion

Subversion has a modular design: it's implemented as a collection of libraries written in C. Each library has a well-defined purpose and application programming interface (API), and that interface is available not only for Subversion itself to use, but for any software that wishes to embed or otherwise programmatically control Subversion. Additionally, Subversion's API is available not only to other C programs, but also to programs written in higher-level languages such as Python, Perl, Java, and Ruby.

This chapter is for those who wish to interact with Subversion through its public API or its various language bindings. If you wish to write robust wrapper scripts around Subversion functionality to simplify your own life, are trying to develop more complex integrations between Subversion and other pieces of software, or just have an interest in Subversion's various library modules and what they offer, this chapter is for you. If, however, you don't foresee yourself participating with Subversion at such a level, feel free to skip this chapter with the confidence that your experience as a Subversion user will not be affected.

Layered Library Design

Each of Subversion's core libraries can be said to exist in one of three main layers—the Repository layer, the Repository Access (RA) layer, or the Client layer (see Figure 1, “Subversion's architecture” in the Preface). We will examine these layers shortly, but first, let's briefly summarize Subversion's various libraries. For the sake of consistency, we will refer to the libraries by their extensionless Unix library names (`libsvn_fs`, `libsvn_wc`, `mod_dav_svn`, etc.).

libsvn_client
: Primary interface for client programs

libsvn_delta
: Tree and byte-stream differencing routines

libsvn_diff
: Contextual differencing and merging routines

libsvn_fs
: Filesystem commons and module loader

libsvn_fs_base
: The Berkeley DB filesystem backend

libsvn_fs_fs
: The native filesystem (FSFS) backend

libsvn_ra
: Repository Access commons and module loader

libsvn_ra_local
: The local Repository Access module

libsvn_ra_neon
: The WebDAV Repository Access module

libsvn_ra_serf
: Another (experimental) WebDAV Repository Access module

libsvn_ra_svn
: The custom protocol Repository Access module

libsvn_repos
: Repository interface

libsvn_subr
: Miscellaneous helpful subroutines

libsvn_wc
: The working copy management library

mod_authz_svn
: Apache authorization module for Subversion repositories access via WebDAV

mod_dav_svn
: Apache module for mapping WebDAV operations to Subversion ones

The fact that the word “miscellaneous” appears only once in the previous list is a good sign. The Subversion development team is serious about making sure that functionality lives in the right layer and libraries. Perhaps the greatest advantage of the modular design is its lack of complexity from a developer's point of view. As a developer, you can quickly formulate that kind of “big picture” that allows you to pinpoint the location of certain pieces of functionality with relative ease.

Another benefit of modularity is the ability to replace a given module with a whole new library that implements the same API without affecting the rest of the code base. In some sense, this happens within Subversion already. The `libsvn_ra_local`, `libsvn_ra_neon`, `libsvn_ra_serf`, and `libsvn_ra_svn` libraries each implement the same interface, all working as plug-ins to `libsvn_ra`. And all four communicate with the Repository layer—`libsvn_ra_local` connects to the repository directly; the other three do so over a network. The `libsvn_fs_base` and `libsvn_fs_fs` libraries are another pair of libraries that implement the same functionality in different ways—both are plug-ins to the common `libsvn_fs` library.

The client itself also highlights the benefits of modularity in the Subversion design. Subversion's `libsvn_client` library is a one-stop shop for most of the functionality necessary for designing a working Subversion client (see the section called “Client Layer”). So while the Subversion distribution provides only the **svn** command-line client program, several third-party programs provide various forms of graphical client UIs. These GUIs use the same APIs that the stock command-line client does. This type of modularity has played a large role in the proliferation of available Subversion clients and IDE integrations and, by extension, to the tremendous adoption rate of Subversion itself.

Repository Layer

When referring to Subversion's Repository layer, we're generally talking about two basic concepts—the versioned filesystem implementation (accessed via `libsvn_fs`, and supported by its `libsvn_fs_base` and `libsvn_fs_fs` plug-ins), and the repository logic that wraps it (as implemented in `libsvn_repos`). These libraries provide the storage and reporting mechanisms for the various revisions of your version-controlled data. This layer is connected to the Client layer via the Repository Access layer, and is, from the perspective of the Subversion user, the stuff at the “other end of the line.”

The Subversion filesystem is not a kernel-level filesystem that one would install in an operating system (such as the Linux ext2 or NTFS), but instead is a virtual filesystem. Rather than storing “files” and “directories” as real files and directories (the kind you can navigate through using your favorite shell program), it uses one of two available abstract storage backends—either a Berkeley DB database environment or a flat-file representation. (To learn more about the two repository backends, see the section called “Choosing a Data Store”.) There has even been considerable interest by the development community in giving future releases of Subversion the ability to use other backend database systems, perhaps through a mechanism such as Open Database Connectivity (ODBC). In fact, Google did something similar to this before launching the Google Code Project Hosting service: they announced in mid-2006 that members of its open source team had written a new proprietary Subversion filesystem plug-in that used Google's ultra-scalable Bigtable database for its storage.

The filesystem API exported by `libsvn_fs` contains the kinds of functionality you would expect from any other filesystem API—you can create and remove files and directories, copy and move them around, modify file contents, and so on. It also has features that are not quite as common, such as the ability to add, modify, and remove metadata (“properties”) on each file or directory. Furthermore, the Subversion filesystem is a versioning filesystem, which means that as you make changes to your directory tree, Subversion remembers what your tree looked like before those changes. And before the previous changes. And the previous ones. And so on, all the way back through versioning time to (and just beyond) the moment you first started adding things to the filesystem.

All the modifications you make to your tree are done within the context of a Subversion commit transaction. The following is a simplified general routine for modifying your filesystem:

1. Begin a Subversion commit transaction.
2. Make your changes (adds, deletes, property modifications, etc.).
3. Commit your transaction.

Once you have committed your transaction, your filesystem modifications are permanently stored as historical artifacts. Each of these cycles generates a single new revision of your tree, and each revision is forever accessible as an immutable snapshot of “the way things were.”

The Transaction Distraction

The notion of a Subversion transaction can become easily confused with the transaction support provided by the underlying database itself, especially given the former's close proximity to the Berkeley DB database code in `libsvn_fs_base`. Both types of transaction exist to provide atomicity and isolation. In other words, transactions give you the ability to perform a set of actions in an all-or-nothing fashion—either all the actions in the set complete with success, or they all get treated as though *none* of them ever happened—and in a way that does not interfere with other processes acting on the data.

Database transactions generally encompass small operations related specifically to the modification of data in the database itself (such as changing the contents of a table row). Subversion transactions are larger in scope, encompassing higher-level operations such as making modifications to a set of files and directories that are intended to be stored as the next revision of the filesystem tree. If that isn't confusing enough, consider the fact that Subversion uses a database transaction during the creation of a Subversion transaction (so that if the creation of a Subversion transaction fails, the database will look as though we had never attempted that creation in the first place)!

Fortunately for users of the filesystem API, the transaction support provided by the database system itself is hidden almost entirely from view (as should be expected from a properly modularized library scheme). It is only when you start digging into the implementation of the filesystem itself that such things become visible (or interesting).

Most of the functionality the filesystem interface provides deals with actions that occur on individual filesystem paths. That is, from outside the filesystem, the primary mechanism for describing and accessing the individual revisions of files and directories comes through the use of path strings such as `/foo/bar`, just as though you were addressing files and directories through your favorite shell program. You add new files and directories by passing their paths-to-be to the right API functions. You query for information about them by the same mechanism.

Unlike most filesystems, though, a path alone is not enough information to identify a file or directory in Subversion. Think of a directory tree as a two-dimensional system, where a node's siblings represent a sort of left-and-right motion, and navigating into the node's subdirectories represents a downward motion. Figure 8.1, “Files and directories in two dimensions” shows a typical representation of a tree as exactly that.

Figure 8.1. Files and directories in two dimensions

The difference here is that the Subversion filesystem has a nifty third dimension that most filesystems do not have—Time![1] In the filesystem interface, nearly every function that has a *`path`* argument also expects a *`root`* argument. This `svn_fs_root_t` argument describes either a revision or a Subversion transaction (which is simply a revision in the making) and provides that third dimension of context needed to understand the difference between `/foo/bar` in revision 32, and the same path as it exists in revision 98. Figure 8.2, "Versioning time—the third dimension!" shows revision history as an added dimension to the Subversion filesystem universe.

Figure 8.2. Versioning time—the third dimension!

As we mentioned earlier, the `libsvn_fs` API looks and feels like any other filesystem, except that it has this wonderful versioning capability. It was designed to be usable by any program interested in a versioning filesystem. Not coincidentally, Subversion itself is interested in that functionality. But while the filesystem API should be sufficient for basic file and directory versioning support, Subversion wants more—and that is where `libsvn_repos` comes in.

The Subversion repository library (`libsvn_repos`) sits (logically speaking) atop the `libsvn_fs` API, providing additional functionality beyond that of the underlying versioned filesystem logic. It does not completely wrap each and every filesystem func-

[1]We understand that this may come as a shock to sci-fi fans who have long been under the impression that Time was actually the *fourth* dimension, and we apologize for any emotional trauma induced by our assertion of a different theory.

tion—only certain major steps in the general cycle of filesystem activity are wrapped by the repository interface. Some of these include the creation and commit of Subversion transactions and the modification of revision properties. These particular events are wrapped by the repository layer because they have hooks associated with them. A repository hook system is not strictly related to implementing a versioning filesystem, so it lives in the repository wrapper library.

The hooks mechanism is but one of the reasons for the abstraction of a separate repository library from the rest of the filesystem code. The `libsvn_repos` API provides several other important utilities to Subversion. These include the abilities to:

- Create, open, destroy, and perform recovery steps on a Subversion repository and the filesystem included in that repository.
- Describe the differences between two filesystem trees.
- Query for the commit log messages associated with all (or some) of the revisions in which a set of files was modified in the filesystem.
- Generate a human-readable “dump” of the filesystem—a complete representation of the revisions in the filesystem.
- Parse that dump format, loading the dumped revisions into a different Subversion repository.

As Subversion continues to evolve, the repository library will grow with the filesystem library to offer increased functionality and configurable option support.

Repository Access Layer

If the Subversion Repository layer is at “the other end of the line,” the Repository Access (RA) layer is the line itself. Charged with marshaling data between the client libraries and the repository, this layer includes the `libsvn_ra` module loader library, the RA modules themselves (which currently includes `libsvn_ra_neon`, `libsvn_ra_local`, `libsvn_ra_serf`, and `libsvn_ra_svn`), and any additional libraries needed by one or more of those RA modules (such as the `mod_dav_svn` Apache module or `libsvn_ra_svn`'s server, **svnserve**).

Since Subversion uses URLs to identify its repository resources, the protocol portion of the URL scheme (usually `file://`, `http://`, `https://`, `svn://`, or `svn+ssh://`) is used to determine which RA module will handle the communications. Each module registers a list of the protocols it knows how to “speak” so that the RA loader can, at runtime, determine which module to use for the task at hand. You can determine which RA modules are available to the Subversion command-line client, and what protocols they claim to support, by running **`svn --version`**:

```
$ svn --version
svn, version 1.7.0
   compiled Nov 15 2011, 10:10:24

Copyright (C) 2011 The Apache Software Foundation.
This software consists of contributions made by many people; see the NOTICE
file for more information.
Subversion is open source software, see http://subversion.apache.org/

The following repository access (RA) modules are available:

* ra_neon : Module for accessing a repository via WebDAV protocol using Neon.
  - handles 'http' scheme
  - handles 'https' scheme
* ra_svn : Module for accessing a repository using the svn network protocol.
  - with Cyrus SASL authentication
  - handles 'svn' scheme
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' scheme
* ra_serf : Module for accessing a repository via WebDAV protocol using serf.
  - handles 'http' scheme
```

```
  - handles 'https' scheme
$
```

The public API exported by the RA layer contains functionality necessary for sending and receiving versioned data to and from the repository. And each of the available RA plug-ins is able to perform that task using a specific protocol—`libsvn_ra_neon` and `libsvn_ra_serf` speak HTTP/WebDAV (optionally using SSL encryption) with an Apache HTTP Server that is running the `mod_dav_svn` Subversion server module; `libsvn_ra_svn` speaks a custom network protocol with the **svnserve** program; and so on.

For those who wish to access a Subversion repository using still another protocol, that is precisely why the Repository Access layer is modularized! Developers can simply write a new library that implements the RA interface on one side and communicates with the repository on the other. Your new library can use existing network protocols or you can invent your own. You could use inter-process communication (IPC) calls, or—let's get crazy, shall we?—you could even implement an email-based protocol. Subversion supplies the APIs; you supply the creativity.

Client Layer

On the client side, the Subversion working copy is where all the action takes place. The bulk of functionality implemented by the client-side libraries exists for the sole purpose of managing working copies—directories full of files and other subdirectories that serve as a sort of local, editable “reflection” of one or more repository locations—and propagating changes to and from the Repository Access layer.

Subversion's working copy library, `libsvn_wc`, is directly responsible for managing the data in the working copies. To accomplish this, the library stores administrative information about the working copy within a special subdirectory. This subdirectory, named `.svn`, is present in each working copy and contains various other files and directories that record state and provide a private workspace for administrative action. For those familiar with CVS, this `.svn` subdirectory is similar in purpose to the `CVS` administrative directories found in CVS working copies.

The Subversion client library, `libsvn_client`, has the broadest responsibility; its job is to mingle the functionality of the working copy library with that of the Repository Access layer, and then to provide the highest-level API to any application that wishes to perform general revision control actions. For example, the function `svn_client_checkout()` takes a URL as an argument. It passes this URL to the RA layer and opens an authenticated session with a particular repository. It then asks the repository for a certain tree, and sends this tree into the working copy library, which then writes a full working copy to disk (`.svn` directories and all).

The client library is designed to be used by any application. While the Subversion source code includes a standard command-line client, it should be very easy to write any number of GUI clients on top of the client library. New GUIs (or any new client, really) for Subversion need not be clunky wrappers around the included command-line client—they have full access via the `libsvn_client` API to the same functionality, data, and callback mechanisms that the command-line client uses. In fact, the Subversion source code tree contains a small C program (which you can find at `tools/examples/minimal_client.c`) that exemplifies how to wield the Subversion API to create a simple client program.

Binding Directly—A Word About Correctness

Why should your GUI program bind directly with a `libsvn_client` instead of acting as a wrapper around a command-line program? Besides simply being more efficient, it can be more correct as well. A command-line program (such as the one supplied with Subversion) that binds to the client library needs to effectively translate feedback and requested data bits from C types to some form of human-readable output. This type of translation can be lossy. That is, the program may not display all of the information harvested from the API or may combine bits of information for compact representation.

If you wrap such a command-line program with yet another program, the second program has access only to already interpreted (and as we mentioned, likely incomplete) information, which it must *again* translate into *its* representation format. With each layer of wrapping, the integrity of the original data is potentially tainted more and more, much like the result of

making a copy of a copy (of a copy…) of a favorite audio or video cassette.

But the most compelling argument for binding directly to the APIs instead of wrapping other programs is that the Subversion project makes compatibility promises regarding its APIs. Across minor versions of those APIs (such as between 1.3 and 1.4), no function's prototype will change. In other words, you aren't forced to update your program's source code simply because you've upgraded to a new version of Subversion. Certain functions might be deprecated, but they still work, and this gives you a buffer of time to eventually embrace the newer APIs. These kinds of compatibility promises do not exist for Subversion command-line program output, which is subject to change from release to release.

Using the APIs

Developing applications against the Subversion library APIs is fairly straightforward. Subversion is primarily a set of C libraries, with header (`.h`) files that live in the `subversion/include` directory of the source tree. These headers are copied into your system locations (e.g., `/usr/local/include`) when you build and install Subversion itself from source. These headers represent the entirety of the functions and types meant to be accessible by users of the Subversion libraries. The Subversion developer community is meticulous about ensuring that the public API is well documented—refer directly to the header files for that documentation.

When examining the public header files, the first thing you might notice is that Subversion's datatypes and functions are namespace-protected. That is, every public Subversion symbol name begins with `svn_`, followed by a short code for the library in which the symbol is defined (such as `wc`, `client`, `fs`, etc.), followed by a single underscore (_), and then the rest of the symbol name. Semipublic functions (used among source files of a given library but not by code outside that library, and found inside the library directories themselves) differ from this naming scheme in that instead of a single underscore after the library code, they use a double underscore (_ _). Functions that are private to a given source file have no special prefixing and are declared `static`. Of course, a compiler isn't interested in these naming conventions, but they help to clarify the scope of a given function or datatype.

Another good source of information about programming against the Subversion APIs is the project's own hacking guidelines, which you can find at http://subversion.apache.org/docs/community-guide/. This document contains useful information, which, while aimed at developers and would-be developers of Subversion itself, is equally applicable to folks developing against Subversion as a set of third-party libraries.[2]

The Apache Portable Runtime Library

Along with Subversion's own datatypes, you will see many references to datatypes that begin with `apr_`—symbols from the Apache Portable Runtime (APR) library. APR is Apache's portability library, originally carved out of its server code as an attempt to separate the OS-specific bits from the OS-independent portions of the code. The result was a library that provides a generic API for performing operations that differ mildly—or wildly—from OS to OS. While the Apache HTTP Server was obviously the first user of the APR library, the Subversion developers immediately recognized the value of using APR as well. This means that there is practically no OS-specific code in Subversion itself. Also, it means that the Subversion client compiles and runs anywhere that the Apache HTTP Server does. Currently, this list includes all flavors of Unix, Win32, BeOS, OS/2, and Mac OS X.

In addition to providing consistent implementations of system calls that differ across operating systems,[3] APR gives Subversion immediate access to many custom datatypes, such as dynamic arrays and hash tables. Subversion uses these types extensively. But perhaps the most pervasive APR datatype, found in nearly every Subversion API prototype, is the `apr_pool_t`—the APR memory pool. Subversion uses pools internally for all its memory allocation needs (unless an external library requires a different memory management mechanism for data passed through its API),[4] and while a person coding against the Subversion APIs is not required to do the same, she *is* required to provide pools to the API functions that need them. This means that users of the Subversion API must also link against APR, must call `apr_initialize()` to initialize the APR subsystem, and then must create and manage pools for use with Subversion API calls, typically by using `svn_pool_create()`, `svn_pool_clear()`, and `svn_pool_destroy()`.

[2] After all, Subversion uses Subversion's APIs, too.
[3] Subversion uses ANSI system calls and datatypes as much as possible.
[4] Neon and Berkeley DB are examples of such libraries.

Programming with Memory Pools

Almost every developer who has used the C programming language has at some point sighed at the daunting task of managing memory usage. Allocating enough memory to use, keeping track of those allocations, freeing the memory when you no longer need it—these tasks can be quite complex. And of course, failure to do those things properly can result in a program that crashes itself, or worse, crashes the computer.

Higher-level languages, on the other hand, either take the job of memory management away from you completely or make it something you toy with only when doing extremely tight program optimization. Languages such as Java and Python use *garbage collection*, allocating memory for objects when needed, and automatically freeing that memory when the object is no longer in use.

APR provides a middle-ground approach called *pool-based memory management*. It allows the developer to control memory usage at a lower resolution—per chunk (or "pool") of memory, instead of per allocated object. Rather than using `malloc()` and friends to allocate enough memory for a given object, you ask APR to allocate the memory from a memory pool. When you're finished using the objects you've created in the pool, you destroy the entire pool, effectively de-allocating the memory consumed by *all* the objects you allocated from it. Thus, rather than keeping track of individual objects that need to be de-allocated, your program simply considers the general lifetimes of those objects and allocates the objects in a pool whose lifetime (the time between the pool's creation and its deletion) matches the object's needs.

Functions and Batons

To facilitate "streamy" (asynchronous) behavior and provide consumers of the Subversion C API with hooks for handling information in customizable ways, many functions in the API accept pairs of parameters: a pointer to a callback function, and a pointer to a blob of memory called a *baton* that carries context information for that callback function. Batons are typically C structures with additional information that the callback function needs but which is not given directly to the callback function by the driving API function.

URL and Path Requirements

With remote version control operation as the whole point of Subversion's existence, it makes sense that some attention has been paid to internationalization (i18n) support. After all, while "remote" might mean "across the office," it could just as well mean "across the globe." To facilitate this, all of Subversion's public interfaces that accept path arguments expect those paths to be canonicalized—which is most easily accomplished by passing them through `svn_dirent_canonicalize()` or `svn_uri_canonicalize()` (depending on whether you are canonicalizing a local system path or a URL, respectively)—and encoded in UTF-8. This means, for example, that any new client binary that drives the `libsvn_client` interface needs to first convert paths from the locale-specific encoding to UTF-8 before passing those paths to the Subversion libraries, and then reconvert any resultant output paths from Subversion back into the locale's encoding before using those paths for non-Subversion purposes. Fortunately, Subversion provides a suite of functions (see `subversion/include/svn_utf.h`) that any program can use to do these conversions.

Also, Subversion APIs require all URL parameters to be properly URI-encoded. So, instead of passing `file:///home/username/My File.txt` as the URL of a file named `My File.txt`, you need to pass `file:///home/username/My%20File.txt`. Again, Subversion supplies helper functions that your application can use—`svn_path_uri_encode()` and `svn_path_uri_decode()`, for URI encoding and decoding, respectively.

Using Languages Other Than C and C++

If you are interested in using the Subversion libraries in conjunction with something other than a C program—say, a Python or Perl script—Subversion has some support for this via the Simplified Wrapper and Interface Generator (SWIG). The SWIG bindings for Subversion are located in `subversion/bindings/swig`. They are still maturing, but they are usable. These bindings allow you to call Subversion API functions indirectly, using wrappers that translate the datatypes native to your scripting language into the datatypes needed by Subversion's C libraries.

Significant efforts have been made toward creating functional SWIG-generated bindings for Python, Perl, and Ruby. To some extent, the work done preparing the SWIG interface files for these languages is reusable in efforts to generate bindings for other languages supported by SWIG (which include versions of C#, Guile, Java, MzScheme, OCaml, PHP, and Tcl, among others). However, some extra programming is required to compensate for complex APIs that SWIG needs some help translating between languages. For more information on SWIG itself, see the project's web site at http://www.swig.org/.

Subversion also has language bindings for Java. The javahl bindings (located in `subversion/bindings/java` in the Subversion source tree) aren't SWIG-based, but are instead a mixture of Java and hand-coded JNI. Javahl covers most Subversion client-side APIs and is specifically targeted at implementors of Java-based Subversion clients and IDE integrations.

Subversion's language bindings tend to lack the level of developer attention given to the core Subversion modules, but can generally be trusted as production-ready. A number of scripts and applications, alternative Subversion GUI clients, and other third-party tools are successfully using Subversion's language bindings today to accomplish their Subversion integrations.

It's worth noting here that there are other options for interfacing with Subversion using other languages: alternative bindings for Subversion that aren't provided by the Subversion development community at all. There are a couple of popular ones we feel are especially noteworthy. First, Barry Scott's PySVN bindings (http://pysvn.tigris.org/) are a popular option for binding with Python. PySVN boasts of a more Pythonic interface than the more C-like APIs provided by Subversion's own Python bindings. And if you're looking for a pure Java implementation of Subversion, check out SVNKit (http://svnkit.com/), which is Subversion rewritten from the ground up in Java.

SVNKit Versus javahl

In 2005, a small company called TMate announced the 1.0.0 release of JavaSVN—a pure Java implementation of Subversion. Since then, the project has been renamed to SVNKit (available at http://svnkit.com/) and has seen great success as a provider of Subversion functionality to various Subversion clients, IDE integrations, and other third-party tools.

The SVNKit library is interesting in that, unlike the javahl library, it is not merely a wrapper around the official Subversion core libraries. In fact, it shares no code with Subversion at all. But while it is easy to confuse SVNKit with javahl, and easier still to not even realize which of these libraries you are using, folks should be aware that SVNKit differs from javahl in some significant ways. First, while SVNKit is developed as open source software just like Subversion, SVNKit's license is more restrictive than that of Subversion.[5] Finally, by aiming to be a pure Java Subversion library, SVNKit is limited in which portions of Subversion can be reasonably cloned while still keeping up with Subversion's releases. This has already happened once—SVNKit cannot access BDB-backed Subversion repositories via the `file://` protocol because there's no pure Java implementation of Berkeley DB that is file-format-compatible with the native implementation of that library.

That said, SVNKit has a well-established track record of reliability. And a pure Java solution is much more robust in the face of programming errors—a bug in SVNKit might raise a catchable Java Exception, but a bug in the Subversion core libraries as accessed via javahl can bring down your entire Java Runtime Environment. So, weigh the costs when choosing a Java-based Subversion implementation.

Code Samples

Example 8.1, “Using the repository layer” contains a code segment (written in C) that illustrates some of the concepts we've been discussing. It uses both the repository and filesystem interfaces (as can be determined by the prefixes `svn_repos_` and `svn_fs_` of the function names, respectively) to create a new revision in which a directory is added. You can see the use of an APR pool, which is passed around for memory allocation purposes. Also, the code reveals a somewhat obscure fact about Subversion error handling—all Subversion errors must be explicitly handled to avoid memory leakage (and in some cases, application failure).

Example 8.1. Using the repository layer

[5]Redistributions in any form must be accompanied by information on how to obtain complete source code for the software that uses SVNKit and any accompanying software that uses the software that uses SVNKit. See http://svnkit.com/license.html for details.

```
/* Convert a Subversion error into a simple boolean error code.
 *
 * NOTE:  Subversion errors must be cleared (using svn_error_clear())
 *        because they are allocated from the global pool, else memory
 *        leaking occurs.
 */
#define INT_ERR(expr)                           \
  do {                                          \
    svn_error_t *__temperr = (expr);            \
    if (__temperr)                              \
      {                                         \
        svn_error_clear(__temperr);             \
        return 1;                               \
      }                                         \
    return 0;                                   \
  } while (0)

/* Create a new directory at the path NEW_DIRECTORY in the Subversion
 * repository located at REPOS_PATH.  Perform all memory allocation in
 * POOL.  This function will create a new revision for the addition of
 * NEW_DIRECTORY.  Return zero if the operation completes
 * successfully, nonzero otherwise.
 */
static int
make_new_directory(const char *repos_path,
                   const char *new_directory,
                   apr_pool_t *pool)
{
  svn_error_t *err;
  svn_repos_t *repos;
  svn_fs_t *fs;
  svn_revnum_t youngest_rev;
  svn_fs_txn_t *txn;
  svn_fs_root_t *txn_root;
  const char *conflict_str;

  /* Open the repository located at REPOS_PATH.
   */
  INT_ERR(svn_repos_open(&repos, repos_path, pool));

  /* Get a pointer to the filesystem object that is stored in REPOS.
   */
  fs = svn_repos_fs(repos);

  /* Ask the filesystem to tell us the youngest revision that
   * currently exists.
   */
  INT_ERR(svn_fs_youngest_rev(&youngest_rev, fs, pool));

  /* Begin a new transaction that is based on YOUNGEST_REV.  We are
   * less likely to have our later commit rejected as conflicting if we
   * always try to make our changes against a copy of the latest snapshot
   * of the filesystem tree.
   */
  INT_ERR(svn_repos_fs_begin_txn_for_commit2(&txn, repos, youngest_rev,
                                             apr_hash_make(pool), pool));

  /* Now that we have started a new Subversion transaction, get a root
   * object that represents that transaction.
   */
  INT_ERR(svn_fs_txn_root(&txn_root, txn, pool));

  /* Create our new directory under the transaction root, at the path
```

```
   * NEW_DIRECTORY.
   */
  INT_ERR(svn_fs_make_dir(txn_root, new_directory, pool));

  /* Commit the transaction, creating a new revision of the filesystem
   * which includes our added directory path.
   */
  err = svn_repos_fs_commit_txn(&conflict_str, repos,
                                &youngest_rev, txn, pool);
  if (! err)
    {
      /* No error?  Excellent!  Print a brief report of our success.
       */
      printf("Directory '%s' was successfully added as new revision "
             "'%ld'.\n", new_directory, youngest_rev);
    }
  else if (err->apr_err == SVN_ERR_FS_CONFLICT)
    {
      /* Uh-oh.  Our commit failed as the result of a conflict
       * (someone else seems to have made changes to the same area
       * of the filesystem that we tried to modify).  Print an error
       * message.
       */
      printf("A conflict occurred at path '%s' while attempting "
             "to add directory '%s' to the repository at '%s'.\n",
             conflict_str, new_directory, repos_path);
    }
  else
    {
      /* Some other error has occurred.  Print an error message.
       */
      printf("An error occurred while attempting to add directory '%s' "
             "to the repository at '%s'.\n",
             new_directory, repos_path);
    }

  INT_ERR(err);
}
```

Note that in Example 8.1, “Using the repository layer”, the code could just as easily have committed the transaction using `svn_fs_commit_txn()`. But the filesystem API knows nothing about the repository library's hook mechanism. If you want your Subversion repository to automatically perform some set of non-Subversion tasks every time you commit a transaction (e.g., sending an email that describes all the changes made in that transaction to your developer mailing list), you need to use the `libs-vn_repos`-wrapped version of that function, which adds the hook triggering functionality—in this case, `svn_repos_fs_commit_txn()`. (For more information regarding Subversion's repository hooks, see the section called “Implementing Repository Hooks”.)

Now let's switch languages. Example 8.2, “Using the repository layer with Python” is a sample program that uses Subversion's SWIG Python bindings to recursively crawl the youngest repository revision, and to print the various paths reached during the crawl.

Example 8.2. Using the repository layer with Python

```
#!/usr/bin/python

"""Crawl a repository, printing versioned object path names."""
```

```
import sys
import os.path
import svn.fs, svn.core, svn.repos

def crawl_filesystem_dir(root, directory):
    """Recursively crawl DIRECTORY under ROOT in the filesystem, and return
    a list of all the paths at or below DIRECTORY."""

    # Print the name of this path.
    print directory + "/"

    # Get the directory entries for DIRECTORY.
    entries = svn.fs.svn_fs_dir_entries(root, directory)

    # Loop over the entries.
    names = entries.keys()
    for name in names:
        # Calculate the entry's full path.
        full_path = directory + '/' + name

        # If the entry is a directory, recurse.  The recursion will return
        # a list with the entry and all its children, which we will add to
        # our running list of paths.
        if svn.fs.svn_fs_is_dir(root, full_path):
            crawl_filesystem_dir(root, full_path)
        else:
            # Else it's a file, so print its path here.
            print full_path

def crawl_youngest(repos_path):
    """Open the repository at REPOS_PATH, and recursively crawl its
    youngest revision."""

    # Open the repository at REPOS_PATH, and get a reference to its
    # versioning filesystem.
    repos_obj = svn.repos.svn_repos_open(repos_path)
    fs_obj = svn.repos.svn_repos_fs(repos_obj)

    # Query the current youngest revision.
    youngest_rev = svn.fs.svn_fs_youngest_rev(fs_obj)

    # Open a root object representing the youngest (HEAD) revision.
    root_obj = svn.fs.svn_fs_revision_root(fs_obj, youngest_rev)

    # Do the recursive crawl.
    crawl_filesystem_dir(root_obj, "")

if __name__ == "__main__":
    # Check for sane usage.
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: %s REPOS_PATH\n"
                         % (os.path.basename(sys.argv[0])))
        sys.exit(1)

    # Canonicalize the repository path.
    repos_path = svn.core.svn_dirent_canonicalize(sys.argv[1])

    # Do the real work.
    crawl_youngest(repos_path)
```

This same program in C would need to deal with APR's memory pool system. But Python handles memory usage automatically, and Subversion's Python bindings adhere to that convention. In C, you'd be working with custom datatypes (such as those provided

by the APR library) for representing the hash of entries and the list of paths, but Python has hashes (called “dictionaries”) and lists as built-in datatypes, and it provides a rich collection of functions for operating on those types. So SWIG (with the help of some customizations in Subversion's language bindings layer) takes care of mapping those custom datatypes into the native datatypes of the target language. This provides a more intuitive interface for users of that language.

The Subversion Python bindings can be used for working copy operations, too. In the previous section of this chapter, we mentioned the `libsvn_client` interface and how it exists for the sole purpose of simplifying the process of writing a Subversion client. Example 8.3, “A Python status crawler” is a brief example of how that library can be accessed via the SWIG Python bindings to re-create a scaled-down version of the **svn status** command.

Example 8.3. A Python status crawler

```
#!/usr/bin/env python

"""Crawl a working copy directory, printing status information."""

import sys
import os.path
import getopt
import svn.core, svn.client, svn.wc

def generate_status_code(status):
    """Translate a status value into a single-character status code,
    using the same logic as the Subversion command-line client."""
    code_map = { svn.wc.svn_wc_status_none        : ' ',
                 svn.wc.svn_wc_status_normal      : ' ',
                 svn.wc.svn_wc_status_added       : 'A',
                 svn.wc.svn_wc_status_missing     : '!',
                 svn.wc.svn_wc_status_incomplete  : '!',
                 svn.wc.svn_wc_status_deleted     : 'D',
                 svn.wc.svn_wc_status_replaced    : 'R',
                 svn.wc.svn_wc_status_modified    : 'M',
                 svn.wc.svn_wc_status_conflicted  : 'C',
                 svn.wc.svn_wc_status_obstructed  : '~',
                 svn.wc.svn_wc_status_ignored     : 'I',
                 svn.wc.svn_wc_status_external    : 'X',
                 svn.wc.svn_wc_status_unversioned : '?',
                }
    return code_map.get(status, '?')

def do_status(wc_path, verbose, prefix):
    # Build a client context baton.
    ctx = svn.client.svn_client_create_context()

    def _status_callback(path, status):
        """A callback function for svn_client_status."""

        # Print the path, minus the bit that overlaps with the root of
        # the status crawl
        text_status = generate_status_code(status.text_status)
        prop_status = generate_status_code(status.prop_status)
        prefix_text = ''
        if prefix is not None:
            prefix_text = prefix + " "
        print '%s%s%s  %s' % (prefix_text, text_status, prop_status, path)

    # Do the status crawl, using _status_callback() as our callback function.
    revision = svn.core.svn_opt_revision_t()
    revision.type = svn.core.svn_opt_revision_head
    svn.client.svn_client_status2(wc_path, revision, _status_callback,
```

```
                                    svn.core.svn_depth_infinity, verbose,
                                    0, 0, 1, ctx)

def usage_and_exit(errorcode):
    """Print usage message, and exit with ERRORCODE."""
    stream = errorcode and sys.stderr or sys.stdout
    stream.write("""Usage: %s OPTIONS WC-PATH

  Print working copy status, optionally with a bit of prefix text.

Options:
  --help, -h    : Show this usage message
  --prefix ARG  : Print ARG, followed by a space, before each line of output
  --verbose, -v : Show all statuses, even uninteresting ones
""" % (os.path.basename(sys.argv[0])))
    sys.exit(errorcode)

if __name__ == '__main__':
    # Parse command-line options.
    try:
        opts, args = getopt.getopt(sys.argv[1:], "hv",
                                   ["help", "prefix=", "verbose"])
    except getopt.GetoptError:
        usage_and_exit(1)
    verbose = 0
    prefix = None
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage_and_exit(0)
        if opt in ("--prefix"):
            prefix = arg
        if opt in ("-v", "--verbose"):
            verbose = 1
    if len(args) != 1:
        usage_and_exit(2)

    # Canonicalize the working copy path.
    wc_path = svn.core.svn_dirent_canonicalize(args[0])

    # Do the real work.
    try:
        do_status(wc_path, verbose, prefix)
    except svn.core.SubversionException, e:
        sys.stderr.write("Error (%d): %s\n" % (e.apr_err, e.message))
        sys.exit(1)
```

As was the case in Example 8.2, “Using the repository layer with Python”, this program is pool-free and uses, for the most part, normal Python datatypes.

Run user-provided paths through the appropriate canonicalization function (`svn_dirent_canonicalize()` or `svn_uri_canonicalize()`) before passing them to other API functions. Failure to do so can trigger assertions in the underlying Subversion C library which translate into rather immediate and unceremonious program abortion.

Of particular interest to users of the Python flavor of Subversion's API is the implementation of callback functions. As previously mentioned, Subversion's C API makes liberal use of the callback function/baton paradigm. API functions which in C accept a function and baton pair only accept a callback function parameter in Python. How, then, does the caller pass arbitrary context information to the callback function? In Python, this is done by taking advantage of Python's scoping rules and default argument values. You can see this in action in Example 8.3, “A Python status crawler”. The `svn_client_status2()` function is given a callback function (`_status_callback()`) but no baton—`_status_callback()` gets access to the user-provided prefix string

because that variable falls into the scope of the function automatically.

Summary

One of Subversion's greatest features isn't something you get from running its command-line client or other tools. It's the fact that Subversion was designed modularly and provides a stable, public API so that others—like yourself, perhaps—can write custom software that drives Subversion's core logic.

In this chapter, we took a closer look at Subversion's architecture, examining its logical layers and describing that public API, the very same API that Subversion's own layers use to communicate with each other. Many developers have found interesting uses for the Subversion API, from simple repository hook scripts, to integrations between Subversion and some other application, to completely different version control systems. What unique itch will *you* scratch with it?

Chapter 9. Subversion Complete Reference

This chapter is intended to be a complete reference to using Subversion. It includes command summaries and examples for all the command-line tools provided as part of the stock Subversion distribution, configuration information for the Subversion server modules, and other information that lends itself to a reference format.

svn—Subversion Command-Line Client

svn is the official command-line client of Subversion. Its functionality is offered via a collection of task-specific subcommands, most of which accept a number of options for fine-grained control of the program's behavior.

When using the **svn** program, subcommands and other non-option arguments must appear in a specified order on the command line. Options, on the other hand, may appear anywhere on the command line (after the program name, of course), and in general, their order is irrelevant. For example, all of the following are valid ways to use **svn status**, and are interpreted in exactly the same way:

```
$ svn -vq status myfile
$ svn status -v -q myfile
$ svn -q status -v myfile
$ svn status -vq myfile
$ svn status myfile -qv
```

The following sections describe each of the various subcommands and options provided by the **svn** command-line client program, including some examples of each subcommand's typical uses.

svn Options

While Subversion has different options for its subcommands, all options exist in a single namespace—that is, each option is guaranteed to mean the roughly same thing regardless of the subcommand you use it with. For example, `--verbose` (`-v`) always means “verbose output,” regardless of the subcommand you use it with.

The **svn** command-line client usually exits quickly with an error if you pass it an option which does not apply to the specified subcommand. But as of Subversion 1.5, several of the options which apply to all—or nearly all—of the subcommands have been deemed acceptable by all subcommands, even if they have no effect on some of them. (This change was made primarily to improve the client's ability to called from custom wrapping scripts.) These options appear grouped together in the command-line client's usage messages as global options, as can be seen in the following bit of output:

```
$ svn help upgrade
upgrade: Upgrade the metadata storage format for a working copy.
usage: upgrade [WCPATH...]

  Local modifications are preserved.

Valid options:
  -q [--quiet]             : print nothing, or only summary information

Global options:
  --username ARG           : specify a username ARG
  --password ARG           : specify a password ARG
  --no-auth-cache          : do not cache authentication tokens
  --non-interactive        : do no interactive prompting
```

```
  --trust-server-cert      : accept SSL server certificates from unknown
                             certificate authorities without prompting (but only
                             with '--non-interactive')
  --config-dir ARG         : read user configuration files from directory ARG
  --config-option ARG      : set user configuration option in the format:
                                 FILE:SECTION:OPTION=[VALUE]
                             For example:
                                 servers:global:http-library=serf
$
```

svn subcommands recognize the following global options:

`--config-dir` *DIR*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--config-option` *CONFSPEC*
: Sets, for the duration of the command, the value of a runtime configuration option. *CONFSPEC* is a string which specifies the configuration option namespace, name and value that you'd like to assign, formatted as *FILE*:*SECTION*:*OPTION*=[*VALUE*]. In this syntax, *FILE* and *SECTION* are the runtime configuration file (either `config` or `servers`) and the section thereof, respectively, which contain the option whose value you wish to change. *OPTION* is, of course, the option itself, and *VALUE* the value (if any) you wish to assign to the option. For example, to temporarily disable the use of the automatic property setting feature, use `--config-option=config:miscellany:enable-auto-props=no`. You can use this option multiple times to change multiple option values simultaneously.

`--no-auth-cache`
: Prevents caching of authentication information (e.g., username and password) in the Subversion runtime configuration directories.

`--non-interactive`
: Disables all interactive prompting. Some examples of interactive prompting include requests for authentication credentials and conflict resolution decisions. This is useful if you're running Subversion inside an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--password` *PASSWD*
: Specifies the password to use when authenticating against a Subversion server. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

`--trust-server-cert`
: When used with `--non-interactive`, instructs Subversion to accept SSL server certificates issued by unknown certificate authorities without first prompting the user. For security's sake, you should use this option only when the integrity of the remote server and the network path between it and your client is known to be trustworthy.

`--username` *NAME*
: Specifies the username to use when authenticating against a Subversion server. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

The rest of the options apply and are accepted by only a subset of the subcommand. They are as follows:

`--accept` *ACTION*
: Specifies an action for automatic conflict resolution, disabling the interactive prompts which ask the user how to handle each conflict as it is noticed. Though which of the specific actions are applicable differs depending on which subcommand is in use, Subversion supports the following long (and short) values for *ACTION*:

`postpone` (`p`)
Take no resolution action at all and instead allow the conflicts to be recorded for future resolution.

`edit` (`e`)
Open each conflicted file in a text editor for manual resolution of line-based conflicts.

`launch` (`l`)
Launch an interactive merge conflict resolution tool for each conflicted file.

`base`
Choose the file that was the (unmodified) `BASE` revision before you tried to integrate changes from the server into your working copy.

`working`
Assuming that you've manually handled the conflict resolution, choose the version of the file as it currently stands in your working copy.

`mine-full` (`mf`)
Resolve conflicted files by preserving all local modifications and discarding all changes fetched from the server during the operation which caused the conflict.

`theirs-full` (`tf`)
Resolve conflicted files by discarding all local modifications and integrating all changes fetched from the server during the operation which caused the conflict.

`mine-conflict` (`mc`)
Resolve conflicted files by preferring local modifications over the changes fetched from the server in conflicting regions of each file's content.

`theirs-conflict` (`tc`)
Resolve conflicted files by preferring the changes fetched from the server over local modifications in conflicting regions of each file's content.

Consult the output of **svn help *SUBCOMMAND*** to see exactly which actions are supported by the specific subcommand of interest.

`--allow-mixed-revisions`
Disables the verification—performed by default by **svn merge** as of Subversion 1.7—that the target of a merge operation and all of its children are at a uniform revision. While merging into a single-revision working copy target is the recommended best practice, this option may be used to permit merges into mixed-revision working copies as necessary.

`--auto-props`
Enables automatic property assignment (per runtime configuration rules), overriding the `enable-auto-props` runtime configuration directive.

`--change` (`-c`) *`ARG`*
Perform the requested operation using a specific “change”. Generally speaking, this option is syntactic sugar for **`-r ARG-1:ARG`**. Some subcommands permit a comma-separated list of revision number arguments (e.g., **`-c ARG1,ARG2,ARG3`**). Alternatively, you can provide two arguments separated by a dash (as in **`-c ARG1-ARG2`**) to identify the range of revisions between *`ARG1`* and *`ARG2`*, inclusive. Finally, if the revision argument is negated, the implied revision range is reversed: **`-c -45`** is equivalent to **`-r 45:44`**.

`--changelist` (`--cl`) *`ARG`*
Instructs Subversion to operate only on members of the changelist named *`ARG`*. You can use this option multiple times to specify sets of changelists.

`--depth` *`ARG`*
Instructs Subversion to limit the scope of an operation to a particular tree depth. *`ARG`* is one of `empty` (only the target itself), `files` (the target and any immediate file children thereof), `immediates` (the target and any immediate children thereof), or

`infinity` (the target and all of its descendants—full recursion).

`--diff`
Enables a special output mode for **svn log** which includes a difference report (a la **svn diff**) as part of each revision's information.

`--diff-cmd` *CMD*
Specifies an external program to use to show differences between files. When **svn diff** is invoked without this option, it uses Subversion's internal differencing engine, which provides unified diffs by default. If you want to use an external differencing program, use `--diff-cmd`. You can then pass options to the specified program using the `--extensions` (`-x`) option.

`--diff3-cmd` *CMD*
Specifies an external 3-way differencing program (used to merge line-based changes into files).

`--dry-run`
Goes through all the motions of running a command, but makes no actual changes—either on disk or in the repository.

`--editor-cmd` *CMD*
Specifies an external program to use to edit a log message or a property value. See the `editor-cmd` section in the section called "Config" for ways to specify a default editor.

`--encoding` *ENC*
Tells Subversion that your commit message is composed using the character encoding provided. The default character encoding is derived from your operating system's native locale; use this option if your commit message is composed using any other encoding.

`--extensions` (`-x`) *ARG*
Specifies customizations which Subversion should make when performing difference calculations. Valid extensions include:

`--ignore-space-change` (`-b`)
Ignore changes in the amount of white space.

`--ignore-all-space` (`-w`)
Ignore all white space.

`--ignore-eol-style`
Ignore changes in EOL (end-of-line) style.

`--show-c-function` (`-p`)
Show C function names in the diff output.

`--unified` (`-u`)
Show three lines of unified diff context.

The default value of *ARG* is `-u`. If you wish to pass multiple arguments, you must enclose all of them in quotes.

Note that when Subversion is configured to invoke an external diff command, the value of the `--extension` (`-x`) option isn't restricted to the previously mentioned options, but may be *any* additional arguments which Subversion should pass to that command.

`--file` (`-F`) *FILENAME*
Uses the contents of the named file for the specified subcommand. Different subcommands do different things with this content. For example, **svn commit** uses the content as a commit log message, whereas **svn propset** uses it as a property value.

`--force`
Forces a particular command or operation to run. Subversion will prevent you from performing some operations in normal usage, but you can pass this option to tell Subversion "I know what I'm doing as well as the possible repercussions of doing it, so let me at 'em." This option is the programmatic equivalent of doing your own electrical work with the power on—if you don't know what you're doing, you're likely to get a nasty shock.

`--force-log`
Forces a suspicious parameter passed to the `--message` (`-m`) or `--file` (`-F`) option to be accepted as valid. By default, Subversion will produce an error if parameters to these options look like they might instead be targets of the subcommand. For example, if you pass a versioned file's path to the `--file` (`-F`) option, Subversion will assume you've made a mistake, that the path was instead intended as the target of the operation, and that you simply failed to provide some other—unversioned—file as the source of your log message. To assert your intent and override these types of errors, pass the `--force-log` option to subcommands that accept log messages.

`--git`
Enables a special output mode for **svn diff** designed for cross-compatibility with the popular Git distributed version control system.

`--help` (`-h`, `-?`)
If used with one or more subcommands, shows the built-in help text for each. If used alone, it displays the general client help text.

`--ignore-ancestry`
Tells Subversion to ignore ancestry when calculating differences (rely on path contents alone). Also disables Merge Tracking when used with the **svn merge** subcommand.

`--ignore-externals`
Tells Subversion to ignore externals definitions and the external working copies managed by them.

`--ignore-keywords`
Disables keyword expansion.

`--ignore-whitespace`
Instructs **svn patch** to ignore whitespace when attempting to identify patch context.

`--incremental`
Prints output in a format suitable for concatenation to prior similar output.

