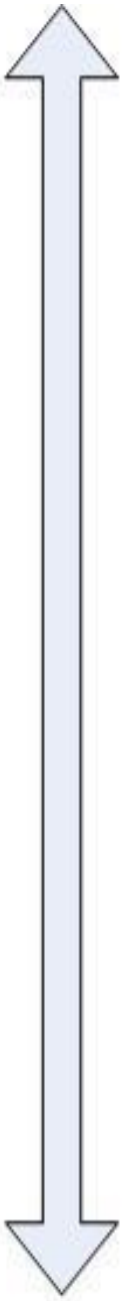
Unit – II

Agile Software Design & Development

Agile Design Practices

There is a range of agile design practices, Figure in next slide shows, high-level architectural practices to low-level programming practices. Each of these practices are important, and each are needed if your team is to be effective at agile design.



Architecture envisioning – Light-weight modeling at the beginning of a project to identify and think through critical architecture-level issues.

Iteration modeling – Light-weight modeling for a few minutes at the beginning of an iteration/sprint to help identify your team's strategy for that iteration. Part of your iteration/sprint planning effort.

Model storming – Light-weight modeling for a few minutes on a just-in-time (JIT) basis to think through an aspect of your solution.

Test-first design (TFD) – Write a single test before writing enough production code to fulfill that test.

Refactoring – Make small changes to a part of your solution which improves the quality without changing the semantics of that part.

} Test-driven design = TFD + Refactoring

Continuous integration – Automatically compile, test, and validate the components of your solution whenever one of those components changes.

Various Design Principles

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

S

O

L

I

D

Single Responsibility Principle (SRP)

In object-oriented programming, the single responsibility principle states that –

“Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. “

All its services should be narrowly aligned with that responsibility.

- Also some of the researchers described it as being based on the **principle of cohesion.**

- Robert C. Martin defines a ***responsibility*** as **a reason to change, and concludes that a class or module should have one, and only one, reason to change.**

  The single responsibility principle says that two aspects (component and format) of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust.

The responsibility is defined as a charge assigned to a unique actor to signify its accountabilities concerning a unique business task.

Example : Document – *1. Content*

2. Formatting

Both of them are different responsibility, so they should be well comprised in separate classes

Open Closed Principle

In object-oriented programming, the **open/closed principle** states -

"***software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification***"

That is, such an entity can allow its behavior to be extended without modifying its source code.

- This is especially valuable in a production environment, where changes to source code may necessitate code reviews, unit tests, and other such procedures to qualify it for use in a product: code obeying the principle doesn't change when it is extended, and therefore needs no such effort.

- The name *open/closed principle* has been used in two ways. Both ways use inheritance to resolve the apparent dilemma, but the goals, techniques, and results are different.

Meyer's open/closed principle – Says "once completed, the implementation of a class could only be modified to correct errors; new or changed features would require that a different class be created. That class could reuse coding from the original class through inheritance. The derived subclass might or might not have the same interface as the original class."

Meyer's definition advocates implementation inheritance. Implementation can be reused through inheritance but interface specifications need not be. The existing implementation is closed to modifications, and new implementations need not implement the existing interface.

Polymorphic open/closed principle advocates inheritance from abstract base classes. Interface specifications can be reused through inheritance but implementation need not be.

The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface.

Liskov Substitution Principle

Substitutability is a principle in object-oriented programming.

It states that, in a computer program –

"if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may *substitute* objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)."

- More formally, the **Liskov substitution principle** (**LSP) is a particular definition of a sub typing relationship,** called **(strong) behavioral sub typing,** that was initially introduced by Barbara Liskov in a 1987**.**

- **It is a semantic rather than merely syntactic relation** because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular**.**

Barbara Liskov and Jeannette Wing formulated the principle succinctly in a 1994 paper as follows:

"Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S, where S is a subtype of T."

<u>Note Points –</u>

1. **Getter & Setter Methods** *(What, Why, Where and How ??????)*
2. **Behavioral Sub typing** *(What, Why, Where and How ?????)*

Interface Segregation Principle

The **interface-segregation principle (ISP) states that no** client should be forced to depend on methods it does not use.

ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.

Such shrunken interfaces are also called ***role interfaces***

ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of Object-Oriented Design, similar to the High Cohesion Principle of GRASP ????.

Within object-oriented design, interfaces provide layers of abstraction that facilitate conceptual explanation of the code and create a barrier preventing dependencies.

Using interfaces to further describe the intent of the software is often a good idea.

A system may become so coupled at multiple levels that it is no longer possible to make a change in one place without necessitating many additional changes. Using an interface or an abstract class can prevent this side effect.

Dependency Inversion Principle

In object-oriented programming, the **dependency inversion principle refers to a specific form of decoupling** software modules.

When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (i.e. reversed), thus rendering high-level modules independent of the low-level module implementation details.

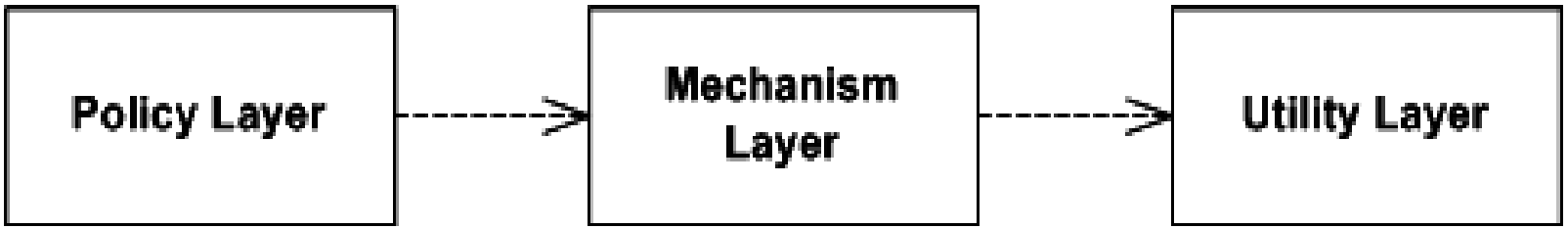
The principle states:

- ***High-level modules should not depend on low-level modules. Both should depend on abstractions.***
- ***Abstractions should not depend on details. Details should depend on abstractions.***

The principle *inverts the way some people may think* about object-oriented design, dictating that *both high- and* low-level objects must depend on the same abstraction.

In conventional application architecture, **lower-level components are designed to be consumed by higher-level components** which enable increasingly complex systems to be built.

