Version control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

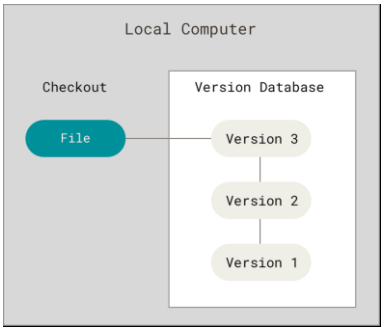
Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, revisions can be merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may concurrently make changes to the same files.

Version control systems

Version control systems (VCS) are most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, collaborative web docs and in various content management systems, e.g., Wikipedia's page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming in wikis.

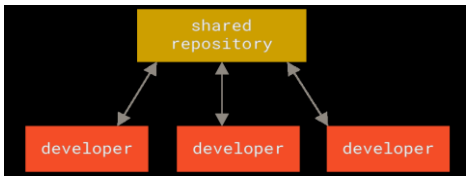
1. Local Version Control Systems:



Local VCSs have a simple database that keep all the changes to files under revision control. One of the most popular VCS tools was a system called **Revision Control** System (RCS), which is still distributed with many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

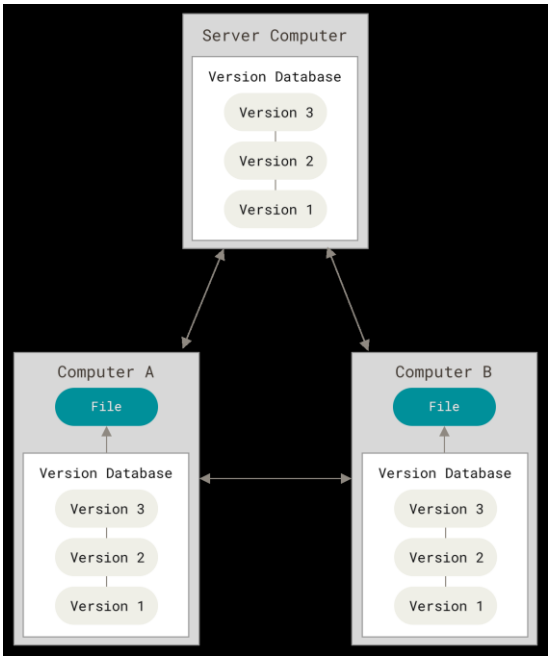
2. Centralized Version Control Systems:

Centralized Version Control Systems (CVCSs, such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



3. Distributed Version Control Systems:

In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.



Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

Basic concepts of software testing (test levels, test types, test design techniques).

1. Test Levels:
   a) Component (Unit) Test
      i. Objective: Find Bugs, build confidence and reduce risk in the individual pieces of the system under test prior to system integration.
      ii. Item Under Test (IUT): functions, classes or larger components

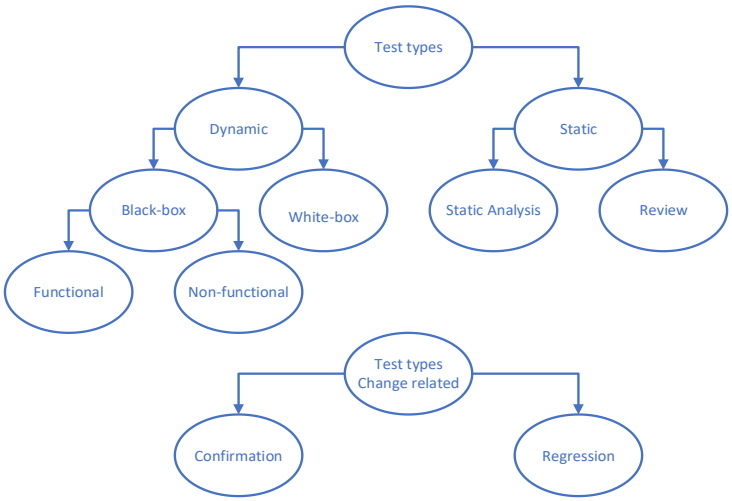
iii. Basis: Code, technical requirements
iv. Environment, tools: Development environment, drivers and stubs
v. Responsible: Usually programmers, but level of proficiency and degree of execution varies.