`--internal-diff`
Instructs Subversion to use its built-in differencing engine despite any external differencing mechanism that may be specified for use in the user's runtime configuration.

`--keep-changelists`
Tells Subversion not to remove the changelist assigments from working copy items after committing.

`--keep-local`
Keeps the local copy of a file or directory (used with the **svn delete** command).

`--limit` (`-l`) *`NUM`*
Shows only the first *`NUM`* log messages.

`--message` (`-m`) *`MESSAGE`*
Indicates that you will specify either a log message or a lock comment on the command line, following this option. For example:

```
$ svn commit -m "They don't make Sunday."
```

`--native-eol` *`ARG`*
Causes **svn export** to use a specific end-of-line sequence as if it was the native sequence for the client platform. *`ARG`* may be one of `CR`, `LF`, or `CRLF`.

`--new` *ARG*
Uses *ARG* as the newer target (for use with **svn diff**).

`--no-auto-props`
Disables automatic property setting, overriding the `enable-auto-props` runtime configuration directive.

`--no-diff-deleted`
Prevents Subversion from printing differences for deleted files. The default behavior when you remove a file is for **svn diff** to print the same differences that you would see if you had kept the file but removed all of its content.

`--no-ignore`
Shows files in the status listing that would normally be omitted since they match a pattern in the `global-ignores` configuration option or the `svn:ignore` property. See the section called "Config" and the section called "Ignoring Unversioned Items" for more information.

`--no-unlock`
Tells Subversion not to automatically unlock files. (The default commit behavior is to unlock all files listed as part of the commit.) See the section called "Locking" for more information.

`--non-recursive` (`-N`)
Deprecated. Stops a subcommand from recursing into subdirectories. Most subcommands recurse by default, but some do not. Users should avoid this option and use the more precise `--depth` option instead. For most subcommands, specifying `--non-recursive` produces behavior which is the same as if you'd specified `--depth=files`, but there are exceptions: non-recursive **svn status** operates at the `immediates` depth, and the non-recursive forms of **svn revert**, **svn add**, and **svn commit** operate at an `empty` depth.

`--notice-ancestry`
Pays attention to ancestry when calculating differences.

`--old` *ARG*
Uses *ARG* as the older target (for use with **svn diff**).

`--parents`
Creates and adds nonexistent or nonversioned parent subdirectories to the working copy or repository as part of an operation. This is useful for automatically creating multiple subdirectories where none currently exist. If performed on a URL, all the directories will be created in a single commit.

`--quiet` (`-q`)
Requests that the client print only essential information while performing an operation.

`--record-only`
Enables a special mode of **svn merge** in which the specified merge operation is recorded in the local merge tracking information, but is not actually performed.

`--recursive` (`-R`)
Makes a subcommand recurse into subdirectories. (Most subcommands recurse by default.)

`--reintegrate`
Used with the **svn merge** subcommand to merge all of the source URL's changes into the working copy. See the section called "Keeping a Branch in Sync" for details.

`--relocate`
Deprecated. When used with the **svn switch** subcommand, changes the location of the repository that your working copy references. The preferred approach as of Subversion 1.7, however, is to use the **svn relocate** subcommand. See svn relocate for more details and an example.

`--remove`
Used with **svn changelist** to disassociate—rather than associate (which is the default operation)—the target(s) from a changelist.

`--reverse-diff`
Causes **svn patch** to interpret the input patch instructions in reverse—treating added lines as removed ones and vice-versa.

`--revision (-r) REV`
Specifies a revision (or range of revisions) on with which to operate. You can provide revision numbers, keywords, or dates (in curly braces) as arguments to the revision option. If you wish to offer a range of revisions, you can provide two revisions separated by a colon. For example:

```
$ svn log -r 1729
$ svn log -r 1729:HEAD
$ svn log -r 1729:1744
$ svn log -r {2001-12-04}:{2002-02-17}
$ svn log -r 1729:{2002-02-17}
```

See the section called "Revision Keywords" for more information.

`--revprop`
Operates on a revision property instead of a property specific to a file or directory. This option requires that you also pass a revision with the `--revision (-r)` option.

`--set-depth ARG`
Sets the sticky depth on a directory in a working copy to one of `exclude`, `empty`, `files`, `immediates`, or `infinity`. For detailed coverage of what these mean and how to use this option, see the section called "Sparse Directories".

`--show-copies-as-adds`
Enables a special output mode for **svn diff** in which the content difference for a file created via a copy operation appears as it would for a brand new file (with each line therein appearing as an addition to an empty file) rather than as a delta against the original file from which the copy was created.

`--show-revs ARG`
Used to make **svn mergeinfo** display certain classes of merge tracking information. *ARG* may be either `merged` or `eligible`, indicating a desire to see revisions either already merged or eligible for future merge from the specified source URL, respectively.

`--show-updates (-u)`
Causes the client to display information about which files in your working copy are out of date. This doesn't actually update any of your files—it just shows you which files will be updated if you then use **svn update**.

`--stop-on-copy`
Causes a Subversion subcommand that traverses the history of a versioned resource to stop harvesting that historical information when a copy—that is, a location in history where that resource was copied from another location in the repository—is encountered.

`--strict`
Causes Subversion to use strict semantics, a notion that is rather vague unless talking about specific subcommands (namely, **svn propget**).

`--strip NUM`
Used by **svn patch** to ignore *NUM* leading path components found on paths specified in the patch input file.

`--summarize`
Display only high-level summary notifications about the operation instead of its detailed output.

`--targets FILENAME`
Tells Subversion to read additional target paths for the operation from *FILENAME*. *FILENAME* should contain one path per

line, with each path expected to use the same encoding and formatting that it would if you had specified it directly as an argument on the command line.

`--use-merge-history` (`-g`)
Uses or displays additional information from merge history.

`--verbose` (`-v`)
Requests that the client print out as much information as it can while running any subcommand. This may result in Subversion printing out additional fields, detailed information about every file, or additional information regarding its actions.

`--version`
Prints the client version info. This information includes not only the version number of the client, but also a listing of all repository access modules that the client can use to access a Subversion repository. With `--quiet` (`-q`) it prints only the version number in a compact form.

`--with-all-revprops`
Used with the `--xml` option to **svn log**, instructs Subversion to retrieve and display all revision properties—the standard ones used internally by Subversion as well as any user-defined ones—in the log output.

`--with-no-revprops`
Used with the `--xml` option to **svn log**, instructs Subversion to omit all revision properties—including the standard log message, author, and revision datestamp—from the log output.

`--with-revprop` *`ARG`*
When used with any command that writes to the repository, sets the revision property, using the *`NAME=VALUE`* format, *`NAME`* to *`VALUE`*. When used with **svn log** in `--xml` mode, this displays the value of *`ARG`* in the log output.

`--xml`
Prints output in XML format.

svn Subcommands

Here are the various subcommands for the **svn** program. For the sake of brevity, we omit the global options (described in the section called “svn Options”) from the subcommand descriptions which follow.

Name

svn add — Add files, directories, or symbolic links.

Synopsis

```
svn add PATH...
```

Description

Schedule files, directories, or symbolic links in your working copy for addition to the repository. They will be uploaded and added to the repository on your next commit. If you add something and change your mind before committing, you can unschedule the addition using **svn revert**.

Options

```
--auto-props
--depth ARG
--force
--no-auto-props
--no-ignore
--parents
--quiet (-q)
--targets FILENAME
```

Examples

To add a file to your working copy:

```
$ svn add foo.c
A         foo.c
```

When adding a directory, the default behavior of **svn add** is to recurse:

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

You can add a directory without adding its contents:

```
$ svn add --depth=empty otherdir
A         otherdir
```

Attempts to schedule the addition of an item which is already versioned will fail by default. This behavior foils the most common scenario under which users attempt this: when trying to get to Subversion to recursively examine a versioned directory and add any unversioned items inside of it. To override the default behavior and force Subversion to recurse into already-versioned directories, pass the `--force` option:

```
$ svn add versioned-dir
svn: warning: W150002: '/home/cmpilato/projects/subversion/site' is already un\
der version control
$ svn add versioned-dir --force
A         versioned-dir/foo.c
A         versioned-dir/somedir/bar.c
A  (bin)  versioned-dir/otherdir/docs/baz.doc
…
```

Name

svn blame (praise, annotate, ann) — Show author and revision information inline for the specified files or URLs.

Synopsis

```
svn blame TARGET[@REV]...
```

Description

Show author and revision information inline for the specified files or URLs. Each line of text is annotated at the beginning with the author (username) and the revision number for the last change to that line.

Options

```
--extensions (-x) ARG
--force
--incremental
--revision (-r) REV
--use-merge-history (-g)
--verbose (-v)
--xml
```

Examples

If you want to see blame-annotated source for `readme.txt` in your test repository:

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry Don't bother reading it.  The boss is a knucklehead.
     3      sally
…
```

Now, just because **svn blame** says that Harry last modified `readme.txt` in revision 5, understand that this subcommand is by default very picky about what constitutes a change. Before clubbing Harry over the head for what appears to be insubordination, first consider that perhaps the change he made to the file might have been only to its specific character content, not to its overall semantic meaning. Perhaps his changes were the result of blindly running a whitespace cleanup script on this file. You might need to examine the specific differences and related log message to understand exactly what Harry did to this file in revision 5.

```
$ svn log -c 5 http://svn.red-bean.com/repos/test/readme.txt
------------------------------------------------------------------------
r5 | harry | 2008-05-29 07:26:12 -0600 (Thu, 29 May 2008) | 1 line

Commit the results of 'double-space-after-period.sh'.

------------------------------------------------------------------------
$ svn diff -c 5 http://svn.red-bean.com/repos/test/readme.txt
Index: http://svn.red-bean.com/repos/test/readme.txt
===================================================================
--- http://svn.red-bean.com/repos/test/readme.txt (revision 4)
```

```
+++ http://svn.red-bean.com/repos/test/readme.txt (revision 5)
@@ -1,5 +1,5 @@
 This is a README file.
-Don't bother reading it. The boss is a knucklehead.
+Don't bother reading it.  The boss is a knucklehead.

 INSTRUCTIONS
 ============
$
```

Sure enough, Harry only changed the whitespace in that line. Fortunately, the `--extensions` (`-x`) option can help you better determine the last time that a *meaningful* change was made to a given line of text. For example, here's how you can see the annotation information while disregarding mere whitespace changes:

```
$ svn blame -x -b http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     4       jess Don't bother reading it.  The boss is a knucklehead.
     3      sally
…
```

If you use the `--xml` option, you can get XML output describing the blame annotations, but not the contents of the lines themselves:

```
$ svn blame --xml http://svn.red-bean.com/repos/test/readme.txt
<?xml version="1.0"?>
<blame>
<target
   path="readme.txt">
<entry
   line-number="1">
<commit
   revision="3">
<author>sally</author>
<date>2008-05-25T19:12:31.428953Z</date>
</commit>
</entry>
<entry
   line-number="2">
<commit
   revision="5">
<author>harry</author>
<date>2008-05-29T13:26:12.293121Z</date>
</commit>
</entry>
<entry
   line-number="3">
…
</entry>
</target>
</blame>
$
```

Name

svn cat — Output the contents of the specified files or URLs.

Synopsis

```
svn cat TARGET[@REV]...
```

Description

Output the contents of the specified files or URLs. For listing the contents of directories, see **svn list** later in this chapter.

Options

`--revision (-r) REV`

Examples

If you want to view `readme.txt` in your repository without checking it out:

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
Don't bother reading it.  The boss is a knucklehead.

INSTRUCTIONS
============

Step 1:  Do this.

Step 2:  Do that.
$
```

You can view specific versions of files, too.

```
$ svn cat -r 3 http://svn.red-bean.com/repos/test/readme.txt
This is a README file.

INSTRUCTIONS
============

Step 1:  Do this.

Step 2:  Do that.
$
```

You might develop a reflex action of using **svn cat** to view your working file contents. But keep in mind that the default peg revision for **svn cat** when used on a working copy file target is BASE, the unmodified base revision of that

file. Don't be surprised when a simple **`svn cat /path/to/file`** invocation fails to display your local modifications to that file!

If your working copy is out of date (or you have local modifications) and you want to see the `HEAD` revision of a file in your working copy, use the `--revision` (`-r`) option: **`svn cat -r HEAD`** ***`FILENAME`***

Name

svn changelist (cl) — Associate (or deassociate) local paths with a changelist.

Synopsis

```
changelist CLNAME TARGET...
```

```
changelist --remove TARGET...
```

Description

Used for dividing files in a working copy into a changelist (logical named grouping) in order to allow users to easily work on multiple file collections within a single working copy.

Options

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--remove
--targets FILENAME
```

Example

Edit three files, add them to a changelist, then commit only files in that changelist:

```
$ svn changelist issue1729 foo.c bar.c baz.c
A [issue1729] foo.c
A [issue1729] bar.c
A [issue1729] baz.c
$ svn status
A       someotherfile.c
A       test/sometest.c

--- Changelist 'issue1729':
A       foo.c
A       bar.c
A       baz.c
$ svn commit --changelist issue1729 -m "Fixing Issue 1729."
Adding         bar.c
Adding         baz.c
Adding         foo.c
Transmitting file data ...
Committed revision 2.
$ svn status
A       someotherfile.c
A       test/sometest.c
$
```

Note that in the previous example, only the files in changelist `issue1729` were committed.

Name

svn checkout (co) — Check out a working copy from a repository.

Synopsis

```
svn checkout URL[@REV]... [PATH]
```

Description

Check out a working copy from a repository. If *PATH* is omitted, the basename of the URL will be used as the destination. If multiple URLs are given, each will be checked out into a subdirectory of *PATH*, with the name of the subdirectory being the basename of the URL.

Options

```
--depth ARG
--force
--ignore-externals
--quiet (-q)
--revision (-r) REV
```

Examples

Check out a working copy into a directory called `mine`:

```
$ svn checkout file:///var/svn/repos/test mine
A    mine/a
A    mine/b
A    mine/c
A    mine/d
Checked out revision 20.
$ ls
mine
$
```

Check out two different directories into two separate working copies:

```
$ svn checkout file:///var/svn/repos/test \
               file:///var/svn/repos/quiz
A    test/a
A    test/b
A    test/c
A    test/d
Checked out revision 20.
A    quiz/l
A    quiz/m
Checked out revision 13.
$ ls
quiz  test
```

```
$
```

Check out two different directories into two separate working copies, but place both into a directory called `working-copies`:

```
$ svn checkout file:///var/svn/repos/test \
               file:///var/svn/repos/quiz \
               working-copies
A    working-copies/test/a
A    working-copies/test/b
A    working-copies/test/c
A    working-copies/test/d
Checked out revision 20.
A    working-copies/quiz/l
A    working-copies/quiz/m
Checked out revision 13.
$ ls
working-copies
```

If you interrupt a checkout (or something else interrupts your checkout, such as loss of connectivity, etc.), you can restart it either by issuing the identical checkout command again or by updating the incomplete working copy:

```
$ svn checkout file:///var/svn/repos/test mine
A    mine/a
A    mine/b
^C
svn: E200015: Caught signal
$ svn checkout file:///var/svn/repos/test mine
A    mine/c
^C
svn: E200015: Caught signal
$ svn update mine
Updating 'mine':
A    mine/d
Updated to revision 20.
$
```

If you wish to check out some revision other than the most recent one, you can do so by providing the `--revision` (`-r`) option to the **svn checkout** command:

```
$ svn checkout -r 2 file:///var/svn/repos/test mine
A    mine/a
Checked out revision 2.
$
```

Prior to version 1.7, Subversion would complain by default if you try to check out a directory atop an existing directory which contains files or subdirectories that the checkout itself would have created. Subversion 1.7 handles this situation differently, allowing the checkout to proceed but marking any obstructing objects as tree conflicts. Use the `--force` option to override this safeguard. When you check out with the `--force` option, any unversioned file in the checkout target tree which ordinarily would obstruct

the checkout will still become versioned, but Subversion will preserve its contents as-is. If those contents differ from the repository file at that path (which was downloaded as part of the checkout), the file will appear to have local modifications—the changes required to transform the versioned file you checked out into the unversioned file you had before checking out—when the checkout completes.

```
$ mkdir project
$ mkdir project/lib
$ touch project/lib/file.c
$ svn checkout file:///var/svn/repos/project/trunk project --force
E    project/lib
A    project/lib/subdir
E    project/lib/file.c
A    project/lib/anotherfile.c
A    project/include/header.h
Checked out revision 21.
$ svn status wc
M       project/lib/file.c
$ svn diff wc
Index: project/lib/file.c
===================================================================
--- project/lib/file.c (revision 1)
+++ project/lib/file.c (working copy)
@@ -3 +0,0 @@
-/* file.c: Code for acting file-ishly. */
-#include <stdio.h>
-/* Not feeling particularly creative today. */

$
```

As in another other working copy, you have the choices typically available: reverting some or all of those local “modifications”, committing them, or continuing to modify your working copy.

This feature is especially useful for performing in-place imports of unversioned directory trees. By first importing the tree into the repository, and then checking out new repository location atop the unversioned tree with the `--force` option, you effectively transform the unversioned tree into a working copy.

```
$ svn mkdir -m "Create newproject project root." \
      file://var/svn/repos/newproject
$ svn import -m "Import initial newproject codebase." newproject \
      file://var/svn/repos/newproject/trunk
Adding         newproject/include
Adding         newproject/include/newproject.h
Adding         newproject/lib
Adding         newproject/lib/helpers.c
Adding         newproject/lib/base.c
Adding         newproject/notes
Adding         newproject/notes/README

Committed revision 22.
$ svn checkout file://`pwd`/repos-1.6/newproject/trunk newproject --force
E    newproject/include
E    newproject/include/newproject.h
E    newproject/lib
E    newproject/lib/helpers.c
E    newproject/lib/base.c
E    newproject/notes
E    newproject/notes/README
Checked out revision 2.
```

```
$ svn status newproject
$
```

Name

svn cleanup — Recursively clean up the working copy

Synopsis

```
svn cleanup [PATH...]
```

Description

Recursively clean up the working copy, removing working copy locks and resuming unfinished operations. If you ever get a `working copy locked` error, run this command to remove stale locks and get your working copy into a usable state again.

If, for some reason, an **svn update** fails due to a problem running an external diff program (e.g., user input or network failure), pass the `--diff3-cmd` to allow the cleanup process to complete any required merging using your external diff program. You can also specify any configuration directory with the `--config-dir` option, but you should need these options extremely infrequently.

Options

`--diff3-cmd` *CMD*

Examples

Well, there's not much to the examples here, as **svn cleanup** generates no output. If you pass no *PATH*, then "`.`" is used:

```
$ svn cleanup
$ svn cleanup /var/svn/working-copy
```

Name

svn commit (ci) — Send changes from your working copy to the repository.

Synopsis

```
svn commit [PATH...]
```

Description

Send changes from your working copy to the repository. If you do not supply a log message with your commit by using either the `--file` (`-F`) or `--message` (`-m`) option, **svn** will launch your editor for you to compose a commit message. See the `editor-cmd` list entry in the section called "Config".

svn commit will send any lock tokens that it finds and will release locks on all *PATH*s committed (recursively) unless `--no-unlock` is passed.

If you begin a commit and Subversion launches your editor to compose the commit message, you can still abort without committing your changes. If you want to cancel your commit, just quit your editor without saving your commit message and Subversion will prompt you to either abort the commit, continue with no message, or edit the message again.

Options

```
--changelist (--cl) ARG
--depth ARG
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force-log
--keep-changelists
--message (-m) MESSAGE
--no-unlock
--quiet (-q)
--targets FILENAME
--with-revprop ARG
```

Examples

Commit a simple modification to a file with the commit message on the command line and an implicit target of your current directory ("`.`"):

```
$ svn commit -m "added howto section."
Sending        a
Transmitting file data .
Committed revision 3.
```

Commit a modification to the file `foo.c` (explicitly specified on the command line) with the commit message in a file named `msg`:

```
$ svn commit -F msg foo.c
Sending        foo.c
Transmitting file data .
Committed revision 5.
```

If you want to use a file that's under version control for your commit message with `--file` (`-F`), you need to pass the `--force-log` option:

```
$ svn commit -F file_under_vc.txt foo.c
svn: E205004: Log message file is a versioned file; use '--force-log' to override

$ svn commit --force-log -F file_under_vc.txt foo.c
Sending        foo.c
Transmitting file data .
Committed revision 6.
```

To commit a file scheduled for deletion:

```
$ svn commit -m "removed file 'c'."
Deleting       c

Committed revision 7.
```

Name

svn copy (cp) — Copy a file or directory in a working copy or in the repository.

Synopsis

```
svn copy SRC[@REV]... DST
```

Description

Copy one or more files in a working copy or in the repository. *SRC* and *DST* can each be either a working copy (WC) path or URL. When copying multiple sources, add the copies as immediate children of *DST* (which, of course, must be a directory).

WC # WC
: Copy and schedule an item for addition (with history).

WC # URL
: Immediately commit a copy of WC to URL.

URL # WC
: Check out URL into WC and schedule it for addition.

URL # URL
: Complete server-side copy. This is usually used to branch and tag.

If no peg revision (i.e., *@REV*) is supplied, by default the BASE revision will be used for files copied from the working copy, while the HEAD revision will be used for files copied from a URL.

You can only copy files within a single repository. Subversion does not support cross-repository copying.

Options

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force-log
--ignore-externals
--message (-m) MESSAGE
--parents
--quiet (-q)
--revision (-r) REV
--with-revprop ARG
```

Examples

Copy an item within your working copy (this schedules the copy—nothing goes into the repository until you commit):

```
$ svn copy foo.txt bar.txt
A         bar.txt
```

```
$ svn status
A  +    bar.txt
```

Copy several files in a working copy into a subdirectory:

```
$ svn copy bat.c baz.c qux.c src
A         src/bat.c
A         src/baz.c
A         src/qux.c
```

Copy revision 8 of `bat.c` into your working copy under a different name:

```
$ svn copy -r 8 bat.c ya-old-bat.c
A         ya-old-bat.c
```

Copy an item in your working copy to a URL in the repository (this is an immediate commit, so you must supply a commit message):

```
$ svn copy near.txt file:///var/svn/repos/test/far-away.txt -m "Remote copy."

Committed revision 8.
```

Copy an item from the repository to your working copy (this just schedules the copy—nothing goes into the repository until you commit):

```
$ svn copy file:///var/svn/repos/test/far-away -r 6 near-here
A         near-here
```

This is the recommended way to resurrect a dead file in your repository!

And finally, copy between two URLs:

```
$ svn copy file:///var/svn/repos/test/far-away \
           file:///var/svn/repos/test/over-there -m "remote copy."

Committed revision 9.
```

```
$ svn copy file:///var/svn/repos/test/trunk \
           file:///var/svn/repos/test/tags/0.6.32-prerelease -m "tag tree"

Committed revision 12.
```

This is the easiest way to “tag” a revision in your repository—just **svn copy** that revision (usually `HEAD`) into your `tags` directory.

And don't worry if you forgot to tag—you can always specify an older revision and tag anytime:

```
$ svn copy -r 11 file:///var/svn/repos/test/trunk \
           file:///var/svn/repos/test/tags/0.6.32-prerelease \
           -m "Forgot to tag at rev 11"

Committed revision 13.
```

Name

svn delete (del, remove, rm) — Delete an item from a working copy or the repository.

Synopsis

```
svn delete PATH...
```

```
svn delete URL...
```

Description

Items specified by *PATH* are scheduled for deletion upon the next commit. Files (and directories that have not been committed) are immediately removed from the working copy unless the `--keep-local` option is given. The command will not remove any unversioned or modified items; use the `--force` option to override this behavior.

Items specified by URL are deleted from the repository via an immediate commit. Multiple URLs are committed atomically.

Options

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--keep-local
--message (-m) MESSAGE
--quiet (-q)
--targets FILENAME
--with-revprop ARG
```

Examples

Using **svn** to delete a file from your working copy deletes your local copy of the file, but it merely schedules the file to be deleted from the repository. When you commit, the file is deleted in the repository.

```
$ svn delete myfile
D         myfile

$ svn commit -m "Deleted file 'myfile'."
Deleting       myfile
Transmitting file data .
Committed revision 14.
```

Deleting a URL, however, is immediate, so you have to supply a log message:

```
$ svn delete -m "Deleting file 'yourfile'" \
             file:///var/svn/repos/test/yourfile

Committed revision 15.
```

Here's an example of how to force deletion of a file that has local mods:

```
$ svn delete over-there
svn: E195006: Use --force to override this restriction (local modifications m\
ay be lost)
svn: E195006: '/home/sally/project/over-there' has local modifications -- com\
mit or revert them first
$ svn delete --force over-there
D         over-there
$
```

Use the `--keep-local` option to override the default **svn delete** behavior of also removing the target file that was scheduled for versioned deletion. This is helpful when you realize that you've accidentally committed the addition of a file that you need to keep around in your working copy, but which shouldn't have been added to version control.

```
$ svn delete --keep-local conf/program.conf
D         conf/program.conf

$ svn commit -m "Remove accidentally-added configuration file."
Deleting       conf/program.conf
Transmitting file data .
Committed revision 21.
$ svn status
?       conf/program.conf
$
```

Name

svn diff (di) — This displays the differences between two revisions or paths.

Synopsis

```
diff [-c M | -r N[:M]] [TARGET[@REV]...]

diff [-r N[:M]] --old=OLD-TGT[@OLDREV] [--new=NEW-TGT[@NEWREV]] [PATH...]

diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

Description

Display the differences between two paths. You can use **svn diff** in the following ways:

- Use just **svn diff** to display local modifications in a working copy.
- Display the changes made to *TARGET*s as they are seen in *REV* between two revisions. *TARGET*s may be all working copy paths or all *URL*s. If *TARGET*s are working copy paths, *N* defaults to BASE and *M* to the working copy; if *TARGET*s are *URL*s, *N* must be specified and *M* defaults to HEAD. The `-c M` option is equivalent to `-r N:M` where `N = M-1`. Using `-c -M` does the reverse: `-r M:N` where `N = M-1`.
- Display the differences between *OLD-TGT* as it was seen in *OLDREV* and *NEW-TGT* as it was seen in *NEWREV*. *PATH*s, if given, are relative to *OLD-TGT* and *NEW-TGT* and restrict the output to differences for those paths. *OLD-TGT* and *NEW-TGT* may be working copy paths or *URL[@REV]*. *NEW-TGT* defaults to *OLD-TGT* if not specified. `-r N` makes *OLDREV* default to N; `-r N:M` makes *OLDREV* default to *N* and *NEWREV* default to *M*.

svn diff OLD-URL[@OLDREV] NEW-URL[@NEWREV] is shorthand for **svn diff --old=OLD-URL[@OLDREV] --new=NEW-URL[@NEWREV].**

svn diff -r N:M URL is shorthand for **svn diff -r N:M --old=URL --new=URL**.

svn diff [-r N[:M]] URL1[@N] URL2[@M] is shorthand for **svn diff [-r N[:M]] --old=URL1 --new=URL2**.

If *TARGET* is a URL, then revs N and M can be given either via the `--revision` (`-r`) option or by using the "@" notation as described earlier.

If *TARGET* is a working copy path, the default behavior (when no `--revision` (`-r`) option is provided) is to display the differences between the base and working copies of *TARGET*. If a `--revision` (`-r`) option is specified in this scenario, though, it means:

`--revision N:M`
: The server compares *TARGET@N* and *TARGET@M*.

`--revision N`
: The client compares *TARGET@N* against the working copy.

If the alternate syntax is used, the server compares *URL1* and *URL2* at revisions *N* and *M*, respectively. If either *N* or *M* is omitted, a value of HEAD is assumed.

By default, **svn diff** ignores the ancestry of files and merely compares the contents of the two files being compared. If you use `--notice-ancestry`, the ancestry of the paths in question will be taken into consideration when comparing revisions (i.e., if

you run **svn diff** on two files with identical contents but different ancestry, you will see the entire contents of the file as having been removed and added again).

Options

```
--change (-c) ARG
--changelist (--cl) ARG
--depth ARG
--diff-cmd CMD
--extensions (-x) ARG
--force
--git
--internal-diff
--new ARG
--no-diff-deleted
--notice-ancestry
--old ARG
--revision (-r) REV
--show-copies-as-adds
--summarize
--xml
```

Examples

Compare BASE and your working copy (one of the most popular uses of **svn diff**):

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
…
```

See what changed in the file COMMITTERS revision 9115:

```
$ svn diff -c 9115 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
…
```

See how your working copy's modifications compare against an older revision:

```
$ svn diff -r 3900 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

…

Compare revision 3000 to revision 3500 using “@” syntax:

```
$ svn diff http://svn.collab.net/repos/svn/trunk/COMMITTERS@3000 \
           http://svn.collab.net/repos/svn/trunk/COMMITTERS@3500
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

Compare revision 3000 to revision 3500 using range notation (pass only the one URL in this case):

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

Compare revision 3000 to revision 3500 of all the files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk
```

Compare revision 3000 to revision 3500 of only three files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 --old http://svn.collab.net/repos/svn/trunk \
           COMMITTERS README HACKING
```

If you have a working copy, you can obtain the differences without typing in the long URLs:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

Use `--diff-cmd` *`CMD`* `--extensions` (`-x`) to pass arguments directly to the external diff program:

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
> This is a test
>
$
```

Lastly, you can use the `--xml` option along with the `--summarize` option to view XML describing the changes that occurred between revisions, but not the contents of the diff itself:

```
$ svn diff --summarize --xml http://svn.red-bean.com/repos/test@r2 \
           http://svn.red-bean.com/repos/test
<?xml version="1.0"?>
<diff>
<paths>
<path
   props="none"
   kind="file"
   item="modified">http://svn.red-bean.com/repos/test/sandwich.txt</path>
<path
   props="none"
   kind="file"
   item="deleted">http://svn.red-bean.com/repos/test/burrito.txt</path>
<path
   props="none"
   kind="dir"
   item="added">http://svn.red-bean.com/repos/test/snacks</path>
</paths>
</diff>
```

Name

svn export — Export a clean directory tree.

Synopsis

```
svn export [-r REV] URL[@PEGREV] [PATH]
```

```
svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

Description

The first form exports a clean directory tree from the repository specified by *URL*—at revision *REV* if it is given; otherwise, at HEAD, into *PATH*. If *PATH* is omitted, the last component of the *URL* is used for the local directory name.

The second form exports a clean directory tree from the working copy specified by *PATH1* into *PATH2*. All local changes will be preserved, but files not under version control will not be copied.

Options

```
--depth ARG
--force
--ignore-externals
--ignore-keywords
--native-eol ARG
--quiet (-q)
--revision (-r) REV
```

Examples

Export from your working copy (doesn't print every file and directory):

```
$ svn export a-wc my-export
Export complete.
```

Export directly from the repository (prints every file and directory):

```
$ svn export file:///var/svn/repos my-export
A    my-export/test
A    my-export/quiz
…
Exported revision 15.
```

When rolling operating-system-specific release packages, it can be useful to export a tree that uses a specific EOL character for line endings. The `--native-eol` option will do this, but it affects only files that have `svn:eol-style = native` properties attached to them. For example, to export a tree with all CRLF line endings (possibly for a Windows `.zip` file distribution):

```
$ svn export file:///var/svn/repos my-export --native-eol CRLF
A    my-export/test
A    my-export/quiz
…
Exported revision 15.
```

You can specify `LR`, `CR`, or `CRLF` as a line-ending type with the `--native-eol` option.

Name

svn help (h, ?) — Help!

Synopsis

```
svn help [SUBCOMMAND...]
```

Description

This is your best friend when you're using Subversion and this book isn't within reach!

Options

None

Name

svn import — Commit an unversioned file or tree into the repository.

Synopsis

```
svn import [PATH] URL
```

Description

Recursively commit a copy of *PATH* to *URL*. If *PATH* is omitted, "`.`" is assumed. Parent directories are created in the repository as necessary. Unversionable items such as device files and pipes are ignored even if `--force` is specified.

Options

```
--auto-props
--depth ARG
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--message (-m) MESSAGE
--no-auto-props
--no-ignore
--quiet (-q)
--with-revprop ARG
```

Examples

This imports the local directory `myproj` into `trunk/misc` in your repository. The directory `trunk/misc` need not exist before you import into it—**svn import** will recursively create directories for you.

```
$ svn import -m "New import" myproj \
             http://svn.red-bean.com/repos/trunk/misc
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

Be aware that this will *not* create a directory named `myproj` in the repository. If that's what you want, simply add `myproj` to the end of the URL:

```
$ svn import -m "New import" myproj \
            http://svn.red-bean.com/repos/trunk/misc/myproj
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

After importing data, note that the original tree is *not* under version control. To start working, you still need to **svn checkout** a fresh working copy of the tree.

Name

svn info — Display information about a local or remote item.

Synopsis

```
svn info [TARGET[@REV]...]
```

Description

Print information about the working copy paths or URLs specified. The information displayed for each path may include (as pertinent to the object at that path):

- information about the repository in which the object is versioned
- the most recent commit made to the specified version of the object
- any user-level locks held on the object
- local scheduling information (added, deleted, copied, etc.)
- local conflict information

Options

```
--changelist (--cl) ARG
--depth ARG
--incremental
--recursive (-R)
--revision (-r) REV
--targets FILENAME
--xml
```

Examples

svn info will show you all the useful information that it has for items in your working copy. It will show information for files:

```
$ svn info foo.c
Path: foo.c
Name: foo.c
Working Copy Root Path: /home/sally/projects/test
URL: http://svn.red-bean.com/repos/test/foo.c
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 4417
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 20
Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)
Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan 2003)
```

```
Checksum: d6aeb60b0662ccceb6bce4bac344cb66
```

It will also show information for directories:

```
$ svn info vendors
Path: vendors
Working Copy Root Path: /home/sally/projects/test
URL: http://svn.red-bean.com/repos/test/vendors
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 19
Node Kind: directory
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 19
Last Changed Date: 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-16 23:39:02 -0600 (Thu, 16 Jan 2003)
```

svn info also acts on URLs (also note that the file `readme.doc` in this example is locked, so lock information is also provided):

```
$ svn info http://svn.red-bean.com/repos/test/readme.doc
Path: readme.doc
Name: readme.doc
URL: http://svn.red-bean.com/repos/test/readme.doc
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 1
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 42
Last Changed Date: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Lock Token: opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2
Lock Owner: harry
Lock Created: 2003-01-15 17:35:12 -0600 (Wed, 15 Jan 2003)
Lock Comment (1 line):
My test lock comment
```

Lastly, **svn info** output is available in XML format by passing the `--xml` option:

```
$ svn info --xml http://svn.red-bean.com/repos/test
<?xml version="1.0"?>
<info>
<entry
   kind="dir"
   path="."
   revision="1">
<url>http://svn.red-bean.com/repos/test</url>
<repository>
<root>http://svn.red-bean.com/repos/test</root>
```

```
<uuid>5e7d134a-54fb-0310-bd04-b611643e5c25</uuid>
</repository>
<wc-info>
<schedule>normal</schedule>
<depth>infinity</depth>
</wc-info>
<commit
   revision="1">
<author>sally</author>
<date>2003-01-15T23:35:12.847647Z</date>
</commit>
</entry>
</info>
```

Name

svn list (ls) — List directory entries in the repository.

Synopsis

```
svn list [TARGET[@REV]...]
```

Description

List each *TARGET* file and the contents of each *TARGET* directory as they exist in the repository. If *TARGET* is a working copy path, the corresponding repository URL will be used.

The default *TARGET* is "`.`", meaning the repository URL of the current working copy directory.

With `--verbose` (`-v`), **svn list** shows the following fields for each item:

- Revision number of the last commit
- Author of the last commit
- If locked, the letter "O" (see the preceding section on svn info for details).
- Size (in bytes)
- Date and time of the last commit

With `--xml`, output is in XML format (with a header and an enclosing document element unless `--incremental` is also specified). All of the information is present; the `--verbose` (`-v`) option is not accepted.

Options

```
--depth ARG
--incremental
--recursive (-R)
--revision (-r) REV
--verbose (-v)
--xml
```

Examples

svn list is most useful if you want to see what files a repository has without downloading a working copy:

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
...
```

You can pass the `--verbose` (`-v`) option for additional information, rather like the Unix command **ls -l**:

```
$ svn list -v file:///var/svn/repos
     16 sally         28361 Jan 16 23:18 README.txt
     27 sally             0 Jan 18 15:27 INSTALL
     24 harry               Jan 18 11:27 examples/
```

You can also get **svn list** output in XML format with the `--xml` option:

```
$ svn list --xml http://svn.red-bean.com/repos/test
<?xml version="1.0"?>
<lists>
<list
   path="http://svn.red-bean.com/repos/test">
<entry
   kind="dir">
<name>examples</name>
<size>0</size>
<commit
   revision="24">
<author>harry</author>
<date>2008-01-18T06:35:53.048870Z</date>
</commit>
</entry>
...
</list>
</lists>
```

For further details, see the earlier section the section called “Listing versioned directories”.

Name

svn lock — Lock working copy paths or URLs in the repository so that no other user can commit changes to them.

Synopsis

```
svn lock TARGET...
```

Description

Lock each *TARGET*. If any *TARGET* is already locked by another user, print a warning and continue locking the rest of the *TARGET*s. Use `--force` to steal a lock from another user or working copy.

Options

```
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--message (-m) MESSAGE
--targets FILENAME
```

Examples

Lock two files in your working copy:

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user 'harry'.
'house.jpg' locked by user 'harry'.
```

Lock a file in your working copy that is currently locked by another user:

```
$ svn lock tree.jpg
svn: warning: W160035: Path '/tree.jpg is already locked by user 'sally' in fi
lesystem '/var/svn/repos/db'
$ svn lock --force tree.jpg
'tree.jpg' locked by user 'harry'.
```

Lock a file without a working copy:

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user 'harry'.
```

For further details, see the section called “Locking”.

Name

svn log — Display commit log messages.

Synopsis

```
svn log [PATH]
```

```
svn log URL[@REV] [PATH...]
```

Description

Shows log messages from the repository. If no arguments are supplied, **svn log** shows the log messages for all files and directories inside (and including) the current working directory of your working copy. You can refine the results by specifying a path, one or more revisions, or any combination of the two. The default revision range for a local path is `BASE:1`.

If you specify a URL alone, it prints log messages for everything the URL contains. If you add paths past the URL, only messages for those paths under that URL will be printed. The default revision range for a URL is `HEAD:1`.

With `--verbose` (`-v`), **svn log** will also print all affected paths with each log message. With `--quiet` (`-q`), **svn log** will not print the log message body itself, this is compatible with `--verbose` (`-v`).

Each log message is printed just once, even if more than one of the affected paths for that revision were explicitly requested. Logs follow copy history by default. Use `--stop-on-copy` to disable this behavior, which can be useful for determining branch points.

Options

```
--change (-c) ARG
--depth ARG
--diff
--diff-cmd CMD
--extensions (-x) ARG
--incremental
--internal-diff
--limit (-l) NUM
--quiet (-q)
--revision (-r) REV
--stop-on-copy
--targets FILENAME
--use-merge-history (-g)
--verbose (-v)
--with-all-revprops
--with-no-revprops
--with-revprop ARG
--xml
```

Examples

You can see the log messages for all the paths that changed in your working copy by running **`svn log`** from the top:

```
$ svn log
------------------------------------------------------------------------
r20 | harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line
```

```
Tweak.
------------------------------------------------------------------------
r17 | sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines
…
```

Examine all log messages for a particular file in your working copy:

```
$ svn log foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

If you don't have a working copy handy, you can log a URL:

```
$ svn log http://svn.red-bean.com/repos/test/foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

If you want several distinct paths underneath the same URL, you can use the `URL [PATH...]` syntax:

```
$ svn log http://svn.red-bean.com/repos/test/ foo.c bar.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r31 | harry | 2003-01-10 12:25:08 -0600 (Fri, 10 Jan 2003) | 1 line

Added new file bar.c
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

The `--verbose` (`-v`) option causes **svn log** to include information about the paths that were changed in each displayed revision. These paths appear, one path per line of output, with action codes that indicate what type of change was made to the path.

```
$ svn log -v http://svn.red-bean.com/repos/test/ foo.c bar.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line
Changed paths:
   M /foo.c

Added defines.
------------------------------------------------------------------------
r31 | harry | 2003-01-10 12:25:08 -0600 (Fri, 10 Jan 2003) | 1 line
Changed paths:
   A /bar.c

Added new file bar.c
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

svn log uses just a handful of action codes, and they are similar to the ones the **svn update** command uses:

`A`
: The item was added.

`D`
: The item was deleted.

`M`
: Properties or textual contents on the item were changed.

`R`
: The item was replaced by a different one at the same location.

In addition to the action codes which precede the changed paths, **svn log** with the `--verbose` (`-v`) option will note whether a path was added or replaced as the result of a copy operation. It does so by printing (`from` *`COPY-FROM-PATH`*`:`*`COPY-FROM-REV`*) after such paths.

When you're concatenating the results of multiple calls to the log command, you may want to use the `--incremental` option. **svn log** normally prints out a dashed line at the beginning of a log message, after each subsequent log message, and following the final log message. If you ran **svn log** on a range of two revisions, you would get this:

```
$ svn log -r 14:15
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r15 | …

------------------------------------------------------------------------
```

However, if you wanted to gather two nonsequential log messages into a file, you might do something like this:

```
$ svn log -r 14 > mylog
```

```
$ svn log -r 19 >> mylog
$ svn log -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
------------------------------------------------------------------------
r27 | …

------------------------------------------------------------------------
```

You can avoid the clutter of the double dashed lines in your output by using the `--incremental` option:

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | …

------------------------------------------------------------------------
r19 | …

------------------------------------------------------------------------
r27 | …
```

The `--incremental` option provides similar output control when using the `--xml` option:

```
$ svn log --xml --incremental -r 1 sandwich.txt
<logentry
   revision="1">
<author>harry</author>
<date>2008-06-03T06:35:53.048870Z</date>
<msg>Initial Import.</msg>
</logentry>
```

Sometimes when you run **svn log** on a specific path and a specific revision, you see no log information output at all, as in the following:

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
------------------------------------------------------------------------
```

That just means the path wasn't modified in that revision. To get log information for that revision, either run the log operation against the repository's root URL, or specify a path that you happen to know was changed in that revision:

```
$ svn log -r 20 touched.txt
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.
------------------------------------------------------------------------
```

Beginning with Subversion 1.7, users can take advantage of a special output mode which combines the information from **svn log** with what you would see when running **svn diff** on the same targets for each revision of the log. Simply invoke **svn log** with the `--diff` option to trigger this output mode.

```
$ svn log -r 20 touched.txt --diff
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.

Index: touched.txt
===================================================================
--- touched.txt (revision 19)
+++ touched.txt (revision 20)
@@ -1 +1,2 @@
 This is the file 'touched.txt'.
+We add such exciting text to files around here!
------------------------------------------------------------------------
$
```

As with **svn diff**, you may also make use of many of the various options which control the way the difference is generated, including `--depth`, `--diff-cmd`, and `--extensions` (`-x`).

Name

svn merge — Apply the differences between two sources to a working copy path.

Synopsis

```
svn merge [-c M[,N...] | -r N:M ...] SOURCE[@REV] [TARGET_WCPATH]
```

```
svn merge --reintegrate SOURCE[@REV] [TARGET_WCPATH]
```

```
svn merge SOURCE1[@N] SOURCE2[@M] [TARGET_WCPATH]
```

Description

In all three forms *TARGET_WCPATH* is the working copy path that will receive the differences. If *TARGET_WCPATH* is omitted, the changes are applied to the current working directory, unless the sources have identical basenames that match a file within the current working directory. In this case, the differences will be applied to that file.

In the first two forms, *SOURCE* can be either a URL or a working copy path (in which case its corresponding URL is used). If the peg revision *REV* is not specified, then HEAD is assumed. In the third form the same rules apply for *SOURCE1*, *SOURCE2*, *M*, and *N* with the only difference being that if either source is a working copy path, then the peg revisions *must* be explicitly stated.

- Sync and Cherrypick Merges

  The first form, when used without either the `-c` or `-r` options, is called a “sync” merge and `-r 1:REV` is implied. This variant is used to merge all eligible changes to a branch from its immediate ancestor branch, see the section called “Keeping a Branch in Sync”.

  When the first form is used with the `-c` or `-r` options, this is called a “cherrypick” merge and is used to merge an explicitly defined set of changes from one branch to another, see the section called “Cherrypicking”

  Multiple `-c` and/or `-r` instances may be specified, and mixing of forward and reverse ranges is allowed— the ranges are internally compacted to their minimum representation before merging begins (which may result in a no-op merge or conflicts that cause the merge to stop before merging all of the requested revisions).

  In both variants of the first form, *SOURCE* in revision *REV* is compared as it existed between revisions *N* and *M* for each revision range provided.

- Reintegrate Merges

  The second form is called a “reintegrate merge” and is used to bring changes from a feature branch (*SOURCE*) back into the feature branch's immediate ancestor branch (*TARGET_WCPATH*).

  Reintegrate merges support only this specialized use case and as such have a number of special requirements and limitations that the other two merge forms do not posses. See the section called “Keeping a Branch in Sync”, the section called “Reintegrating a Branch”, the section called “Keeping a Reintegrated Branch Alive”, and the section called “Feature Branches”.

- 2-URL Merges

  In the third form, called a “2-URL Merge”, the difference between *SOURCE1* at revision *N* and *SOURCE2* at revision *M* is generated and applied to *TARGET_WCPATH*. The revisions default to HEAD if omitted.

If Merge Tracking is active, then Subversion will internally track metadata (i.e. the `svn:mergeinfo` property) about merge operations when the two merge sources are ancestrally related—if the first source is an ancestor of the second or vice versa—this is guaranteed to be the case when using the first two forms. Subversion will also take preexisting merge metadata on the working copy target into account when determining what revisions to merge and in an effort to avoid repeat merges and needless conflicts it may only merge a subset of the requested ranges.

Merge Tracking can be disabled by using the `--ignore-ancestry` option.

Unlike **svn diff**, the merge command takes the ancestry of a file into consideration when performing a merge operation. This is very important when you're merging changes from one branch into another and you've renamed a file on one branch but not the other.

Options

```
--accept ACTION
--allow-mixed-revisions
--change (-c) ARG
--depth ARG
--diff3-cmd CMD
--dry-run
--extensions (-x) ARG
--force
--ignore-ancestry
--quiet (-q)
--record-only
--reintegrate
--revision (-r) REV
```

Examples

Merge a branch back into the trunk (assuming that you have an up-to-date working copy of the trunk):

```
$ svn merge --reintegrate \
            http://svn.example.com/repos/calc/branches/my-calc-branch
--- Merging differences between repository URLs into '.':
U    button.c
U    integer.c
U    Makefile
 U   .
--- Recording mergeinfo for merge between repository URLs into '.':
 U   .

$ # build, test, verify, ...

$ svn commit -m "Merge my-calc-branch back into trunk!"
Sending        .
Sending        button.c
Sending        integer.c
Sending        Makefile
Transmitting file data ..
Committed revision 391.
```

To merge changes to a single file:

```
$ svn merge -c 31 ^/trunk/thhgttg.txt thhgttg.txt
--- Merging r31 into 'thhgttg.txt':
U    thhgttg.txt
--- Recording mergeinfo for merge of r31 into 'thhgttg.txt':
 U   thhgttg.txt
```

Name

svn mergeinfo — Query merge-related information. See the section called “Mergeinfo and Previews” for details.