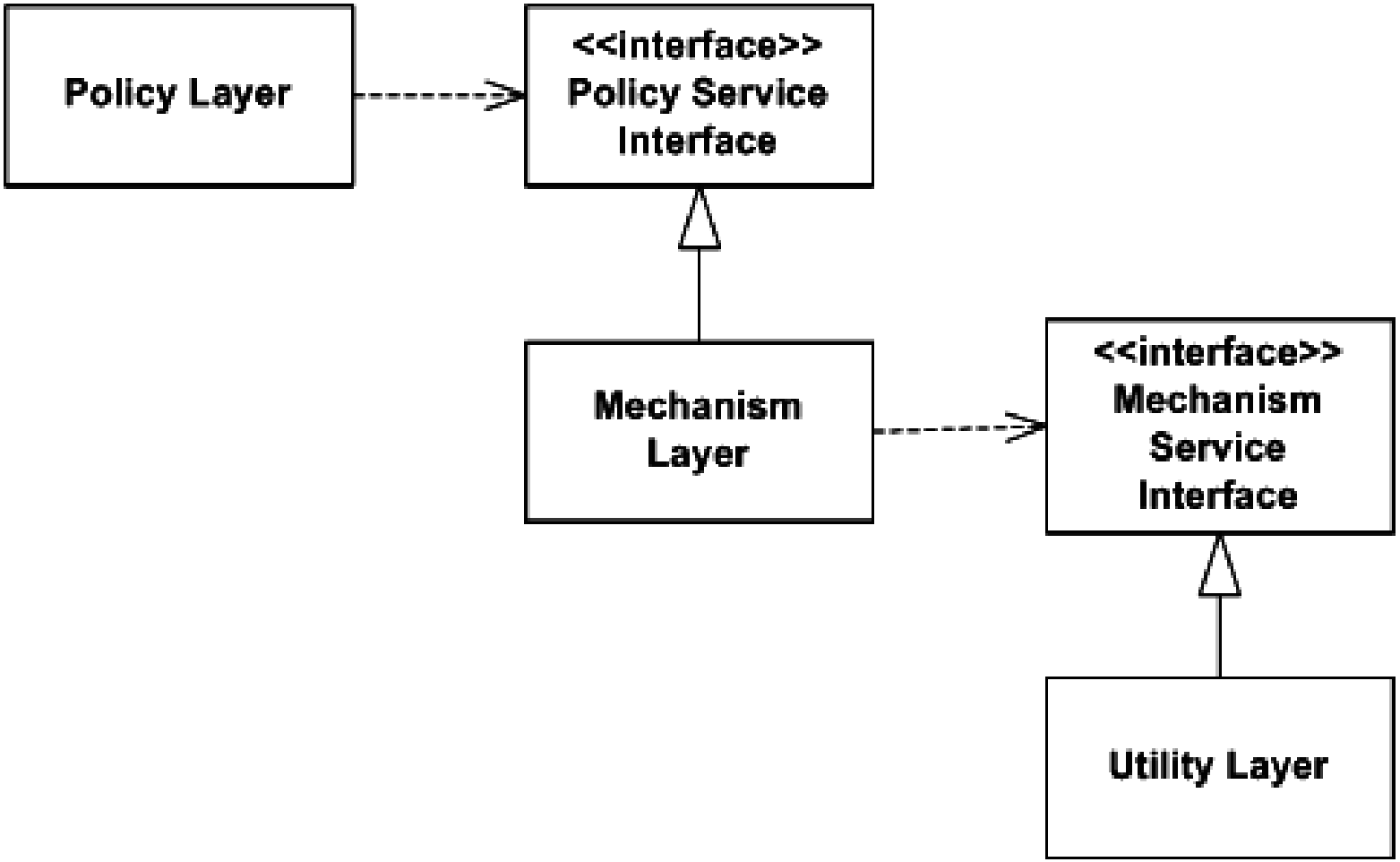
In this composition, **higher-level components depend directly upon lower-level components to achieve some task.** This dependency upon lower-level components limits the reuse opportunities of the higher-level components.



Traditional Dependency Layer Pattern

“The goal of the dependency inversion principle is **to avoid this highly coupled distribution with the mediation of an abstract layer**, and **to increase the re-usability of higher/policy layers.**”

Ownership Inversion



With the addition of an abstract layer, both high- and lower-level layers avoid the traditional dependencies from top to bottom.

Nevertheless the ″inversion″ concept does not mean that lower-level layers depend on higher-level layers.

Both layers should depend on abstractions that draw the behavior needed by higher-level layers.

In a direct application of dependency inversion, the abstracts are owned by the upper/policy layers.

This architecture groups the higher/policy components and the abstracts that define lower services together in the

same package.

The lower-level layers are created by inheritance/implementation of these abstracts classes or interfaces.

How to achieve it ???????

Dependency Injection

DI is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it.

The intent behind dependency injection is to decouple objects to the extent that no client code has to be changed simply because an object it depends on needs to be changed to a different one.

Dependency injection is one form of the broader technique of inversion of control. dependency injection supports the dependency inversion principle.

There are three common means for a client to accept a dependency injection:

Setter based injection

Interface based injection and

Constructor based injection.

Setter and constructor injection differ mainly by when they can be used.

Interface injection differs in that the dependency is given a chance to control its own injection.

All require that separate construction code (the injector) take responsibility for introducing a client and its dependencies to each other.

Dependency injection involves four roles:

- **Service -** *object(s) to be used*
- **Client -** *object that is depending on the services it uses*
- **Interfaces -** *that define how the client may use the services*
- **Injector -** *which is responsible for constructing the services and injecting them into the client*

Find out merits and demerits of using dependency injections.