b) Integration Test
i. Objective: find bugs, build confidence, and reduce risk in the relationships and interfaces between pairs and groups of components in the system under test as the pieces come together.
ii. Item Under Test: Two or more integrated components
iii. Basis: architecture, technical requirements, dataflows
iv. Environment, tools: Development environment, Test Environment, drivers and stubs
v. Responsible: Ideally both testers and programmers

c) System Test
i. Objective: find bugs, build confidence and reduce risk in the overall particular behaviors, functions and responses of the system under test as a whole
ii. Item Under Test: Whole system in an as-realistic-as-possible test environment, including user and operator manuals and configuration data
iii. Basis: Business requirements, use cases, business/domain knowledge, common sense
iv. Environment, Tools: Test Environment, GUI
v. Responsible: typically, independent tester

d) Acceptance Test
i. Objective: Demonstrate that the product is ready for deployment/release
ii. Item Under Test: Fully integrated system, including business, operational and maintenance processes, user procedures and configuration data
iii. Basis: Business, use cases, business/domain knowledge, common sense
iv. Environment and tools: GUI
v. Responsible: Often users and customers but also independent tester.

2. Test types



1. Static testing is the testing of the software work products manually, or with a set of tools, but they are not executed.

2. Static Analysis: Using a tool to evaluate the Item Under Test
3. Review: Manual evaluation of the Item Under Test
4. White-box Test or Structured-based testing: In white-box testing the tester is concentrating on how the software is put together. The internal structure of the software is known by the tester.
5. Black Box Testing or Specification-based testing: In black-box testing the tester is concentrating on what the software does, not how it does it. The internal structure of the software is not known by the tester
6. Grey box testing: We make assumptions about the internal structure of the software. Example: Boundary Value Analysis (BVA)
7. Functional testing: Functional testing is a type of testing which verifies that each function of the software application operates in conformance with the requirement specification. What the system does.
8. Non-functional testing: Non-functional testing is a type of testing to check non-functional aspects of a software application. It is explicitly designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing. How well the system does it.
9. Non-functional Testing - Performance Testing:
    a) Load testing - checks the application's ability to perform under anticipated user loads. The objective is to identify performance bottlenecks before the software application goes live.
    b) Stress testing - involves testing an application under extreme workloads to see how it handles high traffic or data processing. The objective is to identify the breaking point of an application.
    c) Endurance testing - is done to make sure the software can handle the expected load over a long period of time.
10. Non-functional Testing - Maintainability Testing:
    a) It basically checks how easy it is to maintain the system. How easy it is to analyze, change and test the application or product.
    b) Update/patch install and uninstall processes don’t work
    c) Configurations can’t be changed appropriately
    d) Code not maintainable
    e) Software is not efficiently testable
11. Usability Testing:
    a) In usability testing basically the testers tests the ease with which the user interfaces can be used. It tests that whether the application or the product built is user-friendly or not
    b) Learnability: How easy is it for users to accomplish basic tasks the first time they encounter the design?
    c) Efficiency: How fast can experienced users accomplish tasks?
    d) Memorability: When users return to the design after a period of not using it, does the user remember enough to use it effectively the next time, or does the user have to start over again learning everything?
    e) Errors: How many errors do users make, how severe are these errors and how easily can they recover from the errors?
    f) Satisfaction: How much does the user like using the system?

3. Test design techniques

   Test design is a significant step in the Software Development Life Cycle (SDLC), also known as creating test suites or testing a program. In other words, its primary purpose is to create a set of inputs that can provide a set of expected outputs, to address these concerns:

   a) What to test and what not to test
   b) How to stimulate the system and with what data values
   c) How the system should react and respond to the stimuli

Principles of object-oriented programming (GoF, SOLID).