Synopsis

```
svn mergeinfo SOURCE_URL[@REV] [TARGET[@REV]]
```

Description

Query information related to merges (or potential merges) between *SOURCE-URL* and *TARGET*. If the `--show-revs` option is not provided, display revisions which have been merged from *SOURCE-URL* to *TARGET*. Otherwise, display either `merged` or `eligible` revisions as specified by the `--show-revs` option.

Options

```
--depth ARG
--recursive (-R)
--revision (-r) REV
--show-revs ARG
```

Examples

Find out which changesets your have been merged from your trunk directory into your test branch:

```
$ svn propget svn:mergeinfo ^/branches/test
/branches/other:3-4
/trunk:11-13,14,16
$ svn mergeinfo --show-revs merged ^/trunk ^/branches/test
r11
r12
r13
r14
r16
$
```

Note that the default output from the **svn mergeinfo** command is to display merged revisions, so the `--show-revs` option shown in the command line of the previous example is not strictly required.

Find out which changesets from your trunk directory have not yet been merged into your test branch:

```
$ svn mergeinfo --show-revs eligible ^/trunk ^/branches/test
r15
r17
r20
r21
r22
$
```

Name

svn mkdir — Create a new directory under version control.

Synopsis

```
svn mkdir PATH...
```

```
svn mkdir URL...
```

Description

Create a directory with a name given by the final component of the *PATH* or *URL*. A directory specified by a working copy *PATH* is scheduled for addition in the working copy. A directory specified by a URL is created in the repository via an immediate commit. Multiple directory URLs are committed atomically. In both cases, all the intermediate directories must already exist unless the `--parents` option is used.

Options

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force-log
--message (-m) MESSAGE
--parents
--quiet (-q)
--with-revprop ARG
```

Examples

Create a directory in your working copy:

```
$ svn mkdir newdir
A         newdir
```

Create one in the repository (this is an instant commit, so a log message is required):

```
$ svn mkdir -m "Making a new dir." http://svn.red-bean.com/repos/newdir
```

```
Committed revision 26.
```

Name

svn move (mv) — Move a file or directory.

Synopsis

```
svn move SRC... DST
```

Description

This command moves files or directories in your working copy or in the repository.

This command is equivalent to an **svn copy** followed by **svn delete**.

When moving multiple sources, they will be added as children of *DST*, which must be a directory.

Subversion does not support moving between working copies and URLs. In addition, you can only move files within a single repository—Subversion does not support cross-repository moving. Subversion supports the following types of moves within a single repository:

WC # WC
: Move and schedule a file or directory for addition (with history).

URL # URL
: Complete server-side rename.

Options

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--message (-m) MESSAGE
--parents
--quiet (-q)
--revision (-r) REV
--with-revprop ARG
```

Examples

Move a file in your working copy:

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

Move several files in your working copy into a subdirectory:

```
$ svn move baz.c bat.c qux.c src
A         src/baz.c
D         baz.c
A         src/bat.c
D         bat.c
A         src/qux.c
D         qux.c
```

Move a file in the repository (this is an immediate commit, so it requires a commit message):

```
$ svn move -m "Move a file" http://svn.red-bean.com/repos/foo.c \
                            http://svn.red-bean.com/repos/bar.c

Committed revision 27.
```

Name

svn patch — Apply changes represented in a unidiff patch to the working copy.

Synopsis

```
svn patch PATCHFILE [WCPATH]
```

Description

This subcommand will apply changes described a unidiff-formatted patch file *PATCHFILE* to the working copy *WCPATH*. As with most other working copy subcommands, if *WCPATH* is omitted, the changes are applied to the current working directory. A unidiff patch suitable for application to a working copy can be produced with the **svn diff** command or third-party differencing tools. Any non-unidiff content found in the patch file is ignored.

Changes listed in the patch file will either be applied or rejected. If a change does not match at its exact line offset, it may be applied earlier or later in the file if a match is found elsewhere for the surrounding lines of context provided by the patch. A change may also be applied with *fuzz*—meaning, one or more lines of context are ignored when attempting to match the change location. If no matching context can be found for a change, the change conflicts and will be written to a reject file which bears the extension `.svnpatch.rej`.

svn patch reports a status line for patched file or directory using letter codes, very similar to the way that **svn update** provides notification. The letter codes have the following meanings:

`A`
: Added

`D`
: Deleted

`C`
: Conflicted

`G`
: Merged

`U`
: Updated

Changes applied with an offset or fuzz are reported on lines starting with the '>' symbol. You should review such changes carefully.

If the patch removes all content from a file, that file is automatically scheduled for deletion. Likewise, if the patch creates a new file, that file is automatically scheduled for addition. Use **svn revert** to undo undesired deletions and additions.

Options

```
--dry-run
--ignore-whitespace
--quiet (-q)
--reverse-diff
--strip NUM
```

Examples

Apply a simple patch file generated by the **svn diff** command. Our patch file will create a new file, delete another file, and modify a third's contents and properties. Here's the patch file itself (which we'll assume is creatively named `PATCH`):

```
Index: deleted-file
===================================================================
--- deleted-file (revision 3)
+++ deleted-file (working copy)
@@ -1 +0,0 @@
-This file will be deleted.
Index: changed-file
===================================================================
--- changed-file (revision 4)
+++ changed-file (working copy)
@@ -1,6 +1,6 @@
 The letters in a line of text
 Could make your day much better.
 But expanded into paragraphs,
-I'd tell of kangaroos and calves
+I'd tell of monkeys and giraffes
 Until you were all smiles and laughs
 From my letter made of letters.

Property changes on: changed-file
___________________________________________________________________
Added: propname
## -0,0 +1 ##
+propvalue
Index: added-file
===================================================================
--- added-file (revision 0)
+++ added-file (working copy)
@@ -0,0 +1 @@
+This is an added file.
```

We can apply the previous patch file to another working copy from our repository using **svn patch**, and verify that it did the right thing by using **svn diff**:

```
$ cd /some/other/workingcopy
$ svn patch /path/to/PATCH
D         deleted-file
UU        changed-file
A         added-file
$ svn diff
Index: deleted-file
===================================================================
--- deleted-file (revision 3)
+++ deleted-file (working copy)
@@ -1 +0,0 @@
-This file will be deleted.
Index: changed-file
===================================================================
--- changed-file (revision 4)
+++ changed-file (working copy)
@@ -1,6 +1,6 @@
 The letters in a line of text
 Could make your day much better.
```

```
 But expanded into paragraphs,
-I'd tell of kangaroos and calves
+I'd tell of monkeys and giraffes
 Until you were all smiles and laughs
 From my letter made of letters.

Property changes on: changed-file
___________________________________________________________________
Added: propname
## -0,0 +1 ##
+propvalue
Index: added-file
===================================================================
--- added-file (revision 0)
+++ added-file (working copy)
@@ -0,0 +1 @@
+This is an added file.
$
```

Sometimes you might need Subversion to interpret a patch “in reverse”—where added things get treated as removed things, and vice-versa. Use the `--reverse-diff` option for this purpose. In the following example, we'll squirrel away a patch file which describes the changes in our working copy, and then use a reverse patch operation to undo those changes.

```
$ svn status
M       foo.c
$ svn diff > PATCH
$ cat PATCH
Index: foo.c
===================================================================
--- foo.c (revision 128)
+++ foo.c (working copy)
@@ -1003,7 +1003,7 @@
     return ERROR_ON_THE_G_STRING;

   /* Do something in a loop. */
-  for (i = 0; i < txns->nelts; i++)
+  for (i = 0; i < txns->nelts; i--)
     {
       status = do_something(i);
       if (status)
$ svn patch --reverse-diff PATCH
U         foo.c
$ svn status
$
```

Name

svn propdel (pdel, pd) — Remove a property from an item.

Synopsis

```
svn propdel PROPNAME [PATH...]
```

```
svn propdel PROPNAME --revprop -r REV [TARGET]
```

Description

This removes properties from files, directories, or revisions. The first form removes versioned properties in your working copy, and the second removes unversioned remote properties on a repository revision (*TARGET* determines only which repository to access).

Options

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
```

Examples

Delete a property from a file in your working copy:

```
$ svn propdel svn:mime-type some-script
property 'svn:mime-type' deleted from 'some-script'.
```

Delete a revision property:

```
$ svn propdel --revprop -r 26 release-date
property 'release-date' deleted from repository revision '26'
```

Name

svn propedit (pedit, pe) — Edit the property of one or more items under version control. See svn propset (pset, ps) later in this chapter.

Synopsis

```
svn propedit PROPNAME TARGET...
```

```
svn propedit PROPNAME --revprop -r REV [TARGET]
```

Description

Edit one or more properties using your favorite editor. The first form edits versioned properties in your working copy, and the second edits unversioned remote properties on a repository revision (*TARGET* determines only which repository to access).

Options

```
--editor-cmd CMD
--encoding ENC
--file (-F) FILENAME
--force
--force-log
--message (-m) MESSAGE
--revision (-r) REV
--revprop
--with-revprop ARG
```

Examples

svn propedit makes it easy to modify properties that have multiple values:

```
$ svn propedit svn:keywords foo.c

    # svn will open in your favorite text editor a temporary file
    # containing the current contents of the svn:keywords property.  You
    # can add multiple values to a property easily here by entering one
    # value per line.  When you save the temporary file and exit,
    # Subversion will re-read the temporary file and use its updated
    # contents as the new value of the property.

Set new value for property 'svn:keywords' on 'foo.c'
$
```

Name

svn propget (pget, pg) — Print the value of a property.

Synopsis

```
svn propget PROPNAME [TARGET[@REV]...]
```

```
svn propget PROPNAME --revprop -r REV [URL]
```

Description

Print the value of a property on files, directories, or revisions. The first form prints the versioned property of an item or items in your working copy, and the second prints unversioned remote properties on a repository revision. See the section called "Properties" for more information on properties.

Options

```
--changelist (--cl) ARG
--depth ARG
--recursive (-R)
--revision (-r) REV
--revprop
--strict
--verbose (-v)
--xml
```

Examples

Examine a property of a file in your working copy:

```
$ svn propget svn:keywords foo.c
Author
Date
Rev
```

The same goes for a revision property:

```
$ svn propget svn:log --revprop -r 20
Began journal.
```

For a more structured display of properties, use the `--verbose` (`-v`) option:

```
$ svn propget svn:keywords foo.c --verbose
Properties on 'foo.c':
  svn:keywords
```

```
    Author
    Date
    Rev
```

By default, **svn propget** will append a trailing end-of-line sequence to the property value it prints. Most of the time, this is a desirable feature that has a positive effect on the printed output. But there are times when you might wish to capture the precise property value, perhaps because that value is not textual in nature, but of some binary format (such as a JPEG thumbnail stored as a property value, for example). To disable pretty-printing of property values, use the `--strict` option.

Lastly, you can get **svn propget** output in XML format with the `--xml` option:

```
$ svn propget --xml svn:ignore .
<?xml version="1.0"?>
<properties>
<target
   path="">
<property
   name="svn:ignore">*.o
</property>
</target>
</properties>
```

Name

svn proplist (plist, pl) — List all properties.

Synopsis

```
svn proplist [TARGET[@REV]...]
```

```
svn proplist --revprop -r REV [TARGET]
```

Description

List all properties on files, directories, or revisions. The first form lists versioned properties in your working copy, and the second lists unversioned remote properties on a repository revision (*TARGET* determines only which repository to access).

Options

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--verbose (-v)
--xml
```

Examples

You can use **proplist** to see the properties on an item in your working copy:

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

But with the `--verbose` (`-v`) flag, **svn proplist** is extremely handy as it also shows you the values for the properties:

```
$ svn proplist -v foo.c
Properties on 'foo.c':
  svn:mime-type
    text/plain
  svn:keywords
    Author Date Rev
  owner
    sally
```

Lastly, you can get **svn proplist** output in xml format with the `--xml` option:

```
$ svn proplist --xml
<?xml version="1.0"?>
<properties>
<target
   path=".">
<property
   name="svn:ignore"/>
</target>
</properties>
```

Name

svn propset (pset, ps) — Set *PROPNAME* to *PROPVAL* on files, directories, or revisions.

Synopsis

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...
```

```
svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [TARGET]
```

Description

Set *PROPNAME* to *PROPVAL* on files, directories, or revisions. The first example creates a versioned, local property change in the working copy, and the second creates an unversioned, remote property change on a repository revision (*TARGET* determines only which repository to access).

Subversion has a number of “special” properties that affect its behavior. See the section called “Subversion Properties” later in this chapter for more on these properties.

Options

```
--changelist (--cl) ARG
--depth ARG
--encoding ENC
--file (-F) FILENAME
--force
--quiet (-q)
--recursive (-R)
--revision (-r) REV
--revprop
--targets FILENAME
```

Examples

Set the MIME type for a file:

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property 'svn:mime-type' set on 'foo.jpg'
```

On a Unix system, if you want a file to have the executable permission set:

```
$ svn propset svn:executable ON somescript
property 'svn:executable' set on 'somescript'
```

Perhaps you have an internal policy to set certain properties for the benefit of your coworkers:

```
$ svn propset owner sally foo.c
property 'owner' set on 'foo.c'
```

If you made a mistake in a log message for a particular revision and want to change it, use `--revprop` and set `svn:log` to the new log message:

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New York."
property 'svn:log' set on repository revision '25'
```

Or, if you don't have a working copy, you can provide a URL:

```
$ svn propset --revprop -r 26 svn:log "Document nap." \
              http://svn.red-bean.com/repos
property 'svn:log' set on repository revision '25'
```

Lastly, you can tell **propset** to take its input from a file. You could even use this to set the contents of a property to something binary:

```
$ svn propset owner-pic -F sally.jpg moo.c
property 'owner-pic' set on 'moo.c'
```

By default, you cannot modify revision properties in a Subversion repository. Your repository administrator must explicitly enable revision property modifications by creating a hook named `pre-revprop-change`. See the section called “Implementing Repository Hooks” for more information on hook scripts.

Name

svn relocate — Relocate the working copy to point to a different repository root URL.

Synopsis

```
svn relocate FROM-PREFIX TO-PREFIX [PATH...]
```

```
svn relocate TO-URL [PATH]
```

Description

Sometimes an administrator might change the location (or apparent location, from the client's perspective) of a repository. The content of the repository doesn't change, but the repository's root URL does. The hostname may change because the repository is now being served from a different computer. Or, perhaps the URL scheme changes because the repository is now being served via SSL (using `https://`) instead of over plain HTTP. There are many different reasons for these types of repository relocations. But ideally, a “change of address” for a repository shouldn't suddently cause all the working copies which point to that repository to become forever unusable. And fortunately, that's not the case. Rather than force users to check out a new working copy when a repository is relocated, Subversion provides the **svn relocate** command, which “rewrites” the working copy's administrative metadata to refer to the new repository location.

The first **svn relocate** syntax allows you to update one or more working copies by what essentially amounts to a find-and-replace within the repository root URLs recorded in those working copies. Subversion will replace the initial substring *`FROM-PREFIX`* with the string *`TO-PREFIX`* in those URLs. These initial URL substrings can be as long or as short as is necessary to differentiate between them. Obviously, to use this syntax form, you need to know both the current root URL of the repository to which the working copy is pointing, and the new URL of that repository. (You can use **svn info** to determine the former.)

The second syntax does not require that you know the current repository root URL with which the working copy is associated at all—only the new repository URL (*`TO-URL`*) to which it should be pointing. In this syntax form, only one working copy may be relocated at a time.

Users often get confused about the difference between **svn switch** and **svn relocate**. Here's the rule of thumb:

- If the working copy needs to reflect a new directory *within* the repository, use **svn switch**.
- If the working copy still reflects the same repository directory, but the location of the repository itself has changed, use **svn relocate**.

Options

`--ignore-externals`

Examples

Let's start with a working copy that reflects a local repository URL:

```
$ svn info | grep URL:
URL: file:///var/svn/repos/trunk
```

```
$
```

One day the administrator decides to rename the on-disk repository directory. We missed the memo, so we see an error the next time we try to update our working copy.

```
$ svn up
Updating '.':
svn: E180001: Unable to connect to a repository at URL 'file:///var/svn/repos/trunk'
```

After cornering the administrator over by the vending machines, we learn about the repository being moved and are told the new URL. Rather than checkout a new working copy, though, we simply ask Subversion to rewrite the working copy metadata to point to the new repository location.

```
$ svn relocate file:///var/svn/new-repos/trunk
$
```

Subversion doesn't tell us much about what it did, but hey—error-free operation is really all we need, right? Our working copy is functional for online operations again.

```
$ svn up
Updating '.':
A    lib/new.c
M    src/code.h
M    src/headers.h
…
```

Once again, this type of relocation affects *working copy metadata only*. It will not change your versioned or unversioned file contents, perform any version control operations (such as commits or updates), and so on.

A few months later, we're told that the company is moving development to separate machines and that we'll be using HTTP to access the repository. So we relocate our working copy again.

```
$ svn relocate http://svn.company.com/repos/trunk
$
```

Now, each time we perform a relocation of this sort, Subversion contacts the repository—at its new URL, of course—to verify a few things.

First, it wants to compare the UUID of the repository against what is stored in the working copy. If these UUIDs don't match, the working copy relocation is disallowed. Maybe this isn't the same repository (just in a new location) after all?

Secondly, Subversion wants to ensure that the updated working copy metadata jives with respect to the directory location *inside* the repository. Subversion won't let you accidentally relocate a working copy of a branch in your repository to the URL of a different branch in the same repository. (That's what **svn switch**, described in svn switch (sw), is for.)

Also, Subversion will not allow you to relocate a subtree of the working copy. If you're going to relocate the working copy at all, you must relocate the whole thing. This is to protect the integrity of the working copy metadata and behavior as a whole. (And really, you'd be hard pressed to come up with a compelling reason to relocate only a piece of your working copy anyway.)

Let's look at one final relocation opportunity. After using HTTP access for some time, the company moves to SSL-only access. Now, the only change to the repository URL is that the scheme goes from being `http://` to being `https://`. There are two different ways that we could make our working copy reflect ths change. The first is to do exactly as we've done before and relocate to the new repository URL.

```
$ svn relocate http://svn.company.com/repos/trunk
$
```

But we have another option here, too. We could simply ask Subversion to swap out the changed prefixes of the URL.

```
$ svn relocate http https
$
```

Either approach leaves us a working copy whose metadata has been updated to point to the right repository location.

By default, **svn relocate** will traverse any external working copies nested within your working copy and attempt relocation of those working copies, too. Use the `--ignore-externals` option to disable this behavior.

Name

svn resolve — Resolve conflicts on working copy files or directories.

Synopsis

```
svn resolve PATH...
```

Description

Resolve “conflicted” state on working copy files or directories. This routine does not semantically resolve conflict markers; however, it replaces *PATH* with the version specified by the `--accept` argument and then removes conflict-related artifact files. This allows *PATH* to be committed again—that is, it tells Subversion that the conflicts have been “resolved.”

See the section called “Resolve Any Conflicts” for an in-depth look at resolving conflicts.

Options

```
--accept ACTION
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

Examples

Here's an example where, after a postponed conflict resolution during update, **svn resolve** replaces the all conflicts in file `foo.c` with your edits:

```
$ svn update
Updating '.':
Conflict discovered in 'foo.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options: p
C    foo.c
Updated to revision 5.
Summary of conflicts:
  Text conflicts: 1
$ svn resolve --accept mine-full foo.c
Resolved conflicted state of 'foo.c'
$
```

Name

svn resolved — *Deprecated*. Remove “conflicted” state on working copy files or directories.

Synopsis

```
svn resolved PATH...
```

Description

This command has been deprecated in favor of running **`svn resolve --accept working PATH`**. See svn resolve in the preceding section for details.

Remove “conflicted” state on working copy files or directories. This routine does not semantically resolve conflict markers; it merely removes conflict-related artifact files and allows *PATH* to be committed again; that is, it tells Subversion that the conflicts have been “resolved.” See the section called “Resolve Any Conflicts” for an in-depth look at resolving conflicts.

Options

```
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

Examples

If you get a conflict on an update, your working copy will sprout three new files:

```
$ svn update
Updating '.':
C    foo.c
Updated to revision 31.
Summary of conflicts:
  Text conflicts: 1
$ ls foo.c*
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
$
```

Once you've resolved the conflict and `foo.c` is ready to be committed, run **svn resolved** to let your working copy know you've taken care of everything.

You *can* just remove the conflict files and commit, but **svn resolved** fixes up some bookkeeping data in the working copy administrative area in addition to removing the conflict files, so we recommend that you use this command.

Name

svn revert — Undo all local edits.

Synopsis

```
svn revert PATH...
```

Description

Reverts any local changes to a file or directory and resolves any conflicted states. **svn revert** will revert not only the contents of an item in your working copy, but also any property changes. Finally, you can use it to undo any scheduling operations that you may have performed (e.g., files scheduled for addition or deletion can be “unscheduled”).

Options

```
--changelist (--cl) ARG
--depth ARG
--quiet (-q)
--recursive (-R)
--targets FILENAME
```

Examples

Discard changes to a file:

```
$ svn revert foo.c
Reverted foo.c
```

If you want to revert a whole directory of files, use the `--depth=infinity` option:

```
$ svn revert --depth=infinity .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

Lastly, you can undo any scheduling operations:

```
$ svn add mistake.txt whoops
A         mistake.txt
A         whoops
A         whoops/oopsie.c

$ svn revert mistake.txt whoops
Reverted mistake.txt
Reverted whoops
```

```
$ svn status
?       mistake.txt
?       whoops
```

svn revert is inherently dangerous, since its entire purpose is to throw away data—namely, your uncommitted changes. Once you've reverted, Subversion provides *no way* to get back those uncommitted changes.

If you provide no targets to **svn revert**, it will do nothing. To protect you from accidentally losing changes in your working copy, **svn revert** requires you to explicitly provide at least one target.

Name

svn status (stat, st) — Print the status of working copy files and directories.

Synopsis

```
svn status [PATH...]
```

Description

Print the status of working copy files and directories. With no arguments, it prints only locally modified items (no repository access). With `--show-updates` (`-u`), it adds working revision and server out-of-date information. With `--verbose` (`-v`), it prints full revision information on every item. With `--quiet` (`-q`), it prints only summary information about locally modified items.

The first seven columns in the output are each one character wide, and each column gives you information about a different aspect of each working copy item.

The first column indicates that an item was added, deleted, or otherwise changed:

`' '`
: No modifications.

`'A'`
: Item is scheduled for addition.

`'D'`
: Item is scheduled for deletion.

`'M'`
: Item has been modified.

`'R'`
: Item has been replaced in your working copy. This means the file was scheduled for deletion, and then a new file with the same name was scheduled for addition in its place.

`'C'`
: The contents (as opposed to the properties) of the item conflict with updates received from the repository.

`'X'`
: Item is present because of an externals definition.

`'I'`
: Item is being ignored (e.g., with the `svn:ignore` property).

`'?'`
: Item is not under version control.

`'!'`
: Item is missing (e.g., you moved or deleted it without using **svn**). This also indicates that a directory is incomplete (a checkout or update was interrupted).

`'~'`
: Item is versioned as one kind of object (file, directory, link), but has been replaced by a different kind of object.

The second column tells the status of a file's or directory's properties:

`' '`
: No modifications.

`'M'`
: Properties for this item have been modified.

`'C'`
: Properties for this item are in conflict with property updates received from the repository.

The third column is populated only if the working copy directory is locked (see the section called “Sometimes You Just Need to Clean Up”):

`' '`
: Item is not locked.

`'L'`
: Item is locked.

The fourth column is populated only if the item is scheduled for addition-with-history:

`' '`
: No history scheduled with commit.

`'+'`
: History scheduled with commit.

The fifth column is populated only if the item is switched relative to its parent (see the section called “Traversing Branches”):

`' '`
: Item is a child of its parent directory.

`'S'`
: Item is switched.

The sixth column is populated with lock information:

`' '`
: When `--show-updates` (`-u`) is used, this means the file is not locked. If `--show-updates` (`-u`) is *not* used, this merely means that the file is not locked in this working copy.

`'K'`
: File is locked in this working copy.

`'O'`
: File is locked either by another user or in another working copy. This appears only when `--show-updates` (`-u`) is used.

`'T'`
: File was locked in this working copy, but the lock has been “stolen” and is invalid. The file is currently locked in the repository. This appears only when `--show-updates` (`-u`) is used.

`'B'`

File was locked in this working copy, but the lock has been “broken” and is invalid. The file is no longer locked. This appears only when `--show-updates` (`-u`) is used.

The seventh column is populated only if the item is the victim of a tree conflict:

`' '`
: Item is not the victim of a tree conflict.

`'C'`
: Item is the victim of a tree conflict.

The eighth column is always blank.

The out-of-date information appears in the ninth column (only if you pass the `--show-updates` (`-u`) option):

`' '`
: The item in your working copy is up to date.

`'*'`
: A newer revision of the item exists on the server.

The remaining fields are variable width and delimited by spaces. The working revision is the next field if the `--show-updates` (`-u`) or `--verbose` (`-v`) option is passed.

If the `--verbose` (`-v`) option is passed, the last committed revision and last committed author are displayed next.

The working copy path is always the final field, so it can include spaces.

Options

```
--changelist (--cl) ARG
--depth ARG
--ignore-externals
--incremental
--no-ignore
--quiet (-q)
--show-updates (-u)
--verbose (-v)
--xml
```

Examples

This is the easiest way to find out what changes you have made to your working copy:

```
$ svn status wc
 M       wc/bar.c
A  +     wc/qax.c
```

If you want to find out what files in your working copy are out of date, pass the `--show-updates` (`-u`) option (this will *not*

make any changes to your working copy). Here you can see that `wc/foo.c` has changed in the repository since we last updated our working copy:

```
$ svn status -u wc
 M            965    wc/bar.c
        *     965    wc/foo.c
A  +          965    wc/qax.c
Status against revision:    981
```

`--show-updates` (`-u`) *only* places an asterisk next to items that are out of date (i.e., items that will be updated from the repository if you later use **svn update**). `--show-updates` (`-u`) does *not* cause the status listing to reflect the repository's version of the item (although you can see the revision number in the repository by passing the `--verbose` (`-v`) option).

The most information you can get out of the status subcommand is as follows:

```
$ svn status -u -v wc
 M            965       938 sally        wc/bar.c
        *     965       922 harry        wc/foo.c
A  +          965       687 harry        wc/qax.c
              965       687 harry        wc/zig.c
Status against revision:   981
```

Lastly, you can get **svn status** output in XML format with the `--xml` option:

```
$ svn status --xml wc
<?xml version="1.0"?>
<status>
<target
   path="wc">
<entry
   path="qax.c">
<wc-status
   props="none"
   item="added"
   revision="0">
</wc-status>
</entry>
<entry
   path="bar.c">
<wc-status
   props="normal"
   item="modified"
   revision="965">
<commit
   revision="965">
<author>sally</author>
<date>2008-05-28T06:35:53.048870Z</date>
</commit>
</wc-status>
</entry>
</target>
```

```
</status>
```

For many more examples of **svn status**, see the section called “See an overview of your changes”.

Name

svn switch (sw) — Update working copy to a different URL.

Synopsis

```
svn switch URL[@PEGREV] [PATH]
```

```
switch --relocate FROM TO [PATH...]
```

Description

The first variant of this subcommand (without the `--relocate` option) updates your working copy to point to a new URL. This is the Subversion way to make a working copy begin tracking a new branch. If specified, *PEGREV* determines in which revision the target is first looked up. See the section called “Traversing Branches” for an in-depth look at switching.

Beginning with Subversion 1.7, the **svn switch** command will demand by default that the URL to which you are switching your working copy shares a common ancestry with item that the working copy currently reflects. You can override this behavior by specifying the `--ignore-ancestry` option.

If `--force` is used, unversioned obstructing paths in the working copy do not automatically cause a failure if the switch attempts to add the same path. If the obstructing path is the same type (file or directory) as the corresponding path in the repository, it becomes versioned but its contents are left untouched in the working copy. This means that an obstructing directory's unversioned children may also obstruct and become versioned. For files, any content differences between the obstruction and the repository are treated like a local modification to the working copy. All properties from the repository are applied to the obstructing path.

As with most subcommands, you can limit the scope of the switch operation to a particular tree depth using the `--depth` option. Alternatively, you can use the `--set-depth` option to set a new “sticky” working copy depth on the switch target.

The `--relocate` option is deprecated as of Subversion 1.7. Use **svn relocate** (described in svn relocate) to perform working copy relocation instead.

Options

```
--accept ACTION
--depth ARG
--diff3-cmd CMD
--force
--ignore-ancestry
--ignore-externals
--quiet (-q)
--relocate
--revision (-r) REV
--set-depth ARG
```

Examples

If you're currently inside the directory `vendors`, which was branched to `vendors-with-fix`, and you'd like to switch your working copy to that branch:

```
$ svn switch http://svn.red-bean.com/repos/branches/vendors-with-fix .
```

```
U    myproj/foo.txt
U    myproj/bar.txt
U    myproj/baz.c
U    myproj/qux.c
Updated to revision 31.
```

To switch back, just provide the URL to the location in the repository from which you originally checked out your working copy:

```
$ svn switch http://svn.red-bean.com/repos/trunk/vendors .
U    myproj/foo.txt
U    myproj/bar.txt
U    myproj/baz.c
U    myproj/qux.c
Updated to revision 31.
```

You *can* switch just part of your working copy to a branch if you don't want to switch your entire working copy, but this is not generally recommended. It's too easy to forget that you've done this and wind up accidentally making and committing changes both to the switched and unswitched portions of your tree.

Name

svn unlock — Unlock working copy paths or URLs.

Synopsis

```
svn unlock TARGET...
```

Description

Unlock each *TARGET*. If any *TARGET* is locked by another user or no valid lock token exists in the working copy, print a warning and continue unlocking the rest of the *TARGET*s. Use `--force` to break a lock belonging to another user or working copy.

Options

```
--force
--targets FILENAME
```

Examples

Unlock two files in your working copy:

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

Unlock a file in your working copy that is currently locked by another user:

```
$ svn unlock tree.jpg
svn: E195013: 'tree.jpg' is not locked in this working copy
$ svn unlock --force tree.jpg
'tree.jpg' unlocked.
```

Unlock a file without a working copy:

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg unlocked.
```

For further details, see the section called “Locking”.

Name

svn update (up) — Update your working copy.

Synopsis

```
svn update [PATH...]
```

Description

svn update brings changes from the repository into your working copy. If no revision is given, it brings your working copy up to date with the `HEAD` revision. Otherwise, it synchronizes the working copy to the revision given by the `--revision` (`-r`) option. As part of the synchronization, **svn update** also removes any stale locks (see the section called "Sometimes You Just Need to Clean Up") found in the working copy.

For each updated item, it prints a line that starts with a character reporting the action taken. These characters have the following meaning:

`A`
: Added

`B`
: Broken lock (third column only)

`D`
: Deleted

`U`
: Updated

`C`
: Conflicted

`G`
: Merged

`E`
: Existed

A character in the first column signifies an update to the actual file, whereas updates to the file's properties are shown in the second column. Lock information is printed in the third column.

As with most subcommands, you can limit the scope of the update operation to a particular tree depth using the `--depth` option. Alternatively, you can use the `--set-depth` option to set a new "sticky" working copy depth on the update target.

Options

```
--accept ACTION
--changelist (--cl) ARG
--depth ARG
--diff3-cmd CMD
--editor-cmd CMD
--force
--ignore-externals
--parents
```

```
--quiet (-q)
--revision (-r) REV
--set-depth ARG
```

Examples

Pick up repository changes that have happened since your last update:

```
$ svn update
Updating '.':
A    newdir/toggle.c
A    newdir/disclose.c
A    newdir/launch.c
D    newdir/README
Updated to revision 32.
```

You can also “update” your working copy to an older revision (Subversion doesn't have the concept of “sticky” files like CVS does; see Appendix B, *Subversion for CVS Users*):

```
$ svn update -r30
Updating '.':
A    newdir/README
D    newdir/toggle.c
D    newdir/disclose.c
D    newdir/launch.c
U    foo.c
Updated to revision 30.
```

If you want to examine an older revision of a single file, you may want to use **svn cat** instead—it won't change your working copy.

svn update is also the primary mechanism used to configure sparse working copies. When used with the `--set-depth`, the update operation will omit or reenlist individual working copy members by modifying their recorded ambient depth to the depth you specify (fetching information from the repository as necessary). See the section called “Sparse Directories” for more about sparse directories.

You can update multiple targets with a single invocation, and Subversion will not only gracefully skip any unversioned targets you provide it, but as of Subversion 1.7 will also include a post-update summary of all the updates it performed:

```
$ cd my-projects
$ svn update *
Updating 'calc':
U    button.c
U    integer.c
Updated to revision 394.
Skipped 'tempfile.tmp'
Updating 'paint':
A    palettes.c
```

```
U    brushes.c
Updated to revision 60.
Updating 'ziptastic':
At revision 43.
Summary of updates:
  Updated 'calc' to r394.
  Updated 'paint' to r60.
  Updated 'ziptastic' to r43.
Summary of conflicts:
  Skipped paths: 1
$
```

Name

svn upgrade — Upgrade the metadata storage format for a working copy.

Synopsis

```
svn upgrade [PATH...]
```

Description

As new versions of Subversion are released, the format used for the working copy metadata changes to accomodate new features or fix bugs. Older versions of Subversion would automatically upgrade working copies to the new format the first time the working copy was used by the new version of the software. Beginning with Subversion 1.7, working copy upgrades must be explicitly performed at the user's request. **svn upgrade** is the subcommand used to trigger that upgrade process.

Options

`--quiet (-q)`

Examples

If you attempt to use Subversion 1.7 on a working copy created with an older version of Subversion, you will see an error:

```
$ svn status
svn: E155036: Please see the 'svn upgrade' command
svn: E155036: Working copy '/home/sally/project' is too old (format 10, create
d by Subversion 1.6)
$
```

Use the **svn upgrade** command to upgrade the working copy to the most recent metadata format supported by your version of Subversion.

```
$ svn upgrade
Upgraded '.'
Upgraded 'A'
Upgraded 'A/B'
Upgraded 'A/B/E'
Upgraded 'A/B/F'
Upgraded 'A/C'
Upgraded 'A/D'
Upgraded 'A/D/G'
Upgraded 'A/D/H'
$ svn status
D       A/B/E/alpha
M       A/D/gamma
A       A/newfile
$
```

Notice that **svn upgrade** preserved the local modifications present in the working copy at the time of the upgrade, which were introduced by the version of Subversion previously used to manipulate this working copy.

As was the case with automatically upgraded working copies in the past, explicitly upgraded working copies will be unusable by older versions of Subversion, too.

svnadmin—Subversion Repository Administration

svnadmin is the administrative tool for monitoring and repairing your Subversion repository. For detailed information on repository administration, see the maintenance section for the section called “svnadmin”.

Since **svnadmin** works via direct repository access (and thus can only be used on the machine that holds the repository), it refers to the repository with a path, not a URL.

svnadmin Options

Options in **svnadmin** are global, just as they are in **svn**:

`--bdb-log-keep`
: (Berkeley DB-specific.) Disable automatic log removal of database logfiles. Having these logfiles around can be convenient if you need to restore from a catastrophic repository failure.

`--bdb-txn-nosync`
: (Berkeley DB-specific.) Disables fsync when committing database transactions. Used with the **svnadmin create** command to create a Berkeley DB-backed repository with `DB_TXN_NOSYNC` enabled (which improves speed but has some risks associated with it).

`--bypass-hooks`
: Bypass the repository hook system.

`--bypass-prop-validation`
: When loading a dump file, disable the logic which validates property values.

`--clean-logs`
: Remove unused Berkeley DB logs.

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--deltas`
: When creating a repository dump file, specify changes in versioned properties and file contents as deltas against their previous state.

`--fs-type` *`ARG`*
: When creating a repository, use *`ARG`* as the requested filesyste type. *`ARG`* may be either `bdb` or `fsfs`.

`--force-uuid`
: By default, when loading data into a repository that already contains revisions, **svnadmin** will ignore the UUID from the dump stream. This option will cause the repository's UUID to be set to the UUID from the stream.

`--ignore-uuid`
: By default, when loading data into an empty repository, **svnadmin** will set the repository's UUID to the UUID from the dump stream. This option will cause the UUID from the stream to be ignored.

`--incremental`
: Dump a revision only as a diff against the previous revision, instead of the usual fulltext.

`--memory-cache-size` (`-M`) *`ARG`*
: Configures the size (in Megabytes) of the extra in-memory cache used to minimize redundant operations. The default value is `16`. (This cache is used for FSFS-backed repositories only.)

`--parent-dir` *`DIR`*
: When loading a dump file, root paths at *`DIR`* instead of `/`.

`--pre-1.4-compatible`
: When creating a new repository, use a format that is compatible with versions of Subversion earlier than Subversion 1.4.

`--pre-1.5-compatible`
: When creating a new repository, use a format that is compatible with versions of Subversion earlier than Subversion 1.5.

`--pre-1.6-compatible`
: When creating a new repository, use a format that is compatible with versions of Subversion earlier than Subversion 1.6.

`--revision` (`-r`) *`ARG`*
: Specify a particular revision to operate on.

`--quiet` (`-q`)
: Do not show normal progress—show only errors.

`--use-post-commit-hook`
: When loading a dump file, runs the repository's `post-commit` hook after finalizing each newly loaded revision.

`--use-post-revprop-change-hook`
: When changing a revision property, runs the repository's `post-revprop-change` hook after changing the revision property.

`--use-pre-commit-hook`
: When loading a dump file, runs the repository's `pre-commit` hook before finalizing each newly loaded revision. If the hook fails, aborts the commit and terminates the load process.

`--use-pre-revprop-change-hook`
: When changing a revision property, runs the repository's `pre-revprop-change` hook before changing the revision property. If the hook fails, aborts the modification and terminates.

`--wait`
: For operations which require exclusive repository access, wait until the requisite repository lock has been obtained instead of immediately erroring out when it cannot be.

svnadmin Subcommands

Here are the various subcommands for the **svnadmin** program.

Name

svnadmin crashtest — Simulate a process that crashes.

Synopsis

```
svnadmin crashtest REPOS_PATH
```

Description

Open the repository at *REPOS_PATH*, then abort, thus simulating a process that crashes while holding an open repository handle. This is used for testing automatic repository recovery (a new feature in Berkeley DB 4.4). It's unlikely that you'll need to run this command.

Options

None

Examples

```
$ svnadmin crashtest /var/svn/repos
Aborted
```

Exciting, isn't it?

Name

svnadmin create — Create a new, empty repository.

Synopsis

```
svnadmin create REPOS_PATH
```

Description

Create a new, empty repository at the path provided. If the provided directory does not exist, it will be created for you.[1] As of Subversion 1.2, **svnadmin** creates new repositories with the `FSFS` filesystem backend by default.

While **svnadmin create** will create the base directory for a new repository, it will not create intermediate directories. For example, if you have an empty directory named `/var/svn`, creating `/var/svn/repos` will work, while attempting to create `/var/svn/subdirectory/repos` will fail with an error. Also, keep in mind that, depending on where on your system you are creating your repository, you might need to run **svnadmin create** as a user with elevated privileges (such as the `root` user).

Options

```
--bdb-log-keep
--bdb-txn-nosync
--config-dir DIR
--fs-type ARG
--pre-1.4-compatible
--pre-1.5-compatible
--pre-1.6-compatible
```

Examples

Creating a new repository is this easy:

```
$ cd /var/svn
$ svnadmin create repos
$
```

In Subversion 1.0, a Berkeley DB repository is always created. In Subversion 1.1, a Berkeley DB repository is the default repository type, but an FSFS repository can be created using the `--fs-type` option:

```
$ cd /var/svn
$ svnadmin create repos --fs-type fsfs
$
```

[1]Remember, **svnadmin** works only with local *paths*, not *URLs*.

Name

svnadmin deltify — Deltify changed paths in a revision range.

Synopsis

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

Description

svnadmin deltify exists in current versions of Subversion only for historical reasons. This command is deprecated and no longer needed.

It dates from a time when Subversion offered administrators greater control over compression strategies in the repository. This turned out to be a lot of complexity for *very* little gain, and this “feature” was deprecated.

Options

`--memory-cache-size (-M) ARG`
`--quiet (-q)`
`--revision (-r) ARG`

Name

svnadmin dump — Dump the contents of the filesystem to `stdout`.

Synopsis

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental] [--deltas]
```

Description

Dump the contents of the filesystem to `stdout` in a "dump file" portable format, sending feedback to `stderr`. Dump revisions *`LOWER`* rev through *`UPPER`* rev. If no revisions are given, dump all revision trees. If only *`LOWER`* is given, dump that one revision tree. See the section called "Migrating Repository Data Elsewhere" for a practical use.

By default, the Subversion dump stream contains a single revision (the first revision in the requested revision range) in which every file and directory in the repository in that revision is presented as though that whole tree was added at once, followed by other revisions (the remainder of the revisions in the requested range), which contain only the files and directories that were modified in those revisions. For a modified file, the complete full-text representation of its contents, as well as all of its properties, are presented in the dump file; for a directory, all of its properties are presented.

Two useful options modify the dump file generator's behavior. The first is the `--incremental` option, which simply causes that first revision in the dump stream to contain only the files and directories modified in that revision, instead of being presented as the addition of a new tree, and in exactly the same way that every other revision in the dump file is presented. This is useful for generating a relatively small dump file to be loaded into another repository that already has the files and directories that exist in the original repository.

The second useful option is `--deltas`. This option causes **svnadmin dump** to, instead of emitting full-text representations of file contents and property lists, emit only deltas of those items against their previous versions. This reduces (in some cases, drastically) the size of the dump file that **svnadmin dump** creates. There are, however, disadvantages to using this option—deltified dump files are more CPU-intensive to create, cannot be operated on by **svndumpfilter**, and tend not to compress as well as their nondeltified counterparts when using third-party tools such as **gzip** and **bzip2**.

Options

```
--deltas
--incremental
--memory-cache-size (-M) ARG
--quiet (-q)
--revision (-r) ARG
```

Examples

Dump your whole repository:

```
$ svnadmin dump /var/svn/repos > full.dump
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
...
```

Incrementally dump a single transaction from your repository:

```
$ svnadmin dump /var/svn/repos -r 21 --incremental > incr.dump
* Dumped revision 21.
```

Name

svnadmin help (h, ?) — Help!

Synopsis

```
svnadmin help [SUBCOMMAND...]
```

Description

This subcommand is useful when you're trapped on a desert island with neither a Net connection nor a copy of this book.

Name

svnadmin hotcopy — Make a hot copy of a repository.

Synopsis

```
svnadmin hotcopy REPOS_PATH NEW_REPOS_PATH
```

Description

This subcommand makes a full “hot” backup of your repository, including all hooks, configuration files, and, of course, database files. If you pass the `--clean-logs` option, **svnadmin** will perform a hot copy of your repository, and then remove unused Berkeley DB logs from the original repository. You can run this command at any time and make a safe copy of the repository, regardless of whether other processes are using the repository.

Options

`--clean-logs`

As described in the section called “Berkeley DB”, hot-copied Berkeley DB repositories are *not* portable across operating systems, nor will they work on machines with a different “endianness” than the machine where they were created.

Name

svnadmin list-dblogs — Ask Berkeley DB which logfiles exist for a given Subversion repository (applies only to repositories using the `bdb` backend).

Synopsis

```
svnadmin list-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the logfiles accumulate, although most are no longer used and can be deleted to reclaim disk space. See the section called “Managing Disk Space” for more information.

Name

svnadmin list-unused-dblogs — Ask Berkeley DB which logfiles can be safely deleted (applies only to repositories using the `bdb` backend).

Synopsis

```
svnadmin list-unused-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the logfiles accumulate, although most are no longer used and can be deleted to reclaim disk space. See the section called “Managing Disk Space” for more information.

Examples

Remove all unused logfiles from the repository:

```
$ svnadmin list-unused-dblogs /var/svn/repos
/var/svn/repos/log.0000000031
/var/svn/repos/log.0000000032
/var/svn/repos/log.0000000033

$ svnadmin list-unused-dblogs /var/svn/repos | xargs rm
## disk space reclaimed!
```

Name

svnadmin load — Read a repository dump stream from `stdin`.

Synopsis

```
svnadmin load REPOS_PATH
```

Description

Read a repository dump stream from `stdin`, committing new revisions into the repository's filesystem. Send progress feedback to `stdout`.

Options

```
--bypass-prop-validation
--force-uuid
--ignore-uuid
--memory-cache-size (-M) ARG
--parent-dir DIR
--quiet (-q)
--use-post-commit-hook
--use-pre-commit-hook
```

Examples

This shows the beginning of loading a repository from a backup file (made, of course, with **svnadmin dump**):

```
$ svnadmin load /var/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Or if you want to load into a subdirectory:

```
$ svnadmin load --parent-dir new/subdir/for/project \
                /var/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Newer versions of Subversion have grown more strict regarding the format of the values of Subversion's own built-in properties. Of course, properties created with older versions of Subversion wouldn't have benefitted from that strictness, and as such might be improperly formatted. Dump streams carry property values as-is, so using Subversion 1.7 to load dump streams created from repositories with ill-formatted property values will, by default, trigger a validation error. There are several workaround for this problem. First, you can manually repair the problematic property values in the source repository and recreate the dump stream. Or, you can

manually tweak the dump stream itself to fix those property values. Finally, if you'd rather not deal with the problem right now, use the `--bypass-prop-validation` option with **svnadmin load**.

Name

svnadmin lslocks — Print descriptions of all locks.

Synopsis

```
svnadmin lslocks REPOS_PATH [PATH-IN-REPOS]
```

Description

Print descriptions of all locks in repository *REPOS_PATH* underneath the path *PATH-IN-REPOS*. If *PATH-IN-REPOS* is not provided, it defaults to the root directory of the repository.

Options

None

Examples

This lists the one locked file in the repository at `/var/svn/repos`:

```
$ svnadmin lslocks /var/svn/repos
Path: /tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Name

svnadmin lstxns — Print the names of all uncommitted transactions.

Synopsis

```
svnadmin lstxns REPOS_PATH
```

Description

Print the names of all uncommitted transactions. See the section called “Removing dead transactions” for information on how uncommitted transactions are created and what you should do with them.

Examples

List all outstanding transactions in a repository:

```
$ svnadmin lstxns /var/svn/repos/
1w
1x
```

Name

svnadmin pack — Possibly compact the repository into a more efficient storage model.

Synopsis

```
svnadmin pack REPOS_PATH
```

Description

See the section called “Packing FSFS filesystems” for more information.

Options

None

Name

svnadmin recover — Bring a repository database back into a consistent state (applies only to repositories using the `bdb` backend). In addition, if `repos/conf/passwd` does not exist, it will create a default passwordfile .

Synopsis

```
svnadmin recover REPOS_PATH
```

Description

Run this command if you get an error indicating that your repository needs to be recovered.

Options

`--wait`

Examples

Recover a hung repository:

```
$ svnadmin recover /var/svn/repos/
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Recovering the database requires an exclusive lock on the repository. (This is a “database lock”; see the sidebar The Three Meanings of “Lock”.) If another process is accessing the repository, then **svnadmin recover** will error:

```
$ svnadmin recover /var/svn/repos
svn: E165000: Failed to get exclusive repository access; perhaps another proce
ss such as httpd, svnserve or svn has it open?
$
```

The `--wait` option, however, will cause **svnadmin recover** to wait indefinitely for other processes to disconnect:

```
$ svnadmin recover /var/svn/repos --wait
Waiting on repository lock; perhaps another process has it open?

### time goes by…

Repository lock acquired.
Please wait; recovering the repository may take some time...
```

```
Recovery completed.
The latest repos revision is 34.
```

Name

svnadmin rmlocks — Unconditionally remove one or more locks from a repository.