Continuous Integration

- Continuous integration (CI) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day.

- It was first named and proposed by Grady Booch in his method, who did not advocate integrating several times a day.

- It was adopted as part of extreme programming (XP).

Code Refactoring

- Code refactoring is the process of restructuring existing computer code without changing its external behavior.
- While refactoring new code is not developed rather improvement of the existing code is focussed.
- Refactoring improves non functional attributes of the software.
- Advantages include improved code readability and reduced complexity to improve source code maintainability, and create a more expressive internal architecture or object model to improve extensibility.

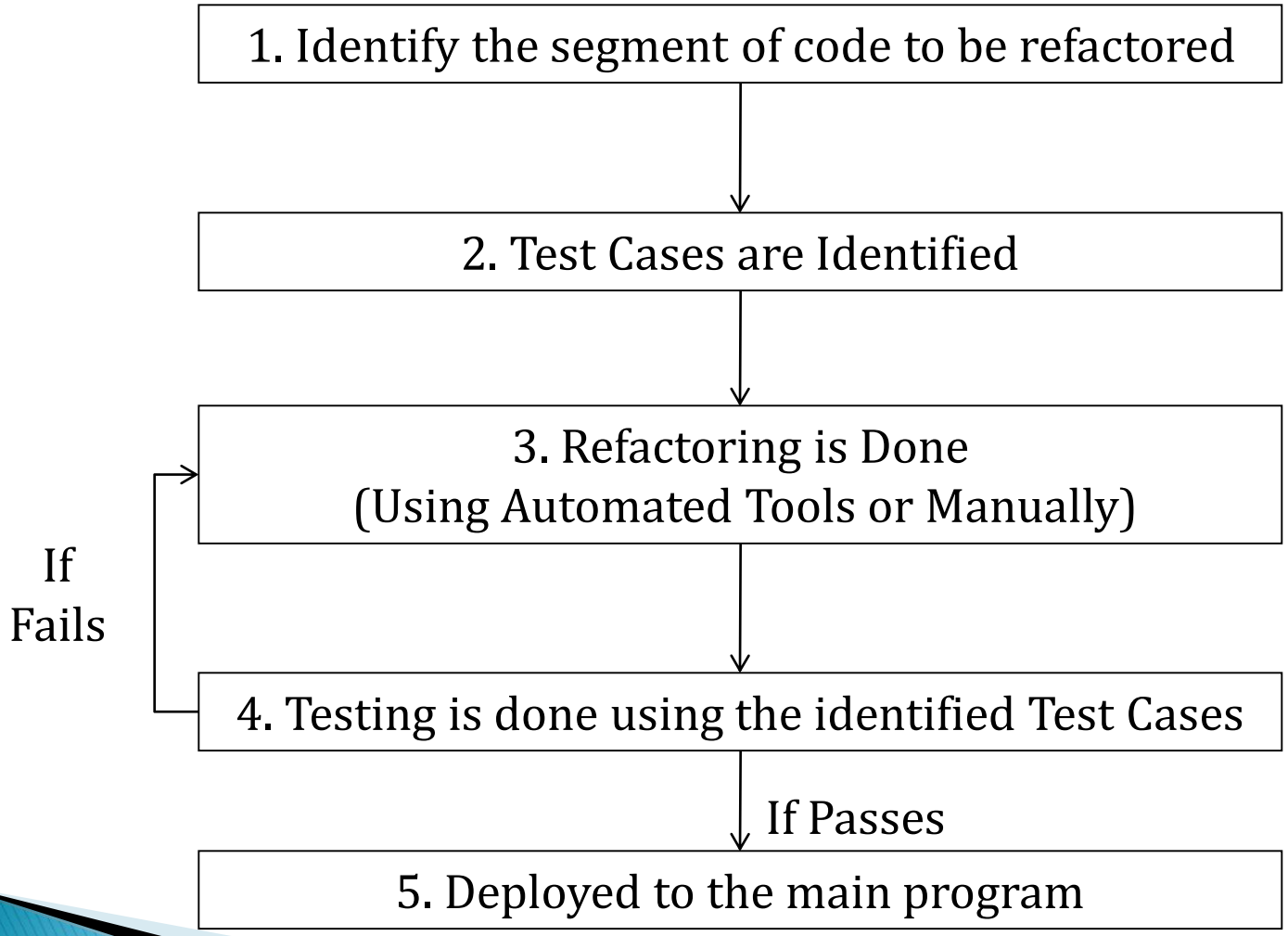
Typically, refactoring applies a series of standardized basic micro-refactorings, each of which is (usually) a tiny change in a computer program's source code that either preserves the behavior of the software, or at least does not modify its conformance to functional requirements.

