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.

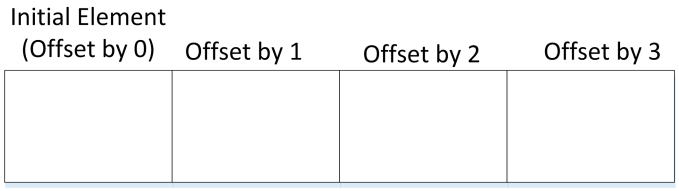
Design of Data Structures

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.



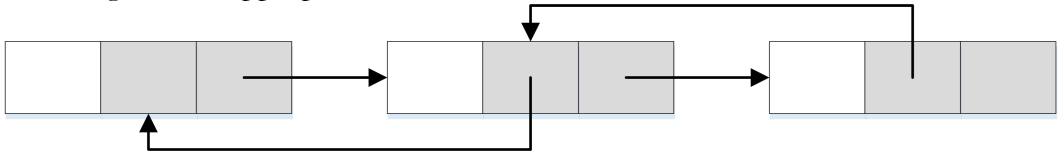
When using a contiguous data structure, the initial value (often called element 0) is stored at a location whose address is recorded; every other location can be calculated using its offset from the initial value and the address of the initial element.

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.