Synopsis

```
svnadmin rmlocks REPOS_PATH LOCKED_PATH...
```

Description

Remove one or more locks from each *`LOCKED_PATH`*.

Options

None

Examples

This deletes the locks on `tree.jpg` and `house.jpg` in the repository at `/var/svn/repos`:

```
$ svnadmin rmlocks /var/svn/repos tree.jpg house.jpg
Removed lock on '/tree.jpg.
Removed lock on '/house.jpg.
```

Name

svnadmin rmtxns — Delete transactions from a repository.

Synopsis

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

Description

Delete outstanding transactions from a repository. This is covered in detail in the section called “Removing dead transactions”.

Options

`--quiet (-q)`

Examples

Remove named transactions:

```
$ svnadmin rmtxns /var/svn/repos/ 1w 1x
```

Fortunately, the output of **lstxns** works great as the input for **rmtxns**:

```
$ svnadmin rmtxns /var/svn/repos/  `svnadmin lstxns /var/svn/repos/`
```

This removes all uncommitted transactions from your repository.

Name

svnadmin setlog — Set the log message on a revision.

Synopsis

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

Description

Set the log message on revision *REVISION* to the contents of *FILE*.

This is similar to using **svn propset** with the `--revprop` option to set the `svn:log` property on a revision, except that you can also use the option `--bypass-hooks` to avoid running any pre- or post-commit hooks, which is useful if the modification of revision properties has not been enabled in the `pre-revprop-change` hook.

Revision properties are not under version control, so this command will permanently overwrite the previous log message.

Options

`--bypass-hooks`
`--revision (-r) ARG`

Examples

Set the log message for revision 19 to the contents of the file `msg`:

```
$ svnadmin setlog /var/svn/repos/ -r 19 msg
```

Name

svnadmin setrevprop — Set a property on a revision.

Synopsis

```
svnadmin setrevprop REPOS_PATH -r REVISION NAME FILE
```

Description

Set the property *NAME* on revision *REVISION* to the contents of *FILE*. Use `--use-pre-revprop-change-hook` or `--use-post-revprop-change-hook` to trigger the revision property-related hooks (e.g., if you want an email notification sent from your `post-revprop-change-hook`).

Options

```
--revision (-r) ARG
--use-post-revprop-change-hook
--use-pre-revprop-change-hook
```

Examples

The following sets the revision property `repository-photo` to the contents of the file `sandwich.png`:

```
$ svnadmin setrevprop /var/svn/repos -r 0 repository-photo sandwich.png
```

As you can see, **svnadmin setrevprop** has no output upon success.

Name

svnadmin setuuid — Reset the repository UUID.

Synopsis

```
svnadmin setuuid REPOS_PATH [NEW_UUID]
```

Description

Reset the repository UUID for the repository located at *`REPOS_PATH`*. If *`NEW_UUID`* is provided, use that as the new repository UUID; otherwise, generate a brand-new UUID for the repository.

Options

None

Examples

If you've **svnsync**ed `/var/svn/repos` to `/var/svn/repos-new` and intend to use `repos-new` as your canonical repository, you may want to change the UUID for `repos-new` to the UUID of `repos` so that your users don't have to check out a new working copy to accommodate the change:

```
$ svnadmin setuuid /var/svn/repos-new 2109a8dd-854f-0410-ad31-d604008985ab
```

As you can see, **svnadmin setuuid** has no output upon success.

Name

svnadmin upgrade — Upgrade a repository to the latest supported schema version.

Synopsis

```
svnadmin upgrade REPOS_PATH
```

Description

Upgrade the repository located at *`REPOS_PATH`* to the latest supported schema version.

This functionality is provided as a convenience for repository administrators who wish to make use of new Subversion functionality without having to undertake a potentially costly full repository dump and load operation. As such, the upgrade performs only the minimum amount of work needed to accomplish this while still maintaining the integrity of the repository. While a dump and subsequent load guarantee the most optimized repository state, **svnadmin upgrade** does not.

You should *always* back up your repository before upgrading.

Options

None

Examples

Upgrade the repository at path `/var/repos/svn`:

```
$ svnadmin upgrade /var/repos/svn
Repository lock acquired.
Please wait; upgrading the repository may take some time...

Upgrade completed.
```

Name

svnadmin verify — Verify the data stored in the repository.

Synopsis

```
svnadmin verify REPOS_PATH
```

Description

Run this command if you wish to verify the integrity of your repository. This basically iterates through all revisions in the repository by internally dumping all revisions and discarding the output—it's a good idea to run this on a regular basis to guard against latent hard disk failures and “bitrot.” If this command fails—which it will do at the first sign of a problem—that means your repository has at least one corrupted revision, and you should restore the corrupted revision from a backup (you did make a backup, didn't you?).

Options

```
--memory-cache-size (-M) ARG
--quiet (-q)
--revision (-r) ARG
```

Examples

Verify a hung repository:

```
$ svnadmin verify /var/svn/repos/
* Verified revision 1729.
```

svnlook—Subversion Repository Examination

svnlook is a command-line utility for examining different aspects of a Subversion repository. It does not make any changes to the repository—it's just used for “peeking.” **svnlook** is typically used by the repository hooks, but a repository administrator might find it useful for diagnostic purposes.

Since **svnlook** works via direct repository access (and thus can be used only on the machine that holds the repository), it refers to the repository with a path, not a URL.

If no revision or transaction is specified, **svnlook** defaults to the youngest (most recent) revision of the repository.

svnlook Options

Options in **svnlook** are global, just as they are in **svn** and **svnadmin**; however, most options apply to only one subcommand since the functionality of **svnlook** is (intentionally) limited in scope:

`--copy-info`
: Causes **svnlook changed** to show detailed copy source information.

`--diff-copy-from`
Print differences for copied items against the copy source.

`--extensions` (`-x`) *`ARG`*
Specifies customizations which Subversion should make when performing difference calculations. Valid extensions include:

`--ignore-space-change` (`-b`)
Ignore changes in the amount of white space.

`--ignore-all-space` (`-w`)
Ignore all white space.

`--ignore-eol-style`
Ignore changes in EOL (end-of-line) style.

`--show-c-function` (`-p`)
Show C function names in the diff output.

`--unified` (`-u`)
Show three lines of unified diff context.

The default value is `-u`.

Note that when Subversion is configured to invoke an external diff command, the value of the `--extension` (`-x`) option isn't restricted to the previously mentioned options, but may be *any* additional arguments which Subversion should pass to that command. If you wish to pass multiple arguments, you must enclose all of them in quotes.

`--full-paths`
Causes **svnlook tree** to display full paths instead of hierarchical, indented path components.

`--limit` (`-l`) *`ARG`*
Limit output to a maximum number of *`ARG`* items.

`--no-diff-deleted`
Prevents **svnlook diff** from printing differences for deleted files. The default behavior when a file is deleted in a transaction/revision is to print the same differences that you would see if you had left the file but removed all the content.

`--no-diff-added`
Prevents **svnlook diff** from printing differences for added files. The default behavior when you add a file is to print the same differences that you would see if you had added the entire contents of an existing (empty) file.

`--non-recursive` (`-N`)
Operate on a single directory only.

`--revision` (`-r`)
Specifies a particular revision number that you wish to examine.

`--revprop`
Operates on a revision property instead of a property specific to a file or directory. This option requires that you also pass a revision with the `--revision` (`-r`) option.

`--transaction` (`-t`)
Specifies a particular transaction ID that you wish to examine.

`--show-ids`
Shows the filesystem node revision IDs for each path in the filesystem tree.

`--verbose` (`-v`)
Be verbose. When used with **svnlook proplist**, for example, this causes Subversion to display not just the list of properties, but

their values also.

`--xml`

svnlook Subcommands

Here are the various subcommands for the **svnlook** program.

Name

svnlook author — Print the author.

Synopsis

```
svnlook author REPOS_PATH
```

Description

Print the author of a revision or transaction in the repository.

Options

```
--revision (-r)
--transaction (-t)
```

Examples

svnlook author is handy, but not very exciting:

```
$ svnlook author -r 40 /var/svn/repos
sally
```

Name

svnlook cat — Print the contents of a file.

Synopsis

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

Description

Print the contents of a file.

Options

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the contents of a file in transaction `ax8`, located at `/trunk/README`:

```
$ svnlook cat -t ax8 /var/svn/repos /trunk/README

               Subversion, a version control system.
               =====================================

$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003) $

Contents:

     I. A FEW POINTERS
    II. DOCUMENTATION
   III. PARTICIPATING IN THE SUBVERSION COMMUNITY
…
```

Name

svnlook changed — Print the paths that were changed.

Synopsis

```
svnlook changed REPOS_PATH
```

Description

Print the paths that were changed in a particular revision or transaction, as well as “svn update-style” status letters in the first two columns:

`'A '`
: Item added to repository

`'D '`
: Item deleted from repository

`'U '`
: File contents changed

`'_U'`
: Properties of item changed; note the leading underscore

`'UU'`
: File contents and properties changed

Files and directories can be distinguished, as directory paths are displayed with a trailing “`/`” character.

Options

```
--copy-info
--revision (-r)
--transaction (-t)
```

Examples

This shows a list of all the changed files and directories in revision 39 of a test repository. Note that the first changed item is a directory, as evidenced by the trailing `/`:

```
$ svnlook changed -r 39 /var/svn/repos
A   trunk/vendors/deli/
A   trunk/vendors/deli/chips.txt
A   trunk/vendors/deli/sandwich.txt
A   trunk/vendors/deli/pickle.txt
U   trunk/vendors/baker/bagel.txt
_U  trunk/vendors/baker/croissant.txt
UU  trunk/vendors/baker/pretzel.txt
D   trunk/vendors/baker/baguette.txt
```

Here's an example that shows a revision in which a file was renamed:

```
$ svnlook changed -r 64 /var/svn/repos
A   trunk/vendors/baker/toast.txt
D   trunk/vendors/baker/bread.txt
```

Unfortunately, nothing in the preceding output reveals the connection between the deleted and added files. Use the `--copy-info` option to make this relationship more apparent:

```
$ svnlook changed -r 64 --copy-info /var/svn/repos
A + trunk/vendors/baker/toast.txt
    (from trunk/vendors/baker/bread.txt:r63)
D   trunk/vendors/baker/bread.txt
```

Name

svnlook date — Print the datestamp.

Synopsis

```
svnlook date REPOS_PATH
```

Description

Print the datestamp of a revision or transaction in a repository.

Options

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the date of revision 40 of a test repository:

```
$ svnlook date -r 40 /var/svn/repos/
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

Name

svnlook diff — Print differences of changed files and properties.

Synopsis

```
svnlook diff REPOS_PATH
```

Description

Print GNU-style differences of changed files and properties in a repository.

Options

```
--diff-copy-from
--no-diff-added
--no-diff-deleted
--revision (-r)
--transaction (-t)
--extensions (-x) ARG
```

Examples

This shows a newly added (empty) file, a modified binary file, and a renamed (that is, copied and deleted) file with modifications:

```
$ svnlook diff -r 40 /var/svn/repos
Copied: trunk/relish.txt (from rev 39, trunk/vendors/deli/pickle.txt)
===================================================================
--- trunk/relish.txt                            (rev 0)
+++ trunk/relish.txt 2013-01-29 20:39:17 UTC (rev 40)
@@ -0,0 +1 @@
+Pickle relish is mostly made from cucumbers.

Deleted: trunk/vendors/deli/pickle.txt
===================================================================
--- trunk/vendors/deli/pickle.txt                           (rev 39)
+++ trunk/vendors/deli/pickle.txt 2013-01-29 20:39:17 UTC (rev 49)
@@ -1 +0,0 @@
-Pickles are mostly made from cucumbers.

Modified: trunk/vendors/deli/logo.jpg
===================================================================
(Binary files differ)

Added: trunk/vendors/deli/soda.txt
===================================================================
$
```

By default, **svnlook diff** will treat copied files very much like any other added file, displaying in their entirety the contents of the new file and merely using a different label to draw the copy/add distinction. However, you can use the `--diff-copy-from` option to cause **svnlook diff** to consider a copied file as worthy of mention only if it differs from the file from which it was copied, and to actually describe those differences.

```
$ svnlook diff -r 40 /var/svn/repos --diff-copy-from
Copied: trunk/relish.txt (from rev 39, trunk/vendors/deli/pickle.txt)
===================================================================
--- trunk/vendors/deli/pickle.txt 2013-01-29 20:39:17 UTC (rev 39)
+++ trunk/relish.txt 2013-01-29 20:47:40 UTC (rev 3)
@@ -1 +1 @@
-Pickles are mostly made from cucumbers.
+Pickle relish is mostly made from cucumbers.

Deleted: trunk/vendors/deli/pickle.txt
===================================================================
--- trunk/vendors/deli/pickle.txt (rev 39)
+++ trunk/vendors/deli/pickle.txt 2013-01-29 20:39:17 UTC (rev 40)
@@ -1 +0,0 @@
-Pickles are mostly made from cucumbers.

Modified: trunk/vendors/deli/logo.jpg
===================================================================
(Binary files differ)

Added: trunk/vendors/deli/soda.txt
================================================================================
$
```

Use the `--no-diff-deleted` option to silence output regarding deleted files.

```
$ svnlook diff -r 40 /var/svn/repos --no-diff-deleted
Copied: trunk/relish.txt (from rev 39, trunk/vendors/deli/pickle.txt)
===================================================================
--- trunk/relish.txt                            (rev 0)
+++ trunk/relish.txt 2013-01-29 20:39:17 UTC (rev 40)
@@ -0,0 +1 @@
+Pickle relish is mostly made from cucumbers.

Modified: trunk/vendors/deli/logo.jpg
===================================================================
(Binary files differ)

Added: trunk/vendors/deli/soda.txt
================================================================================
$
```

Note that in each of the previous examples, when a file has a nontextual `svn:mime-type` property, the differences are not explicitly shown.

Name

svnlook dirs-changed — Print the directories that were themselves changed.

Synopsis

```
svnlook dirs-changed REPOS_PATH
```

Description

Print the directories that were themselves changed (property edits) or whose file children were changed.

Options

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the directories that changed in revision 40 in our sample repository:

```
$ svnlook dirs-changed -r 40 /var/svn/repos
trunk/vendors/deli/
```

Name

svnlook filesize — Print the size (in bytes) of a versioned file.

Synopsis

```
svnlook filesize REPOS_PATH PATH_IN_REPOS
```

Description

Print the file size (in bytes) of the file located at *PATH_IN_REPOS* in the HEAD revision of the repository identified by *REPOS_PATH* as a base-10 integer followed by an end-of-line character. Use the `--revision` (`-r`) and `--transaction` (`-t`) options to specify a version of the file other than HEAD whose file size you wish to display.

Options

```
--revision (-r)
--transaction (-t)
```

Examples

The following demonstrates how to display the size of the `trunk/vendors/deli/soda.txt` file as it appeared in revision 40 of our sample repository:

```
$ svnlook filesize -r 40 /var/svn/repos trunk/vendors/deli/soda.txt
23
$
```

Name

svnlook help (h, ?) — Help!

Synopsis

```
Also svnlook -h and svnlook -?.
```

Description

Displays the help message for **svnlook**. This command, like its brother, **svn help**, is also your friend, even though you never call it anymore and forgot to invite it to your last party.

Options

None

Name

svnlook history — Print information about the history of a path in the repository (or the root directory if no path is supplied).

Synopsis

```
svnlook history REPOS_PATH [PATH_IN_REPOS]
```

Description

Print information about the history of a path in the repository (or the root directory if no path is supplied).

Options

```
--limit (-l) ARG
--revision (-r)
--show-ids
```

Examples

This shows the history output for the path `/branches/bookstore` as of revision 13 in our sample repository:

```
$ svnlook history -r 13 /var/svn/repos /branches/bookstore --show-ids
REVISION   PATH <ID>
--------   ---------
      13   /branches/bookstore <1.1.r13/390>
      12   /branches/bookstore <1.1.r12/413>
      11   /branches/bookstore <1.1.r11/0>
       9   /trunk <1.0.r9/551>
       8   /trunk <1.0.r8/131357096>
       7   /trunk <1.0.r7/294>
       6   /trunk <1.0.r6/353>
       5   /trunk <1.0.r5/349>
       4   /trunk <1.0.r4/332>
       3   /trunk <1.0.r3/335>
       2   /trunk <1.0.r2/295>
       1   /trunk <1.0.r1/532>
```

Name

svnlook info — Print the author, datestamp, log message size, and log message.

Synopsis

```
svnlook info REPOS_PATH
```

Description

Print the author, datestamp, log message size (in bytes), and log message, followed by a newline character.

Options

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the info output for revision 40 in our sample repository:

```
$ svnlook info -r 40 /var/svn/repos
sally
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
16
Rearrange lunch.
```

Name

svnlook lock — If a lock exists on a path in the repository, describe it.

Synopsis

```
svnlook lock REPOS_PATH PATH_IN_REPOS
```

Description

Print all information available for the lock at *PATH_IN_REPOS*. If *PATH_IN_REPOS* is not locked, print nothing.

Options

None

Examples

This describes the lock on the file `tree.jpg`:

```
$ svnlook lock /var/svn/repos tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Name

svnlook log — Print the log message, followed by a newline character.

Synopsis

```
svnlook log REPOS_PATH
```

Description

Print the log message.

Options

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the log output for revision 40 in our sample repository:

```
$ svnlook log /var/svn/repos/
Rearrange lunch.
```

Name

svnlook propget (pget, pg) — Print the raw value of a property on a path in the repository.

Synopsis

```
svnlook propget REPOS_PATH PROPNAME [PATH_IN_REPOS]
```

Description

List the value of a property on a path in the repository.

Options

```
--revision (-r)
--revprop
--transaction (-t)
```

Examples

This shows the value of the “seasonings” property on the file `/trunk/sandwich` in the `HEAD` revision:

```
$ svnlook pg /var/svn/repos seasonings /trunk/sandwich
mustard
```

Name

svnlook proplist (plist, pl) — Print the names and values of versioned file and directory properties.

Synopsis

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

Description

List the properties of a path in the repository. With `--verbose` (`-v`), show the property values too.

Options

```
--revision (-r)
--revprop
--transaction (-t)
--verbose (-v)
--xml
```

Examples

This shows the names of properties set on the file `/trunk/README` in the `HEAD` revision:

```
$ svnlook proplist /var/svn/repos /trunk/README
  original-author
  svn:mime-type
```

This is the same command as in the preceding example, but this time showing the property values as well:

```
$ svnlook -v proplist /var/svn/repos /trunk/README
  original-author : harry
  svn:mime-type : text/plain
```

Name

svnlook tree — Print the tree.

Synopsis

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

Description

Print the tree, starting at *PATH_IN_REPOS* (if supplied; at the root of the tree otherwise), optionally showing node revision IDs.

Options

```
--full-paths
--non-recursive (-N)
--revision (-r)
--show-ids
--transaction (-t)
```

Example

This shows the tree output for revision 13 in our sample repository:

```
$ svnlook tree -r 13 /var/svn/repos
/
 trunk/
  button.c
  Makefile
  integer.c
 branches/
  bookstore/
   button.c
   Makefile
   integer.c
...
```

Use the `--show-ids` option to include node revision IDs (unique internal identifiers for specific nodes in Subversion's versioned filesystem implementation):

```
$ svnlook tree -r 13 /var/svn/repos --show-ids
/ <0.0.r13/811>
 trunk/ <1.0.r9/551>
  button.c <2.0.r9/238>
  Makefile <3.0.r7/41>
  integer.c <4.0.r6/98>
 branches/ <5.0.r13/593>
  bookstore/ <1.1.r13/390>
   button.c <2.1.r12/85>
   Makefile <3.0.r7/41>
   integer.c <4.1.r13/109>
```

…

For output which lends itself more readily to being parsed by scripts, use the `--full-paths` option, which causes **svnlook** to print the full repository path of each tree item and to not use indentation to indicate hierarchy:

```
$ svnlook tree -r 13 /var/svn/repos --show-ids
/ <0.0.r13/811>
trunk/ <1.0.r9/551>
trunk/button.c <2.0.r9/238>
trunk/Makefile <3.0.r7/41>
trunk/integer.c <4.0.r6/98>
branches/ <5.0.r13/593>
branches/bookstore/ <1.1.r13/390>
branches/bookstore/button.c <2.1.r12/85>
branches/bookstore/Makefile <3.0.r7/41>
branches/bookstore/integer.c <4.1.r13/109>
…
```

Name

svnlook uuid — Print the repository's `UUID`.

Synopsis

```
svnlook uuid REPOS_PATH
```

Description

Print the `UUID` for the repository. The `UUID` is the repository's *u*niversal *u*nique *id*entifier. The Subversion client uses this identifier to differentiate between one repository and another.

Options

None

Examples

```
$ svnlook uuid /var/svn/repos
e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

Name

svnlook youngest — Print the youngest revision number.

Synopsis

```
svnlook youngest REPOS_PATH
```

Description

Print the youngest revision number of a repository.

Options

None

Examples

This shows the youngest revision of our sample repository:

```
$ svnlook youngest /var/svn/repos/
42
```

svnsync—Subversion Repository Mirroring

svnsync is the Subversion remote repository mirroring tool. Put simply, it allows you to replay the revisions of one repository into another one.

In any mirroring scenario, there are two repositories: the source repository, and the mirror (or “sink”) repository. The source repository is the repository from which **svnsync** pulls revisions. The mirror repository is the destination for the revisions pulled from the source repository. Each of the repositories may be local or remote—they are only ever addressed by their URLs.

The **svnsync** process requires only read access to the source repository; it never attempts to modify it. But obviously, **svnsync** requires both read and write access to the mirror repository.

svnsync is very sensitive to changes made in the mirror repository that weren't made as part of a mirroring operation. To prevent this from happening, it's best if the **svnsync** process is the only process permitted to modify the mirror repository.

svnsync Options

Options in **svnsync** are global, just as they are in **svn** and **svnadmin**:

`--allow-non-empty`
: Disables the verification (which **svnsync initialize** performs by default) that the repository being initialized is empty of history version.

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location

`.subversion` in the user's home directory).

`--config-option` *`CONFSPEC`*
Sets, for the duration of the command, the value of a runtime configuration option. *`CONFSPEC`* is a string which specifies the configuration option namespace, name and value that you'd like to assign, formatted as *`FILE`*:*`SECTION`*:*`OPTION`*=[*`VALUE`*]. In this syntax, *`FILE`* and *`SECTION`* are the runtime configuration file (either `config` or `servers`) and the section thereof, respectively, which contain the option whose value you wish to change. *`OPTION`* is, of course, the option itself, and *`VALUE`* the value (if any) you wish to assign to the option. For example, to temporarily disable the use of the automatic property setting feature, use `--config-option=servers:global:http-library=serf`. You can use this option multiple times to change multiple option values simultaneously.

`--disable-locking`
Causes **svnsync** to bypass its own exclusive access mechanisms and operate on the assumption that its exclusive access to the mirror repository is being guaranteed through some other, out-of-band mechanism.

`--no-auth-cache`
Prevents caching of authentication information (e.g., username and password) in the Subversion runtime configuration directories.

`--non-interactive`
In the case of an authentication failure or insufficient credentials, prevents prompting for credentials (e.g., username or password). This is useful if you're running Subversion inside an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--quiet` (`-q`)
Requests that the client print only essential information while performing an operation.

`--revision` (`-r`) *`ARG`*
Used by **svnsync copy-revprops** to specify a particular revision or revision range on which to operate.

`--source-password` *`PASSWD`*
Specifies the password for the Subversion server from which you are syncing. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

`--source-prop-encoding` *`ARG`*
Instructs **svnsync** to assume that translatable Subversion revision properties found in the source repository are stored using the character encoding *`ARG`* and to transcode those into UTF-8 when copying them into the mirror repository.

`--source-username` *`NAME`*
Specifies the username for the Subversion server from which you are syncing. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

`--steal-lock`
Causes **svnsync** to steal, as necessary, the lock which it uses on the mirror repository to ensure exclusive repository access. (This option should only be used when a lock exists in the mirror repository and is known to be stale—that is, when you are certain that there are no other **svnsync** processes accessing that repository.)

`--sync-password` *`PASSWD`*
Specifies the password for the Subversion server to which you are syncing. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

`--sync-username` *`NAME`*
Specifies the username for the Subversion server to which you are syncing. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

`--trust-server-cert`
Used with `--non-interactive` to accept any unknown SSL server certificates without prompting.

svnsync Subcommands

Here are the various subcommands for the **svnsync** program.

Name

svnsync copy-revprops — Copy all revision properties for a particular revision (or range of revisions) from the source repository to the mirror repository.

Synopsis

```
svnsync copy-revprops DEST_URL [SOURCE_URL]
```

```
svnsync copy-revprops DEST_URL REV[:REV2]
```

Description

Because Subversion revision properties can be changed at any time, it's possible that the properties for some revision might be changed after that revision has already been synchronized to another repository. Because the **svnsync synchronize** command operates only on the range of revisions that have not yet been synchronized, it won't notice a revision property change outside that range. Left as is, this causes a deviation in the values of that revision's properties between the source and mirror repositories. **svnsync copy-revprops** is the answer to this problem. Use it to resynchronize the revision properties for a particular revision or range of revisions.

When *SOURCE_URL* is provided, **svnsync** will use it as the repository URL which the destination repository is mirroring. Generally, *SOURCE_URL* will be exactly the same source URL as was used with the **svnsync initialize** command when the mirror was first set up. You may choose, however, to omit *SOURCE_URL*, in which case **svnsync** will consult the mirror repository's records to determine the source URL which should be used.

We strongly recommend that you specify the source URL on the command-line, especially when untrusted users have write access to the revision 0 properties which **svnsync** uses to coordinate its efforts.

Options

```
--config-dir DIR
--config-option CONFSPEC
--disable-locking
--no-auth-cache
--non-interactive
--quiet (-q)
--revision (-r) ARG
--source-password PASSWD
--source-prop-encoding ARG
--source-username NAME
--steal-lock
--sync-password PASSWD
--sync-username NAME
--trust-server-cert
```

Examples

Resynchronize the revision properties associated with a single revision (r6):

```
$ svnsync copy-revprops -r 6 file:///var/svn/repos-mirror \
                             http://svn.example.com/repos
```

```
Copied properties for revision 6.
$
```

Name

svnsync help — Help!

Synopsis

```
svnsync help
```

Description

This subcommand is useful when you're trapped in a foreign prison with neither a Net connection nor a copy of this book, but you do have a local Wi-Fi network running and you'd like to sync a copy of your repository over to the backup server that Ira The Knife is running over in cell block D.

Options

None

Name

svnsync info — Print information about the synchronization of a destination repository.

Synopsis

```
svnsync info DEST_URL
```

Description

Print the synchronization source URL, source repository UUID and the last revision merged from the source to the destination repository at *DEST_URL*.

Options

```
--config-dir DIR
--config-option CONFSPEC
--no-auth-cache
--non-interactive
--source-password PASSWD
--source-username NAME
--sync-password PASSWD
--sync-username NAME
--trust-server-cert
```

Examples

Print the synchronization information of a mirror repository:

```
$ svnsync info file:///var/svn/repos-mirror
Source URL: http://svn.example.com/repos
Source Repository UUID: e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
Last Merged Revision: 47
$
```

Name

svnsync initialize (init) — Initialize a mirror repository for synchronization from the source repository.

Synopsis

```
svnsync initialize MIRROR_URL SOURCE_URL
```

Description

svnsync initialize verifies that a repository meets the basic requirements of a new mirror repository and records the initial administrative information that associates the mirror repository with the source repository (specified by *SOURCE_URL*). This is the first **svnsync** operation you run on a would-be mirror repository.

Ordinarily, *SOURCE_URL* is the URL of the root directory of the Subversion repository you wish to mirror. Subversion 1.5 and newer allow you to use **svnsync** for partial repository mirroring, though — simply specify the URL of the source repository subdirectory you wish to mirror as *SOURCE_URL*.

By default, the aforementioned basic requirements of a mirror are that it allows revision property modifications and that it contains no version history. However, as of Subversion 1.7, you may now optionally disable the verification that the target repository is empty using the `--allow-non-empty` option. While the use of this option should not become habitual (as it bypasses a valuable safeguard mechanism), it does aid in one very common use-case: initializing a copy of a repository as a mirror of the original. This is especially handy when setting up new mirrors of repositories which contain a large amount of version history. Rather than initialize a brand new repository as a mirror and then syncronize all of the history into it, administrators will find it *significantly* faster to first make a copy of the mature repository (perhaps using **svnadmin hotcopy**) and then use **svnsync initialize --allow-non-empty** to initialize that copy as a mirror which is now already up-to-date with the original.

Options

```
--allow-non-empty
--config-dir DIR
--config-option CONFSPEC
--disable-locking
--no-auth-cache
--non-interactive
--quiet (-q)
--source-password PASSWD
--source-prop-encoding ARG
--source-username NAME
--steal-lock
--sync-password PASSWD
--sync-username NAME
--trust-server-cert
```

Examples

Fail to initialize a mirror repository due to inability to modify revision properties:

```
$ svnsync initialize file:///var/svn/repos-mirror \
                     http://svn.example.com/repos
svnsync: Repository has not been enabled to accept revision propchanges;
ask the administrator to create a pre-revprop-change hook
$
```

Initialize a repository as a mirror, having already created a `pre-revprop-change` hook that permits all revision property changes:

```
$ svnsync initialize file:///var/svn/repos-mirror \
                     http://svn.example.com/repos
Copied properties for revision 0.
$
```

Name

svnsync synchronize (sync) — Transfer all pending revisions from the source repository to the mirror repository.

Synopsis

```
svnsync synchronize DEST_URL [SOURCE_URL]
```

Description

The **svnsync synchronize** command does all the heavy lifting of a repository mirroring operation. After consulting with the mirror repository to see which revisions have already been copied into it, it then begins to copy any not-yet-mirrored revisions from the source repository.

svnsync synchronize can be gracefully canceled and restarted.

When *SOURCE_URL* is provided, **svnsync** will use it as the repository URL which the destination repository is mirroring. Generally, *SOURCE_URL* will be exactly the same source URL as was used with the **svnsync initialize** command when the mirror was first set up. You may choose, however, to omit *SOURCE_URL*, in which case **svnsync** will consult the mirror repository's records to determine the source URL which should be used.

We strongly recommend that you specify the source URL on the command-line, especially when untrusted users have write access to the revision 0 properties which **svnsync** uses to coordinate its efforts.

Options

```
--config-dir DIR
--config-option CONFSPEC
--disable-locking
--no-auth-cache
--non-interactive
--quiet (-q)
--source-password PASSWD
--source-prop-encoding ARG
--source-username NAME
--steal-lock
--sync-password PASSWD
--sync-username NAME
--trust-server-cert
```

Examples

Copy unsynchronized revisions from the source repository to the mirror repository:

```
$ svnsync synchronize file:///var/svn/repos-mirror \
                      http://svn.example.com/repos
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
Copied properties for revision 3.
```

```
…
Committed revision 45.
Copied properties for revision 45.
Committed revision 46.
Copied properties for revision 46.
Committed revision 47.
Copied properties for revision 47.
$
```

svnrdump—Remote Subversion Repository Data Migration

svnrdump joined the Subversion tool chain in the Subversion 1.7 release. It is best described as a network-aware version of the **svnadmin dump** and **svnadmin load** commands, paired together and released as a separate standalone program. We discuss the process of dumping and loading repository data—using both **svnadmin** and **svnrdump**— in the section called "Migrating Repository Data Elsewhere".

svnrdump Options

Options in **svnrdump** are global, just as they are in **svn** and **svnadmin**:

`--config-dir` *`DIR`*
: Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--config-option` *`FILE`*`:`*`SECTION`*`:`*`OPTION`*`=[`*`VALUE`*`]`
: Sets, for the duration of the command, the value of a runtime configuration option. *`FILE`* and *`SECTION`* are the runtime configuration file (either `config` or `servers`) and the section thereof, respectively, which contain the option whose value you wish to change. *`OPTION`* is, of course, the option itself, and *`VALUE`* the value (if any) you wish to assign to the option. For example, to temporarily disable the use of the automatic property setting feature, use `--config-option=servers:global:http-library=serf`. You can use this option multiple times to change multiple option values simultaneously.

`--incremental`
: Dump a revision or revision range only as a diff against the previous revision, instead of the default, which is begin a dumped revision range with a complete expansion of all contents of the repository as of that revision.

`--no-auth-cache`
: Prevents caching of authentication information (e.g., username and password) in the Subversion runtime configuration directories.

`--non-interactive`
: In the case of an authentication failure or insufficient credentials, prevents prompting for credentials (e.g., username or password). This is useful if you're running Subversion inside an automated script and it's more appropriate to have Subversion fail than to prompt for more information.

`--password` *`PASSWD`*
: Specifies the password to use when authenticating against a Subversion server. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

`--quiet` (`-q`)
: Requests that the client print only essential information while performing an operation.

`--revision (-r) ARG`
: Used by **svnsync copy-revprops** to specify a particular revision or revision range on which to operate.

`--trust-server-cert`
: Used with `--non-interactive` to accept any unknown SSL server certificates without prompting.

`--username NAME`
: Specifies the username to use when authenticating against a Subversion server. If not provided, or if incorrect, Subversion will prompt you for this information as needed.

svnrdump Subcommands

Here are the various subcommands for the **svnrdump** program.

Name

svnrdump dump

Synopsis

```
svnrdump dump SOURCE_URL
```

Description

Dump—that is, generate a repository dump stream of—revisions of the repository item located at *SOURCE_URL*, printing the information to standard output. By default, the entire history will be included in the dump stream, but the scope of the operation can be limited via the use of the `--revision` (`-r`) option.

Options

```
--config-dir DIR
--config-option FILE:SECTION:OPTION=[VALUE]
--incremental
--no-auth-cache
--non-interactive
--password PASSWD
--quiet (-q)
--revision (-r) ARG
--trust-server-cert
--username NAME
```

Examples

Generate a dump stream of the full history of a remote repository (assuming that the user as who this runs has authorization to read all paths in the repository).

```
$ svnrdump dump http://svn.example.com/repos/calc > full.dump
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
```

Incrementally dump a single revision from that same repository:

```
$ svnrdump dump http://svn.example.com/repos/calc \
           -r 21 --incremental > inc.dump
* Dumped revision 21.
$
```

Name

svnrdump help — Help!

Synopsis

```
svnrdump help
```

Description

Use this to get help. Well, certain kinds of help. Need white lab coat and straightjackets kind of help? There's a whole different protocol for that sort of thing.

Options

None

Name

svnrdump load

Synopsis

```
svnrdump load DEST_URL
```

Description

Read from standard input revision information described in a Subversion repository dump stream, and load that information into the repository located at *DEST_URL*.

Options

```
--config-dir DIR
--config-option FILE:SECTION:OPTION=[VALUE]
--no-auth-cache
--non-interactive
--password PASSWD
--quiet (-q)
--trust-server-cert
--username NAME
```

Examples

Dump the contents of a local repository, and use **svnrdump load** to transfer that revision information into an existing remote repository:

```
$ svnadmin dump -q /var/svn/repos/new-project | \
      svnrdump load http://svn.example.com/repos/new-project
* Loaded revision 0
* Loaded revision 1.
* Loaded revision 2.
…
```

To operate properly **svnrdump load** requires that the target repository have revision property modification enabled via the pre-revprop-change hook. For details about that hook, see pre-revprop-change.

svnserve—Custom Subversion Server

svnserve allows access to Subversion repositories using Subversion's custom network protocol.

You can run **svnserve** as a standalone server process (for clients that are using the `svn://` access method); you can have a daemon such as **inetd** or **xinetd** launch it for you on demand (also for `svn://`), or you can have **sshd** launch it on demand for the `svn+ssh://` access method.

Regardless of the access method, once the client has selected a repository by transmitting its URL, **svnserve** reads a file named

`conf/svnserve.conf` in the repository directory to determine repository-specific settings such as what authentication database to use and what authorization policies to apply. See the section called "svnserve, a Custom Server" for details of the `svnserve.conf` file.

svnserve Options

Unlike the previous commands we've described, **svnserve** has no subcommands—it is controlled exclusively by options.

`--cache-fulltexts` *`ARG`*
: Toggles support for fulltext file content caching (in FSFS repositories only).

`--cache-txdeltas` *`ARG`*
: Toggles support for file content delta caching (in FSFS repositories only).

`--compression` *`LEVEL`*
: Specifies the level of compression used for wire transmissions as an integer beween 0 and 9, inclusive. A value of `9` offers the best compression, `5` is the default value, and `0` disables compression altogether.

`--daemon` (`-d`)
: Causes **svnserve** to run in daemon mode. **svnserve** backgrounds itself and accepts and serves TCP/IP connections on the `svn` port (3690, by default).

`--foreground`
: When used together with `-d`, causes **svnserve** to stay in the foreground. This is mainly useful for debugging.

`--inetd` (`-i`)
: Causes **svnserve** to use the `stdin` and `stdout` file descriptors, as is appropriate for a daemon running out of **inetd**.

`--help` (`-h`)
: Displays a usage summary and exits.

`--listen-host` *`HOST`*
: Causes **svnserve** to listen on the interface specified by *`HOST`*, which may be either a hostname or an IP address.

`--listen-once` (`-X`)
: Causes **svnserve** to accept one connection on the `svn` port, serve it, and exit. This option is mainly useful for debugging.

`--listen-port` *`PORT`*
: Causes **svnserve** to listen on *`PORT`* when run in daemon mode. (FreeBSD daemons listen only on tcp6 by default—this option tells them to also listen on tcp4.)

`--log-file` *`FILENAME`*
: Instructs **svnserve** to create (if necessary) and use the file located at *`FILENAME`* for Subversion operational log output of the same sort that **mod_dav_svn** generates. See the section called "High-level Logging" for details.

`--memory-cache-size` (`-M`) *`ARG`*
: Configures the size (in Megabytes) of the extra in-memory cache used to minimize redundant operations. The default value is `16`. (This cache is used for FSFS-backed repositories only.)

`--pid-file` *`FILENAME`*
: Causes **svnserve** to write its process ID to *`FILENAME`*, which must be writable by the user under which **svnserve** is running.

`--prefer-ipv6` (`-6`
: When resolving the listen hostname, prever an IPv6 answer over an IPv4 one. IPv4 is preferred by default.

`--quiet`
: Disables progress notifications. Error output will still be printed.

`--root (-r) ROOT`
Sets the virtual root for repositories served by **svnserve**. The pathname in URLs provided by the client will be interpreted relative to this root and will not be allowed to escape this root.

`--threads (-T)`
When running in daemon mode, causes **svnserve** to spawn a thread instead of a process for each connection (e.g., for when running on Windows). The **svnserve** process still backgrounds itself at startup time.

`--tunnel (-t)`
Causes **svnserve** to run in tunnel mode, which is just like the **inetd** mode of operation (both modes serve one connection over `stdin`/`stdout`, and then exit), except that the connection is considered to be preauthenticated with the username of the current UID. This flag is automatically passed for you by the client when running over a tunnel agent such as **ssh**. That means there's rarely any need for *you* to pass this option to **svnserve**. So, if you find yourself typing `svnserve --tunnel` on the command line and wondering what to do next, see the section called "Tunneling over SSH".

`--tunnel-user NAME`
Used in conjunction with the `--tunnel` option, tells **svnserve** to assume that *NAME* is the authenticated user, rather than the UID of the **svnserve** process. This is useful for users wishing to share a single system account over SSH, but to maintain separate commit identities.

`--version`
Displays version information and a list of repository backend modules available, and then exits.

svndumpfilter—Subversion History Filtering

svndumpfilter is a command-line utility for removing history from a Subversion dump file by either excluding or including paths beginning with one or more named prefixes. For details, see the section called "svndumpfilter".

svndumpfilter Options

Options in **svndumpfilter** are global, just as they are in **svn** and **svnadmin**:

`--drop-empty-revs`
If filtering causes any revision to be empty (i.e., causes no change to the repository), removes these revisions from the final dump file.

`--pattern`
Treat the path prefixes provided to the filtering commands as file glob patterns rather than explicit path substrings.

`--renumber-revs`
Renumbers revisions that remain after filtering.

`--skip-missing-merge-sources`
Skips merge sources that have been removed as part of the filtering. Without this option, **svndumpfilter** will exit with an error if the merge source for a retained path is removed by filtering.

`--preserve-revprops`
If all nodes in a revision are removed by filtering and `--drop-empty-revs` is not passed, the default behavior of **svndumpfilter** is to remove all revision properties except for the date and the log message (which will merely indicate that the revision is empty). Passing this option will preserve existing revision properties (which may or may not make sense since the related content is no longer present in the dump file).

`--targets FILENAME`
Instructs **svndumpfilter** to read additional path prefixes—one per line—from the file located at *FILENAME*. This is especially useful for complex filtering operations which require more prefixes than the operating system allows to be specified on a

single command line.

`--quiet`
Does not display filtering statistics.

svndumpfilter Subcommands

Here are the various subcommands for the **svndumpfilter** program.

Name

svndumpfilter exclude — Filter out nodes with given prefixes from the dump stream.

Synopsis

```
svndumpfilter exclude PATH_PREFIX...
```

Description

This can be used to exclude nodes that begin with one or more *PATH_PREFIX*es from a filtered dump file.

Options

```
--drop-empty-revs
--pattern
--preserve-revprops
--quiet
--renumber-revs
--skip-missing-merge-sources
--targets FILENAME
```

Examples

If we have a dump file from a repository with a number of different picnic-related directories in it, but we want to keep everything *except* the `sandwiches` part of the repository, we'll exclude only that path:

```
$ svndumpfilter exclude sandwiches < dumpfile > filtered-dumpfile
Excluding prefixes:
   '/sandwiches'

Revision 0 committed as 0.
Revision 1 committed as 1.
Revision 2 committed as 2.
Revision 3 committed as 3.
Revision 4 committed as 4.

Dropped 1 node(s):
   '/sandwiches'
$
```

Beginning in Subversion 1.7, **svndumpfilter** can optionally treat the *PATH_PREFIX*s not merely as explicit substrings, but as file patterns instead. So, for example, if you wished to filter out paths which ended with `.OLD`, you would do the following:

```
$ svndumpfilter exclude --pattern "*.OLD" < dumpfile > filtered-dumpfile
Excluding prefix patterns:
   '/*.OLD'

Revision 0 committed as 0.
Revision 1 committed as 1.
```

```
Revision 2 committed as 2.
Revision 3 committed as 3.
Revision 4 committed as 4.

Dropped 3 node(s):
   '/condiments/salt.OLD'
   '/condiments/pepper.OLD'
   '/toppings/cheese.OLD'
$
```

Name

svndumpfilter include — Filter out nodes without given prefixes from dump stream.

Synopsis

```
svndumpfilter include PATH_PREFIX...
```

Description

Can be used to include nodes that begin with one or more *PATH_PREFIX*es in a filtered dump file (thus excluding all other paths).

Options

```
--drop-empty-revs
--pattern
--preserve-revprops
--quiet
--renumber-revs
--skip-missing-merge-sources
--targets FILENAME
```

Example

If we have a dump file from a repository with a number of different picnic-related directories in it, but want to keep only the `sandwiches` part of the repository, we'll include only that path:

```
$ svndumpfilter include sandwiches < dumpfile > filtered-dumpfile
Including prefixes:
   '/sandwiches'

Revision 0 committed as 0.
Revision 1 committed as 1.
Revision 2 committed as 2.
Revision 3 committed as 3.
Revision 4 committed as 4.

Dropped 12 node(s):
   '/condiments'
   '/condiments/pepper'
   '/condiments/pepper.OLD'
   '/condiments/salt'
   '/condiments/salt.OLD'
   '/drinks'
   '/snacks'
   '/supplies'
   '/toppings'
   '/toppings/cheese'
   '/toppings/cheese.OLD'
   '/toppings/lettuce'
$
```

Beginning in Subversion 1.7, **svndumpfilter** can optionally treat the *PATH_PREFIX*s not merely as explicit substrings, but as file

patterns instead. So, for example, if you wished to include only paths which ended with `ks`, you would do the following:

```
$ svndumpfilter include --pattern "*ks" < dumpfile > filtered-dumpfile
Including prefix patterns:
   '/*ks'

Revision 0 committed as 0.
Revision 1 committed as 1.
Revision 2 committed as 2.
Revision 3 committed as 3.
Revision 4 committed as 4.