Why to Refactor a code ?

There may be some stereo- typical situations where program code should be improved. Such situations are known as Code Smells. Some of them are –

1. Duplicate Code
2. Long Methods
3. Switch Case Statements
4. Data Clumping
5. Speculative Generality

Process of Code Refactoring



Benefits of Code Refactoring

1. **Maintainability -** It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp.

2. **Extensibility -** It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed

Code Refactoring Techniques

Techniques that allow for more abstraction

- ***Encapsulate Field*** – *force code to access the* field with getter and setter methods
- ***Generalize Type*** – *create more general types* to allow for more code sharing
- ***Replace type-checking code*** *with* State/Strategy
- ***Replace conditional with polymorphism***

Techniques for breaking code apart into more logical pieces

- ***Componentization breaks code down*** *into* reusable semantic units that present clear, well-defined, simple-to-use interfaces.
- ***Extract Class moves part of the code*** *from an* existing class into a new class.
- ***Extract Method, to turn part of a larger method*** into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.

Techniques for improving names and location of code

- ***Move Method or Move Field*** – *move to a more* appropriate Class or source file

- ***Rename Method or Rename Field*** – *changing* the name into a new one that better reveals its purpose

- ***Pull Up*** – *in OOP, move to a super class*

- ***Push Down*** – *in OOP, move to a sub class*

Enlist any five Code Refactoring tools.

Continuous Integration

- Continuous integration (CI) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day.

- It was first named and proposed by Grady Booch in his method, who did not advocate integrating several times a day.

- It was adopted as part of extreme programming (XP).

- CI typically use a build server to implement continuous processes of applying quality control in general — small pieces of effort, applied frequently.

- In addition to running the unit and integration tests, such processes facilitate manual QA processes.

- This continuous application of quality control aims to improve the quality of software, and to reduce the time taken to deliver it.

Principles of Continuous Integration

1. **Maintain a code repository –**

   This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository.

2. **Automate the build –**

   A single command should have the capability of building the system. Many build-tools, such as make, Debian DEB, Red Hat RPM or Windows MSI files

3. Make the build self-testing -

Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave.

4. Everyone commits to the baseline every day -

By committing regularly, every committer can reduce the number of conflicting changes.

5. Every commit (to baseline) should be built -

The system should build commits to the current working version to verify that they integrate correctly.

6. Keep the build fast -

The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.

7. Test in a clone of the production environment -

Having a test environment can lead to failures in tested systems when they deploy in the production environment, because the production environment may differ from the test environment in a significant way.

8. **Make it easy to get the latest deliverables -**

Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when rebuilding a feature that doesn't meet requirements.

9. Everyone can see the results of the latest build -

It should be easy to find out whether the build breaks and, if so, who made the relevant change.

10. Automate deployment -

Most CI systems allow the running of scripts after a build finishes. o, who made the relevant change.

Benefits of Continuous Integration

1. Integration bugs are detected early and are easy to track down due to small change sets.

2. Avoids last-minute chaos at release dates.

3. Constant availability of a “current” build for testing, demo, or release purposes

4. Frequent code check-in pushes developers to create modular, less complex code

Version Control

- A component of software configuration management, **version control**, also known as **revision control** or **source 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 and it is the management of changes to documents, computer programs, large web sites, and other collections of information.

- Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision".

- Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

- **Version control systems** (**VCS**) 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.

- 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.