1. GoF (Gang of Four Design Patterns):
   a) Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
   b) The GoF Design Patterns are broken into three categories:
      i. Creational Patterns for the creation of objects;
      ii. Structural Patterns to provide relationship between objects;
      iii. Behavioral Patterns to help define how objects interact.
   c) “Program to an interface, not an implementation.”
   d) “Favor object composition over class inheritance.”
      i. The two most common techniques for reusing functionality in object-oriented systems:
         1. Class inheritance (white-box reuse)
         2. Object composition (black-box reuse)
      ii. The term “white-box”/”black box” refers to visibility.
   e) Advantages of inheritance:
      i. Class inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language.
      ii. Class inheritance also makes it easier to modify the implementation being reused. When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.
   f) Disadvantages of class inheritance:
      i. First, you can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time.
      ii. Second, and generally worse, parent classes often define at least part of their subclasses' physical representation.
         1. Because inheritance exposes a subclass to details of its parent's implementation, it's often said that inheritance breaks encapsulation.
         2. The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.
      iii. Implementation dependencies can cause problems when you're trying to reuse a subclass.
         1. Should any aspect of the inherited implementation not be appropriate for new problem domains, the parent class must be rewritten or replaced by something

more appropriate. This dependency limits flexibility and ultimately reusability.

g) Object composition:
   i. Object composition is defined dynamically at run-time through objects acquiring references to other objects.
   ii. Composition requires objects to respect each other's interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others.

h) Advantages of object composition:
   i. Because objects are accessed solely through their interfaces, we don't break encapsulation.
   ii. Any object can be replaced at run-time by another as long as it has the same type.
   iii. Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task.
   iv. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters.

i) Disadvantages of object composition:
   i. A design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

2. SOLID

The SOLID concepts are:

1. The Single-responsibility principle: "There should never be more than one reason for a class to change."[5] In other words, every class should have only one responsibility.[6]
2. The Open–closed principle: "Software entities ... should be open for extension, but closed for modification."[7]
3. The Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it".[8] See also design by contract.[8]
4. The Interface segregation principle: "Many client-specific interfaces are better than one general-purpose interface."[9][4]
5. The Dependency inversion principle: "Depend upon abstractions, [not] concretions."[10][4]

Dependency injection.

1. The term dependency injection (DI) was coined by Martin Fowler.
2. It can be considered as a special case of the application of the architectural pattern called inversion of control (IoC).
3. **Definition:** Dependency Injection is a set of software design principles and patterns that enables you to develop loosely coupled code.
4. Loose coupling makes code extensible, and extensibility makes it maintainable.
5. An object can be seen as a service that are consumed by other objects as clients.
6. Such a client-service relationship between objects is called a dependence. This relationship is transitive.
7. **Dependency:** a specific service that is required by another object to fulfill its function.

8. **Dependent:** a client object that needs a dependency (or dependencies) in order to perform its function.
9. **Object graph:** a set of dependent objects and their dependencies.
10. **Injection:** giving a client its dependency (or dependencies).
11. **DI container:** a software library that provides DI functionality.
    a) DI containers are also known as Inversion of Control (IoC) containers.
12. DI can be applied without using a DI container.
13. Pure DI: the practice of applying DI without a DI container.
14. DI as a subject is primarily concerned with reliably and efficiently building object graphs and the strategies, patterns, and best practices therein.
15. DI frameworks allow clients to delegate the responsibility of creating and injecting their dependencies to external code.
16. Advantages of DI:
    a) Extensibility,
    b) Maintainability,
    c) Testability: DI support unit testing. Instead of real dependencies, test doubles can be injected into the system under test.

Architectural patterns (MVC).

- Name: Model-View-Control
- Context: Interactive applications with a flexible human-computer interface.
- Problem: User interfaces are especially prone to change requests.
  - Forces:
    - The same information is presented differently in different windows, for example, in a bar or pie chart.
    - The display and behavior of the application must reflect data manipulations immediately.
    - Changes to the user interface should be easy, and even possible at run-time.
    - Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.
- Solution: Dividing an interactive application into the following components:
  - The model component encapsulates core data and functionality, and it is independent of specific output representations or input behavior.
  - View components display information to the user.
  - Controller components receive input, usually as events, that are translated to service requests for the model or the view.
- The separation of the model from view and controller components allows multiple views of the same model. The same data can be displayed in various ways.
- Separating the view from the controller is less important. It allows multiple controllers for the same view. The classic example is to support editable and non-editable behavior for the same view with two controllers. In practice, in most cases there is only one controller per view.
- The model component contains the functional core of the application.
  - It encapsulates the appropriate data, and exports procedures that perform application-

specific processing. Controllers call these procedures on behalf of the user.