Dropped 11 node(s):
   '/condiments'
   '/condiments/pepper'
   '/condiments/pepper.OLD'
   '/condiments/salt'
   '/condiments/salt.OLD'
   '/sandwiches'
   '/supplies'
   '/toppings'
   '/toppings/cheese'
   '/toppings/cheese.OLD'
   '/toppings/lettuce'
$
```

Name

svndumpfilter help — Help!

Synopsis

```
svndumpfilter help [SUBCOMMAND...]
```

Description

Displays the help message for **svndumpfilter**. Unlike other help commands documented in this chapter, there is no witty commentary for this help command. The authors of this book deeply regret the omission.

Options

None

svnversion—Subversion Working Copy Version Info

Name

svnversion — Summarize the local revision(s) of a working copy.

Synopsis

```
svnversion [OPTIONS] [WC_PATH [TRAIL_URL]]
```

Description

svnversion is a program for summarizing the revision mixture of a working copy. The resultant revision number, or revision range, is written to standard output.

It's common to use this output in your build process when defining the version number of your program.

TRAIL_URL, if present, is the trailing portion of the URL used to determine whether *WC_PATH* itself is switched (detection of switches within *WC_PATH* does not rely on *TRAIL_URL*).

When *WC_PATH* is not defined, the current directory will be used as the working copy path. *TRAIL_URL* cannot be defined if *WC_PATH* is not explicitly given.

Options

Like **svnserve**, **svnversion** has no subcommands—only options:

`--no-newline` (`-n`)
: Omits the usual trailing newline from the output.

`--committed` (`-c`)
: Uses the last-changed revisions rather than the current (i.e., highest locally available) revisions.

`--help` (`-h`)
: Prints a help summary.

`--quiet` (`-q`)
: Requests that the program print only essential information while performing an operation.

`--version`
: Prints the version of **svnversion** and exit with no error.

Examples

If the working copy is all at the same revision (e.g., immediately after an update), then that revision is printed out:

```
$ svnversion
4168
```

You can add *TRAIL_URL* to make sure the working copy is not switched from what you expect. Note that the *WC_PATH* is required in this command:

```
$ svnversion . /var/svn/trunk
4168
```

For a mixed-revision working copy, the range of revisions present is printed:

```
$ svnversion
4123:4168
```

If the working copy contains modifications, a trailing 'M' is added:

```
$ svnversion
4168M
```

If the working copy is switched, a trailing 'S' is added:

```
$ svnversion
4168S
```

svnversion will also inform you if the target working copy is sparsely populated (see the section called “Sparse Directories”) by attaching the 'P' code to its output:

```
$ svnversion
4168P
```

Thus, here is a mixed-revision, sparsely populated and switched working copy containing some local modifications:

```
$ svnversion
4123:4168MSP
```

mod_dav_svn—Subversion Apache HTTP Server Module

Name

mod_dav_svn Configuration Directives — Apache configuration directives for serving Subversion repositories through the Apache HTTP Server.

Description

This section briefly describes each Subversion Apache configuration directive. For an in-depth description of configuring Apache with Subversion, see the section called “httpd, the Apache HTTP Server”.

Directives

These are the `httpd.conf` directives that apply to **mod_dav_svn**:

`DAV svn`

Must be included in any `Directory` or `Location` block for a Subversion repository. It tells **httpd** to use the Subversion backend for `mod_dav` to handle all requests.

`SVNActivitiesDB` *`directory-path`*

Specifies the location in the filesystem where the activities database should be stored. By default, **mod_dav_svn** creates and uses a directory in the repository called `dav/activities.d`. The path specified with this option must be an absolute path.

If specified for an `SVNParentPath` area, **mod_dav_svn** appends the basename of the repository to the path specified here. For example:

```
<Location /svn>
  DAV svn

  # any "/svn/foo" URL will map to a repository in
  # /net/svn.nfs/repositories/foo
  SVNParentPath         "/net/svn.nfs/repositories"

  # any "/svn/foo" URL will map to an activities db in
  #  /var/db/svn/activities/foo
  SVNActivitiesDB       "/var/db/svn/activities"
</Location>
```

`SVNAdvertiseV2Protocol On|Off`

New to Subversion 1.7, this toggles whether **mod_dav_svn** advertises its support for the new version of its HTTP protocol also introduced in that version. Most admins will not wish to use this directive (which is `On` by default), choosing instead to enjoy the performance benefits that the new protocol offers. However, whena configuring a server as a write-through proxy to another server which does not support the new protocol, set this directive's value to `Off`.

`SVNAllowBulkUpdates On|Off`

Toggles support for all-inclusive responses to update-style `REPORT` requests. Subversion clients use `REPORT` requests to get information about directory tree checkouts and updates from **mod_dav_svn**. They can ask the server to send that information in one of two ways: with the entirety of the tree's information in one massive response, or with a *skelta* (a skeletal representation of a tree delta) which contains just enough information for the client to know what *additional* data to request from the server. When this directive is included with a value of `Off`, **mod_dav_svn** will only ever respond to these `REPORT` requests with skelta responses, regardless of the type of responses requested by the client.

Most folks won't need to use this directive at all. It primarily exists for administrators who wish—for security or auditing reasons—to force Subversion clients to fetch individually all the files and directories needed for updates and checkouts, thus leav-

ing an audit trail of `GET` and `PROPFIND` requests in Apache's logs. The default value of this directive is `On`.

`SVNAutoversioning On|Off`
When its value is `On`, allows write requests from WebDAV clients to result in automatic commits. A generic log message is auto-generated and attached to each revision. If you enable autoversioning, you'll likely want to set `ModMimeUsePathInfo On` so that `mod_mime` can set `svn:mime-type` to the correct MIME type automatically (as best as `mod_mime` is able to, of course). For more information, see Appendix C, *WebDAV and Autoversioning*. The default value of this directive is `Off`.

`SVNCacheFullTexts On|Off`
When set to `On`, this tells Subversion to cache content fulltexts if sufficient in-memory cache is available, which could offer a significant performance benefit to the server. (See also the `SVNInMemoryCacheSize` directive.) The default value of this directive is `Off`.

`SVNCacheTextDeltas On|Off`
When set to `On`, this tells Subversion to cache content deltas if sufficient in-memory cache is available, which could offer a significant performance benefit to the server. (See also the `SVNInMemoryCacheSize` directive.) The default value of this directive is `Off`.

`SVNCompressionLevel` *`level`*
Specifies the compression level used when sending file content over the network. A value of `0` disables compression altogether, and `9` is the maximum value. `5` is the default value.

`SVNIndexXSLT` *`directory-path`*
Specifies the URI of an XSL transformation for directory indexes. This directive is optional.

`SVNInMemoryCacheSize` *`size`*
Specifies the maximum size (in kbytes) per process of Subversion's in-memory object cache. The default value is `16384`; use a value of `0` to deactivate this cache altogether.

`SVNListParentPath On|Off`
When set to `On`, allows a `GET` of `SVNParentPath`, which results in a listing of all repositories under that path. The default setting is `Off`.

`SVNMasterURI` *`url`*
Specifies a URI to the master Subversion repository (used for a write-through proxy).

`SVNParentPath` *`directory-path`*
Specifies the location in the filesystem of a parent directory whose child directories are Subversion repositories. In a configuration block for a Subversion repository, either this directive or `SVNPath` must be present, but not both.

`SVNPath` *`directory-path`*
Specifies the location in the filesystem for a Subversion repository's files. In a configuration block for a Subversion repository, either this directive or `SVNParentPath` must be present, but not both.

`SVNPathAuthz On|Off|short_circuit`
Controls path-based authorization by enabling subrequests (`On`), disabling subrequests (`Off`; see the section called "Disabling path-based checks"), or querying **mod_authz_svn** directly (`short_circuit`). The default value of this directive is `On`.

`SVNReposName` *`name`*
Specifies the name of a Subversion repository for use in `HTTP GET` responses. This value will be prepended to the title of all directory listings (which are served when you navigate to a Subversion repository with a web browser). This directive is optional.

`SVNSpecialURI` *`component`*
Specifies the URI component (namespace) for special Subversion resources. The default is `!svn`, and most administrators will never use this directive. Set this only if there is a pressing need to have a file named `!svn` in your repository. If you change this on a server already in use, it will break all of the outstanding working copies, and your users will hunt you down with pitchforks and flaming torches.

mod_authz_svn—Subversion Apache HTTP Authorization Module

Name

mod_authz_svn Configuration Directives — Apache configuration directives for configuring path-based authorization for Subversion repositories served through the Apache HTTP Server.

Description

This section briefly describes each Apache configuration directive offered by **mod_authz_svn**. For an in-depth description of using path-based authorization in Subversion, see the section called "Path-Based Authorization".

Directives

These are the `httpd.conf` directives that apply to **mod_authz_svn**:

`AuthzForceUsernameCase Upper|Lower`
: Set to `Upper` or `Lower` to perform case conversion of the specified sort on the authenticated username before checking it for authorization. While usernames are compared in a case-sensitive fashion against those referenced in the authorization rules file, this directive can at least normalize variably-cased usernames into something consistent.

`AuthzSVNAccessFile` *`file-path`*
: Consult *`file-path`* for access rules describing the permissions for paths in Subversion repository.

`AuthzSVNAnonymous On|Off`
: Set to `Off` to disable two special-case behaviours of this module: interaction with the `Satisfy Any` directive and enforcement of the authorization policy even when no `Require` directives are present. The default value of this directive is `On`.

`AuthzSVNAuthoritative On|Off`
: Set to `Off` to allow access control to be passed along to lower modules. The default value of this directive is `On`.

`AuthzSVNNoAuthWhenAnonymousAllowed On|Off`
: Set to `On` to suppress authentication and authorization for requests which anonymous users are allowed to perform. The default value of this directive is `On`.

Subversion Properties

Subversion allows users to invent arbitrarily named versioned properties on files and directories, as well as unversioned properties on revisions. The only restriction is on properties whose names begin with `svn:` (those are reserved for Subversion's own use). While these properties may be set by users to control Subversion's behavior, users may not invent new `svn:` properties.

Versioned Properties

These are the versioned properties that Subversion reserves for its own use:

`svn:executable`
: If present on a file, the client will make the file executable in Unix-hosted working copies. See the section called "File Executability".

`svn:mime-type`
: If present on a file, the value indicates the file's MIME type. This allows the client to decide whether line-based contextual merging is safe to perform during updates, and can also affect how the file behaves when fetched via a web browser. See the section called "File Content Type".

`svn:ignore`

If present on a directory, the value is a list of *unversioned* file patterns to be ignored by **svn status** and other subcommands. See the section called "Ignoring Unversioned Items".

`svn:keywords`
: If present on a file, the value tells the client how to expand particular keywords within the file. See the section called "Keyword Substitution".

`svn:eol-style`
: If present on a file, the value tells the client how to manipulate the file's line-endings in the working copy and in exported trees. See the section called "End-of-Line Character Sequences" and svn export earlier in this chapter.

`svn:externals`
: If present on a directory, the value is a multiline list of other paths and URLs the client should check out. See the section called "Externals Definitions".

`svn:special`
: If present on a file, indicates that the file is not an ordinary file, but a symbolic link or other special object.[1]

`svn:needs-lock`
: If present on a file, tells the client to make the file read-only in the working copy, as a reminder that the file should be locked before editing begins. See the section called "Lock Communication".

`svn:mergeinfo`
: Used by Subversion to track merge data. See the section called "Mergeinfo and Previews" for details, but you should never edit this property unless you *really* know what you're doing.

Unversioned Properties

These are the unversioned properties that Subversion reserves for its own use:

`svn:author`
: If present, contains the authenticated username of the person who created the revision. (If not present, the revision was committed anonymously.)

`svn:autoversioned`
: If present, the revision was created via the autoversioning feature. See the section called "Autoversioning".

`svn:date`
: Contains the UTC time the revision was created, in ISO 8601 format. The value comes from the *server* machine's clock, not the client's.

`svn:log`
: Contains the log message describing the revision.

`svn:rdump-lock`
: Used to temporarily enforce mutually exclusive access to the repository by **svnrdump load**. This property is generally only observed when such an operation is active—or when an **svnrdump** command failed to cleanly disconnect from the repository. (This property is only relevant when it appears on revision 0.)

`svn:sync-currently-copying`
: Contains the revision number from the source repository which is currently being mirrored to this one by the **svnsync** tool. (This property is only relevant when it appears on revision 0.)

`svn:sync-from-uuid`
: Contains the UUID of the repository of which this repository has been initialized as a mirror by the **svnsync** tool. (This prop-

[1] As of this writing, symbolic links are indeed the only "special" objects. But there might be more in future releases of Subversion.

erty is only relevant when it appears on revision 0.)

`svn:sync-from-url`
: Contains the URL of the repository directory of which this repository has been initialized as a mirror by the **svnsync** tool. (This property is only relevant when it appears on revision 0.)

`svn:sync-last-merged-rev`
: Contains the revision of the source repository which was most recently and successfully mirrored to this one. (This property is only relevant when it appears on revision 0.)

`svn:sync-lock`
: Used to temporarily enforce mutually exclusive access to the repository by **svnsync** mirroring operations. This property is generally only observed when such an operation is active—or when an **svnsync** command failed to cleanly disconnect from the repository. (This property is only relevant when it appears on revision 0.)

Repository Hooks

These are the repository hooks that Subversion provides:

Name

start-commit — Notification of the beginning of a commit.

Description

The start-commit hook is run before the commit transaction is even created. It is typically used to decide whether the user has commit privileges at all.

If the start-commit hook program returns a nonzero exit value, the commit is stopped before the commit transaction is even created, and anything printed to `stderr` is marshalled back to the client.

Input Parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Authenticated username attempting the commit
3. Colon-separated list of capabilities that a client passes to the server, including `depth`, `mergeinfo`, and `log-revprops` (new in Subversion 1.5).

Common uses

Access control (e.g., temporarily lock out commits for some reason).

A means to allow access only from clients that have certain capabilities.

Name

pre-commit — Notification just prior to commit completion.

Description

The `pre-commit` hook is run just before a commit transaction is promoted to a new revision. Typically, this hook is used to protect against commits that are disallowed due to content or location (e.g., your site might require that all commits to a certain branch include a ticket number from the bug tracker, or that the incoming log message is nonempty).

If the `pre-commit` hook program returns a nonzero exit value, the commit is aborted, the commit transaction is removed, and anything printed to `stderr` is marshalled back to the client.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Commit transaction name

Additionally, Subversion passes any lock tokens provided by the committing client to the hook script via standard input. When present, these are formatted as a single line containing the string `LOCK-TOKENS:`, followed by additional lines—one per lock token—which contain the lock token information. Each lock token information line consists of the URI-escaped repository filesystem path associated with the lock, followed by the pipe (`|`) separator character, and finally the lock token string.

Common uses

Change validation and control

Name

post-commit — Notification of a successful commit.

Description

The `post-commit` hook is run after the transaction is committed and a new revision is created. Most people use this hook to send out descriptive emails about the commit or to notify some other tool (such as an issue tracker) that a commit has happened. Some configurations also use this hook to trigger backup processes.

If the `post-commit` hook returns a nonzero exit status, the commit *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Revision number created by the commit

Common uses

Commit notification; tool integration

Name

pre-revprop-change — Notification of a revision property change attempt.

Description

The `pre-revprop-change` hook is run immediately prior to the modification of a revision property when performed outside the scope of a normal commit. Unlike the other hooks, the default state of this one is to deny the proposed action. The hook must actually exist and return a zero exit value before a revision property modification can happen.

If the `pre-revprop-change` hook doesn't exist, isn't executable, or returns a nonzero exit value, no change to the property will be made, and anything printed to `stderr` is marshalled back to the client.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Revision whose property is about to be modified
3. Authenticated username attempting the property change
4. Name of the property changed
5. Change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes the intended new value of the property to the hook program via standard input.

Common uses

Access control; change validation and control

Name

post-revprop-change — Notification of a successful revision property change.

Description

The `post-revprop-change` hook is run immediately after the modification of a revision property when performed outside the scope of a normal commit. As you can derive from the description of its counterpart, the `pre-revprop-change` hook, this hook will not run at all unless the `pre-revprop-change` hook is implemented. It is typically used to send email notification of the property change.

If the `post-revprop-change` hook returns a nonzero exit status, the change *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Revision whose property was modified
3. Authenticated username of the person making the change
4. Name of the property changed
5. Change description: `A` (added), `D` (deleted), or `M` (modified)

Additionally, Subversion passes to the hook program, via standard input, the previous value of the property.

Common uses

Property change notification

Name

pre-lock — Notification of a path lock attempt.

Description

The `pre-lock` hook runs whenever someone attempts to lock a path. It can be used to prevent locks altogether or to create a more complex policy specifying exactly which users are allowed to lock particular paths. If the hook notices a preexisting lock, it can also decide whether a user is allowed to “steal” the existing lock.

If the `pre-lock` hook program returns a nonzero exit value, the lock action is aborted and anything printed to `stderr` is marshalled back to the client.

The hook program may optionally dictate the lock token which will be assigned to the lock by printing the desired lock token to standard output. Because of this, implementations of this hook should carefully avoid unexpected output sent to standard output.

If the `pre-lock` script takes advantage of lock token dictation feature, the responsibility of generating a *unique* lock token falls to the script itself. Failure to generate unique lock tokens may result in undefined—and very likely, undesired—behavior.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Versioned path that is to be locked
3. Authenticated username of the person attempting the lock
4. Comment provided when the lock was created
5. `1` if the user is attempting to steal an existing lock; `0` otherwise

Common uses

Access control

Name

post-lock — Notification of a successful path lock.

Description

The `post-lock` hook runs after one or more paths have been locked. It is typically used to send email notification of the lock event.

If the `post-lock` hook returns a nonzero exit status, the lock *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Authenticated username of the person who locked the paths

Additionally, the list of paths locked is passed to the hook program via standard input, one path per line.

Common uses

Lock notification

Name

pre-unlock — Notification of a path unlock attempt.

Description

The `pre-unlock` hook runs whenever someone attempts to remove a lock on a file. It can be used to create policies that specify which users are allowed to unlock particular paths. It's particularly important for determining policies about lock breakage. If user A locks a file, is user B allowed to break the lock? What if the lock is more than a week old? These sorts of things can be decided and enforced by the hook.

If the `pre-unlock` hook program returns a nonzero exit value, the unlock action is aborted and anything printed to `stderr` is marshalled back to the client.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Versioned path which is to be unlocked
3. Authenticated username of the person attempting the unlock
4. Lock token associated with the lock which is to be removed
5. `1` if the user is attempting to break the lock; `0` otherwise

Common uses

Access control

Name

post-unlock — Notification of a successful path unlock.

Description

The `post-unlock` hook runs after one or more paths have been unlocked. It is typically used to send email notification of the unlock event.

If the `post-unlock` hook returns a nonzero exit status, the unlock *will not* be aborted since it has already completed. However, anything that the hook printed to `stderr` will be marshalled back to the client, making it easier to diagnose hook failures.

Input parameter(s)

The command-line arguments passed to the hook program, in order, are:

1. Repository path
2. Authenticated username of the person who unlocked the paths

Additionally, the list of paths unlocked is passed to the hook program via standard input, one path per line.

Common uses

Unlock notification

Appendix A. Subversion Quick-Start Guide

If you're eager to get Subversion up and running (and you enjoy learning by experimentation), this appendix will show you how to create a repository, import code, and then check it back out again as a working copy. Along the way, we give links to the relevant chapters of this book.

If you're new to the entire concept of version control or to the “copy-modify-merge” model used by both CVS and Subversion, you should read Chapter 1, *Fundamental Concepts* before going any further.

Installing Subversion

Subversion is built on a portability layer called APR—the Apache Portable Runtime library. The APR library provides all the interfaces that Subversion needs to function on different operating systems: disk access, network access, memory management, and so on. While Subversion is able to use Apache HTTP Server (or, **httpd**) as one of its network server programs, its dependence on APR *does not* mean that **httpd** is a required component. APR is a standalone library usable by any application. It does mean, however, that Subversion clients and servers run on any operating system that **httpd** runs on: Windows, Linux, all flavors of BSD, Mac OS X, NetWare, and others.

The easiest way to get Subversion is to download a binary package built for your operating system. Subversion's web site (http://subversion.apache.org) often has these packages available for download, posted by volunteers. The site usually contains graphical installer packages for users of Microsoft operating systems. If you run a Unix-like operating system, you can use your system's native package distribution system (RPMs, DEBs, the ports tree, etc.) to get Subversion.

Alternatively, you can build Subversion directly from source code, though it's not always an easy task. (If you're not experienced at building open source software packages, you're probably better off downloading a binary distribution instead!) From the Subversion web site, download the latest source code release. After unpacking it, follow the instructions in the `INSTALL` file to build it.

If you're one of those folks that likes to use bleeding-edge software, you can also get the Subversion source code from the Subversion repository in which it lives. Obviously, you'll need to already have a Subversion client on hand to do this. But once you do, you can check out a working copy from http://svn.apache.org/repos/asf/subversion[1]:

```
$ svn checkout http://svn.apache.org/repos/asf/subversion/trunk subversion
A    subversion/HACKING
A    subversion/INSTALL
A    subversion/README
A    subversion/autogen.sh
A    subversion/build.conf
…
```

The preceding command will create a working copy of the latest (unreleased) Subversion source code into a subdirectory named `subversion` in your current working directory. You can adjust that last argument as you see fit. Regardless of what you call the new working copy directory, though, after this operation completes, you will now have the Subversion source code. Of course, you will still need to fetch a few helper libraries (apr, apr-util, etc.)—see the `INSTALL` file in the top level of the working copy for details.

[1]Note that the URL checked out in the example ends not with `subversion`, but with a subdirectory thereof called `trunk`. See our discussion of Subversion's branching and tagging model for the reasoning behind this.

High-Speed Tutorial

> “Please make sure your seat backs are in their full, upright position and that your tray tables are stored. Flight attendants, prepare for take-off….”

What follows is a quick tutorial that walks you through some basic Subversion configuration and operation. When you finish it, you should have a general understanding of Subversion's typical usage.

The examples used in this appendix assume that you have **svn**, the Subversion command-line client, and **svnadmin**, the administrative tool, ready to go on a Unix-like operating system. (This tutorial also works at the Windows command-line prompt, assuming you make some obvious tweaks.) We also assume you are using Subversion 1.2 or later (run **`svn --version`** to check).

Subversion stores all versioned data in a central repository. To begin, create a new repository:

```
$ cd /var/svn
$ svnadmin create repos
$ ls repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
$
```

This command creates a Subversion repository in the directory `/var/svn/repos`, creating the `repos` directory itself if it doesn't already exist. This directory contains (among other things) a collection of database files. You won't see your versioned files if you peek inside. For more information about repository creation and maintenance, see Chapter 5, *Repository Administration*.

Subversion has no concept of a “project.” The repository is just a virtual versioned filesystem, a large tree that can hold anything you wish. Some administrators prefer to store only one project in a repository, and others prefer to store multiple projects in a repository by placing them into separate directories. We discuss the merits of each approach in the section called “Planning Your Repository Organization”. Either way, the repository manages only files and directories, so it's up to humans to interpret particular directories as “projects.” So while you might see references to projects throughout this book, keep in mind that we're only ever talking about some directory (or collection of directories) in the repository.

In this example, we assume you already have some sort of project (a collection of files and directories) that you wish to import into your newly created Subversion repository. Begin by organizing your data into a single directory called `myproject` (or whatever you wish). For reasons explained in Chapter 4, *Branching and Merging*, your project's tree structure should contain three top-level directories named `branches`, `tags`, and `trunk`. The `trunk` directory should contain all of your data, and the `branches` and `tags` directories should be empty:

```
/tmp/
  myproject/
    branches/
    tags/
    trunk/
      foo.c
      bar.c
      Makefile
      …
```

The `branches`, `tags`, and `trunk` subdirectories aren't actually required by Subversion. They're merely a popular convention

that you'll most likely want to use later on.

Once you have your tree of data ready to go, import it into the repository with the **svn import** command (see the section called "Getting Data into Your Repository"):

```
$ svn import /tmp/myproject file:///var/svn/repos/myproject \
      -m "initial import"
Adding         /tmp/myproject/branches
Adding         /tmp/myproject/tags
Adding         /tmp/myproject/trunk
Adding         /tmp/myproject/trunk/foo.c
Adding         /tmp/myproject/trunk/bar.c
Adding         /tmp/myproject/trunk/Makefile
…
Committed revision 1.
$
```

Now the repository contains this tree of data. As mentioned earlier, you won't see your files by directly peeking into the repository; they're all stored within a database. But the repository's imaginary filesystem now contains a top-level directory named `myproject`, which in turn contains your data.

Note that the original `/tmp/myproject` directory is unchanged; Subversion is unaware of it. (In fact, you can even delete that directory if you wish.) To start manipulating repository data, you need to create a new "working copy" of the data, a sort of private workspace. Ask Subversion to "check out" a working copy of the `myproject/trunk` directory in the repository:

```
$ svn checkout file:///var/svn/repos/myproject/trunk myproject
A    myproject/foo.c
A    myproject/bar.c
A    myproject/Makefile
…
Checked out revision 1.
$
```

Now you have a personal copy of part of the repository in a new directory named `myproject`. You can edit the files in your working copy and then commit those changes back into the repository.

- Enter your working copy and edit a file's contents.
- Run **`svn diff`** to see unified diff output of your changes.
- Run **`svn commit`** to commit the new version of your file to the repository.
- Run **`svn update`** to bring your working copy "up to date" with the repository.

For a full tour of all the things you can do with your working copy, read Chapter 2, *Basic Usage*.

At this point, you have the option of making your repository available to others over a network. See Chapter 6, *Server Configuration* to learn about the different sorts of server processes available and how to configure them.

Appendix B. Subversion for CVS Users

This appendix is a guide for CVS users new to Subversion. It's essentially a list of differences between the two systems as “viewed from 10,000 feet.” For each section, we provide references to relevant chapters when possible.

Although the goal of Subversion is to take over the current and future CVS user base, some new features and design changes were required to fix certain “broken” behaviors that CVS had. This means that, as a CVS user, you may need to break habits—ones that you forgot were odd to begin with.

Revision Numbers Are Different Now

In CVS, revision numbers are per file. This is because CVS stores its data in RCS files; each file has a corresponding RCS file in the repository, and the repository is roughly laid out according to the structure of your project tree.

In Subversion, the repository looks like a single filesystem. Each commit results in an entirely new filesystem tree; in essence, the repository is an array of trees. Each of these trees is labeled with a single revision number. When someone talks about “revision 54”, he's talking about a particular tree (and indirectly, the way the filesystem looked after the 54th commit).

Technically, it's not valid to talk about “revision 5 of `foo.c`.” Instead, one would say “`foo.c` as it appears in revision 5.” Also, be careful when making assumptions about the evolution of a file. In CVS, revisions 5 and 6 of `foo.c` are always different. In Subversion, it's most likely that `foo.c` did *not* change between revisions 5 and 6.

Similarly, in CVS, a tag or branch is an annotation on the file or on the version information for that individual file, whereas in Subversion, a tag or branch is a copy of an entire tree (by convention, into the `/branches` or `/tags` directories that appear at the top level of the repository, beside `/trunk`). In the repository as a whole, many versions of each file may be visible: the latest version on each branch, every tagged version, and of course the latest version on the trunk itself. So, to refine the terms even further, one would often say “`foo.c` as it appears in `/branches/REL1` in revision 5.”

For more details on this topic, see the section called “Revisions”.

Directory Versions

Subversion tracks tree structures, not just file contents. It's one of the biggest reasons Subversion was written to replace CVS.

Here's what this means to you, as a former CVS user:

- The **svn add** and **svn delete** commands work on directories now, just as they work on files. So do **svn copy** and **svn move**. However, these commands do *not* cause any kind of immediate change in the repository. Instead, the working items are simply “scheduled” for addition or deletion. No repository changes happen until you run **`svn commit`**.
- Directories aren't dumb containers anymore; they have revision numbers like files. (Or more properly, it's correct to talk about “directory `foo/` in revision 5.”)

Let's talk more about that last point. Directory versioning is a hard problem; because we want to allow mixed-revision working copies, there are some limitations on how far we can abuse this model.

From a theoretical point of view, we define “revision 5 of directory `foo`” to mean a specific collection of directory entries and properties. Now suppose we start adding and removing files from `foo`, and then commit. It would be a lie to say that we still have revision 5 of `foo`. However, if we bumped `foo`'s revision number after the commit, that would be a lie too; there may be other changes to `foo` we haven't yet received, because we haven't updated yet.

Subversion deals with this problem by quietly tracking committed adds and deletes in the `.svn` area. When you eventually run **`svn update`**, all accounts are settled with the repository, and the directory's new revision number is set correctly. *Therefore, only*

after an update is it truly safe to say that you have a "perfect" revision of a directory. Most of the time, your working copy will contain "imperfect" directory revisions.

Similarly, a problem arises if you attempt to commit property changes on a directory. Normally, the commit would bump the working directory's local revision number. But again, that would be a lie, as there may be adds or deletes that the directory doesn't yet have, because no update has happened. *Therefore, you are not allowed to commit property changes on a directory unless the directory is up to date.*

For more discussion about the limitations of directory versioning, see the section called "Mixed-revision working copies".

More Disconnected Operations

In recent years, disk space has become outrageously cheap and abundant, but network bandwidth has not. Therefore, the Subversion working copy has been optimized around the scarcer resource.

The `.svn` administrative directory serves the same purpose as the `CVS` directory, except that it also stores read-only, "pristine" copies of your files. This allows you to do many things offline:

svn status
: Shows you any local changes you've made (see the section called "See an overview of your changes")

svn diff
: Shows you the details of your changes (see the section called "Examine the details of your local modifications")

svn revert
: Removes your local changes (see the section called "Fix Your Mistakes")

Also, the cached pristine files allow the Subversion client to send differences when committing, which CVS cannot do.

The last subcommand in the list—**svn revert**—is new. It will not only remove local changes, but also unschedule operations such as adds and deletes. Although deleting the file and then running **`svn update`** will still work, doing so distorts the true purpose of updating. And, while we're on this subject…

Distinction Between Status and Update

Subversion attempts to erase a lot of the confusion between the **cvs status** and **cvs update** commands.

The **cvs status** command has two purposes: first, to show the user any local modifications in the working copy, and second, to show the user which files are out of date. Unfortunately, because of CVS's hard-to-read status output, many CVS users don't take advantage of this command at all. Instead, they've developed a habit of running **`cvs update`** or **`cvs -n update`** to quickly see their changes. If users forget to use the `-n` option, this has the side effect of merging repository changes they may not be ready to deal with.

Subversion removes this muddle by making the output of **svn status** easy to read for both humans and parsers. Also, **svn update** prints only information about files that are updated, *not* local modifications.

Status

svn status prints all files that have local modifications. By default, the repository is not contacted. While this subcommand accepts a fair number of options, the following are the most commonly used ones:

`-u`
: Contact the repository to determine, and then display, out-of-dateness information.

`-v`
: Show *all* entries under version control.

`-N`
: Run nonrecursively (do not descend into subdirectories).

The **svn status** command has two output formats. In the default “short” format, local modifications look like this:

```
$ svn status
M       foo.c
M       bar/baz.c
```

If you specify the `--show-updates` (`-u`) option, a longer output format is used:

```
$ svn status -u
M             1047   foo.c
        *     1045   faces.html
        *            bloo.png
M             1050   bar/baz.c
Status against revision:   1066
```

In this case, two new columns appear. The second column contains an asterisk if the file or directory is out of date. The third column shows the working copy's revision number of the item. In the previous example, the asterisk indicates that `faces.html` would be patched if we updated, and that `bloo.png` is a newly added file in the repository. (The absence of any revision number next to `bloo.png` means that it doesn't yet exist in the working copy.)

For a more detailed discussion of **svn status**, including an explanation of the status codes shown in the previous example, see the section called “See an overview of your changes”.

Update

svn update updates your working copy, and prints only information about files that it updates.

Subversion has combined CVS's `P` and `U` codes into just `U`. When a merge or conflict occurs, Subversion simply prints `G` or `C`, rather than a whole sentence about it.

For a more detailed discussion of **svn update**, see the section called “Update Your Working Copy”.

Branches and Tags

Subversion doesn't distinguish between filesystem space and “branch” space; branches and tags are ordinary directories within the filesystem. This is probably the single biggest mental hurdle that a CVS user will need to cross. Read all about it in Chapter 4, *Branching and Merging*.

Since Subversion treats branches and tags as ordinary directories, your project's various lines of development probably live in subdirectories of the main project directory. So remember to check out using the URL of the subdirectory that contains the particular line of development you want, not the project's root URL. If you make the mistake of checking out the root of the project, you may very well wind up with a working copy that contains a complete copy of your project's content for each and every one of its branches and tags. [1]

Metadata Properties

A new feature of Subversion is that you can attach arbitrary metadata (or "properties") to files and directories. Properties are arbitrary name/value pairs associated with files and directories in your working copy.

To set or get a property name, use the **svn propset** and **svn propget** subcommands. To list all properties on an object, use **svn proplist**.

For more information, see the section called "Properties".

Conflict Resolution

CVS marks conflicts with inline "conflict markers," and then prints a `C` during an update or merge operation. Historically, this has caused problems, because CVS isn't doing enough. Many users forget about (or don't see) the `C` after it whizzes by on their terminal. They often forget that the conflict markers are even present, and then accidentally commit files containing those conflict markers.

Subversion solves this problem in a pair of ways. First, when a conflict occurs in a file, Subversion records the fact that the file is in a state of conflict, and won't allow you to commit changes to that file until you explicitly resolve the conflict. Second, Subversion provides interactive conflict resolution, which allows you to resolve conflicts as they happen instead of having to go back and do so after the update or merge operation completes. See the section called "Resolve Any Conflicts" for more about conflict resolution in Subversion.

Binary Files and Translation

In the most general sense, Subversion handles binary files more gracefully than CVS does. Because CVS uses RCS, it can only store successive full copies of a changing binary file. Subversion, however, expresses differences between files using a binary differencing algorithm, regardless of whether they contain textual or binary data. That means all files are stored differentially (compressed) in the repository.

CVS users have to mark binary files with `-kb` flags to prevent data from being garbled (due to keyword expansion and line-ending translations). They sometimes forget to do this.

Subversion takes the more paranoid route. First, it never performs any kind of keyword or line-ending translation unless you explicitly ask it to do so (see the section called "Keyword Substitution" and the section called "End-of-Line Character Sequences" for more details). By default, Subversion treats all file data as literal byte strings, and files are always stored in the repository in an untranslated state.

Second, Subversion maintains an internal notion of whether a file is "text" or "binary" data, but this notion is *only* extant in the working copy. During an **svn update**, Subversion will perform contextual merges on locally modified text files, but will not attempt to do so for binary files.

To determine whether a contextual merge is possible, Subversion examines the `svn:mime-type` property. If the file has no `svn:mime-type` property, or has a MIME type that is textual (e.g., `text/*`), Subversion assumes it is text. Otherwise, Subversion assumes the file is binary. Subversion also helps users by running a binary-detection algorithm in the **svn import** and **svn add** commands. These commands will make a good guess and then (possibly) set a binary `svn:mime-type` property on the file being added. (If Subversion guesses wrong, the user can always remove or hand-edit the property.)

Versioned Modules

Unlike CVS, a Subversion working copy is aware that it has checked out a module. That means if somebody changes the definition of a module (e.g., adds or removes components), a call to **svn update** will update the working copy appropriately, adding and removing components.

[1] That is, providing you don't run out of disk space before your checkout finishes.

Subversion defines modules as a list of directories within a directory property; see the section called “Externals Definitions”.

Authentication

With CVS's pserver, you are required to log in to the server (using the **cvs login** command) before performing any read or write operation—you sometimes even have to log in for anonymous operations. With a Subversion repository using Apache **httpd** or **svnserve** as the server, you don't provide any authentication credentials at the outset—if an operation that you perform requires authentication, the server will challenge you for your credentials (whether those credentials are username and password, a client certificate, or even both). So if your repository is world-readable, you will not be required to authenticate at all for read operations.

As with CVS, Subversion still caches your credentials on disk (in your `~/.subversion/auth/` directory) unless you tell it not to by using the `--no-auth-cache` option.

The exception to this behavior, however, is in the case of accessing an **svnserve** server over an SSH tunnel, using the `svn+ssh://` URL scheme. In that case, the **ssh** program unconditionally demands authentication just to start the tunnel.

Converting a Repository from CVS to Subversion

Perhaps the most important way to familiarize CVS users with Subversion is to let them continue to work on their projects using the new system. And while that can be somewhat accomplished using a flat import into a Subversion repository of an exported CVS repository, the more thorough solution involves transferring not just the latest snapshot of their data, but all the history behind it as well, from one system to another. This is an extremely difficult problem to solve; it involves deducing changesets in the absence of atomicity and translating between the systems' completely orthogonal branching policies, among other complications. Still, a handful of tools claim to at least partially support the ability to convert existing CVS repositories into Subversion ones.

The most popular (and mature) conversion tool is cvs2svn (http://cvs2svn.tigris.org/), a Python program originally created by members of Subversion's own development community. This tool is meant to run exactly once: it scans your CVS repository multiple times and attempts to deduce commits, branches, and tags as best it can. When it finishes, the result is either a Subversion repository or a portable Subversion dump file representing your code's history. See the web site for detailed instructions and caveats.

Appendix C. WebDAV and Autoversioning

WebDAV is an extension to HTTP, and it is growing more and more popular as a standard for file sharing. Today's operating systems are becoming extremely web-aware, and many now have built-in support for mounting “shares” exported by WebDAV servers.

If you use Apache as your Subversion network server, to some extent you are also running a WebDAV server. This appendix gives some background on the nature of this protocol, how Subversion uses it, and how well Subversion interoperates with other software that is WebDAV-aware.

What Is WebDAV?

DAV stands for “Distributed Authoring and Versioning.” RFC 2518 defines a set of concepts and accompanying extension methods to HTTP 1.1 that make the Web a more universal read/write medium. The basic idea is that a WebDAV-compliant web server can act like a generic file server; clients can “mount” shared folders over HTTP that behave much like other network filesystems (such as NFS or SMB).

The tragedy, though, is that despite the acronym, the RFC specification doesn't actually describe any sort of version control. Basic WebDAV clients and servers assume that only one version of each file or directory exists, and that it can be repeatedly overwritten.

Because RFC 2518 left out versioning concepts, another committee was left with the responsibility of writing RFC 3253 a few years later. The new RFC adds versioning concepts to WebDAV, placing the “V” back in “DAV”—hence the term “DeltaV.” WebDAV/DeltaV clients and servers are often called just “DeltaV” programs, since DeltaV implies the existence of basic WebDAV.

The original WebDAV standard has been widely successful. Every modern computer operating system has a general WebDAV client built in (details to follow), and a number of popular standalone applications are also able to speak WebDAV—Microsoft Office, Dreamweaver, and Photoshop, to name a few. On the server end, Apache HTTP Server has been able to provide WebDAV services since 1998 and is considered the de facto open source standard. Several other commercial WebDAV servers are available, including Microsoft's own IIS.

DeltaV, unfortunately, has not been so successful. It's very difficult to find any DeltaV clients or servers. The few that do exist are relatively unknown commercial products, and thus it's very difficult to test interoperability. It's not entirely clear as to why DeltaV has remained stagnant. Some opine that the specification is just too complex. Others argue that while WebDAV's features have mass appeal (even the least technical users appreciate network file sharing), its version control features just aren't interesting or necessary for most users. Finally, some believe that DeltaV remains unpopular because there's still no open source server product that implements it well.

When Subversion was still in its design phase, it seemed like a great idea to use Apache as a network server. It already had a module to provide WebDAV services. DeltaV was a relatively new specification. The hope was that the Subversion server module (**mod_dav_svn**) would eventually evolve into an open source DeltaV reference implementation. Unfortunately, DeltaV has a very specific versioning model that doesn't quite line up with Subversion's model. Some concepts were mappable; others were not.

What does this mean, then?

First, the Subversion client is not a fully implemented DeltaV client. It needs certain types of things from the server that DeltaV itself cannot provide, and thus is largely dependent on a number of Subversion-specific HTTP `REPORT` requests that only **mod_dav_svn** understands.

Second, **mod_dav_svn** is not a fully realized DeltaV server. Many portions of the DeltaV specification were irrelevant to Subversion, and thus were left unimplemented.

A long-held debate in the Subversion developer community about whether it was worthfile to remedy either of these situations eventually reached closure, with the Subversion developers officially deciding to abandon plans to fully support DeltaV. As of Subversion 1.7, Subversion clients and servers introduce numerous non-standard simplifications of the DeltaV standards[1], with

more customizations of this sort likely to come. Those versions of Subversion will, of course, continue to provide the same DeltaV feature support already present in older releases, but no new work will be done to increase coverage of the specification—Subversion is intentionally moving away from strict DeltaV as its primary HTTP-based protocol.

Autoversioning

While the Subversion client is not a full DeltaV client, and the Subversion server is not a full DeltaV server, there's still a glimmer of WebDAV interoperability to be happy about: *autoversioning*.

Autoversioning is an optional feature defined in the DeltaV standard. A typical DeltaV server will reject an ignorant WebDAV client attempting to do a `PUT` to a file that's under version control. To change a version-controlled file, the server expects a series of proper versioning requests: something like `MKACTIVITY`, `CHECKOUT`, `PUT`, `CHECKIN`. But if the DeltaV server supports autoversioning, write requests from basic WebDAV clients are accepted. The server behaves as though the client *had* issued the proper series of versioning requests, performing a commit under the hood. In other words, it allows a DeltaV server to interoperate with ordinary WebDAV clients that don't understand versioning.

Because so many operating systems already have integrated WebDAV clients, the use case for this feature can be incredibly appealing to administrators working with non-technical users. Imagine an office of ordinary users running Microsoft Windows or Mac OS. Each user “mounts” the Subversion repository, which appears to be an ordinary network folder. They use the shared folder as they always do: open files, edit them, and save them. Meanwhile, the server is automatically versioning everything. Any administrator (or knowledgeable user) can still use a Subversion client to search history and retrieve older versions of data.

This scenario isn't fiction—it's real and it works. To activate autoversioning in **mod_dav_svn**, use the `SVNAutoversioning` directive within the `httpd.conf` `Location` block, like so:

```
<Location /repos>
  DAV svn
  SVNPath /var/svn/repository
  SVNAutoversioning on
</Location>
```

When Subversion autoversioning is active, write requests from WebDAV clients result in automatic commits. A generic log message is automatically generated and attached to each revision.

Before activating this feature, however, understand what you're getting into. WebDAV clients tend to do *many* write requests, resulting in a huge number of automatically committed revisions. For example, when saving data, many clients will do a `PUT` of a 0-byte file (as a way of reserving a name) followed by another `PUT` with the real file data. The single file-write results in two separate commits. Also consider that many applications auto-save every few minutes, resulting in even more commits.

If you have a post-commit hook program that sends email, you may want to disable email generation either altogether or on certain sections of the repository; it depends on whether you think the influx of emails will still prove to be valuable notifications or not. Also, a smart post-commit hook program can distinguish between a transaction created via autoversioning and one created through a normal Subversion commit operation. The trick is to look for a revision property named `svn:autoversioned`. If present, the commit was made by a generic WebDAV client.

Another feature that may be a useful complement for Subversion's autoversioning comes from Apache's `mod_mime` module. If a WebDAV client adds a new file to the repository, there's no opportunity for the user to set the the `svn:mime-type` property. This might cause the file to appear as a generic icon when viewed within a WebDAV shared folder, not having an association with any application. One remedy is to have a sysadmin (or other Subversion-knowledgeable person) check out a working copy and manually set the `svn:mime-type` property on necessary files. But there's potentially no end to such cleanup tasks. Instead, you can use the `ModMimeUsePathInfo` directive in your Subversion `<Location>` block:

[1]The Subversion developers colloquially refer to this deviation from the DeltaV standard as “HTTPv2”.

```
<Location /repos>
  DAV svn
  SVNPath /var/svn/repository
  SVNAutoversioning on

  ModMimeUsePathInfo on

</Location>
```

This directive allows `mod_mime` to attempt automatic deduction of the MIME type on new files that enter the repository via autoversioning. The module looks at the file's named extension and possibly the contents as well; if the file matches some common patterns, the file's `svn:mime-type` property will be set automatically.

Client Interoperability

All WebDAV clients fall into one of three categories—standalone applications, file-explorer extensions, or filesystem implementations. These categories broadly define the types of WebDAV functionality available to users. Table C.1, "Common WebDAV clients" gives our categorization as well as a quick description of some common pieces of WebDAV-enabled software. You can find more details about these software offerings, as well as their general category, in the sections that follow.

Table C.1. Common WebDAV clients

Software	Type	Windows	Mac	Linux	Description
Adobe Photoshop	Standalone WebDAV application	X			Image editing software, allowing direct opening from, and writing to, WebDAV URLs
cadaver	Standalone WebDAV application		X	X	Command-line WebDAV client supporting file transfer, tree, and locking operations
DAV Explorer	Standalone WebDAV application	X	X	X	Java GUI tool for exploring WebDAV shares
Adobe Dreamweaver	Standalone WebDAV application	X			Web production software able to directly read from and write to WebDAV URLs
Microsoft Office	Standalone WebDAV application	X			Office productivity suite with several components able to directly read from and write to WebDAV URLs
Microsoft Web Folders	File-explorer WebDAV extension	X			GUI file explorer program able to perform tree operations on a WebDAV share

Software	Type	Windows	Mac	Linux	Description
GNOME Nautilus	File-explorer WebDAV extension			X	GUI file explorer able to perform tree operations on a WebDAV share
KDE Konqueror	File-explorer WebDAV extension			X	GUI file explorer able to perform tree operations on a WebDAV share
Mac OS X	WebDAV filesystem implementation		X		Operating system that has built-in support for mounting WebDAV shares.
Novell NetDrive	WebDAV filesystem implementation	X			Drive-mapping program for assigning Windows drive letters to a mounted remote WebDAV share
SRT WebDrive	WebDAV filesystem implementation	X			File transfer software, which, among other things, allows the assignment of Windows drive letters to a mounted remote WebDAV share
davfs2	WebDAV filesystem implementation			X	Linux filesystem driver that allows you to mount a WebDAV share

Standalone WebDAV Applications

A WebDAV application is a program that speaks WebDAV protocols with a WebDAV server. We'll cover some of the most popular programs with this kind of WebDAV support.

Microsoft Office, Dreamweaver, Photoshop

On Windows, several well-known applications contain integrated WebDAV client functionality, such as Microsoft's Office,[2] Adobe's Photoshop and Dreamweaver programs. They're able to directly open and save to URLs, and tend to make heavy use of WebDAV locks when editing a file.

Note that while many of these programs also exist for Mac OS X, they do not appear to support WebDAV directly on that platform. In fact, on Mac OS X, the File#Open dialog box doesn't allow one to type a path or URL at all. It's likely that the WebDAV features were deliberately left out of Macintosh versions of these programs, since OS X already provides such excellent low-level filesystem support for WebDAV.

cadaver, DAV Explorer

cadaver is a bare-bones Unix command-line program for browsing and changing WebDAV shares. Like the Subversion client, it uses the neon HTTP library—not surprisingly, since both neon and cadaver are written by the same author. cadaver is free software

[2]WebDAV support was removed from Microsoft Access for some reason, but it exists in the rest of the Office suite.

(GPL license) and is available at http://www.webdav.org/cadaver/.

Using cadaver is similar to using a command-line FTP program, and thus it's extremely useful for basic WebDAV debugging. It can be used to upload or download files in a pinch, to examine properties, and to copy, move, lock, or unlock files:

```
$ cadaver http://host/repos
dav:/repos/> ls
Listing collection `/repos/': succeeded.
Coll: > foobar                                0  May 10 16:19
      > playwright.el                      2864  May  4 16:18
      > proofbypoem.txt                    1461  May  5 15:09
      > westcoast.jpg                     66737  May  5 15:09

dav:/repos/> put README
Uploading README to `/repos/README':
Progress: [=============================>] 100.0% of 357 bytes succeeded.

dav:/repos/> get proofbypoem.txt
Downloading `/repos/proofbypoem.txt' to proofbypoem.txt:
Progress: [=============================>] 100.0% of 1461 bytes succeeded.
```

DAV Explorer is another standalone WebDAV client, written in Java. It's under a free Apache-like license and is available at http://www.ics.uci.edu/~webdav/. It does everything cadaver does, but has the advantages of being portable and being a more user-friendly GUI application. It's also one of the first clients to support the new WebDAV Access Control Protocol (RFC 3744).

Of course, DAV Explorer's ACL support is useless in this case, since **mod_dav_svn** doesn't support it. The fact that both cadaver and DAV Explorer support some limited DeltaV commands isn't particularly useful either, since they don't allow `MKACTIVITY` requests. But it's not relevant anyway; we're assuming all of these clients are operating against an autoversioning repository.

File-Explorer WebDAV Extensions

Some popular file explorer GUI programs support WebDAV extensions that allow a user to browse a DAV share as though it was just another directory on the local computer, and to perform basic tree editing operations on the items in that share. For example, Windows Explorer is able to browse a WebDAV server as a “network place.” Users can drag files to and from the desktop, or can rename, copy, or delete files in the usual way. But because it's only a feature of the file explorer, the DAV share isn't visible to ordinary applications. All DAV interaction must happen through the explorer interface.

Microsoft Web Folders

Microsoft was one of the original backers of the WebDAV specification, and first started shipping a client in Windows 98, which was known as Web Folders. This client was also shipped in Windows NT 4.0 and Windows 2000.

The original Web Folders client was an extension to Explorer, the main GUI program used to browse filesystems. It works well enough. In Windows 98, the feature might need to be explicitly installed if Web Folders aren't already visible inside My Computer. In Windows 2000, simply add a new “network place,” enter the URL, and the WebDAV share will pop up for browsing.

With the release of Windows XP, Microsoft started shipping a new implementation of Web Folders, known as the WebDAV Mini-Redirector. The new implementation is a filesystem-level client, allowing WebDAV shares to be mounted as drive letters. Unfortunately, this implementation is incredibly buggy. The client usually tries to convert HTTP URLs (`http://host/repos`) into UNC share notation (`\\host\repos`); it also often tries to use Windows Domain authentication to respond to basic-auth HTTP challenges, sending usernames as `HOST\username`. These interoperability problems are severe and are documented in numerous places around the Web, to the frustration of many users. Even Greg Stein, the original author of Apache's WebDAV module, bluntly states that XP Web Folders simply can't operate against an Apache server.

Windows Vista's initial implementation of Web Folders seems to be almost the same as XP's, so it has the same sort of problems.

With luck, Microsoft will remedy these issues in a Vista Service Pack.

However, there seem to be workarounds for both XP and Vista that allow Web Folders to work against Apache. Users have mostly reported success with these techniques, so we'll relay them here.

On Windows XP, you have two options. First, search Microsoft's web site for update KB907306, “Software Update for Web Folders.” This may fix all your problems. If it doesn't, it seems that the original pre-XP Web Folders implementation is still buried within the system. You can unearth it by going to Network Places and adding a new network place. When prompted, enter the URL of the repository, but *include a port number* in the URL. For example, you should enter **`http://host/repos`** as **`http://host:80/repos`** instead. Respond to any authentication prompts with your Subversion credentials.

On Windows Vista, the same KB907306 update may clear everything up. But there may still be other issues. Some users have reported that Vista considers all `http://` connections insecure, and thus will always fail any authentication challenges from Apache unless the connection happens over `https://`. If you're unable to connect to the Subversion repository via SSL, you can tweak the system registry to turn off this behavior. Just change the value of the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\WebClient\Parameters\BasicAuthLevel` key from **`1`** to **`2`**. A final warning: be sure to set up the Web Folder to point to the repository's root directory (`/`), rather than some subdirectory such as `/trunk`. Vista Web Folders seems to work only against repository roots.

In general, while these workarounds may function for you, you might get a better overall experience using a third-party WebDAV client such as WebDrive or NetDrive.

Nautilus, Konqueror

Nautilus is the official file manager/browser for the GNOME desktop (http://www.gnome.org), and Konqueror is the manager/browser for the KDE desktop (http://www.kde.org). Both of these applications have an explorer-level WebDAV client built in, and they operate just fine against an autoversioning repository.

In GNOME's Nautilus, select the File#Open location menu item and enter the URL in the dialog box presented. The repository should then be displayed like any other filesystem.

In KDE's Konqueror, you need to use the `webdav://` scheme when entering the URL in the location bar. If you enter an `http://` URL, Konqueror will behave like an ordinary web browser. You'll likely see the generic HTML directory listing produced by **mod_dav_svn**. When you enter **`webdav://host/repos`** instead of **`http://host/repos`**, Konqueror becomes a WebDAV client and displays the repository as a filesystem.

WebDAV Filesystem Implementation

The WebDAV filesystem implementation is arguably the best sort of WebDAV client. It's implemented as a low-level filesystem module, typically within the operating system's kernel. This means that the DAV share is mounted like any other network filesystem, similar to mounting an NFS share on Unix or attaching an SMB share as a drive letter in Windows. As a result, this sort of client provides completely transparent read/write WebDAV access to all programs. Applications aren't even aware that WebDAV requests are happening.

WebDrive, NetDrive

Both WebDrive and NetDrive are excellent commercial products that allow a WebDAV share to be attached as drive letters in Windows. As a result, you can operate on the contents of these WebDAV-backed pseudodrives as easily as you can against real local hard drives, and in the same ways. You can purchase WebDrive from South River Technologies (http://www.southrivertech.com). Novell's NetDrive is freely available online, but requires users to have a NetWare license.

Mac OS X

Apple's OS X operating system has an integrated filesystem-level WebDAV client. From the Finder, select the Go#Connect to Server menu item. Enter a WebDAV URL, and it appears as a disk on the desktop, just like any other mounted volume. You can also mount a WebDAV share from the Darwin terminal by using the `webdav` filesystem type with the **mount** command:

```
$ mount -t webdav http://svn.example.com/repos/project /some/mountpoint
$
```

Note that if your **mod_dav_svn** is older than version 1.2, OS X will refuse to mount the share as read/write; it will appear as read-only. This is because OS X insists on locking support for read/write shares, and the ability to lock files first appeared in Subversion 1.2.

Also, OS X's WebDAV client can sometimes be overly sensitive to HTTP redirects. If OS X is unable to mount the repository at all, you may need to enable the `BrowserMatch` directive in the Apache server's `httpd.conf`:

```
BrowserMatch "^WebDAVFS/1.[012]" redirect-carefully
```

Linux davfs2

Linux davfs2 is a filesystem module for the Linux kernel, whose development is organized at http://dav.sourceforge.net/. Once you install davfs2, you can mount a WebDAV network share using the usual Linux mount command:

```
$ mount.davfs http://host/repos /mnt/dav
```

Appendix D. Copyright

Copyright (c) 2002-2013 Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

A summary of the license is given below, followed by the full legal text.