- **Distributed revision control systems (DRCS)** take a peer-to-peer approach, as opposed to the client-server approach of centralized systems.

- Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository.

- Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer.

Special Terminology

- **Baseline :** An approved revision of a document or source file from which subsequent changes can be made.
- **Branch :** A set of files under version control may be *branched* or *forked* at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.
- **Change :** A *change* (or *diff*, or *delta*) represents a specific modification to a document under version control. The granularity of the modification considered a change varies between version control systems.

- **Change List** : On many version control systems with atomic multi-change commits, a *change list* (or *CL*), *change set*, *update*, or *patch* identifies the set of *changes* made in a single commit. This can also represent a sequential view of the source code, allowing the examination of source "as of" any particular changelist ID.

- **Clone :** *Cloning* means creating a repository containing the revisions from another repository. This is equivalent to *push*ing or *pull*ing into an empty (newly initialized) repository. As a noun, two repositories can be said to be *clone*s if they are kept synchronized, and contain the same revisions.

- **Checkout :** To *check out* (or *co*) is to create a local working copy from the repository. A user may specify a specific revision or obtain the latest. The term 'checkout' can also be used as a noun to describe the working copy.

- **Commit :** To *commit* (*check in*, *ci* or *submit* or *record*) is to write or merge the changes made in the working copy back to the repository. The terms 'commit' and 'checkin' can also be used as nouns to describe the new revision that is created as a result of committing.

- **Conflict :** A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must *resolve* the conflict by combining the changes, or by selecting one change in favour of the other.

- **Delta Compression** : Most revision control software uses delta compression, which retains only the differences between successive versions of files. This allows for more efficient storage of many different versions of files.

- **Dynamic Stream** : A stream in which some or all file versions are mirrors of the parent stream's versions.

- **Export :** *exporting* is the act of obtaining the files from the repository. It is similar to *checking out* except that it creates a clean directory tree without the version-control metadata used in a working copy. This is often used prior to publishing the contents.

- **Forward integration :** The process of merging changes made in the main *trunk* into a development (feature or team) branch.

- **Head :** Also sometimes called *tip*, this refers to the most recent commit, either to the trunk or to a branch. The trunk and each branch have their own head, though HEAD is sometimes loosely used to refer to the trunk.

- **Import :** *importing* is the act of copying a local directory tree (that is not currently a working copy) into the repository for the first time.

- **Initialize :** to create a new, empty repository.

- **Interleaved deltas :** some revision control software uses Interleaved deltas, a method that allows to store the history of text based files in a more efficient way than by using Delta compression.

- **Mainline :** Similar to *trunk*, but there can be a mainline for each branch.

- **Merge :** A *merge* or *integration* is an operation in which two sets of changes are applied to a file or set of files.

- **Promote :** The act of copying file content from a less controlled location into a more controlled location. For example, from a user's workspace into a repository, or from a stream to its parent.

- **Pull, push :** Copy revisions from one repository into another. *Pull* is initiated by the receiving repository, while *push* is initiated by the source. *Fetch* is sometimes used as a synonym for *pull*, or to mean a *pull* followed by an *update*.

- **Repository :** The *repository* is where files' current and historical data are stored, often on a server. Sometimes also called a *depot*.
- **Resolve :** The act of user intervention to address a conflict between different changes to the same document.
- **Reverse integration :** The process of merging different team branches into the main trunk of the versioning system.
- **Revision :** Also *version*: A version is any change in form. In SVK, a Revision is the state at a point in time of the entire tree in the repository.
- **Share :** The act of making one file or folder available in multiple branches at the same time. When a shared file is changed in one branch, it is changed in other branches.

- **Tag :** A *tag* or *label* refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number. See baselines, labels and tags.
- **Trunk :**The unique line of development that is not a branch (sometimes also called Baseline, Mainline or Master)
- **Update :**An *update* (or *sync*, but *sync* can also mean a combined *push* and *pull*) merges changes made in the repository (by other people, for example) into the local *working copy*.
- **Working copy :** The *working copy* is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy, hence the name. Conceptually, it is a *sandbox*.