- The model also provides functions to access its data that are used by view components to acquire the data to be displayed.
- It registers dependent components (i.e., views and controllers) that are notified by the model about data changes.

Design patterns.

- **Design patterns** are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.
- You can't just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.
- Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.
- Design patterns are medium-scale patterns.
  - They are smaller in scale than architectural patterns, but are at a higher level than the programming language-specific idioms.
- The application of a design pattern has no effect on the fundamental structure of a software system, but may have a strong influence on the architecture of a subsystem.
- Most design patterns are independent of a particular programming language or a programming paradigm.

Free and non-free software.

Free software (or **libre software**)[1][2] is computer software distributed under terms that allow users to run the software for any purpose as well as to study, change, and distribute it and any adapted versions.[3][4][5][6] Free software is a matter of liberty, not price; all users are legally free to do what they want with their copies of a free software (including profiting from them) regardless of how much is paid to obtain the program.[7][2] Computer programs are deemed "free" if they give end-users (not just the developer) ultimate control over the software and, subsequently, over their devices.[5][8]

The right to study and modify a computer program entails that source code—the preferred format for making changes—be made available to users of that program. While this is often called "access to source code" or "public availability", the Free Software Foundation recommends against thinking in those terms,[9] because it might give the impression that users have an obligation (as opposed to a right) to give non-users a copy of the program.

Although the term "free software" had already been used loosely in the past,[10] Richard Stallman is credited with tying it to the sense under discussion and starting the free-software movement in 1983, when he launched the GNU Project: a collaborative effort to create a freedom-respecting operating system, and to revive the spirit of cooperation once prevalent among hackers during the early days of computing.[11][12]

Proprietary software, also known as non-free software or closed-source software, is computer software for which the software's publisher or another person reserves some rights from licensees to use, modify, share modifications, or share the software.[1] It sometimes includes patent rights.[2][1]

Software licenses

--First Definition
A software license is a contract between the entity that created and supplied an application, underlying source code, or related product and its end user. The license is a text document designed to protect the intellectual property of the software developer and to limit any claims against them that may arise from its use.

A software license also provides legally binding definitions for the distribution and use of the software. End-user rights, such as installation, warranties, and liabilities, are also often spelled out in the software license, including protection of the developer's intellectual property.

--Second Definition
A **software license** is a legal instrument (usually by way of **contract law**, with or without printed material) governing the use or redistribution of software. Under United States copyright law, all software is copyright protected, in both source code and object code forms, unless that software was developed by the United States Government, in which case it cannot be copyrighted.[1] Authors of copyrighted software can donate their software to the public domain, in which case it is also not covered by copyright and, as a result, cannot be licensed.

A typical software license grants the licensee, typically an end-user, permission to use one or more copies of software in ways where such a use would otherwise potentially constitute copyright infringement of the software owner's exclusive rights under copyright.

Free and open-source licenses.

Free and open-source software (**FOSS**) is software that can be classified as both free software and open-source software.[a] That is, anyone is freely licensed to use, copy, study, and change the software in any way, and the source code is openly shared so that people are encouraged to voluntarily improve the design of the software.[3] This is in contrast to proprietary software, where the software is under restrictive copyright licensing and the source code is usually hidden from the users.

FOSS maintains the software user's civil liberty rights (see the Four Essential Freedoms, below). Other benefits of using FOSS can include decreased software costs, increased security and stability (especially in regard to malware), protecting privacy, education, and giving users more control over their own hardware. Free and open-source operating systems such as Linux and descendants of BSD are widely utilized today, powering millions of servers, desktops, smartphones (e.g., Android), and other devices.[4][5] Free-software licenses and open-source licenses are used by many software packages. The free-software movement and the open-source software movement are online social movements behind widespread production and adoption of FOSS, with the former preferring **FLOSS** term or simply free or free/libre term used.