```
You are free:

    * to copy, distribute, display, and perform the work
    * to make derivative works
    * to make commercial use of the work

Under the following conditions:

Attribution. You must give the original author credit.

    * For any reuse or distribution, you must make clear to others the
      license terms of this work.

    * Any of these conditions can be waived if you get permission from
      the author.

Your fair use and other rights are in no way affected by the above.

The above is a summary of the full license below.

====================================================================

Creative Commons Legal Code
Attribution 2.0

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES
REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR
DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS
CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS
PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE
WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS
PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND
AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS
YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF
SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Collective Work" means a work, such as a periodical issue,
      anthology or encyclopedia, in which the Work in its entirety in
      unmodified form, along with a number of other contributions,
      constituting separate and independent works in themselves, are
```

assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

b. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

c. "Licensor" means the individual or entity that offers the Work under the terms of this License.

d. "Original Author" means the individual or entity who created the Work.

e. "Work" means the copyrightable work of authorship offered under the terms of this License.

f. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

b. to create and reproduce Derivative Works;

c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e.

For the avoidance of doubt, where the work is a musical composition:

i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society

(e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such

URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable

under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, neither party will use the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.

==========

Index

A
ancestry, 129

B
BASE, 49
branches, 17, 99

C
checkout (see working copy, creation)
CollabNet, xv
COMMITTED, 50
committing (see working copy, commit)
Concurrent Versions System, xiv
conflicts, 6
  conflict markers, 29
  resolution, 32
    interactive, 28
    manual, 30
    postponing, 29
  resolving, 26
  reviewing, 28
CVS (see Concurrent Versions System)

D
delta, 22

H
HEAD, 49

L
log message, 22

M
merge tracking, 105
mergeinfo, 106
mergeinfo elision, 110
mergeinfo inheritance, 114
mod_dav_svn, xviii

P
patches, 25
PREV, 50
project root, 17, 147
properties, 55

R
repository
  defined, 1
  hooks
    post-commit, 418
    post-lock, 422
    post-revprop-change, 420
    post-unlock, 424
    pre-commit, 417
    pre-lock, 421
    pre-revprop-change, 419
    pre-unlock, 423
    start-commit, 416
revisions
  as dates, 50
  defined, 7
  global, 8
  keywords, 49
    BASE, 49
    COMMITTED, 50
    HEAD, 49
    PREV, 50

S
SCM (see software configuration management)
software configuration management, xiv
subtree merge, 110
subtree mergeinfo, 110
Subversion
  architecture, xvi
  components, xviii
  defined, xiv
  history of, xv, xviii
svn, xviii
  options, 16
  subcommands
    add, 21, 263
    blame, 265
    cat, 267
    changelist, 269
    checkout, 11, 18, 270
    cleanup, 274
    commit, 12, 275
    copy, 21, 277
    delete, 21, 280
    diff, 24, 282
    export, 286
    help, 15, 288
    import, 16, 289
    info, 291
    list, 294
    lock, 296
    log, 297
    merge, 302
    mergeinfo, 305
    mkdir, 21, 306
    move, 21, 307
    patch, 309
    propdel, 312
    propedit, 313
    propget, 314
    proplist, 316
    propset, 318

relocate, 320
resolve, 323
resolved, 324
revert, 25, 325
status, 22, 327
switch, 332
unlock, 334
update, 12, 20, 335
upgrade, 338
syntax
URLs, 8
svnadmin, xviii
subcommands
crashtest, 341
create, 342
deltify, 343
dump, 344
help, 346
hotcopy, 347
list-dblogs, 348
list-unused-dblogs, 349
load, 350
lslocks, 352
lstxns, 353
pack, 354
recover, 355
rmlocks, 357
rmtxns, 358
setlog, 359
setrevprop, 360
setuuid, 361
upgrade, 362
verify, 363
svndumpfilter, xviii
subcommands
exclude, 403
help, 407
include, 405
svnlook, xviii
subcommands
author, 366
cat, 367
changed, 368
date, 370
diff, 371
dirs-changed, 373
filesize, 374
help, 375
history, 376
info, 377
lock, 378
log, 379
propget, 380
proplist, 381
tree, 382
uuid, 384
youngest, 385
svnrdump
subcommands
dump, 397
help, 398
load, 399
svnserve, xviii
svnsync, xviii
subcommands
copy-revprops, 388
help, 390
info, 391
initialize, 392
synchronize, 394
syntax
URLs, 8
svnversion, xviii, 408

T

tags, 17, 134
text-base, 22
trunk, 17

U

unified diff, 24
updating (see working copy, update)

V

VCS (see version control systems)
version control
models
copy-modify-merge, 4
lock-modify-unlock, 3
version control systems, xiv, 1

W

working copy
commit, 12
creation, 11
defined, 2, 9
mixed-revision, 12
update, 12

System Testing

Though defect prevention techniques can be quite effective, and are to be encouraged, defects can and will be introduced in a variety of different ways throughout design and construction, regardless of the process used, whether traditional or agile. Hence software testing, which makes use of the product while it is in operation or being operated on, is an essential part of all defect elimination efforts. As discussed in the chapter on Software Quality Assurance, software testing attempts to create trigger conditions (inputs) that will give rise to failures indicating the existence of faults that can be isolated and eliminated. What differentiates system testing from unit testing and integration testing is that the product is treated as if it were atomic, that is, as if it can't be decomposed into constituent parts.

We begin with an overview of testing and a comparison of system testing and sub-system testing (unit testing and integration testing are discussed in the chapter on quality assurance in construction). We then discuss different aspects of system testing in greater detail. Next, we consider system testing in context. That is, we consider how system testing is used in traditional processes and agile processes. We then consider different system testing tools and how design and construction practices can facilitate system testing. We conclude with a discussion of various organizational issues related to system testing.

Overview

In the chapter on Software Quality Assurance, testing was divided into four stages—unit testing, integration testing, alpha testing, and beta testing, with the last two collectively referred to as system testing. To properly discuss system testing, we need a somewhat less abstract view.

As discussed before, testing often begins during the design process (with the creation of unit tests) and certainly begins during construction with the creation and execution of unit tests and integration tests. Unit testing happens in parallel with construction. That is, each developer tests the units that he or she is implementing.

After a sufficient number of units have been constructed and tested, they can be combined into larger sub-systems (sometimes called *assemblages*) and tested together. Conceptually, integration testing begins after unit testing is completed but, in practice, there is considerable iteration and it is often difficult to distinguish one from the other.

After integration testing has been completed, (an increment of) the product can be built from the supposedly fully-tested sub-systems. Despite the fact that the sub-systems are fully tested, there is no reason to expect that the product will not fail. There are several reasons for this. First, the sub-systems could still contain faults. Second, there could be faults at the interfaces between the different sub-systems. Third, because the operation of the sub-systems often depends on the state of the system, it may not have been possible to truly fully test the sub-systems. Hence, system testing is needed.

The first stage of system testing is normally referred to as **alpha testing**, in which developers test an entire product. During this stage, developers simulate the behavior of actual users. Alpha testing involves both functional and non-functional tests, which often can be executed in parallel.

The second stage of system testing is normally referred to as **beta testing**, which involves actual users testing the product in either a simulated environment or the actual environment. Like alpha testing, beta testing involves both functional and non-functional tests, which can be executed in parallel.

This process is summarized in the following activity diagram.

One of the things that can be somewhat misleading about this diagram is the end node. While the testing activity does, indeed, end at some point, the *acceptance criteria* used to determine when testing is complete (and the negotiations that often take place) can be difficult to define. Particularly important in this regard is the fact that one fault often hides or obscures other faults, which means that testing must be conducted iteratively. This aspect of test planning will be ignored for now but

considered later. At this point, we focus attention on the specification, description and categorization of tests.

Differences between System and Sub-System Testing

Sub-system testing (unit testing and integration testing) and system testing differ in two important and related ways that are partially evident from this activity diagram. First, as mentioned above, system testing involves (an increment of) the entire product whereas sub-system testing does not. Second, system testing typically involves units or sub-systems that were constructed by several people. Unit tests, on the other hand, are typically created and executed by the developer who constructed the unit.

Combined, these two differences have important implications for the kinds of faults that give rise to the failures detected during system testing. Inputs that can only be handled by the entire system rather than some part of it can lead to failures of non-functional requirements that apply to the entire system, like low reliability, inadequate throughput, or insufficient precision of results. System tests can also reveal problems arising from different people understanding requirements in different way, or making differ assumptions about the system. For example, one part of the sytem may assume that angles are in degrees while another parts assumes they are in radians. Under this assumption, each sub-system could pass very thorough suites of tests, but completely fail system tests.

Sub-system testing and system testing differ in several other important ways that are not evident from this activity diagram. They differ in the kinds of people that are involved (not just the number), the aspects of the design process that are used to create tests, the dependence of tests on the nature of the product and the design and implementation paradigm used, and the relationships between the different individuals that are involved.

- Sub-system testing is performed by people who have intimate knowledge of the engineering design and implementation of the product, and, hence, can include both clear or white-box and black-box techniques. System testing is performed by people who have an intimate knowledge only of the product design (the requirements) and, hence, only involves black-box techniques.
- Sub-system tests are usually created from detailed requirements specifications or detailed user stories (either sprintable stories or user stories about tasks). System tests, on the other hand, are usually created from descriptions of interactions between users and the system (such as use cases or user stories about features). This has implications for the nature of the tests, the kinds of failures that they are likely to trigger, and the state-dependent nature of the tests (that is, the state of the system is likely to change during a test).
- Unit testing and integration testing depend, to some extent, on the nature of the product (for example, a WWW service vs. a desktop application), whether the product is bought and customized or built, and the approach used to either customize or build the product (for example, object-oriented vs. functional). By the time system testing begins, however, these differences no longer matter (or, at least, matter much less). Regardless of the way in which the product arose, it is going to be tested in essentially the same way.
- Unit testing is more of an individual activity than system testing, which necessarily involves the entire team or a separate testing team. Hence, the problem of finger pointing is less likely to arise in unit testing and more likely to arise in system testing. For example, in system testing opportunities arise for the software people to blame the hardware people, the GUI people to

blame the database people, etc. Obviously, this kind of finger pointing can be very damaging, and steps must be taken to avoid it.

Details of System Testing

System testing can be considered at many different levels of abstraction. In the chapter on quality assurance it was divided into two components, alpha testing and bet testing. In the activity diagram above, it is divided into six different components. This section looks inside each of these six components to provide additional detail. Despite this, the discussion that follows is still an abstraction and considerable detail is omitted.

The differences and similarities between alpha testing, beta acceptance testing, and beta installation testing are summarized in the following table and discussed more fully below.

	Alpha Testing	**Beta Testing**	
		Acceptance Testing	**Installation Testing**
Personnel	Testers	Users	Users
Environment	Controlled	Controlled	Uncontrolled
Purpose	Validation (Indirect) & Verification	Validation (Direct)	Verification
Recording	Extensive Logging	Limited Logging	Limited Logging

Alpha Testing

Alpha tests are used both to validate that a product satisfies the needs and desires of users and to verify that it does so correctly. However, validation is often indirect in that alpha testing does not involve real users. Instead, it typically involves a group of testers attempting to behave like real users. In other words, alpha testing involves internal testers who are part of the organization developing the software product, either structurally or contractually. Hence, they rely on product specifications (for example, the software requirements specification) to validate that the needs and desires of the real users are satisfied.

In many organizations, an attempt is made to create an independent test team that organizes, creates and runs alpha tests. The rationale is that the designers, developers, and authors may make inappropriate assumptions (for example, that what they have implemented is what was required). They may also, knowingly or unknowingly, have conflicts of interest. That is, it may not be in their best interest to identify failures; indeed they have a vested interest in demonstrating that the product is fault free.[1] The test team may include some product designers or requirements analysts so that their expertise about user needs can be incorporated. Finally the test team might include designers or developers involved with the engineering and construction of the product to take advantage of their expertise. In some cases, developers are also needed to create some tests, depending on the type of testing tools used. Obviously, in such situations it is important to take steps to ensure that the testing team and the development team do not become adversaries.

Alpha testing is conducted in a controlled environment, though one that attempts to simulate the real environment in which the product will be used. This helps testers isolate and record failures. Though the testers are responsible for recording failures, the product itself will often keep an

1 Obviously, testing can never demonstrate that a product is "fault free", it can only identify failures. A product that does not fail, no matter how many tests it is subjected to, may still contain faults.

extensive log of activity during alpha testing. Though the instrumentation required for logging may interfere with the operation of the product, at this stage in the process the benefits of logging usually far outweigh the costs.

Functional alpha testing attempts to validate and verify the features provided by the software product. Validation tends to occur when the tests are created and verification when the tests are conducted. For example, consider a simple product that performs various calculations involving weights (such as addition, subtraction, multiplication and division). When constructing the tests, the testers would use the product design documents (requirements) to validate that the product performs all of the required operations and that they will all be tested. Then, the tests would be conducted to verify that these feature operations are performed correctly. As with all testing, the tester must know the expected correct behavior. Hence, functional alpha testing can be very resource intensive.

Functional alpha testing is often conducted in a bottom-up fashion with the most basic functionality tested first, as it makes it easier to isolate failures. For example, again consider a simple weight calculator. The most basic features of such a product are the ability to enter weights and display weights. So, before testing whether the product can add two weights one should test whether it is possible to enter a weight and whether the product can display a weight. Otherwise, the tester might misconstrue or miscategorize a failure. Of course, the tester's job is to find the trigger condition for the failure, not the fault that resulted in the failure, but the tester should avoid adding confusion if at all possible.

Because of the number of paths through a typical product, some organizations use **operational profiles** that contain information about the relative frequency of different use cases and the order in which use cases typically occur. Test cases are then created that reflect these patterns of usage.

Non-Functional Alpha Testing—Recall that there are two broad categories of non-functional requirements: development requirements and execution requirements[2]. Since development requirements are related to the needs and desires of the development team (such as portability and maintainability) they are not, in general, testable. Instead, they must be validated using a review process. Execution requirements, on the other hand, are related to properties of the use of the product (such as response time and memory usage) and are, in general, testable. Some kinds of non-functional execution tests are so common that they have been named, including the following.

> **Timing tests** are used to evaluate non-functional requirements related to the time required to perform a function. Timing tests often make use of **benchmarks**, which are standardized timing tests. Some timing tests involve user interactions and some do not.
>
> *Reliability and availability tests* are used to assess the performance of the system under normal operating conditions. Formally, **reliability tests** aim to determine the probability that the product will fail in a given interval of time while **availability tests** aim to determine the probability that a product will be usable at a particular time. Conceptually, both are concerned with the dependability of the system. For some kinds of products, these kinds of tests involve the calculation of statistics like the *percent up time* and the *mean time to failure*.
>
> **Stress tests** (sometimes called **torture tests**) are used to determine the **robustness** (the ability to operate under a wide range of conditions) and the **safety** (the ability to minimize the damage resulting from a failure) of the system. Stress tests that are focused exclusively on safety are sometimes referred to as **recovery tests**.

2 Execution requirements are commonly called operational requirements. As discussed earlier, the term "execution requirements" is used here so as not to cause any confusion with operation-level requirements.

Configuration tests are used to assess the product on different hardware or software platforms and in different operating environments.

It is important to note that some execution requirements, while testable in principle, are very difficult to test in practice. For example, security requirements are sometimes difficult to test. While it is possible to simulate users for purposes of stress testing, it is almost impossible to simulate attackers or adversaries realistically for purposes of security testing.

It is also important to note that some kinds of tests are difficult to categorize as either functional or non-functional. Perhaps the most important example of this is tests that specifically target the user interface. For example, consider the following.

> When the user slides the units toggle from degrees to radians the units indicator on the display must change from degrees to radians.

Is this a functional requirement about how a product maps inputs (like button clicks) to outputs (what appears in the display) or is it a non-functional requirement that a product must have certain properties? Most people tend to categorize this kind of requirement as functional, but consider a requirement like the following.

> The GUI must be available in both Italian and Arabic.

Most people would consider this a non-functional requirement.

Tests checking requirements about the user interface are called **usability tests** (or, sometimes, **human factors tests**).

Internationalization/localization tests are a particularly important kind of usability test. They involve both translations and locale-specific information like currencies, number formats, and time and date formats. While some of these things can be verified when the product isn't in operation (for example, some translations), many have to be tested. For example, the appropriate word or phrase to use might depend on the context in which it is being used or the part of speech or gender. For example, "New" might be translated as either "Nouveau" or "Nouvelle" depending on the gender of the new thing. In addition, tests are needed to ensure that all strings are translated (and not mistakenly hard coded) and to ensure that character encodings are available and used properly.

Increasingly, an important part of usability testing is **accessibility testing**, which involves ensuring that the user interface is appropriate for all people, regardless of physical abilities. For example, Section 508 of the Rehabilitation Act of 1973, which applies to all Federal agencies, requires that people with visual acuity between 20/70 and 20/200 be able to operate software without relying on audio input. Accessibility testing, and other kinds of usability testing, can often involve hardware or software that measure the amount of time required to complete a task (for example, eye-tracking systems).

Beta Testing

Beta testing, sometimes called *pilot testing*, differs from alpha testing in two important respects. First, whereas alpha testing involves internal testers, beta testing involves external testers. Second, there is increased burden on the user (or trained observers) to record failures. Logging can be used in beta testing, but only to the extent that the code used to perform the logging will remain in the final product (since it could, itself, introduce faults or change the behavior of the system, and therefore is should not be active during this phase of testing).

Acceptance testing is done by clients to validate the product (that is, to determine that it satisfies the clients' needs and desires). Like alpha tests, acceptance tests are conducted in a controlled environment (provided by the developer). Sometimes, though not always, it is the same environment

that was used for alpha testing. Unlike alpha tests, acceptance tests are conducted by real users, hence are a direct form of validation. In some cases, trained observers monitor the tests.

As with alpha testing, acceptance testing involves both functional tests and non-functional execution tests (but not non-functional development tests). Unlike alpha testing, acceptance testing does not typically involve operational profiles. Instead, users interact with the system as they would in a real environment. In other words, the order and frequency in which features are used is not controlled. However, like alpha testing, all features are tested (even if they are used infrequently in practice) so that the product can be validated.

Installation testing involves real users in real uncontrolled environments. Since validation problems should have been found during acceptance testing, installation testing is used principally to verify the product, in particular, to determine that it operates properly in the user's environment. Of course, in practice, validation failures do arise during installation testing and must be noted. Ideally, the users have received some training or instruction about how to report failures, but they are in not expert testers. Hence, the failures they report (and the trigger conditions that resulted in those failures) can be difficult to understand and hence to replicate. For this reason, testing programs should attempt to minimize their dependence on installation testing. Though it is an essential part of the process, it is far and away the least efficient kind of testing (despite the fact that installation testers are often volunteers for mass-market products).

When a new product is replacing an existing product, it is sometimes possible to use the two products in parallel. Though this reduces user productivity and, as a result, can be quite expensive, it is invaluable, especially for mission-critical software products. Using both systems during a changeover period also provides a margin of safety in case the new system fails disastrously—the old system can simply continue in use while the new system is fixed.

Obviously, the details of installation testing vary dramatically with the architecture of the product. For example, standalone products and distributed products will be tested in very different ways. When testing distributed applications, for example, one might test client functionality and server functionality independently, and then conduct transaction tests that involve both. As another example, when testing web apps one might distinguish between content tests, function tests, usability tests, navigability tests, interoperability tests, and security tests. Regardless of the architecture, however, the objective is the same—to verify that the product will work properly when it is released.

System Testing in Context

As with unit testing and integration testing, the timing of system testing varies with the software process being used as does the amount of planning that is devoted to system testing.

Testing in a Traditional Process

In traditional processes, specifications should always be verifiable and hence individually testable. Thus, testing in traditional processes starts with the software requirements specification. That is, in traditional processes, each requirement is mapped to one or more tests. Some requirements will be mapped to unit tests and others will be mapped to system tests. In practice, few requirements tend to be mapped to integration tests.

Unit testing is done during the implementation or construction phase. Hence, it tends to occur early and often in traditional processes. Coders are intimately involved with unit testing in a traditional process but, because of the length of cycles, testers are also often involved in the creation and execution of unit tests. Since integration testing and system testing occur much later, rigorous unit testing (and the debugging that results) is critical in a traditional process.

Integration testing often doesn't begin until fairly late in traditional processes. In other words, it is common for many units to be constructed before they are combined and tested. As a result, in traditional processes it is often necessary to create **stubs**—placeholder code that stands in for units that have not yet been constructed. A stub has the same interface as the code it is standing in for, but provides none (or very little) of its required functionality. Not surprisingly, when the stub is replaced with the fully-functional unit during integration testing, faults are often uncovered. As a result, integration testing is particularly important in traditional processes.

Alpha testing often can't begin until the end of a traditional process. This is because traditional processes make no effort to have a working (if limited) version of the product at all times. Since one can't test the system until it exists, and the system doesn't exist until the end of a traditional process, system testing can't occur until very late. Furthermore, traditional processes tend to have long cycle times, so there is typically a long period between system specification and system test. This is, perhaps, the biggest reason that traditional processes are not considered agile.

Because of the length of the cycles in traditional processes, the product that is released at the end of each cycle is often thought of as the "final product" (even though the product has a version number), and changes are described in relation to it. Of course, though one would like to think that all such changes are improvements, any change to a software product might introduce new faults. Regression testing is crucial in determining whether changes to a product have introduced faults.

In a traditional process, a considerable amount of effort is devoted to the planning of system testing, the result of which is a document called a **test plan**. A test plan typically includes

- The business and technical objectives of the test suite;
- The test cases;
- The review process and acceptance criteria;
- An estimate of the size of the testing effort;
- A schedule for the testing effort.

Testing in Scrum (and Other Agile Processes)

In a traditional process, tests are created from the requirements specification. Since Scrum does not make use of a formal requirements specification, test development in Scrum proceeds differently from test development in a traditional process.

Perhaps most importantly, in Scrum integration testing and alpha testing are conducted each and every sprint. Indeed, most agile proponents argue that frequent integration and alpha testing is a big part of what it means to be agile. In Scrum, integration and alpha testing are used to demonstrate that features in the product backlog have been completed.

Unit testing in Scrum is left entirely to the developers. Those who argue for this approach believe that unit testing will find implementation failures and integration and system testing will find design failures. Alternatively, one can specify acceptance criteria (also known as *conditions of satisfaction*) as unit tests. Using this approach, it becomes very clear when a user-level product backlog item (PBI) has been completed.

As part of alpha testing[3], many agile processes advocate the use of daily **build verification tests** in which the product is built and tested (using automated tools) each day. Such tests are sometimes called **smoke tests**, and ensure that the product remains at a given quality throughout its incremental development. Of course, smoke testing requires discipline on the part of the team—each developer must smoke test her or his code before committing it. To encourage the necessary discipline, many organizations penalize developers that break a build (sometimes in comical ways but sometimes in more serious ways).

The timing of beta testing in Scrum is somewhat more ambiguous. Some practitioners beta test regularly (though, perhaps, not every sprint) and use beta testing as part of the product design process. That is, each increment is used to elicit user needs and refine user stories. Other practitioners only beta test **release candidates** (RCs), which are product versions containing all features planned for a release.

In Scrum, it is much more difficult to isolate regression testing from the rest of the testing process. This is because, without formal a requirements specification, it is difficult to distinguish old features from new features and, hence, difficult to have a baseline set of tests.

Though testing is an important part of Scrum and other agile methods, test planning tends to be informal. Hence, though organizations will have a generic testing process in place, there may not be test plans for individual products.

System Testing Tools

A variety of tools can facilitate system testing. Some assist with developing and executing tests and some assist with recording and reporting results.

Test Development and Execution Tools

Software products that do not have a graphical user interface (GUI) can, in large part, be tested in an automated fashion in much the same way as individual modules can. Though a custom test harness may be required for system testing because existing unit testing frameworks may not be appropriate, the concepts and process are the same as in unit testing.

Automated testing of software products with a graphical user interface (GUI) is rather different, however. Fortunately, many tools exist. Some tools allow the testing team to record an interaction with the product, indicate the anticipated state of the system after the interaction, and compare the two. These recordings, which include such events as mouse movements and key presses, can then be played back to conduct the test again for debugging or regression testing. Though the initial creation of the tests is very labor intensive because a tester has to interact with the product, repeated execution of the tests is not.

Other tools allow the testing team to write scripts that describe the interaction, rather than record an actual interaction. Creation of these scripts is labor intensive and requires specialized expertise, but the scripts are often much easier to understand than recordings (since they are a verbal rather than visual representation of the interaction). Rerunning the scripts is as easy as rerunning the recordings.

Hybrid automation tools support both. That is, user interactions can be recorded, but the recordings generate scripts that can then be edited in various ways. For example, one might record an

3 Some authors consider smoke testing a part of integration testing while others consider it a part of system testing. We consider it a part of system testing because, given the amount of effort required to keep the smoke tests up-to-date and conduct the daily builds, it is often the job of a dedicated person or team. In practice, smoke tests often start as integration tests but, as the product and tests become more complicated and sophisticated, become system tests.

interaction that involves saving a document, copy the resulting script, and then modify it to create a test for opening a document.

Test Recording and Reporting Tools

Tools also exist that help testers record, track, and report test results. Commonly called **bug tracking systems**, they essentially are specialized database systems that manage information about defects, most commonly failures and the trigger conditions that led to those failures.

Bug tracking systems are often used both during development (in alpha and beta testing) and after a product is deployed. Though these are conceptually very different activities, in practice they have enough in common that one tool can often be used for both.[4]

Preparing for System Testing

Many things that can be done during early stages of the software process that facilitate system testing. Most importantly, software products can be both designed and constructed to make them easier to test.

Engineering Design for System Testing—Some design patterns make it much easier to conduct system tests. Perhaps the most important example of this is the Command Pattern discussed in the chapter on engineering design. Recall that the Command Pattern encapsulates functionality in individual objects. This makes it easy to programmatically initiate functions that would normally be triggered by a GUI event. For example, there might be an `Open` class that encapsulates the functionality required to open a document in a word processor. A test case can be written that constructs an `Open` object and call its `execute()` method without having to actually click on the button that would normally initiate this action. While the GUI still must be tested, the Command Pattern makes it easier to test the functionality independently of the events that trigger that functionality.

Other patterns, for example the Observer Pattern and the Proxy Pattern, make it easy to unobtrusively instrument code for system testing. Though the details of these patterns are outside of the scope of this discussion, essentially one can use them to create monitors or loggers that can interact with other components in exactly the same way as components that are providing required functionality. Hence, though they can change the timing of and order in which code is executed, they have minimal impact on the product and are unlikely to introduce or mask defects.

Construction for System Testing—It is also possible to make decisions during the construction phase that facilitate system testing. For example, some languages and libraries have built-in support for logging (for example, the Java Logging Framework includes `Logger`, `Level` and other classes that greatly facilitate logging). As another example, the use of exceptions (that is, using `try-catch` blocks rather than special return values) generally makes it much easier to build a flexible test harness.

References

1. Gilb, T. (1995) "What We Fail to Do in Our Current testing Culture", *Testing Techniques Newsletter*, Software Research, January.
2. Musa, J. (1993) "Operational Profiles in Software Reliability Engineering", IEEE Software, March, pp. 14-32.

4 Post deployment systems are sometimes referred to as *issue tracking systems* and often include a knowledge base of work arounds that can be used by customer support specialists to provide assistance to users. Individual entries in an issue tracking system are often referred to as *tickets*.

3. Jones, Capers. *Software Quality In 2013: A Survey Of The State Of The Art*. Http://Namcookanalytics.Com/Software-Quality-Survey-State-Art/.

Task Identification and Effort Estimation

In order to plan most engineering activities, you must understood all of the work that needs to be done and how much effort will need to be expended to do it. Who can imagine a bridge construction or house construction project being approved without a budget or timetable? Similarly, who can imagine an automobile manufacturer deciding to offer a new model without fairly accurate estimates of the time and cost required to produce the cars?

Despite what many people seem to believe, this information is also important in most software engineering activities, despite the process being used. In a traditional, heavyweight process, this kind of information is needed for almost all aspects of planning and management and is primarily the responsibility of managers or lead technical people. In an agile, lightweight process (like Scrum), functional descriptions (sprintable stories) need to be divided into tasks and team members as a group have to commit to completing them during a sprint.

Thus, two central aspects of the management of a software engineering project are the identification of the tasks that must be completed to build a software product (the scope of the project), and estimating the amount of effort that will be required to complete those tasks. Along the way, it is common to estimate the size of the software product.

Task Identification and Organization

You should recall from our discussion of software processes that a task is an item of work and that a task can be either non-decomposable (in which case we call it an action) or decomposable (in which case we call it an activity or process).

It is usually fairly easy, at a high level of abstraction (which means a low level of detail) to identify the tasks that must be completed to build a software product. What is much more difficult is the identification of tasks at a level of detail that is useful for management purposes. Typically, you do not need to identify all of the actions; tasks can be decomposable but they should not be decomposable into too many different sub-tasks. The ideal level of detail is called a **work package**, which is somewhat tautologically defined as an amount of work that is small and specific enough that it can be estimated reliably.

A good way to both identify tasks at a useful level of abstraction is to use a **work breakdown structure** (**WBS**), which is a deliverable-oriented hierarchical decomposition of work done by the project team to achieve project goals or produce project deliverables. A WBS may be presented in three formats: it may be drawn as a tree, it may be drawn using a hierarchy diagram, or it may be written as a hierarchical list. A tree format has the main item of work at its root. Each level below the root shows a decomposition of the activity at that node into smaller parts. The leaves are work packages. A hierarchy diagram is a tree drawn in a more compact format: at some point child nodes are no longer splayed below their parent but instead are lined up (usually vertically). A hierarchical list uses indentation to show levels of abstraction: the activities or actions decomposing an activity are all listed indented from that activity to the same extent. Trees tend to be impractical for real problems because they use too much space; a hierarchy chart tends to be better for brainstorming (for example, on a whiteboard) and is generally easy to read; hierarchical lists are harder to read but easier to create in documents because they are texts, not images. We will use hierarchy diagrams.

The nodes in the WBS are items of work in the project. The root of a WBS is the project name, representing all the work to be done in the project. The first level is typically (though not necessarily) all the deliverables for the project, including documents, services, hardware, software, etc. Succeeding levels represent increasingly detailed specifications of work, where the leaf nodes are the work packages.

To illustrate, consider the WBS below for a project to develop the software for a home security system.

Figure 1: HomeSafe WBS

The nodes at the first level of this diagram are the three software components of the product: the Controller that runs the home system, the Web Site that sells and services the product, and a Mobile App that allows people to contact and control the home system remotely. The nodes at the third level are the main features comprising each component. The fourth level nodes (below Support) are the features of the Support portion of the Web Site. So far this WBS contains only deliverables. Some experts say that a WBS should contain only deliverables, so that this WBS is finished. But using this WBS for effort estimation would be difficult. Consequently other experts advocate adding more detail to make estimation easier. In this case, we could extend the WBS by adding nodes for design, coding, testing, inspection, deployment, and so forth. This would better display all the work that needs to be done in the project, and produce smaller work packages that are easier to estimate.

The WBS above decomposes deliverables and product features and functions to detail project work. Other decomposition strategies are possible, including the following.

- Project phases
- Organizational units
- Physical decomposition
- Geographical location

While mixing decomposition strategies is often frowned upon, decomposition strategies can be mixed in a WBS (even within a level). For example, in Figure 1, the first level is decomposed by deliverables (three programs), the second and third level by features and functions, and we proposed extending the WBS by project phases or activities (design, coding, etc.). In short, the best decomposition strategy is the one that leads to the best set of work packages.

If more information is to be associated with WBS nodes, a *WBS dictionary* can be written. Each node has an entry in the dictionary with additional information about it, such as its estimated effort, party responsible for the work, planned start and end date, and so forth.

There are several heuristics for making good work breakdown structures.

> *One hundred percent rule*—The nodes descended from a parent represent 100% of the work of the parent, but no work outside the project. Obviously, if the WBS is to accurately represent the

work in the project and be used for estimation and scheduling, it must not omit any work nor include activities that are not part of the work of the project.

Mutually exclusive siblings—No siblings in the WBS have overlapping work. Violation of this rule leads to inflated effort estimates and confusion about work assignments.

8/80 rule—Work packages (leaf nodes) require between 8 and 80 person-hours of effort. This heuristic helps to determine whether a node should be decomposed; work that can be done by one person in one day to two weeks is considered small enough to estimate reliably.

The best process for creating a WBS is to gather the project team and stakeholders in a meeting in which the WBS is created on a whiteboard, perhaps also using sticky notes. Participants should first make a list of items at level one silently on their own, and then the group can come to consensus on level one. Then each item in level one can be decomposed in the same way. Once deliverables (including product features and functions) are all listed in the WBS, stakeholders can leave the meeting. The lower levels consisting of development tasks and activities can then be filled in by the developers.

In traditional approaches, the entire project is planned as thoroughly as possible at the start of the project, progress is monitored closely, and adjustments are made to activities and plans as necessary as the project progresses. The planning process begins by setting the scope of the project and determining requirements.

Effort Estimation in Traditional Processes

There are several approaches to effort estimation in traditional processes.

Analogy—If the current project is similar to a past project, then the effort to complete it should be roughly the same as the effort required for the past project. *Ad hoc* adjustments can be made to compensate for any differences. This method works well when a project indeed closely ressembles some past project, but this is not often the case.

WBS to Effort—Given a complete work breakdown structure for a project, one can estimate the effort required for each work package and then sum these estimates to generate an effort estimate for the entire project. This can work well if the WBS accurately describes all the tasks needed to complete the project, and if the effort estimates for work packages are accurate. It is often hard to get everything right, however.

Size to Effort—Given an estimate of the size of the software product, along with known relationships between product size and effort, one can generate an estimate of project effort. A great deal of research has been devoted to this approach and we consider it in more detail next.

Software size can be measured quantitatively using two kinds of measures, functional and non-functional. **Functional measures of size** are denominated in units that are a direct reflection of the functionality being provided by the program (for example, the number of pages in a WWW application, the number of reports in a database application, the number of windows in a GUI application). Non-functional measures of size are denominated in units that are a direct reflection of the programs structure (for example, lines of code or number of classes).

The obvious shortcoming of all non-functional approaches is that while, at least in principle, it is easy to count lines of code or number of classes after the product has been developed, it is much more difficult to forecast them beforehand from requirements.

Fortunately, software engineers can learn from other disciplines. For example, you can construct a pretty good estimate of the time and cost of constructing a house based on the number of rooms

of different kinds, such as kitchens, bathrooms, bedrooms and living rooms. In other words, you can measure the size based on the functionality requested by or provided to the user. You can then improve the estimate by adding details like the finishes and sizes of those rooms. All of this can be done without detailed blueprints. Similarly, in software engineering, you can use the functionality provided (and the the characteristics of the functionality) to estimate time and cost.

In addition to being available from the requirements phase, functional measures have two other advantages over non-functional measures. First, they are largely independent of the technology and platform being used. Second, they are user-focused and, hence, are easier to explain to non-technical stakeholders.

Lines of Code (LOC)

Perhaps the most obvious measure of program size is the number of physical lines of code. The obvious shortcoming of this approach is that some lines are not "of consequence." For example, commentary lines and blank lines are not executed, and hence may not inflate the "real" size of the program. Hence, many people instead argue that the number of logical lines of code, that is, the number of statements "of consequence," is a better measure of size. Interestingly, however, it is hard to agree on which lines are "of consequence." For example, shouldn't comments be included? They certainly take time and effort to write, so much so that many programmers omit them. As another example, should initialization statements be included? To settle these disagreements, some people argue that each statement of consequence should be weighted by its complexity or difficulty. But this only provides more grounds for argument.

As an alternative to lines of code, you can instead count modules (functions, methods, files, or classes). However, you still have the problem of differences in complexity or difficulty. In addition, such measures tend to be very highly correlated to LOC. Hence, LOC is the most popular non-functional measure, and often people opt for the simplest case of counting every non-commentary line of code without weights.

So how can product size in lines of code be estimated? The usual approach is to decompose the software product into smaller and smaller components until the components are so small and simple that estimators feel comfortable estimating their sizes. A work breakdown structure can be used for this decomposition, but some sort of design composition can be used as well. The component estimates are simply summed to produce a product size estimate.

The problem with this approach is that it may not reflect the actual design of the product (because at the start of a traditional project there is no design yet), and hence the components estimated may not coincide with the components ultimately comprising the product, resulting in an inaccurate estimate.

Function Points

One of the first functional measures was proposed by Albrecht (1979), and is called **function point analysis**. In this approach five different kinds of functionality, sometimes called *base functional component* (BFC) types, are measured. Three of these are related to processes or transactions and two are related to data storage.

Processes or Transactions

External Inputs (**EI**) are processes that provide data that will be used or stored by the product. For example, a process that adds new employees to a file or table of employee records is an EI process.

External Queries (**EQ**) are processes that retrieve stored data (without adding value or information but, perhaps, providing formatting). For example, a process that retrieves all of the employees in a file or table of employee records is an EQ process.

External Outputs (**EO**) are processes that provide derived information to a user (such as reports, messages, or prompts) or other system. For example, a process that calculates the average salary of all employees in a particular division is an EO process.

Data Storage

Internal Logical Files (**ILF**) are logical groupings of data maintained by the product. For example, a file or table of employee records would be counted as an ILF.

External Interface Files (**EIF**) are logical groupings of data that are external to the product (not maintained by the product) but are used or referenced by the product. For example, a file or table used by multiple products that maps machine-readable codes to human-readable descriptions is an example of an EIF.

To account for the fact that these different types of components can vary in complexity or difficulty both within a type and between different types, these counts are weighted. Specifically, each component is categorized as being either simple, average, or complex, and the following weights are applied.

Measure	Simple	Average	Complex
External Inputs	3	4	6
External Queries	3	4	6
External Outputs	4	5	7
Internal Logical Files	7	10	15
External Interface Files	5	7	10

Finally, the following fourteen value adjustment questions about the product are answered using a Likert scale that ranges from 0 (Not Important) to 5 (Essential).

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require input transactions over multiple screens or operations?
8. Are the ILFs updated on-line?
9. Are the inputs, outputs, files or inquiries complex?
10. Is the internal processing complex?
11. Is the code to be designed to be reusable?
12. Are conversion and installation included in the design?

13. Is the system designed for multiple installations in different organizations?

14. Is the application designed to facilitate change and ease of use by the user?

With this information in hand, the number of **function points** can be calculated. Specifically, letting M_{md} denote measure m of difficulty d, W_{md} denote the weighting factor for measure m of difficulty d, and V_q denote the value adjustment factor for question q, the number of function points, F, is defined as follows.

$$F = \sum_{m=1}^{5}\sum_{d=1}^{3} M_{md} \cdot W_{md} \cdot \left[0.65 + 0.01 \cdot \sum_{q=1}^{14} V_q\right]$$

For example, suppose a small product has the following components.

Measure	Simple	Average	Complex
External Inputs	2	0	1
External Queries	0	3	0
External Outputs	1	1	2
Internal Logical Files	3	0	0
External Interface Files	1	0	0

Then, the weighted EI measure is 2·3+0·4+1·6=12, the weighted EQ measure is 0·3+3·4+0·6=12, the weighted EO measure is 1·4+1·5+2·7=23, the weighted ILF measure is 3·7+0·10+0·15=21, and the weighted EIF measure is 1·5+0·7+0·10=5. Hence, if the answer to all 14 questions is 0 then the number of function points is (12+12+23+21+5)·0.65=73·0.65=47. If, on the other hand, the answer to three of the questions is 1, the answer to 2 questions is 2, and the answer to one question is 5 (it doesn't matter which one), the number of function points is 73·(0.65+0.12)=73·(0.77)=56.

This approach has been formalized and standardized by the International Function Point Users Group (IPFUG) as ISO20926 (2009). In addition, several variants have been developed. Function points are highly correlated with LOC counts, but they do not vary by programming language as LOC counts do, and it is often easier to estimate function points than LOC early in the life cycle.

Other Functional Measures

The Common Software Measurement International Consortium (COSMIC) created a functional measure that is suitable for both application software and system software. Their methodology has been standardized as ISO19761 (2011). The COSMIC methodology counts the *data movements* (movements of data from or to users and from or to persistent storage) associated with functional processes (steps required to complete an action) that are initiated as a result of *triggering events* (indivisible events that occur external to the product). The size of a functional process is the number of data movements associated with it, and the size of a software product is the size of all of its functional processes.

Roetzheim (2000) describes a version of function points that is applicable for Web Apps. When using this measure EI corresponds to input screens or forms, EQ corresponds to externally published interfaces, EO corresponds to HTML pages, ILF corresponds to internal database tables and XML files, and EIF corresponds to internal database tables and XML files.

Boehm (1996) suggests using the notion of **object points** rather than function points. In this variant there are three measures (i.e., screens in the user interface, reports, and components or

modules), each of which is characterized as being simple, medium, or difficult, each with appropriate weights. The number of object points is then the sum of the weighted measures times a proportion of reuse (that is between 0 and 1).

Effort Estimation

Given a size estimate, we now need to construct an estimate of the effort that will be required to complete the product. Effort can be measured either in absolute units or in relative units but should be independent (to the extent possible) of the size of the staff that will conduct the work.

Effort estimates in traditional processes invariably use as their unit the **person-month**, which is the amount of effort a typical developer expends in one month of work. For example, if it is estimated that an average developer would need eleven days of uninterrupted work to complete a task, then the task's size is half a person-month (a month is four and a half weeks, and assuming five work days per week, a person-month has about 22 person-days).

Effort estimates generated from product size estimates include all life cycle work writing requirements and designs, doing testing, generating documentation, collecting data, managing the project, and so forth, as well as actually writing the code.

Standard statistical techniques can be used to determine the relationship between program size and effort. These analyses can be conducted either using industry-standard data or organization-specific data. The latter, if collected and used properly, will almost certainly lead to better estimates. However, in practice, both the collection and use of these data requires a level of sophistication that only mature (and large) organizations possess. Hence, we will only consider existing models that have been estimated using industry-standard data. While the same kinds of models can be estimated by individual organizations, we will not discuss the details of collecting such data or of the estimation techniques that are most appropriate.

Albrecht and Gafney (1983) and Kemerer (1987) both posited models of the form

$$E = \alpha + \beta F$$

where E denotes the effort (in person-months) and F denotes the size in function points.

Using several data sets, they used ordinary least squares (a statistical technique for estimating parameters) to determine the following values for α and β.

Study	α	β
Albrecht and Gafney	-91.4	0.255
Kemerer	-37.0	0.960

Obviously, given the negative values of α, neither model is useful for small products (since small values of F will yield negative values of E). Aside from that similarity, the models have very different implications. Albrecht and Gafney found that each function point adds about a quarter of a person-month of effort, whereas Kemerer found that each function point adds almost one person-month of effort. This difference can be explained in several different ways. It may be the result of differences in the calculation of function points. It also may also be the result of using data from different organizations. In either case, the differences tend to argue for the need for organization-specific data except when only coarse estimates are needed.

Several groups have also developed models that relate the size measured in (thousands of) lines of code to the effort. These groups have posited models of the form

$$E = \alpha \cdot L^{\beta}$$

where L represents the number of thousands of lines of code (KLOC). In models of this form, the parameter β accounts for economies or diseconomies of scale. When $\beta < 1$ the project exhibits economies of scale and when $\beta > 1$ it exhibits diseconomies of scale. Diseconomies of scale result from such things as communications problems and integration problems. Economies of scale are very rare in software engineering and creative work in general.

Early models of this kind were estimated by Watson and Felix (1977), Basili and Freburger (1981), and Boehm (1981). Their results can be summarized as follows.

Study	α	β
Watson and Felix	5.20	0.91
Basili and Freburger	1.38	0.93
Boehm	3.20	1.05

A comparison of these models shows that they are quite different, especially for larger products. Consider the graph below and note how widely the predictions diverge as L grows larger.

Perhaps the most comprehensive set of "simple" models of these kinds have been estimated by the International Software Benchmarking Standards Group. They have estimated the parameters of

models for various different platforms (e.g., mainframes, mid-range systems, desktops) and different kinds of languages. The general form of their models is:

$$E = \alpha \cdot F^{\beta} \cdot N^{\gamma}$$

were F is the product size in function points and N denotes the maximum size of the team. One representative instance is the following model.

$$E = 0.512 \cdot F^{0.392} \cdot N^{0.791}$$

It's interesting to consider the impact of the maximum size of the team in this model. Specifically, note that for a product with a given number of function points, increasing the maximum size of the team from five to six actually increases the number of person-months of effort by a factor of about 0.5. This is because larger teams tend to be less efficient than smaller teams.

One of the most complicated models is the **Constructive Cost Model** (COCOMO) developed by Boehm in 1981 and modified in 2000 (and now called COCOMO II). At its core, COCOMO II uses size measured in either lines of code or functional measures. However, it includes models that can be used to convert from functional measures to LOC (in a variety of different languages). COCOMO II actually has three different (but closely related) models used at different stages of the software process. We present the "Post-Architecture" model (which uses the same functional form as the "Early Design" model but a different number of effort multipliers). We will use notation that is consistent with the models presented in this book, rather than that in the documentation of the model.

The nominal-schedule effort (excluding the required development schedule adjustments) is given by the following equation.

$$E = 2.94 \cdot L^{\beta} \cdot \prod_{i=1}^{16} W_i$$

Here W_i denotes effort multiplier i and β is defined as follows.

$$\beta = 0.91 + 0.01 \cdot \sum_{j=1}^{5} \psi_j$$

Here ψ_j denotes scale factor j. This model can be used on a module-by-module basis. After estimating the effort required for each module and summing to find the effort required for the product, COCOMO II also includes an adjustment factor that captures the impact of schedule constraints. Note that while COCOMO II is presented here using thousands of lines of code (L) as the independent variable, one can also use it with function points (F). The model contains conversion factors for a wide variety of programming languages.

In COCOMO II, the parameter that accounts for diseconomies of scale (β) is calculated from the five scale factors in the following table.

Index	Description	Very Low	Low	Nominal	High	Very High	Extra High
1	Precedentedness	6.20	4.96	3.72	2.48	1.24	0.00
2	Flexibility	5.07	4.05	3.04	2.03	1.01	0.00
3	Risk Resolution	7.07	5.65	4.24	2.83	1.41	0.00
4	Team Cohesion	5.48	4.38	3.29	2.19	1.10	0.00
5	Process Maturity	7.80	6.24	4.68	3.12	1.56	0.00

Values in this table increase from right to left because diseconomies of scale increase from right to left. Given the definition of β, anytime the sum of the ψ_j values is greater than 9, the project will exhibit diseconomies of scale.

The effort multipliers (W_i) are given in the following table.

Index	Description	Very Low	Low	Nominal	High	Very High	Extra High
1	Req. Reliability	0.82	0.91	1.00	1.10	1.26	N/A
2	Database Size	N/A	0.90	1.00	1.14	1.28	N/A
3	Complexity	0.73	0.87	1.00	1.17	1.34	1.74
4	Req. Reuse	N/A	0.95	1.00	1.07	1.15	1.24
5	Req. Document.	0.81	0.91	1.00	1.11	1.23	N/A
6	Exec. Time Const.	N/A	N/A	1.00	1.11	1.29	1.63
7	Storage Const.	N/A	N/A	1.00	1.05	1.17	1.46
8	Platofrom Volatility	N/A	0.87	1.00	1.15	1.30	N/A
9	Analyst Capability	1.42	1.19	1.00	0.85	0.71	N/A
10	Programmer Capability	1.34	1.15	1.00	0.88	0.76	N/A
11	Personnel Continuity	1.29	1.12	1.00	0.90	0.81	N/A
12	Applications Experience	1.22	1.10	1.00	0.88	0.81	N/A
13	Platform Experience	1.19	1.09	1.00	0.91	0.85	N/A
14	Lang. & Tool Experience	1.20	1.09	1.00	0.91	0.85	N/A
15	Use of Tools	1.17	1.09	1.00	0.90	078	N/A
16	Multisite Development	1.22	1.09	1.00	0.93	0.86	0.80

The first group consists of characteristics of the product, the next group consists of characteristics of the platform, the next group consists of characteristics of the personnel, and the last group consists of characteristics of the project. Detailed definitions of each of the different levels is available in the model documentation.

As an example, consider the application of this model to a program with 50,000 lines of code ($L = 50$) in which all of the scale factors are nominal except for the team cohesion (which is low) and all of the effort multipliers are nominal except for the required reliability (which is very high) and the language and tool experience (which is very low).

$$\beta = 0.91 + 0.01 \cdot (3.72+3.04+4.24+4.38+4.68) = 0.91 + 0.01 \cdot 20.06 = 1.11$$

Since fourteen of the effort multipliers are 1.00 and the other two are 1.26 and 1.20,

$$\prod_i W_i = 1 \cdot 1 \cdot \cdots 1 \cdot \quad 1 \cdot \quad 1.26 \cdot \quad 1.20 = 1.51.$$

Hence

$$E = 2.94 \cdot \quad 50^{1.11} \cdot \quad 1.51 = 2.94 \cdot \quad 76.89 \cdot \quad 1.51 = 341.35$$

COCOMO II also includes techniques for differentiating new code from *adapted code* (that is, preexisting code that requires some changes) and *re-used code* (code that is used "as is"). These techniques essentially adjust the size of the product (they adjust *L*).

Effort Estimation in Scrum

In Scrum, planners do not estimate software size but instead go directly to effort estimates for stories and tasks. Effort estimates in Scrum can use person-days or person-months (called **ideal days** or **ideal months**) or a unit called a *story point*.

A **story point** or **task point** is a relative unit of size. Story point estimates are made by choosing one or a few small stories or tasks of about the same size and using them as a baseline (by declaring them to be of size one). Then all stories or tasks are estimated relative to this standard. In other words, if story or task X is defined to be the baseline and story or task Y is about four times as big, then the size of Y is four story points. Although it takes a little getting used to, the advantage of story points is that they do not require thinking in detail about the time that is actually required to accomplish a story or task; this abstraction may make estimation easier and hence more accurate. Story points are the most-used effort unit in Scrum projects, so we will restrict our attention to them in the remainder of our discussion.

In Scrum, effort estimations are refined during the project and become more detailed (and presumably more accurate) over time. Epic PBIs tend to have rough qualitative or categorical effort estimates (like small, medium, large, and extra-large) because there is no need to do the work to make more accurate estimates for them. But once PBIs begin to be refined as they rise in the product backlog, more accurate estimates are needed for release and sprint planning. PBIs chosen for sprints must be estimated especially accurately so that the team can ensure a successful sprint.

By not bothering with software size estimates and only making detailed effort estimates when they are needed, Scrum teams avoid much work and rework. The trade-off for this gain is that accurate early effort (and hence time and cost) estimates for a release or a product cannot be made. Of course, given the fluidity and changeability of products developed with agile methods, agilists would argue that accurate early estimates are impossible in any case, so nothing is really lost.

Planning Poker

Planning poker is a technique for an agile team to come to consensus on effort estimates. The team first chooses a sequence of numbers to be used as estimates. Usually this sequence is a modified Fibonacci sequence such as 1, 2, 3, 5, 8, 13, 20, 40, and 100, or the powers of two, that is, 1, 2, 4, 8, 16, and so forth up to some large value, like 128. The point of using such a scale is to try to attain estimation *accuracy* (an estimate that reflects reality) without attempting to achieve unrealistic *precision* (being very close to reality). By leaving out most of the larger numbers, agile estimators acknowledge that estimates for large stories will not be very precise (they will not be very exact) but they will be accurate (a very large story will indeed require a big effort). The numbers from the chosen scale are written on cards, and each team member gets a full set.

The team, PO and SM participate in planning poker, but only team members actually make estimates. The PO presents, explains, and clarifies PBIs when questioned about them during planning poker, and the SM facilitates participation, making sure that all team members contribute to discussion and have a voice in the results. The PO also records story estimates.

The process proceeds in rounds, and a PBI or task is estimated in each round. A round begins when the PO presents and explains a PBI and the team and PO discuss the PBI or task to make clear exactly what it involves. Then all team members simultaneously throw cards (one each) with his or

her estimate of the PBI or task. If the cards all agree, the round is over and the thrown value is the estimate. If not, then the team members discuss why they made the estimates they did. It is especially important for the team members with the highest and lowest estimates to explain their reasoning. The PO may be further questioned about the PBI. Once everyone understands the rationales for the estimates and any confusions have been cleared up, the cards are collected and the process repeats from the point where all team members throw cards with his or her estimate.

Usually consensus is reached in two or three rounds. The the estimates are often quite good because they are based on discussion and reasoning. Furthermore, the discussion of PBIs and tasks that results from planning poker helps everyone on the team understand the PBIs and tasks better.

References

1. A.J. Albrecht, "Measuring application development productivity", *Proceedings of the IBM Application Development Symposium*, Monterey, CA, Oct. 14-17, 1979.
2. International Standards Organization, *IFPUG functional size mFunctional Size Measurement*, ISO/IEC 20926, 2009.
3. International Standards Organization, *COSMIC: A functional size measurement mFunctional Size Measurement Method*, ISO/IEC 19761, 2011.
4. W. Roetzheim "Estimating Internet Development", *Software Development*, www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm, Aug. 2000.
5. B. Boehm, "Anchoring the Software Process", *IEEE Software*, Vol. 13, pp. 73-82, 1996.
6. A.J. Albrecht and J.E. Gafney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", *IEEE Trans.actions on Software Eng.ineering*, SE-9, pp. 639-648, 1983.
7. C.F. Kemerer, "An Empirical Validation of Software Cost Estimation Models", *Communications of the ACM*, Vol. 30, pp. 416-429, 1987.
8. C.E. Watson and C.P. Felix, "A method of program measurement and estimation", *IBM Systems Journal*, Vol. 16, pp. 54-73, 1977.
9. V.R. Basili and K. Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, *The Journal of Systems and Software*, Vol. 2, pp. 47-57, 1981.
10. B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.
11. B. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D.J. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*, Prentice-Hall, 2000.

User Interaction Design

Interaction Design Overview

Interaction design is concerned with specifying the user experience for a software product. As such, it is concerned with the visual and aural appearance of the program and with the sequence of events that occur as a user and a program exchange data. The quality of a user interface is often referred to as its *usability*, but this term is both vague and incomplete because there are user experience goals that may go beyond usability (such as the aesthetic quality of an interface). Consequently it is better to focus on more specific and complete interaction design goals, including the following.

Effectiveness—Users can access all the features needed to achieve their goals.

Efficiency—Users achieve their goals quickly.

Safety—Neither users, their computers, nor their data are harmed by the product.

Learnability—Users become proficient in a short time.

Memorability—Users regain proficiency quickly after not using the program for some time.

Enjoyability—Users experience positive emotions when using the product.

Beauty—Users find the product is aesthetically pleasing.

Interaction design draws material from many fields, in particular from ergonomics (the study of how to design and arrange things so that people can use them easily and safely), perceptual physiology and psychology (the study of the senses and our perceptual mechanisms), cognitive psychology (the study of memory, learning, and thinking), and graphic design (the study of communication through the use of type, space, color, and image). We cannot do more than the broadest survey of this rich field in this chapter. We will concentrate on the place of interaction design in the software life cycle, models and notations for interaction design, interaction design processes, and a few interaction design principles For a far more detailed treatment, see [1] or [2].

Interaction Design In the Life Cycle

Interaction design is a major part of product design, and so it is part of software requirements specification. In traditional approaches, interaction design has often been done as an after-thought or a mini-phase between requirements specification and engineering design. Designing user interaction after most of the features and behaviors of a product have been specified makes it very difficult to do well because interaction is in large part *about* the features that a product should provide and the behaviors it should exhibit. A better approach is for interaction design to be a driver of requirements specification.

Agile methods tend to omit mention of interaction design altogether, but the fundamental directive that agile teams include all the skills needed to create a product means that interaction design skills should be present in the team and that interaction design should be part of agile processes. Several practitioners have proposed that interaction design be conducted during a sprint for the features the team expects to be implemented in the following sprint. Then the designers have time to prototype and test designs with users before they are incorporated into the product.

Interaction Design Models and Notations

Interaction designers specify both the appearance and the behavior of products, and hence they need both static and dynamic models. A **static interaction design model** depicts the visual and

aural features of products that do not change during execution, such as screen layouts and sounds associated with states or events. A **dynamic interaction design model** depicts product behavior during execution, such as product states and state transitions and the communications between the product and its users. There are several notations for making static and dynamic models.

Static Model Notations

A **use case** is a type of complete interaction between a product and its environment [3]. For example, some use cases for an e-reader are reading a book, browsing an e-bookstore, buying a book, or browsing books on the e-reader. An **actor** is a type of agent that interacts with a product. For example, an e-reader's actors might be a reader and an e-bookstore. A use case diagram is a UML notation for showing the use cases in a product. A **use case diagram** represents a product's use cases and the actors involved in each use case. The diagram in Figure 1 shows an example UML use case diagram.

Figure 1: A Use Case Diagram for an E-Book Reader

This diagram shows use cases for an e-book reader. The diagram shows two actors: the Reader is the person operating the device, and the E-Bookstore is a computer accessed over the Internet. Use cases are shown as ovals. A line between a use case and an actor indicates that the actor is involved in that use case. The Browse Book Store use case is an interaction that occurs when the Reader uses the device to look at e-books available in the bookstore—both actors are involved in this interaction. The Reader may also Buy a Book from the E-Bookstore, again involving both actors. The other use cases do not involve the E-Bookstore, only the Reader.

Use case diagrams are an easy way to represent collections of features that a product might have. They are useful in product design because several diagrams representing different collections of features show design alternatives. It is helpful to make several of these diagrams and manipulate them to consider various combinations of product features.

A **screen** (or **page**) **layout diagram** is a drawing of (part of) a product's visual display when it is in a particular state. Screen layout diagrams are easy to sketch and modify and are a simple but essential tool in designing graphical user interfaces. Layout diagrams can be drawn at various levels of abstraction. A good place to start is with a low-fidelity diagram, often called a **wireframe**, that roughly indicates the layout of screen elements without many details. There are many free tools on the Internet for making wireframes. Once a satisfactory overall design is formulated, a high-fidelity diagrams showing colors, fonts, controls, and so forth can be made as the basis for implementation. Low- and high-fidelity screen layout diagrams are easy to draw and manipulate, so they are good

tools for considering many design alternatives. They also are a great help to stakeholders in visualizing what a product will look like.

The following is an example of a wireframe for a screen from a document search system.

Figure 2: A Wireframe for a Screen in a Document Retrieval System

This window has a title (Pyrex) and an indication of window controls (the circles at the top left). It depicts a frame with two tabs. The rectangles with crossed lines indicate text fields. Labels, buttons, and a combo box are also shown. This drawing could be part of a a preliminary design, but it shows enough detail that it could also be the basis for an implementation.

Dynamic Model Notations

A **use case description** is a specification of the interaction between the product and the actors in a use case. Use case descriptions are text, usually conforming to a template. They provide some context information for the use case, along with a narrative resembling a script that describes the sequence of actions by actors and the product as they interact. Figure 3 below shows a typical use case description template.

Use Case Name: To identify the use case

Actors: The agents participating in the use case

Stakeholders and Needs: What this use case does to meet stakeholder needs

Preconditions: What must be true before this use case begins

Post conditions: What will be true when this use case ends

Trigger: The event that causes this use case to begin

Basic Flow: The steps in a typical successful instance of this use case

Extensions: The steps in alternative instances of this use case occurring either because of variations in the normal flow or because of errors.

Figure 3: A Use Case Description Template

Use case descriptions fill in the details of the use cases named in a use case diagram. They provide the notation for the next step in product design after choosing features, which is to describe the interactions realizing each feature. Use case descriptions can be used to explore alternative

interaction flows. This is important because the order in which interactions occur has a big influence over many interaction design criteria, such as efficiency, safety, learnability, memorability, and enjoyability.

A **storyboard** is a collection of screen layout diagrams linked by arrows depicting events or the passage of time. A storyboard shows how the screen changes over time or in response to various inputs. Storyboards can be made to elaborate use case descriptions by filling in the details of the visual display. They can even be linked to capture the connections between use cases. Storyboard development usually begins with wireframe nodes. As design progresses higher-fidelity nodes can be filled in, or placeholders can be used to refer to higher fidelity screen layout diagrams outside the storyboard.

Like screen layout diagrams, storyboards are an excellent tool for investigating design alternatives and for helping stakeholders visualize designs before their implementation.

Use Case Models and Requirements

Many people have suggested that use case models be used as requirements specifications. Use cases model the interactions between a system and its users and other things with which it exchanges data (like other computer systems). Use case models are valuable for interaction design, and can document some requirements. But in principle they cannot document all requirements because they are not able to model requirements that don't have to do with interactions. Thus, for example, they are not intended to capture specifications about how things must be computed (for example, equations relating inputs to outputs), how a system must be implemented (for example, what sort of database system it must use), and most non-functional requirements (like reliability, safety, security, portability, and performance requirements).

Another problem with use case models for documenting requirements is that they are organized to trace interactions and not to specify behaviors in relation to system features. This can sometimes make it difficult to see the overall organization of a user interface, and to check whether an interaction specification is complete.

In summary, use case models are good for interaction design and they can help in expressing interaction requirements, but they cannot capture all requirements, and they may not always be the best vehicle for expressing interaction requirements.

Product backlog items (PBIs) in Scrum are the standard means for specifying features to be included in a product. As explained in the chapter about requirements, PBIs are high-level (mainly) functional requirements that must be elaborated before they can be implemented. A PBI may specify some aspect of a product's interaction with its environment, but as high-level specifications they are really not suited to capture the details of interactions. Use case descriptions do capture such details, and so they are a good tool for elaborating interaction PBIs during sprints. They can also provide supplementary information if a PBI is elaborated to a fairly low-level of detail during backlog grooming.

Interaction Design Processes

Like other design processes, interaction design is mainly a top-down activity that begins with the most abstract models and gradually refines them until a detailed specification is complete. After eliciting needs, desires, ideas, and preferences from stakeholders, a use case diagram can be formulated to pin down product features and capabilities. Use case descriptions should follow as a means to determine the best overall flow of interaction. Screen layout diagrams and storyboards can

be made to aid in design, to obtain feedback from stakeholders, and as specifications for implementation.

In both traditional and agile approaches, it is often advisable to conduct usability testing at various points during interaction design. **Usability testing** is empirical evaluation of (parts of) an interaction design to determine whether it meets interaction design goals [4]. Usability testing can take many forms, but it typically requires specifying the interaction design goals to be tested and the response that will be considered adequate. Several test subjects are chosen to be representative of the user population and a testing scenario and environment are selected and set up. The scenario can be "executed" on anything from a rough paper prototype to a full-featured product. Users are exposed to the scenario and their behavior measured. Measurements might include pre- and post-test questionnaires or surveys, observation of user actions, timing how long it takes users to achieve goals, counting user mistakes, and so forth. The data is analyzed and conclusions drawn about the effectiveness of the design and how it might be improved. Several rounds of usability testing may be done to refine important parts of an interaction design.

Interaction Design Principles

A **design principle** is statement that certain characteristics make a design better, used as a quality criterion in generating and evaluating designs. There are many interaction design principles; here we mention nine of them in three categories that seem particularly helpful to novice interaction designers.

The first group of three principles applies to interaction design in general. The principles can be remembered using the acronym SAC, which stand for Simplicity, Accessibility, and Consistency.

> **Simplicity**—Simpler designs are better. Simplicity is a principle that is fundamental in every design discipline. An example in interaction design is proliferation of interface features. Many commercial products include (apparently) every feature that anyone has ever thought of, making them very complex. The user is confronted with a plethora of controls, and long, multi-level menus. Just finding basic features becomes difficult, not to mention the difficulty of choosing between a wealth of similar options. A better alternative is a product that offers all basic functions in a simple, elegant design. Rarely done tasks can still be completed, though perhaps with a bit more work. The principle of simplicity says it is better to opt for streamlined, simplified access to basic functions in exchange for slightly more effort to accomplish rarely done tasks.
>
> **Accessibility**—Designs that can be used by more people are better. In the US, about eight percent of the population has red-green color blindness; older people generally have difficulty reading text in small fonts; many people have tremors or fine-motor control problems that make it difficult or impossible for them to use small controls with a mouse. Designs that rely on red or green to make important distinctions, use small fonts, or have small controls will be unusable by these populations. Better design will accomplish their ends while still being accessible to a wide range of users.
>
> **Consistency**—Designs that present similar data in similar ways, and provide similar ways of accomplishing similar tasks, are better. For example, all the pages of web site should have the same navigation mechanisms in the same places on the page, and invoking them should take the user to the same location. Fonts, colors, and layout should be used in a consistent fashion throughout a site.

The SAC principles apply to both product appearance and behavior. The CAP principles apply in particular to appearance. CAP stands for Contrast, Alignment, and Proximity.

Contrast—Designs that make things that are different appear different are better. This principle has a long history in typography and has lead to conventions about the use of serif versus sans-serif fonts (use one for headings, the other for text), italics and boldface (for emphasis), and font-size (for distinguishing different levels of headers, for example) [5]. It applies as well to colors, images, and controls.

Alignment—Designs that line-up elements in a grid are better. Alignment makes designs look neater and may also convey information about relationships between elements. For example, text that is aligned on both the left and right looks neater and more formal than text that is only aligned on the left (so-called ragged-right text). As another example, it is conventional to use indentation (and alignment on the left) for different levels of an outline or hierarchical list, and of course we use left alignment in source code to show the bodies of control structures.

Proximity—Designs that group related items together spatially are better. For example, when arranging a group of related radio buttons, it is better to place them neatly in a tight group separated from other elements by vertical or horizontal space. This signals to users that these buttons are connected and also makes it easier for users to discern the range of choices available to them.

The final group of three design principles applies especially to behavior. These are the FeVER principles, which stands for Feedback, Visibility, and Error prevention and Recovery.

Feedback—Designs that acknowledge user actions are better. Without feedback users often cannot know whether their actions are having any effect. For example, imagine a mail program that did not indicate in any way whether a message had actually been sent when the send button was pressed. Most users would probably press it several times to make sure their mail was sent, resulting in involuntary spam. Feedback can be very subtle, such as changing the color of an icon or emitting a small noise. Nevertheless, some sort of feedback is essential for efficient and effective use of most programs.

Visibility—Designs that prominently display their state and available operations are better. The state of a program often has consequences for what a user can do at a given time. For example, the popular UNIX text editor vi has two states: insertion mode and edit mode. In insertion mode, whatever the user types is inserted into the text, but in edit mode, whatever the user types is interpreted as editing commands. Unfortunately, vi makes its state visible only through display of "insert" or nothing at the bottom of the screen, so it is hard for users to keep track of the program state. Veteran users learn to press the escape key whenever they lose track of the program state because it forces the program into edit mode. Wouldn't it be better for the program to display its state in a more overt fashion? Programs that do not make available operations visible are similarly hard to use, and again vi provides a bad example. The only way to discover editor commands is to type ":h" in edit mode (for help)—and there is no way to discover this incantation in the program. Then the user has to remember the commands when returning to editing. This makes vi very difficult to use until the user has mastered many of its commands.

Error Prevention and Recovery—Designs that prevent user errors and provide error recovery mechanisms are better. There are many ways that a design may prevent errors. For example, buttons that should not be pressed or menu items that should not be selected in some program state should be disabled. If the program needs a value from a small set of values (like months or a range of numbers), don't ask the user to type in the value: have the user select from a list or menu. Not all errors can be prevented, so programs should also provide ways to recover from them. A common example is the Undo feature in many programs.

References

1. Preece, Jenny, Helen Sharp, and Yvonne Rogers. *Interaction Design: Beyond Human-Computer Interaction 4th Edition*. Wiley, 2015.
2. Shneiderman, Ben, Catherine Plaisant, Maxine Cohen, and Steven Jacobs. *Designing the User Interface: Strategies for Effective Human-Computer Interaction 5th Edition*. Pearson, 2009.
3. Cockburn, Alistair. *Writing Effective Use Cases*. Addison-Wesley, 2000.
4. Leventhal, Laura, and Julie Barnes. *Usability Engineering: Process, Products, and Examples*. Prentice Hall, 2008.
5. Williams, Robin. *The Non-Designers Design Book*. Peachpit Press, 1994.

CSC131 Fall’18
Computer Software Engineering

Introduction

Syllabus

- Class time & location
  - Section **1**: Tu/Th, 12:00PM - 1:15PM, Riverside Hall 1010
  - Section **2**: Tu/Th, 1:30PM - 2:45PM, Riverside Hall 5029
  - Section **5**: M/W, 4:00PM - 5:15PM, Mariposa Hall 2030
- Instructor
  - Homepage: http://www.ecs.csus.edu/csc/yang/
  - Email: yang@csus.edu
  - Office: Riverside Hall 3006
  - Phone: (916) 278-7328
  - Tentative office hours: M/W 11:00 AM – 1:00 PM (subject to change)

Syllabus (cont.)

- Course Content
  - How to develop high-quality software systems that are delivered on time and within budget using modern development tools.
  - Two aspects:
    - Managing the effort and,
    - Applying effective tools and techniques.
  - We will survey both aspects and apply them by building a system in teams of 5-6 members during the semester.

Syllabus (cont.)

- Goals
- Text Book
  - Software Engineering: A Practitioner's Approach, R. Pressman, 8th edition, 2014, McGraw Hill
  - Safari books (free through library)
  - Text (to be posted in Canvas)
- Attendance & Participation
  - Lecture
  - Labs
  - In-class activities

Syllabus (cont.)

- Grades
  - Criteria:

Assignments	Weight
Labs + In-class Activities	10%
Homework Assignments	10%
Team Project	30%
Midterm Exam	20%
Final Exam	30%

  - No curving
  - Note: Your overall grade in Canvas grade book will NOT be accurate until all grades (homework, labs, in-class activities, project, exams) are entered. However, it is a good indicator of your current performance.

Syllabus (cont.)

- Labs
  - Attendance is required.
- Homework
  - Individual work unless advised otherwise
  - Late homework will be accepted within 2 days after due dates, with 25% penalty for each day.
- Project
  - 30% of your grade
  - Work as teams
    - Be professional
    - Peer evaluation

Syllabus (cont.)

- Important Dates
  - Midterm – Week 8 (Oct 15 – 19)
  - Last day to withdraw – Nov 16, 2018
  - Final – Week 16 (Dec 10 – 14)
- Academic Honesty
  - Reference to others' work
- Questions?
- Survey in Canvas

A Question for You

- Why do you take this class?

- What is this class all about?
  - Programming skills?
  - Algorithm analysis?
  - Team building?
  - …

The Process

- Why does it matter?

A Card Game

Step 1 Preparation

- Form your groups
  - Each group should have 5-6 members
- Introduce yourself to your group mates
- Shuffle your cards, and exchange with your adjacent groups
  - Try all means necessary to shuffle the cards. Your goal is to make it hard for the other group to sort the cards

Step 2 Understanding the Rules

- Goal – to sort the cards like this:

- At any moment, only one person can touch one card with one hand only.

Step 3: Assigning Roles & Planning

- **Company Alpha**

- **Company Beta**

Plans:

Player A can spread cards.
Player B can flip cards.
Player C can touch ”Club” only.
Player D can touch ”Spade” only.
Player E can touch “Heart” only.
Player G can touch “Diamond” only.
Order to touch cards: A –> B –> C –> D –> E -> G

Form your own plan.

Game Start!

You have 2 mins to sort the cards!

Time is up.

Step 5 Reflection

- Company Alpha
  - What went well with the plan?
  - What went wrong?
  - What can you do better as an individual?
  - What can you do better as a team?
- Company Beta
  - Did you like your plan?
  - What would you like to change? (It is OK if you think your plan is perfect.)

Let’s rematch!

Shuffle Cards

Game Start!

You have 2 mins to sort the cards!

Time is up.

A Follow-up Interview

- Did you feel like a team?
- What would happen if something happened to kill the project 75% of the way through? (Considering a full suit of same rank is a feature)
- For Company Alpha, what do you like/dislike about your plan?
- For Company Beta, did you change your process after Game 1 at all? How would you change things next time to be even better?

Factors influencing our process

- Individual strength
- Collaboration
- Communication
- Risk management
- Exception handling
- …

Before Wed Class

- Complete the survey in Canvas
- Reading assignment in Week01 Module
- Programming assignment on CodingBat

1. Introduction to SE

What is Software?

- To the Hardware:
  - A collection of **instructions** to be executed by a computer/processor
- To the Developer:
  - A collection of **human-readable statements** (in a language) that can be converted to a collection of instructions to be executed by a computer/processor
- To the User?

The Organization of Software

- Program:
  - Any piece of software that can "**run on its own**" (including utilities, tools, scripts, apps, etc...)
- Sub-Program:
  - A collection of instructions/statements that implement an **algorithm** for accomplishing **a specific task or tasks**
- Library:
  - A group of related sub-programs for accomplishing **a specific collection of (usually related) tasks**

From Software to Software Product

- What People (Other than Us) Care About:
  - Having ***everything*** they need to **solve** one or more problems or achieve one or more goals (i.e., a complete means to one or more ends)
- Definition:
  - A *software product* is one or more programs, sub-programs, or libraries, along with the data and supporting materials and services, that a client can use to **solve problems** or **achieve goals**

Kinds of Software Products

- Bespoke:
  - Software products that are developed (usually under contract) for **a specific client**
- Generic:
  - Software products that are developed (usually speculatively) and then sold to **market** clients (either a mass market or a niche market)
- Examples?

Software Engineering

- Defined:
  - The application of (scientific) theories, methods and tools to the specification, design, creation, verification/validation, deployment, operation, and maintenance of software products
- Scope:
  - From specification to maintenance
  - Involves technical and managerial concerns
  - Involves theories/methods from psychology, mathematics/statistics, computer science, and management (of people and resources)
  - Consists of science and art

Software Engineering vs. Computer Science/Engineering

- Computer Engineering:
  - The application of theories (often from physics) to the creation of **computational devices**
  - Usually thought of as a subset of electrical and electronic engineering
- Computer Science:
  - The **theories and methods** that underlie computation and the use of computational devices

Why Software Engineering?

- My experience

2. System

Motivation

- Some Commonly Used Phrases:
  - The systems needs to...
  - ... is a complex system...
- An Interesting/Troubling Observation:
  - Most people can't define the word "system"

Systems

- System
  - A set of entities, their attributes, and the relationships between them.
- *Entities*
  - The **components** of the system that are **known/observed**.
- *Attributes*
  - The **properties** of the entities (i.e., the external manifestations of the way the objects are known or observed).
- *Relationships*
  - The **bonds** that link entities and attributes

Abstraction

- A Definition:
  - Abstract (verb): To extract what is important
- Two Observations:
  - An object/concept can be conceptualized in an enormous number of ways
  - The *level of detail* (which is the inverse of the *level of abstraction*) is a defining feature of the system
- Example: A Car
  - Low Detail - A source of emissions
  - Medium Detail - A contributor to traffic congestion at an intersection
  - High Detail - A mechanical motive device

Systems (cont.)

- We gauge the level of detail/abstraction by viewing the system under consideration in relation to:

Environment	The set of all other systems (also known as the context and the domain)
Subsystems	Parts of the whole which display a richness of interrelationships
Elements/Atoms	The smallest parts of the system (i.e., the black boxes)

- Example: A Car
  - Low Detail - A source of emissions
  - Medium Detail - A contributor to traffic congestion at an intersection
  - High Detail - A mechanical motive device

Physical and Conceptual Systems

- Physical Systems:
  - Can be "pointed to"
  - Include entities that exist in space and time
- Conceptual Systems:
  - All other systems

System State

- Defined:
  - An unvarying (over space, and/or time, and/or some other dimension) description or summary of that system
- Specification:
  - All of the attributes (at an appropriate level of abstraction) of the entities that **comprise** the system

3. Modeling

A Definition

- A mini-activity:

- A *model* is a **description** of a system (that is created to help understand the system)

Modeling Methods

- Physical Models:
  - Paper, clay, plastic, wood, metal, etc...
- Conceptual Models:
  - Mathematical, computational/numerical, diagrammatic/visual/graphical, textual, etc...

Model Types

- Static:
  - Models of the aspects of the system that **do not change** (sometimes called *structural*)
- Dynamic:
  - Models of how the system **changes** over time (sometimes called *behavioral*)

Importance of Modeling

- Some Observations:
  - Complex systems are hard to understand
  - Models can be used to make them easier to understand
- An Example:
  - You can probably build a doghouse without a model
  - A house would be very difficult to build without a model
  - A skyscraper would be impossible to build without a model

Modeling and Software

- A Common Problem:
  - Start with a doghouse
  - Keep adding on until you have a skyscraper
- What Models Provide:
  - An understanding of a system (as it is or should be)
  - A specification of a system
  - A template for construction
  - A record of the decisions that were made

Abstraction Revisited

- The same object/concept can be conceptualized at different levels of abstraction
- A system that is conceptualized at a particular level of abstraction can be modeled at different levels of abstraction
- Examples?

Abstraction Revisited: An Example

- Viewing a Car as a System:
  - Low Detail - A source of emissions
  - Medium Detail - A contributor to traffic congestion at an intersection
  - High Detail - A mechanical motive device
- Modeling a Car as a Source of Emissions
  - Low Detail - Relate the number of cars to total emissions
  - Medium Detail - Consider the number of miles and driven and the average drive cycle (e.g., EPA estimates)
  - High Detail - Consider car following behavior and engine performance

Announcement

- Office hours:
  - M/W 10:30am – noon

UML Activity Diagrams
An Introduction

Overview

- Purpose:
  - Describe a task/procedure/process
- Components:
  - Activity - a non-atomic (i.e., decomposable) task
  - Action - an atomic (i.e., not decomposable) task
  - Data - inputs to or outputs from actions (and may include state information in square brackets)
  - Flow - flow of control and flow of data

An Example

Representation and Interpretation

- Directed Graph:
  - Nodes/Vertexes - represent actions or objects
  - Edges/Arcs/Links - represent control flows and/or data flows
- Tokens:
  - Produced and consumed by nodes (though not represented on the diagram)
  - Flow instantaneously along edges
- Execution:
  - An action node begins executing when tokens are available on all incoming edges
  - When an action node begins execution it consumes all incoming tokens
  - While/after an action node executes it produces outgoing tokens

Symbology

- Activities:

- Actions:

- Objects/Classes/Data:

- Flows:

Additional Elements

- Initial Node: ●
- Final Node: ◉
- Decision Node: ◊
  - One incoming edge
  - One or more outgoing edges
  - Outgoing edges can have a Boolean *guard*
- Merge Node: ◊
  - One or more incoming edges
  - One outgoing edge

An Example Revisited

Why Use

- Compare th

An action does NOT start until the tokens on all of its incoming edges have arrived.

Decision/Merge nodes do NOT consume incoming token, but pass it along.
While actions consume incoming token and produce a new one after execution.

Still More Elements

- Pin:
  - A terminator for data flows into (input pins) or out of (output pins) an action node
- Activity Parameter:
  - An object node on the boundary of an activity
  - Contains the name of a particular object and the name of its type
  - Syntax: *name:type*

A Larg

Courtesy of *Introduction to Software Engineering and Design* (Fox, 2006)

The Larger Nodes

Sequential and Parallel Actions

- Fork Node:
  - Splits a single (incoming) flow into multiple concurrent/parallel (outgoing) flows

- Join Node:
  - Synchronizes multiple (incoming) flows into a single (outgoing) flow

An E

Prepare Breakfast Special
order
Announce Order
Cook Bacon
Cook Eggs
Cook Hash Browns
Plate Meal
meal

Do Laundry
dirtyClothes : Clothes
Sort Clothes
Wash Whites
Wash Colors
Dry Clothes
Fold Clothes
cleanClothes: Clothes

Do Laundry
dirtyClothes : Clothes
Sort Clothes
Wash Whites
Wash Colors
Dry Clothes
Fold Clothes
cleanClothes: Clothes

Do Laundry
dirtyClothes : Clothes
Sort Clothes
Wash Whites
Wash Colors
Dry Clothes
Fold Clothes
cleanClothes: Clothes

Abstraction and Sub-Activities

- Abstraction in Activity Diagrams:
  - As with any model, a decision must be made about how much detail to include
- Associating an Activity with an Action:
  - It's possible to indicate that a sub-activity exists for an action using the following notation:

In-class Activity

Exercises

- 1. Actions and activities are the same. **(F)**
- 2. There can be several activity final nodes in a diagram. **(T)**
- 3. There can be several initial nodes in a diagram. **(T)**
- 4. A decision node always has two branches. **(F)**
- 5. In general, there are as many merge nodes as decision nodes in a diagram. **(T)**

Act00-Activity Diagrams

- Draw an activity diagram describing an algorithm for finding the maximum value in a list. The input is the list, and output is the maximum value.

- Draw an activity diagram describing two people peeling a pot of potatoes. Note that two people do this in parallel and they stop when all the potatoes are peeled.

Software Processes

Background

- A Definition:
  - A *process* (a.k.a. *activity*) is a **collection of related tasks** (a.k.a. actions) that transforms a set of **inputs** into a set of **outputs**.
- Processes and Algorithms:
  - The set of algorithms is a **subset** of the set of processes (which, includes, for example heuristics)
- Describing Processes:
  - Specify the **inputs** to and **outputs** from the process
  - Specify the **tasks**
  - Specify the inputs to and outputs from each task (a.k.a. **data flows**)
  - Specify the **order** of tasks and the **conditions** under which they occur (a.k.a. **control flows**)

Think of a process that you have been through

- In terms of:
  - **Inputs** to and **outputs** from the process
  - **Tasks**
  - **Data flows -** inputs to and outputs from each task
  - **Control flows -** the **order** of tasks and the **conditions** under which they occur

Recall the Card Game…

- Input:

- Output:

Your Plans – Your Processes

- **Company Alpha**

- **Company Beta**

Plans:

Player A can spread and flip cards.
Player B can touch ”Club” only.
Player C can touch ”Spade” only.
Player D can touch “Heart” only.
Player E can touch “Diamond” only.
Order to touch cards: A –> B –> C –> D –> E

Tasks

Control Flow

Form your own plan.

Software Processes - Overview

- Definition of a Software Process:
  - A set of actions/tasks (and corresponding inputs and outputs) that results in the specification, development, validation, and/or evolution of a software product
- An Observation:
  - Software processes are very complex and, hence, our discussion of them will necessarily involve abstraction (i.e., we will really be discussing process models)

Common Actions/Tasks in Software Processes

- **Project Planning**:
  - Financial/economic analysis
  - Scheduling
  - Resource allocation
  - Cost estimation
  - Risk management
- **Product Design**:
  - Identification of needs and desires
  - Specification of requirements
  - Prototyping

Common Actions/Tasks in Software Processes (cont.)

- **Engineering Design:**
  - Creation of static models
  - Creation of dynamic models
  - Consideration of architectural styles and design patterns
- **Implementation:**
  - Development
  - Code/documentation management
  - Debugging
  - Verification and validation
- **Deployment**
- **Support and Maintenance**

Models of Software Processes

- As Always:
  - Since they are models they are **simplifications** (and some are very abstract)
- Types of Models of Software Processes:
  - Descriptive - what actually happens
  - Prescriptive - what should (i.e., is supposed to) happen

The Waterfall Model

SRS: Software Requirements Specification
SDD: Software Design Document

The Waterfall Model (cont.)

- Questions:
  - Is it supposed to be descriptive or prescriptive?
  - How idealized/abstract is it?

Pros:

- Predictable
- Easy to estimate
- Easy to fix
- Documentation
- Distributable

Cons:

- Requiring stable Req.
- Requiring complete & correct Req.
- Documentation
- Communication overhead
- Management
- Taking too long

So the Waterfall model works, but…

Prototyping

- A **prototype** is a working model of some or all of a finished product.
  - **Throwaway** prototypes
  - **Evolutionary** prototypes

Prototyping

Prototyping

- Pros
  - Easy to handle changes
  - Getting a working product quickly
  - Less documentation or management required
  - Lightweight
- Cons
  - Hard to predict (time & cost)
  - Product may be unmaintainable
  - Little discipline -> bad quality control -> buggy product

So customers like prototypes a lot, but…

Risk Management

- Risk
  - An occurrence with negative consequences.
- Risk management
  - The practice of identifying, analyzing, and controlling or mitigating risks
  - Can (and should) be incorporated into any life cycle processes. For example, prototype evolution model.

The Spiral

- Boehm (1988)

The Spiral Model

- Boehm (1988)

The Spiral Model (cont.)

- Contributions:
  - Explicit incorporation of increases in fidelity/detail
  - Explicit incorporation of *risk*
  - One of the early *iterative* processes
- Cons:
  - Not very many people are trained in and good at risk management
  - Very general and adaptable, demands expertise in tailoring software processes

Iterative Process

Incremental Processes

- Definition:
  - An *incremental process* is one that produces its output **in phases** and the output of each phase is **a working version** of the software product
- Example:
  - When a customer orders the breakfast special, the chef can first cook and deliver some of the eggs, then cook and deliver the rest of the eggs and the bacon (an incremental process) or cook all of the eggs and bacon and deliver them at the same time (not an incremental process)

Iterative vs. Incremental Software Processes

- Iterative Processes:
  - Any software process can be made iterative and most are naturally iterative
- Incremental Processes:
  - Software processes are sharply distinguished by whether they are incremental (e.g., the waterfall process is not -- the intermediate outputs are not versions of the product)

The Rational Unified Process (RUP)

- An iterative & Incremental Process
  - Jacobson, Booch, Rumbaugh (1999)

RUP Stages

- **Inception**:
  - Develop use cases
  - Estimate the schedule and resource requirements
  - Estimate risks
  - Create a business case
- **Elaboration**:
  - Understand the problem domain
  - Create baseline design
  - Create test plan
  - Decide upon metrics
  - Organize resources

RUP Stages (cont.)

- **Construction**:
  - Complete the implementation
  - Release code to α testers
  - Identify remaining activities
- **Transition**:
  - Complete the final release
  - Confirm user readiness and acceptance
  - Confirm support readiness

The Rational Unified Process (cont.)

- Iteration:
  - Can occur within a phase
  - Can occur between phases
- Core Process Workflows:
  - Business modeling
  - Requirements
  - Analysis and design
  - Implementation
  - Testing
  - Deployment
  - Configuration and change management
  - Project management

PHASES
Inception
Elaboration
Construction
Transition
DISCIPLINES
BUSINESS MODELLING
REQUIREMENTS
ANALYSIS & DESIGN
IMPLEMENTATION
TEST
DEPLOYMENT
CONFIGURATION & CHANGE MANAGEMENT
PROJECT MANAGEMENT
ENVIRONMENT
Initial
Elab #1
Elab #2
Const #1
Const #2
Const #3
Tran #1
Tran #2
ITERATIONS

RUP

- Pros
  - Fully specified process, comes with templates
  - Adaptable to projects of different sizes/needs
  - Less risk
  - Getting usable product quickly
  - Iterative
- Cons
  - Requiring knowledgeable and experienced developers
  - A lot of documentation and management effort (heavyweight)
  - Cannot handle requirements change very well

OK, iterative & incremental processes have their strength, but…

Agile Processes

- A Generic Model

Extreme Programming - XP (Beck; 2004)

- Basic Principles:
  - Ensure **rapid** feedback
  - Embrace **change**
  - **Collective** ownership
- Practices:
  - **Small releases** (every release should be as small as possible containing the most valuable business requirements)
  - Have a **metaphor** for the project (that replaces the architecture)
  - **Unit tests** for every production module and functional tests for features
  - Make a **small initial investment** (i.e., the simplest solution that works is the best) and **refactor** (i.e., improve) later
  - **Pair programming** (all production code is written with two people working at one computer; one person thinks about the simplest implementation the other things strategically)
  - **Continuous integration** (code is integrated and tested every few hours)
  - Strict **coding standards**

Extreme Programming (cont.)

- Participants in the Planning Process:
  - Business
  - Development
- Planning Tools:
  - **Story cards** describe the necessary functionality (in the form of a task description) and an estimate of how long the functionality will take to implement
- The Planning Process:
  - Exploration - found out what new things the system could do
  - Commitment - Decide what subset of all possible requirements to pursue next (Business sorts by value; Development sorts by risk
  - Steer - Guide the development (involves iteration, recovery, new story construction, and re-estimation

Extreme Programming (cont.)

- Story board

Scrum (Sutherland and Schwaber; 1995)

- **Sprints**:
  - Fixed duration development cycles (usually 1-4 weeks) that end on a specific date (whether the work has been completed or not)
- Scrum Roles:
  - **Product Owner** (responsible for achieving maximum business value)
  - Team (multi-functional; 5-10 people)
  - ScrumMaster (helps the Team be successful; protects the Team)
- Process:
  - Start of the Sprint: There is a Sprint Planning Meeting at which the Team selects items from a prioritized list of requirements/features and commits to completing them
  - Each Day: Members of the Team report (at a standup meeting of 15 minutes or less) on progress

Scrum (cont.)

- Terminology:
  - The prioritized list of requirements/features is called the **Product Backlog**
  - The tasks being completed are called the **Sprint Backlog**
  - The daily meeting is called the **Daily Scrum**
  - Members of the Team are sometimes called "Pigs" (because they are truly committed to making bacon and eggs) and everyone else is called a "Chicken" (because they are only involved in making bacon and eggs)
- Some Details about the Product Backlog:
  - The Product Backlog includes a rough estimate of the "**size**"
  - The Product Backlog also contains developer "**to dos**", exploratory work, and known bugs
  - The Product Backlog is updated by the Product Owner

Scrum (cont.)

The Scrum Process
Product Vision
Create Product Backlog
Product Backlog
Sprint Planning
Sprint Backlog
Sprint Execution
Product Backlog
Shippable Product
Sprint Review
Sprint Backlog
Product Backlog
Sprint Retrospective
[else]
[project finished]

Scrum - Sprint Planning Meetings

- Team members report on their **availability**
- The top item in the product backlog is broken down into tasks that are recorded in the Sprint Backlog
- Team members volunteer for the tasks (i.e., take ownership of them) and provide time estimates
- The process continues until all available hours are accounted for (Note: Since commitments are being made these meetings can take hours)

Scrum - Sprints

- The team should strive to have a working product at the end of every sprint
- The Sprint Backlog contains, at a minimum, the tasks, the name of the task owner, the initial estimate, and the hours of work remaining

				Hours of Work Remaining		
Product Backlog Item	Task	Owner	Initial Estimate	Day 1	Day 2	Day 3

Scrum – Sprints (cont.)

- Teams often post a **Sprint Burndown Chart** that graphically illustrates the numbers of hours remaining
- It is very important that the team have a definition of "Done" that can be used to assess when work is complete on the current increment

Nerd Humor - From Product Backlog to Sprint Backlog

IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.

Courtesy of xkcd

Agile Processes

- Pros:
  - Handle changes very well
  - Quick delivery of a working product to customer
  - Can recognize bad projects early (then cancel them)
  - Reduce waste and duplication of effort
- Cons:
  - Customers and users have to commit time & effort
  - Incremental design may not be a good thing
  - Difficult to use on large projects
  - Hard to predict outcomes

"Heavyweight" vs. "Agile"/"Lean" Methods

- A Common Categorization:
  - Heavyweight: Waterfall, Incremental Delivery, Spiral, RUP
  - Agile/Lean: XP, Scrum
- The Agile Manifesto:
  - Value individuals and interactions over processes and tools
  - Value working software over comprehensive documentation
  - Value customer collaboration over contract negotiation
  - Value responding to change over following a plan
- Another View:
  - The different process involve the same actions/tasks but vary dramatically in the cycle times

Act 01 Software Processes

Comparison

Waterfall Lifecycle Model
Product Vision
Requirements
SRS
Design
SDD
Implementation
Code
Testing
Software Product
Maintenance

Prototype Evolution Lifecycle Model
Product Vision
Design
Implementation
Prototype
Demo & Evaluation
[complete]
[else]
Software Product

Comparison (cont.)

Comparison (cont.)

PHASES
Inception
Elaboration
Construction
Transition
DISCIPLINES
BUSINESS MODELLING
REQUIREMENTS
ANALYSIS & DESIGN
IMPLEMENTATION
TEST
DEPLOYMENT
CONFIGURATION & CHANGE MANAGEMENT
PROJECT MANAGEMENT
ENVIRONMENT
Initial
Elab #1
Elab #2
Const #1
Const #2
Const #3
Tran #1
Tran #2
ITERATIONS

Comparison (cont.)

Software Product Requirements

Definition

- A **requirement** is a property or **behavior** that something must exhibit—it is something demanded or necessary.

- A **Specification** is a precise **description** of something.

- A **requirements specification** is a precise description of properties or behaviors that something must have.

Required According to Whom?

- A **product stakeholder**, or just a **stakeholder** for short, is anyone affected by a product or its development.
  - Customers
  - Users
  - Clients
  - Developers
  - Regulators
  - Marketers

Required According to Whom? (cont.)

- Stakeholders have a say in the requirements for a product, but they **can't** have the final word. **Why?**
  - Conflicting desires and ideas;
  - Incomplete and incorrect;
  - Too abstract.
- Hence, we need to distinguish between ***stakeholder needs*** and ***requirements***.
  - Stakeholder needs are features or properties that **particular** stakeholders desire in a product
  - Requirements are product features or properties agreed by **all**.

Functional vs. Non-Functional

- **Functional needs and requirements** are about how a software product must map inputs to outputs.

- **Non-functional needs and requirements** are about *properties* that a software product must have.

Levels of Abstraction in Specifications

- Software engineers have terms for specifications at various levels of abstraction.
  - Business requirement specifications
  - User-level needs and requirements
  - Operational-level needs and requirements
  - Physical-level needs and requirements.

Requirements in Traditional Processes

- In the waterfall model, *all* the requirements for the product (or as near as possible) are determined before almost anything else is done.

Requirements in Traditional Processes (cont.)

- In the waterfall model, *all* the requirements for the product (or as near as possible) are determined before almost anything else is done.
- This has often not worked well.
  - It is very **difficult (perhaps impossible) to determine all** requirements for non-trivial products before any implementation is done.
  - Stakeholder needs and desires **changes** over time.
  - **Documenting** all the requirements is difficult.
  - The whole process takes **too long**.

Requirements in Agile Processes

Requirements in Agile Processes (cont.)

- Scrum can make requirements specification and change as easy and cheap as possible, by
  - *Delay **choosing requirements** as long as possible*
  - *Delay **refinement** as long as possible*
  - *Avoid writing requirements **altogether***
  - *Determine requirements in light of current product features*

Stating Specifications in Traditional Processes

- Most specifications are stated in English (or other natural languages).
- English specifications may be vague or ambiguous, and still hard to understand.

Stating Specifications in Traditional Processes

- Rules to specification writing:
  - Write complete, simple sentences in the **active voice**.
  - Define terms clearly and use them **consistently**.
  - Use the **same** words for the same thing—avoid synonyms.
  - **Group** related material into sections.
  - Use tables, lists, indentation, white space, and other formatting aids to present and organize material clearly and concisely.
  - Expressing all specifications using the words "**must**" or "**shall**."
  - Specifications should be **testable** or **verifiable**.
  - Specifications should be **easy to understand and to change**.
  - Each specification should state only a single need or requirement – **Atomic**.
  - Each one should have a unique **identifier**.

Stating Specifications in Traditional Processes

- Recording Requirements as Structured Text

Nonfunctional Requirements by Type

1. The product must be accessible as a WWW page.
    - 1.1. The main file for the product must be written in HTML. (S1, S2, S3, S4, S5, S6, S7, S8, S9)
    - 1.2. The form/presentation of the user interface must be written in CSS.
    - 1.3. The algorithms required by the product must be written in Javascript.

Functional Requirements by Type

1. Startup Requirements
    - 1.1. Startup Study Guide Requirements
        - 1.1.1. The product must be able to start with out a study guide.
        - 1.1.2. The product must be able to open a particular study guide at startup. (U3)
            - 1.1.2.1. It must be possible to identify the guide in the URL of the main file.
    - 1.2 It must be possible to include start-up option the URL of the main file.
        - 1.2.1. It must be possible to specify the default server as a start-up option.
        - 1.2.2. It must be possible to specify the directory as a start-up option.

Stating Specifications in Traditional Processes

- Recording Requirements as Use Cases
- What is a use case?
  - An **abstract episode or interaction** between a system and its environment
  - A **description** of one or more **sequences of actions** that a system performs to yield an observable result
  - A **dialog** that a system and its environment **share** as they interact
  - A way of **using** a system

Use Cases (cont.)

- Properties of Good Use Cases
  - A use case should be an **entire transaction**
    - A withdrawal from an ATM machine
    - A telephone call
  - The collection of uses cases should **entirely characterize** the functionality of the system

Use Cases (cont.)

- Developing Use Cases
  - Context
  - Outside entities
  - System behaviors
- Identifying the Actors
  - Principal actors
  - Secondary actors
  - External hardware
  - Other Systems
- Identifying the Flow
  - The beginning of the use case
  - The origin of information
  - The exchange of information
  - The end of the use case

An example of use case diagram for Bank ATM subsystem - top level use cases

Credits to uml-diagrams.org

Use Cases (cont.)

Bank ATM Transactions and Customer Authentication Use Cases

Use Cases (cont.)

Bank ATM Maintenance, Repair, Diagnostics Use Cases

Credits to uml-diagrams.org

https://www.uml-diagrams.org/bank-atm-uml-use-case-diagram-example.html?context=uc-examples

Stating Specifications in Agile Processes

- A **user story** is a description of a function a product must provide for a user.
- User stories can be expressed at many levels of abstraction.
  - Epics
  - Features
  - Sprintable (implementable) stories

Stating Specifications in Agile Processes (cont.)

- User stories are usu
  - As a <*role*>, I want
  - E.g. - As a bank tell customer inquiries.
- User stories, in part **acceptance criteria**
- Three Cs of user sto
  - Card
  - Conversation
  - Confirmation

Eliciting Stakeholder Needs in Traditional Processes

- Common techniques:
  - Interviews
  - Observation
  - Focus groups
  - Workshops
  - Prototypes
  - Document studies
  - Competitive product studies

Eliciting Stakeholder Needs in Agile Processes

- Needs are elicited **continuously** during development.
- Pros:
  - Stakeholders can express needs that have only just arisen in response to the **current** situation
  - Stakeholders can see the **current** state of the product and react to it in formulating needs
- Cons:
  - Stakeholders may **over-react** to recent events, losing track of the big picture.
  - Agile methods usually place great trust in a few (**not all**) stakeholders or stakeholder representatives.

Verifying and Validating Requirements in Traditional Processes

- Five important characteristics:
  - Clarity
  - Consistency
  - Completeness
  - Correctness
  - Well-formedness
- Main technique - reviews

Verifying and Validating Requirements in Agile Processes

- The most careful check of requirements is during the reviews at the **conclusion** of each sprint.

- Major threat to this approach
  - Usually, **not all** stakeholders can attend every sprint review.

Requirements Management in Traditional Processes

- A project is launched with a **product mission statement**.
- Then One or more require
  requirements analysis.
  - **Requirements analysis** is t
    understanding stakeholder
    and evaluating them to en
    correct, and well formed.
  - The output is a **software r**

1. Introduction
   - 1.1 Product Vision
   - 1.2 Project Scope
   - 1.3 Stakeholders
   - 1.4 Design and Implementation Constraints
2. Functional Requirements
   - 2.1 Product Behavior
   - 2.2 User Interfaces
   - 2.3 System Interfaces
   - 2.4 Data Requirements
3. Non-Functional Requirements
4. Other Requirements
5. Glossary

Requirements Management in Agile Processes

- Agile projects begin with some project mission statement in the form of **epics** or **feature**.
- The product owner refines these high-level features into **user stories**.
- During each sprint, user stories are refined into **physical-level** requirements and then implemented.
- At the conclusion of each sprint, stakeholders and the product owner validate requirements during sprint review.
  - Requirements management is a **core activity** of agile methods, but it is incorporated into the broader processes of backlog grooming and sprinting, and not distinguished as a separate process.

Requirements vs. Product Design

- Are they the same thing?

Interaction Design

Overview

- **Interaction design**
  - is concerned with specifying the **user experience** for a software product
  - is concerned with the **visual and aural appearance** of the program and with the **sequence of events** that occur as a user and a program exchange data
  - Specifies both the **appearance** and the **behavior** of products.

Overview (cont.)

- The **quality** of a user interface:
  - Effectiveness
  - Efficiency
  - Safety
  - Learnability
  - Memorability
  - Enjoyability
  - Beauty
- Related fields:
  - Ergonomics
  - Perceptual physiology & psychology
  - Cognitive psychology
  - Graphic design
  - …

Interaction Design In the Life Cycle

- In traditional approaches –
  - Done as an **after-thought** or a mini-phase between requirements specification and engineering design.
- A better approach –
  - Use interaction design as a **driver** of requirements specification.
  - In SCRUM, interaction design should be conducted during a sprint for the features to be implemented in the following sprint.

Interaction Design Models and Notations

- Static model – Use case diagrams
  - Are an easy way to represent **collection of features**
  - Several diagrams can represent different collections of features as **design alternatives**

Static Model (cont.)

- Screen (or page) layout diagrams
  - Drawing of (part of) a product's **visual display** when it is in a particular state.
  - Are easy to draw and manipulate, so they are good tools for considering many **design alternatives**
  - Are a great help to stakeholders in **visualizing** what a product will look like
- Can be drawn at various levels of abstraction
  - A good starting point – **Wireframe**
  - A high-fidelity diagram showing colors, fonts, controls, and so forth can be made as the **basis for implementation**

Static Model (cont.)

A Wireframe for a Screen in a Document Retrieval System

Dynamic Model

Use Case Name: To identify the use case

Actors: The agents participating in the use case

Stakeholders and Needs: What this use case does to meet stakeholder needs

Preconditions: What must be true before this use case begins

Post conditions: What will be true when this use case ends

Trigger: The event that causes this use case to begin

Basic Flow: The steps in a typical successful instance of this use case

Extensions: The steps in alternative instances of this use case occurring either because of variations in the normal flow or because of errors.

Dynamic Model (cont.)

- Use case descriptions can
  - Provide the notation for the **next step** in product design after choosing features.
  - Be used to describe the **interactions** realizing by each feature.
  - Be used to explore **alternative** interaction flows.
  - Have a big influence over many interaction design **criteria**, such as efficiency, safety, learnability, memorability, and enjoyability.

Dynamic Model (cont.)

- Storyboards are an excellent tool for investigating design alternatives and for helping stakeholders visualize designs before their implementation.

Use Case Models vs. Requirements

- Should use case models be used as requirements specification?
  - Yes?
    - They can model the interactions between a system and its users and other things with which it exchanges data.
    - They are valuable for interaction design, and can document some requirements.
  - No.
    - They are not able to model requirements that don't have to do with interactions.
      - E.g., algorithms, non-functional requirements
    - They are organized to **trace interactions** and not to specify behaviors in relation to system features.
- Hence, use case models cannot capture all requirements, and they may not always be the best vehicle for expressing interaction requirements.

Use Case Models vs. Requirements (cont.)

- In SCRUM
  - A PBI may specify **some aspect** of a product's interaction with its environment
  - Use case descriptions can help
    - capture details of interaction, and so they are a good tool for **elaborating** interaction PBIs during sprints.
    - provide **supplementary** information if a PBI is elaborated to a fairly low-level of detail during backlog grooming.

Interaction Design Processes

- A **top-down** activity:
  - It begins with the **most abstract** models and gradually refines them until a **detailed specification** is complete.
  - A use case diagram can be formulated to **pin down** product features and capabilities.
  - Use case descriptions should follow as a means to determine the **best overall flow** of interaction.
  - Screen layout diagrams and storyboards can be made to **aid** in design, to obtain **feedback** from stakeholders, and as specifications for implementation.

Interaction Design Processes (cont.)

- **Usability testing**
  - It is **empirical evaluation** of (parts of) an interaction design to determine whether it meets interaction design goals.
  - A **testing scenario** can be “executed” on anything from a rough paper prototype to a full-featured product.
  - Measurements:
    - Pre- and post- test questionnaires or surveys
    - Observation of user actions
    - How long it takes users to achieve goals
    - Number of user mistakes
    - …

Interaction Design Principles

- General principles for p

- **Consistency**—Designs that present similar data in similar ways, and provide similar ways of accomplishing similar tasks, are better.

Interaction Design Principles (cont.)

- Principles that apply in particular to appearance – CAP
  - **Contrast**—Designs that r
    better.
  - **Alignment**—
    elements in
  - **Proximity**—Designs that group related items together spatially are better.

Interaction Design Principles (cont.)

- Principles that apply especially to behavior – FeVER
  - **Feedback**—Designs that acknowl
  - **Visibility**—Designs that
    display their state and a
  - **Error Prevention and Re** Designs that prevent
    user errors and provide error recovery mechanisms are
    better.

Open Discussion

- The “Best” design
  - Does it exist?
  - How to evaluate a design?
    - Effectiveness
    - Efficiency
    - Safety
    - Learnability
    - Memorability
    - Enjoyability
    - Beauty

Software Quality Assurance

Getting Started

- Definition:
  - *Quality assurance* (QA) is **a systematic pattern** of activities intended to ensure that a product properly satisfies the needs and desires of its stakeholders.
- Activities:
  - ***Validation*** is the process of determining if a product (or its specification) satisfies stakeholders' needs and desires ("**Are we building the right product?**")
  - ***Verification*** is the process of determining if a product (or its specification) satisfies those needs and desires properly ("**Are we building the product right?**")

An Example

- The Setting:
  - You are a newspaper reporter and you are told to write an article about homelessness (the product)
- Validation:
  - You must write about homelessness
  - Note: **All validation activities are product specific**
- Verification:
  - You must write properly (e.g., use proper grammar, follow the newspaper's style guide, make a compelling argument)
  - Note: Some verification activities are product-specific (e.g., the quality of the argument) and **some are not** (e.g., proper grammar)

Software Quality - ISO/IEC 25010

- Functional Suitability:
  - Functional completeness, correctness, and appropriateness
- Performance Efficiency:
  - Processing times and throughput rates, resources used, capacity
- Compatibility:
  - The degree to which the product can co-exist with and interoperate with other products

Software Quality - ISO/IEC 25010 (cont.)

- Usability:
  - Understandability, learnability, and operability
- Security:
  - Confidentiality, integrity, authentication, and accountability
- Maintainability:
  - Modifiability/Adaptability and reusability

Software Products vs. Large-Scale Engineering Products

- Examples:
  - Structures (bridges, buildings, dams)
  - Networks (e.g., roads, water, electrical, communication)
- How is Quality Defined?
  - In very similar ways
    - Functional Suitability
    - Performance Efficiency
    - Compatibility
    - Usability
    - Security
    - Maintainability

Software Products vs. Manufactured Products

- Other Products:
  - After the product is designed and engineered it must be produced/manufactured and each instance is different
  - The concern is often with the quality of the instances of the product (sometimes called "quality control")
- Software Products:
  - After the product is designed and engineered each instance is identical
  - The concern is with the quality of the product

Defect Elimination:

- Defects:
  - Any undesirable aspect of a product
- Defect Elimination Activities:
  - Detection and Removal
  - Prevention

Defect Elimination
Defect Prevention
Process Guides
Analysis and Design Methods
Reference Architectures
Design Patterns
Standard Data Structures & Algorithms
Software Reuse
Prototyping
Version Control
Configuration Management
CASE Tools
Training & Education
Defect Detection & Removal
Review & Correct
Style & Standards Checkers
Spelling & Grammar Checkers
Reviews
Desk Checks
Walkthroughs
Inspections
Test & Debug
Regression Testing
Unit Testing
Integration Testing
System Testing
Alpha Testing
Beta Testing

The Quality Tradeoff

- Costs of Low Quality Software:
  - Negligence and liability costs
  - Security costs
  - Disgruntled customers (hence lost sales)
- Costs of High Quality Software:
  - Prevention costs
  - Appraisal costs
  - Delays to market (hence lost sales)

Architectures & Architectural Styles

Definitions

- Architecture:
  - An abstract model of a system
- Architectural Style/Idiom:
  - Some architectures have **similar entities, attributes and relationships** which allow them to be grouped into a a *style* or *idiom* (i.e., when we abstract even further we see commonalities)
- A Common Mistake:
  - Using the term "architecture" when one really means "architectural style"

The Formalization of Architectural Styles

- Define a Vocabulary:
  - **Components** (e.g., modules, processes, tools, databases)
  - **Connectors** (e.g., calls, events, queries)
- Specify Constraints:
  - How components and connectors can/can't be combined (e.g., topological constraints)

Studying Architectural Styles/Idioms

- Rationale:
  - Makes it **easier to understand/discuss** the architectures of existing systems
  - Helps in the design of architectures for new systems
- Approach:
  - Identify the **commonalities** that define the style
  - Consider the style's advantages and disadvantages

Layered Architectures

Layered Architectures

- Commonalities:
  - Entities are grouped based on the **services** that they provide
  - Each layer hides all of the layers above and below
- Visualizations:

Wedding Cake Diagram

Architecture of an HVAC Unit

Layered Architecture (cont.)

- The Classic Three Tiers:
  - Presentation (Front End)
  - Application Logic
  - Storage (Back End)
- Visualization:

Layered Architectures (cont.)

- Visualization (cont.)

Onion Diagram

Layered Architectures (cont.)

- Advantages:
  - **Improved cohesion** which allows for multiple physical deployment options (e.g., presentation and application logic on one machine; storage on another)
  - **Decreased coupling** which promotes re-use
  - **Specialized expertise** (by level)
- Disadvantages:
  - **Communicating** through multiple layers can be inefficient
  - Often results in "cheating"

Client-Server Architectures

Client-Server Architectures

- Commonalities:
  - Each entity has exactly **one role**, service **provider** or service **consumer**
  - Service providers **wait for and then respond to requests**
  - Service consumers **initiate** requests
- The Alternative:
  - The entities are peers (i.e., all play the same roles)

Client-Server Example

- The WWW:
  - Identify the client(s)
  - Identify the server(s)
- X-Windows:
  - Identify the client(s)
  - Identify the server(s)

Client-Server Architectures (cont.)

- Advantages:
  - Same as for layered architectures
  - **Centralized** control
- Disadvantages:
  - "Single" point of failure
  - Difficult to scale (especially to balance loads)

Pipe and Filter Architecture

Pipe and Filter Architectures

- Commonalities:
  - Treat inputs/outputs as ***streams*** (i.e., infinite sequences of bytes)
  - Group entities into ***filters*** (i.e., those that **process streams**) and ***pipes*** (i.e., those that **connect sources and sinks**)
- A Visualization:

Pipe and Filter Examples

- The Problem:
  - Read a file and print all of the lines that contain a particular string. The output should be sorted alphabetically.
- The Solution (in Unix):

```
cat in.txt | grep "Overdue" | sort > out.txt
```

- A Diagram:

Pipe and Filter Examples (cont.)

- The Problem:
  - Read a file that contains Java source code and produce the associated byte code.
- A Diagram of the Java Compiler:

Pipe and Filter Examples (cont.)

- The Problem:
  - Read a file that contains invoices and payments and prints receipts and reminders.
- A Diagram:

Pipe and Filter Architectures (cont.)

- Advantages:
  - **Simple**
  - Easy to extend and/or modify
- Disadvantages:
  - Forces **lowest-common-denominator** data transmission
  - Queue **overflows**
  - **Unidirectional** data flow
  - Not good for **interactive** systems

Event Driven Architectures

Event Driven Architectures

- Commonalities:
  - Entities are grouped into ***event generators/producers*** and ***event receivers/handlers***
  - Generators and receivers communicate through an **intermediary** (called an *event queue*)
- Visualization:

- Terminology:
  - Event generators ***post/announce*** events
  - The event queue ***invokes/fires*** *to* event handlers/receiver

Event Driven Architectures (cont.)

- Bubbling/Hierarchical:
  - A (usually predefined) **hierarchy** of objects that handle events

- Listening:
  - The receivers/handlers (called *listeners*/*observers*) inform the queue (using a process called ***registration***) that they want to be notified of specific events

Event Driven Architectures (cont.)

- Advantages:
  - Promotes re-use
  - Easy to modify
- Disadvantages:
  - Makes **synchronization** difficult

MVC

Model-View-Controller

- Models – to **realize** problem-domain function
- View – to **display** data to users
- Controllers – to **receive and carry out** commands from users

Model-View-Controller (cont.)

- An Example

Model-View-Controller (cont.)

- Advantages:
  - Views and controllers can be **added, removed, and changed without disturbing the model.**
  - Views can be added or changed, even during **execution**
  - The components of the user **interface** can be changed, even at **runtime**.
- Disadvantages:
  - **Views and controllers are often hard to separate**
  - **Frequent updates may slow data display and degrade user interface performance.**
  - User interface components highly depend on model components.

Architectural Styles in Context

- Some Domains:
  - Instruction set architectures
  - Microarchitectures
  - Application architectures
  - Network architectures
- Some Observations:
  - Many styles are used in multiple domains
  - Some issues arise in some domains but not others (e.g., geometry matters in chip design but not in application programming)

Identifying Architectural Styles

- Abstract away from the architectures used in existing systems
- Create an idealization (sometimes called a ***reference*** *architecture*)

Software Engineering Design

Definitions

- Software Design:
  - Software design is the process of specifying the **nature** and **composition** of a software system that satisfies client needs and desires, subject to constraints. (Fox, 2006)
- Software Engineering Design:
  - Software engineering design is the process of specifying **programs** and **sub-systems**, and **their constituent parts and workings**, to meet software product specifications. (Fox, 2006)

The Process

- The general software Design process

Parts of the Process

- Analysis:
  - **Understanding** the problem
- Resolution:
  - **Solving** the problem

Design Principles

Evaluating Software Designs

Overview

- Rationale
- Qualitative Goals
- The Fundamental Principle
- Other Software Design Principles

Rationale

- **Generation** of Alternative:
  - Several alternative designs should/will be developed
- **Evaluation** of Alternatives:
  - Some "measure(s)" must be used to evaluate/compare alternatives
  - They can be quantitative or qualitative "measures" of quality

The Fundamental Principle of Engineering Design

- The Principle:
  - Designs should be ***modular***
- The Rationale:
  - **Understanding** is inversely proportional to scope/scale (due, in part, to human memory limitations of 7±2 "chunks")
  - **Small** (and simple) components are easier to debug and fix than large (and complex) components
  - **Reusability** is inversely proportional to scope/scale
  - Systems constructed from small (and simple) components are more **portable**

Modular Designs: Things to Think About

- Decomposability:
  - Do the modules aid in **decomposition** (i.e., top-down development)?
- Composability:
  - Do the modules aid the **constructions** of new systems (i.e., bottom-up development)?
- Continuity:
  - Do the modules "**localize**" small changes?
- Protection:
  - Do the modules **confine** problems?

Some Other Software Design Principles

- Standardization
- Simplicity
- Elegance/Beauty

Standardization

- The Principle:
  - Use standard designs, algorithms, data structures, and documentation practices if possible
- The Rationale:
  - **Easier** to understand and use
  - More likely to be **re-used**
  - More likely to be **portable**

Simplicity

- The Principle:
  - Simpler designs are better
- The Rationale:
  - **Easier** to understand and use
  - More likely to be **re-used**
- To Think About:
  - Have I developed a complex solution to a simple problem?

Elegance/Beauty

- The Principle:
  - Elegance matters
- The Rationale:
  - Elegant modules are more likely to be **re-used**
  - Elegance is usually directly proportional to **clarity**

Software Design Practices

- Information Hiding
- Coupling
- Cohesion

Information Hiding

- The Principle:
  - Hide the **internal** details of a component from all other components
- The Rationale:
  - **Prevents damage** from errant external code
    - You can't hurt what you can't see
  - Makes components **easier** to understand/use
    - It enhances abstraction (i.e., shields users from the details)
  - **Simplifies** modification and repair
    - Changing internal details should not have any impact other components
  - Facilitates **re-use**

Information Hiding (cont.)

- Examples of Private Information:
  - Local variables, data types, and data structures
  - Internal flow of control
  - Algorithms
- Examples of Public Information:
  - Interface information (inputs and outputs)
  - Behavior/Functionality
  - Errors and exceptions

Coupling

- The Principle:
  - **Minimize** the coupling between modules
- The Rationale:
  - Strongly coupled modules are difficult to **change**
  - Strongly coupled modules are difficult to **debug**
  - Weakly coupled/decoupled modules are **easier to understand**
  - Weakly coupled/decoupled modules are more likely to be **re-used**

Coupling (cont.)

- Techniques:
  - **Information hiding** decreases coupling
  - The use of **global entities** increases coupling
  - Modules that communicate **complex data** structures are more tightly coupled
- To Think About:
  - Could I use module A elsewhere without module B?

Levels of Coupling

- Direct Coupling:
  - Modules share variables
- Common Coupling:
  - Modules share **"global"/"common"** variables
- Import/Export Coupling:
  - Only share variables/operations that are explicitly "exported"

Cohesion

- The Principle:
  - The parts of a module should be **closely related** to each other.
- The Rationale:
  - Highly cohesive modules are **easier** to understand (like a well-written paragraph and document)
  - High cohesion tends to **reduce coupling** between modules
  - **Separation** of Concerns

Cohesion (cont.)

- Techniques:
  - **Separate** input, input validation, calculations, and output
  - Modules should not **cross** levels of abstraction
  - Design modules that relate to **real-world entities**
- To Think About:
  - Does this module have **a single, clear** responsibility?

Levels of Cohesion

- **Coincidental** Cohesion:
  - No meaningful connection
- **Logical** Cohesion:
  - Collection of functionally related components (e.g., I/O)
- **Temporal** Cohesion:
  - Functions are executed together (e.g., initialization)
- **Abstract** Cohesion:
  - Module provides services associated with a single data type

Exercise

Examples

- Design 1:

- Discussion:
  - It's OK but the methods should be static.

Examples (cont.)

- Design 2:

Utilities
indexOf (text : String, word : String) : int median (data : double[ ]) : double open (name : String) : FileHandle

- Discussion:
  - The class is not cohesive.

Examples (cont.)

- Design 3

- Discussion:
  - The classes are strongly coupled. The Calculator would not have to use messages[].

Examples (cont.)

- Design: 4

- Discussion:
  - It isn't cohesive. Though the buttons seem similar, they aren't. Some are in the elevator and some are outside of the elevator. So, it would be better to have an Elevator class and a CallStation class.

Examples (cont.)

- Design: 5

- Discussion:
  - It is tightly coupled. It could be improved by adding a Content interface.

Examples (cont.)

- Design: 6

- Discussion:
- It isn't a good design because, in the real world, Bold, Italic and Plain do not have a **part-whole** relationship to Paragraph.

Examples (cont.)

- Design: 7

- Discussion:
  - It is not cohesive. A better design would be to have a ZipCodeReader class and a PostageFinder class.

Design Patterns

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

In Our Class

- Iterator
- Factory method
- Composite
- Singleton
- Observer
- Strategy
- Decorator

Iterator

Motivation

- Computers vs. Calculators:
  - Computers can perform the same operation many times
- Programming Languages:
  - Harness this power using loops
- A Problem:
  - Traditional looping requires an understanding of the structure of the "aggregate"

Looping and Aggregates

- Looping Over an Array

```
1.          String     city;
2.
3.          for (int i=0; i < cities.length; i++)
4.          {
5.              city = cities[i];
6.              System.out.println(city);
7.          }
```

Looping and Aggregates (cont.)

- Looping Over an ArrayList

```
1.          String    city;
2.
3.          for (int i=0; i < cities.size(); i++)
4.          {
5.              city = cities.get(i);
6.              System.out.println(city);
7.          }
```

Looping and Aggregates (cont.)

- Looping Over a Linked Structure

```
1.          Node<String>        current;
2.          String              city;
3.
4.          current = first;
5.          while (current != null)
6.          {
7.              city = current.value;
8.              System.out.println(city);
9.              current = current.next;
0.          }
```

```java
String    city;

for (int i=0; i < cities.length; i++)
{
    city = cities[i];
    System.out.println(city);
}
```

```java
String    city;

for (int i=0; i < cities.size(); i++)
{
    city = cities.get(i);
    System.out.println(city);
}
```

```java
Node<String>      current;
String            city;

current = first;
while (current != null)
{
    city = current.value;
    System.out.println(city);
    current = current.next;
}
```

Some Observations

- Applications often "**loop**" over the same aggregate object in many classes and methods
- Changing from one aggregate to another is, as a result, very **inconvenient**
- The ***Iterator*** design pattern enables us to access the elements of an aggregate object while hiding its internal structure

Important Operations - Commonalities

- Reset its "pointer" (or cursor) to the first element.
- Determine if there are any more elements in the sequence.
- Move its "pointer" to the next element.
- Retrieve the "current" element.

```
1.          String    city;
2.
3.          for (int i=0; i < cities.size(); i++)
4.          {
5.              city = cities.get(i);
6.              System.out.println(city);
7.          }
```

```
1.          String    city;
2.
3.          for (int i=0; i < cities.length; i++)
4.          {
5.              city = cities[i];
6.              System.out.println(city);
7.          }
```

```
1.          Node<String>      current;
2.          String            city;
3.
4.          current = first;
5.          while (current != null)
6.          {
7.              city = current.value;
8.              System.out.println(city);
9.              current = current.next;
0.          }
```

The Iterator Pattern

Examples in Java

- The Aggregates:
  - Classes that realize the Iterable interface (including classes that implement the List interface like ArrayList and classes that realize the Set interface like HashSet.
- The Iterator:
  - Classes that realize the Iterator interface

Exar

- Note that the Map interface does not extend Iterable or Collection, but one can get a Set or Collection object from a Map.

Benefits of the Iterator

- What code must change to use a HashSet instead of an ArrayList?

```
1.      Iterable<String>    cities;
2.
3.      cities = new ArrayList<String>();
4.
5.      // Important code
6.
7.      // An example loop
8.      Iterator<String>    i = cities.iterator();
9.      while (i.hasNext())
0.      {
1.          String city = i.next();
2.          System.out.println(city);
3.      }
```

Benefits of the Iterator (cont.)

- The Principle Benefit:
  - You can **easily change the type of aggregate** being used without having to change any loops (and, if you declare variables properly, without having to change any of the declarations, only the instantiations)
- Other Benefits:
  - Several objects can be "looping" over the elements in the aggregate at the same time
  - A "filtered" list (e.g., names starting with the letter "A") is handled in exactly the same way that the "unfiltered" version is

The "New" Loop

- The Idea:
  - Include a control structure that takes advantage of the Iterator pattern (i.e., create a "**for each**" loop for any Object that implements the Iterable interface)
- Syntax:
  - for (*Class element* : *Iterable*) *statement*
- Example:

```
1.      Iterable<String>    cities;
2.
3.      cities = new ArrayList<String>();
4.
5.      // Important code
6.
7.      // An example loop
8.      for (String city: cities)
9.      {
0.         System.out.println(city);
1.      }
```

To Recap

- Problems that Iterator can solve:
  - The elements of an aggregate object should be **accessed** and **traversed** without **exposing its representation** (data structures).
  - New traversal operations should be defined for an aggregate object without changing its interface.
- Solution
  - **Define a separate (iterator) object** that encapsulates accessing and traversing an aggregate object.
  - Clients use an iterator to access and traverse an aggregate without knowing its representation (data structures).

Composite

Motivation

- The Need:
  - Ignore differences between **individual** objects and **compositions** of objects
- Examples:
  - Group "primitive" shapes in a drawing application

One Version

- In UML:

An Example

- A Message Distribution System

To Recap

- Problems that Composite pattern can solve:
  - A part-whole hierarchy should be represented so that clients can **treat part and whole objects uniformly**.
  - A part-whole hierarchy should be represented as **tree** structure.
- Solution
  - Define a **unified Component interface** for both part (Leaf) objects and whole (Composite) objects.
  - Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.

Singleton

Motivation

- Some applications require **exactly one instance** of a class:
  - Windowing systems with one event queue
  - Word processors with one menu bar (for all documents)
- Unfortunately, constructors can be called repeatedly:
  - In one method
  - In one class
  - Across multiple classes

Requirements of a Singleton

- Create and return an instance of itself if one doesn't exist
- Return the existing instance if one does exist

One Implementation of the Singleton Pattern

An Example

- A FileViewer

```
1.  public class FileViewer
2.  {
3.
4.      private static boolean      exists = false;
5.      private static FileViewer   instance;
6.
7.      private FileViewer()
8.      {
9.
0.
1.          exists = true;
2.      }
3.
4.  }
```

```
1.      public static FileViewer createInstance()
2.      {
3.          if (!exists) instance = new FileViewer();
4.
5.          return instance;
6.      }
```

An Example (cont.)

- Using a FileViewer

```
1.      public void valueChanged(ListSelectionEvent lse)
2.      {
3.          FileViewer   fv;
4.          String       fn;
5.
6.          fn = (String)list.getSelectedValue();
7.
8.          fv = FileViewer.createInstance();
9.          fv.load(fn);
0.      }
```

Thread Safety

- A Potential Problem:
  - If the createInstance() method might be called by multiple threads, problems can arise
- Resolutions:
  - Make createInstance() synchronized
  - Use eager initialization (i.e., instantiate instance when it is declared)
  - More: External Link

An Example (cont.)

- Using Eager initialization

```
1.      private static FileViewer   instance = new FileViewer();
```

```
1.      public static FileViewer createInstance()
2.      {
3.          return instance;
4.      }
```

To Recap

- Problems that Singleton pattern can solve:
  - How can it be ensured that a class has only one instance?
  - How can the sole instance of a class be accessed easily?
  - How can a class control its instantiation?
  - How can the number of instances of a class be restricted?
- Solution
  - Hide the constructor of the class.
  - Define a public static operation (getInstance()) that returns the sole instance of the class.

Observer

Motivation

- Objects often need to communicate with each other
- Traditional message passing techniques tend to couple objects too tightly

A Trading System Example

- A Bad Design

- Drawbacks:
  - Because TickReader calls the TickWriter and TickerTape it is not very re-usable
  - Every TickReader must have an associated TickWriter and TickerTape

A Trading System Example (cont.)

- A Better Design

- Advantages:
  - A TickReader need only have a list of (zero or more) TickListener objects that it will inform

The Observer Pattern

- Also known as Listener or Publish-Subscribe Pattern
- Intent:
  - Define a **one-to-many** dependency between objects so that when one object changes state, all of its dependents are notified
- Participants:
  - A subject
  - Some observers

The Observer Pattern (cont.)

A Complete Example

- A Silly Text Processor:
  - Counts the number of words that start with an uppercase letter
  - Save the lines to a file
  - Shows the progress (e.g., then number of lines processed)

A Complete Example (cont.)

- A Bad Design

A Complete Example (cont.)

- A Better Design

- This design is better
- It is, however, too tightly coupled

A Complete Example (cont.)

- Using the Observer Pattern

Decorator

Motivation

- Adding Capabilities to a Class:
  - Specialization
- Adding Capabilities to an Object:
  - Needs to be done "dynamically" (i.e., at run time)
- Example:
  - Borders, etc.. on GUI components

<<Interface>>
Component
+operation()
ConcreteComponent
<<abstract>>
Decorator
-decorated : Component
+Decorator(decorated : Component)
+operation()
operation() is delegated to decorated
ConcreteDecorator
-additionalAttribute
+additionalBehavior()
Could also override operation() to change its behavior

Abstract Classes

- An abstract class is a class that is declared abstract
  - it may or may not include abstract methods
    - An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

  - If a class includes abstract methods, then the class itself must be declared abstract
  - When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Abstract classes vs. Interfaces

- Use abstract classes if:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
  - An example of an abstract class in the JDK is AbstractMap, which is part of the Collections Framework. Its subclasses (which include HashMap, TreeMap, and ConcurrentHashMap) share many methods (including get, put, isEmpty, containsKey, and containsValue) that AbstractMap defines.

Abstract classes vs. Interfaces

- Use interfaces if:
  - You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.
  - An example of a class in the JDK that implements several interfaces is HashMap, which implements the interfaces Serializable, Cloneable, and Map<K, V>. An instance of HashMap can be cloned, is serializable (which means that it can be converted into a byte stream; see the section Serializable Objects), and has the functionality of a map.

An Example

<<Interface>>
Printer
+print(text : String)
ConsolePrinter
<<abstract>>
PrinterDecorator
-decorated : Printer
UppercasePrinter
+print(text : String)
WrappingPrinter
+print(text : String)
print() modifies the String and then delegates to decorated.print()

An Example (cont.)

- Implementation

Strategy

Overview

- The "Problem":
  - Have a family of *interchangeable* algorithms for accomplishing the same objective
- Examples:
  - Different text formatting algorithms (e.g., for paragraph formatting in a word processor)
  - Different metrics for finding the distance between, for example, colors

A Model of the Strategy Pattern

- In UML

An Example of the Strategy Pattern

- In UML

An Example (cont.)

- Implementation

Factory Method

Overview

- The "Problem":
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

A Model

Product
~Constructor()
Producer
+createProduct() : Product
creates
Constructor() usually has package visibility
createProduct() calls the Product Constructor()

A Generic Example

A Specific Example

- DirectoryListing

To Recap

- Problems that factory method pattern can solve:
  - How can an object be created so that subclasses can redefine which class to instantiate?
  - How can a class defer instantiation to subclasses?
- Solution:
  - Define a separate operation (*factory method*) for creating an object.
  - Create an object by calling a *factory method*.

Introduction to Java Swing

AWT vs. Swing

- AWT – Abstract Window Toolkit
- Swing – GUI Widget Toolkit

AWT vs. Swing (cont.)

AWT	Swing
Heavy-weight components	Light-weight components
Native look and feel	Pluggable look and feel
Does not have MVC	Supports MVC
Not available in AWT	With advanced features like JTable, JTabbedPane
Components are platform dependent	Components are platform independent
Requires java.awt package	Requires javax.swing package
Slower	Faster than AWT

An Introduction

- By the end of this class, you should know how to…
  - Create a window with menus
  - Use different layouts
  - Use action listeners

Windowing

Windows in Java

- Class Hierarchy

Windows in Java (cont.)

- **Window**
  - show()
  - hide()
  - dispose()
- Frame
  - getTitle(), setTitle()
  - getIconImage(), setIconImage()
  - isResizable(), setResizable()
  - isVisible(), setVisible()

Menus

- Using Something Like the **Composite** Pattern

- An Example – MenuBarFrame.java

Dialog Boxes

- Standard Types:
  - Message Dialog
  - Option Dialog
  - Input Dialog
  - File Dialog
  - Color Dialog
- Examples
  - MessageDialogDriver.java
  - OptionDialogDriver.java
  - InputDialogDriver.java
  - FileDialogDriver.java
  - ColorDialogDriver.java

GUI Layout

GUI Layout

- Organizing GUI components
- Grouping GUI components
- Positioning GUI components

Absolute Layout

- The "Graph Paper" Approach

- An example – NullLayoutFrame.java

Relative Layout

- Defined:
  - Components are positioned relative to each other, rather than an absolute position
- Pros and Cons:
  - Better than absolute layout across platforms
  - Can be fairly inflexible
- Example – FlowLayoutFrame.java

Template Layout

- Defined:
  - Divide the container into sections and place one component in each section
- Pros and Cons:
  - Easy to use
  - The right template may not exist
- Example – GridLayoutFrame.java

Template Layout in Java (cont.)

- Using BoxLayout
  - One row or column only
- Fillers:
  - Struts (one-dimensional, fixed-size)
  - Rigid Areas (two-dimensional, fixed-size)
  - Glue (one-dimensional, variable-size)
- Example – BoxLayoutFrame.java

Template Layout in Java (cont.)

- Using BorderLayout

- Example – BorderLayoutFrame.java

Event-Driven Programming in GUIs

Overview

- Traditional Programs:
  - Start at the first line and proceed step-by-step
- Event-Driven Programs:
  - Some objects generate events and other objects respond to them

The Focus of Event-Driven Design

- The events that can occur (e.g., mouse clicks, timing signals, key presses)
- The classes that can generate events of different kinds (often called *event generators*)
- The classes that need to respond to events of different kinds (often called *event receivers*)

The Event Queue

- Responsibility:
  - A central **repository** of events that ensures that everything happens in the right order
- Other Participants:
  - Event generators add events to the back of the queue (a process known as *posting*)
  - Event receivers are sent events as they are removed from the front of the queue (a process that is known as *firing* or *dispatching*)

Using a **JButton**

- Implement ActionListener
- Handle the event in the actionPerformed() method
- Example – ButtonFrame.java

Using a **JCheckBox**

- One Way:
  - It extends **JToggleButton**
  - So, it behaves like a **Jbutton**
- Another Way:
  - Implement **ItemListener**
- Example – CheckBoxFrame.java

Using a **JRadioButton**

- Use the same model as buttons
- Are grouped using **ButtonGroup**
- Example – RadioButtonFrame.java

Event-Driven Programming in GUIs (cont.)

Using a Text Component

- Kinds:
  - **JTextField**
  - **JTextArea**
- ActionEvents:
  - ActionEvent is generated when [Enter] is pressed
- DocumentEvents:
  - Most applications only need to respond to changes in the contents of the model
  - The model is a Document
  - So, the important interface is **DocumentListener**
- Example – TextFieldFrame.java

Using a JList

- Some Issues:
  - Doesn't support scrolling directly (more on this later)
  - Interval selection is supported by the DefaultListSelectionModel
- Handling Events:
  - Implement ListSelectionListener
- Example – ListFrame.java

Using a JComboBox

- A combination of a JTextField and a Jlist
- So, implement either ItemListener or ActionListener (or both)
- Example – ComboBoxFrame.java

Responding to **JSlider** Objects

- The Event(s):
  - ChangeEvent
- The Interface(s):
  - ChangeListener
- The Method(s)/Message(s):
  - void stateChanged(ChangeEvent event)
- Example – SliderFrame.java

Advanced GUI Layout

Hierarchical Template Layout

- Overview:
  - Subdivide the sections in a template layout
- Pros and Cons:
  - Platform independent and flexible
  - Difficult to get used to
- Using the Composite Pattern
- Example: HierarchicalLayoutFrame.java

Scrollable Components

- The Idea:
  - Use a "viewport" to make only a portion of their total area visible at a time
  - Enable the visible portion to change (i.e., appear to scroll)
- In Java:
  - The Scrollable interface
  - A JScrollPane controls a Scrollable
- Example: ScrollFrame.java

Docking Toolbars

- The Concept:
  - A container that can be repositioned
- In Java:
  - JToolBar
- Some Views

- Example: DockFrame.java

Borders

- Motivation:
  - Visually group components (e.g., a group of RadioButton objects)
- In Java:
  - A Border can be applied to any Jcomponent
  - Because a large number are used they should be created using a BorderFactory
- Example: BorderExample.java

Layered Layout

- Use the same "screen real estate" for multiple purposes
- Put the different purposes on top of each other and provide a mechanism for moving between them
- In Java use the JTabbedPane

Constrained Template Layout

- The Idea:
  - Add constraints (e.g., doesn't expand) to sections of a template
- In Java:
  - GridBagLayout

Serialization

Serialization

- Defined:
  - The process of creating a "frozen" representation of a "live" (i.e., in-memory) object
- Uses:
  - Copy objects
  - Provide *persistence*
  - Transport objects over a communications channel

Overview of Serialization in Java

- The Serializable Interface:
  - Implemented by classes to indicate that their Objects can be serialized
  - Has no methods (serves only to identify the semantics of being serializable)
- Serialization:
  - Handled by the **writeObject()** method in the **ObjectOutputStream** class
- Deserialization:
  - Handled by the **readObject()** method in the **ObjectInputStream** class

The Serialization Process

- An ObjectOutputStream maintains a **mapping** from instances and classes to handles.
- When writeObject( ) is passed an instance that has not yet been written to the stream, the instance is assigned a **reference handle**, the handle is written to the stream, and the instance data is written to the stream.
- When writeObject() is called with an argument that has already been written to the stream, the handle is written to the stream, and no further operations are necessary.
- When an instance is being serialized and it contains another serializable instance, the serialization of the first instance is **suspended** and the second instance is serialized. After the second instance is fully serialized, the serialization of the first instance resumes.

Examples

```
Date                    d;
FileOutputStream        out;
ObjectOutputStream      serializer;
String                  s;


out = new FileOutputStream("objects.dat");
serializer = new ObjectOutputStream(out);

s = new String("Today");
d = new Date();

serializer.writeObject(s);
serializer.writeObject(d);

serializer.flush();
out.close();
```

```
Date                    d;
FileInputStream         in;
ObjectInputStream       deserializer;
String                  s;


in = new FileInputStream("objects.dat");
deserializer = new ObjectInputStream(in);

s = (String)deserializer.readObject();
d = (Date)deserializer.readObject();

in.close();
```