UNIT-3 → ~~Agile Design Principle with UML eg - SOLID — Need & significance of Refactoring, Refactoring Techniques, Continuous Integration, Automated Build tools, Version Control, TDD, xUNIT framework & tools for TDD.~~

SOLID principles → Solid principles are design principles that enable us manage most of s/w design problems.

→ The term SOLID is an acronym of "5" design principles intended to make software designs more understandable, flexible & maintainable.

→ It was first introduced by ~~Michael Feathers~~ Robert C Martin

Q) Why SOLID ?? Ans: We may: ~~end up~~:

→ end up with tight/strong coupling of code

→ Tight coupling causes unknown issues/causes time to implement new feature/any bug fixes...

→ End up with duplication of code...

→ End up with code which is not testable

→ End up creating new bugs/errors

Following SOLID principles helps us to:

→ achieve/in reduction of complexity of code...

→ increase readability, extensibility & maintenance

→ Reduce error & implement reusability

→ Achieve better testability

→ Reduce tight coupling

Solutions to develop a successful application depends on

- Architecture → It is the 1st step in designing app. based on requirements. Eg:- MVC, MVVM etc.
- Design principles → Application development process needs to follow the design principles.
- Design patterns → We need to choose correct design patterns to build the s/w.

1)# Single Responsibility Principle →

→ "A class should have only one reason to change"
→ Every module or class should have responsibility over a single part of functionality provided by s/w & that responsibility should be entirely encapsulated by class.

#Motivation →
- Maintainability
- Testability
- Flexibility & Extendability
- Parallel development
- Loose Coupling → makes app easier to make any change in 1 area of system.

S
→ Each class & module should focus on single task at a time.
→ Everything in class should be related to that single purpose.
→ There can be many members in class as long as they are related to single responsibility.
→ Classes become smaller & cleaner.
→ Code is less fragile.

Code →

```
using System; using System.Collection.Generic;
using System.Ling; using System.Text; using System.
Threading.Tasks;
namespace SRPDemo { interface IUser {
    bool login(string uname, string pass);
    bool Register(string uname, string pass, string
        email); }
interface ILogger {
    void LogError(string error);
```

```
interface IEmail {
    bool SendEmail(string emailContent);
}
```

→ using interface is good idea
→ provides layer of abstraction
→ simplifies code

2) Interface Segregation Principle → "No client should not be forced to depend on methods it does not use". means...

→ one big interface needs to be split to many smaller & relevant interfaces so that clients can know about the interfaces that are relevant to them.

→ Case Study → # Problem

Xerox operation manufatures printer system. In their dev. process of new systems Xerox had created a new printer system that could perform variety of tasks such as stapling & faxing along with regular printing task.

↳ The s/w for this system was created from ground up
→ Modification became more difficult
→ A single Job class was used by all tasks. Whenever a print job or stapling job needed to be performed a call was made to Job class.
→ This resulted in 'fat' class
→ Because of this design, a staple job knows all methods of print job

Solution → One large job class is segregated to multiple interfaces

3) Open Closed Principle → abstraction & polymorphism

"Software entities such as classes, modules, functions should be open for extension but closed for modification." means...

→ any new functionality should be implemented by adding new classes, attributes & methods instead of changing the current ones or existing ones.

Implementation Guidelines of OCP →

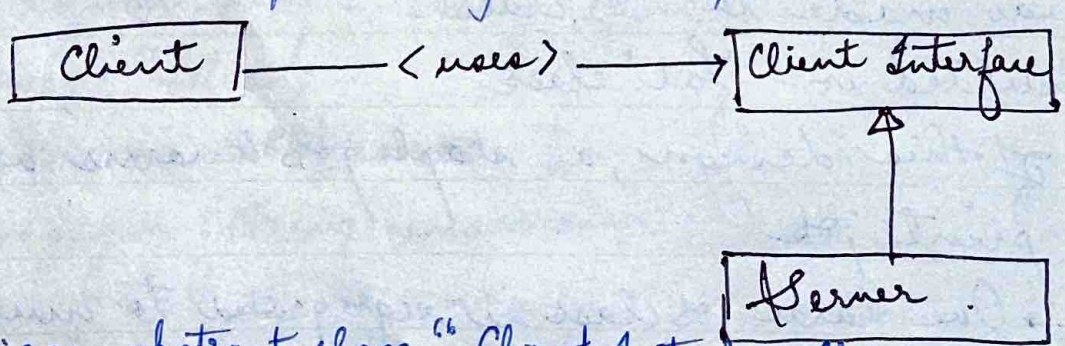
→ implement new functionality on new derived classes.

→ allow clients to access original class with abstract interface.

Why OCP?

If not followed then

- ▷ if class allows addition of new logic always then we end up testing the entire functionality.
- ▷ QA team will have to test once again
- ▷ It is costly.
- ▷ Maintainance overhead increases.
- ▷ It breaks single responsibility since class/func end up doing multiple tasks.



In this eg, abstract class "Client Interface" was added & Client uses this abstraction. However Client class will actually use Server class which implements "Client Interface" class. If in future one wishes to use another type of server all you need to do is implement a new class derived from ClientInterface.

4)

Liskov Substitution Principle → "Subtypes must be substitutable for their base Types"

"Substitutability is a principle in OOP such as C++ & it states that in a computer program if "S" is a subtype of "T", then object of type "T" may be replaced with objects of type "S". Which means ...

→ Derived types must be completely substitutable for their base types.

→ Liskov substitution principle is a definition of subtyping relation, called (strong) behavioral subtyping.

→ It is an extension of Open Closed principle...

Implementation Guidelines →

i) No new exceptions can be thrown by subtype

ii) Client shouldn't know which specific subtype they are calling.

iii) New derived classes are extended without replacing functionality of old classes.

example 1: Green is base class of Blue. When new object of Blue will be assigned in object of Super class, getColor() of Green will be replaced by GetColor() of Blue. In this, we are expecting "green" from Green class but getting Blue from "Blue" Class.

```
public class Green {
    public void GetColor() {
        Sop("Green"); }
}
```

```
public class Blue extends Green {
    public void getColor() {
        S.o.p ("Blue");
    }
}
public class Main {
    p.s.v.m (String [] args) {
    Green green = new Blue();
    green.getColor();
    }  // output : Blue
}
```

Advantage

→ Code reusability

→ Improved code quality

To solve this

```
public interface IColor {
    public void getColor();
}
public class Green implements IColor {
    public void getColor() {
        S.o.p ("Green");
    }
}
public class Blue implements IColor {
    public void getColor() {
        S.o.p ("Blue");
    }
}
public class Main {
    p.s.v.m( String [] args) {
    IColor color = new Blue();
    Color.getColor();
    }
}
```

5) Dependency Inversion Principle →

→ It states that high level modules should not depend on low level modules. Both should depend on abstractions.

→ Abstraction should not depend on details but details should depend on abstractions.

→ The interaction b/w high level & low level module should be thought as an abstract interaction b/w them.

Advantages

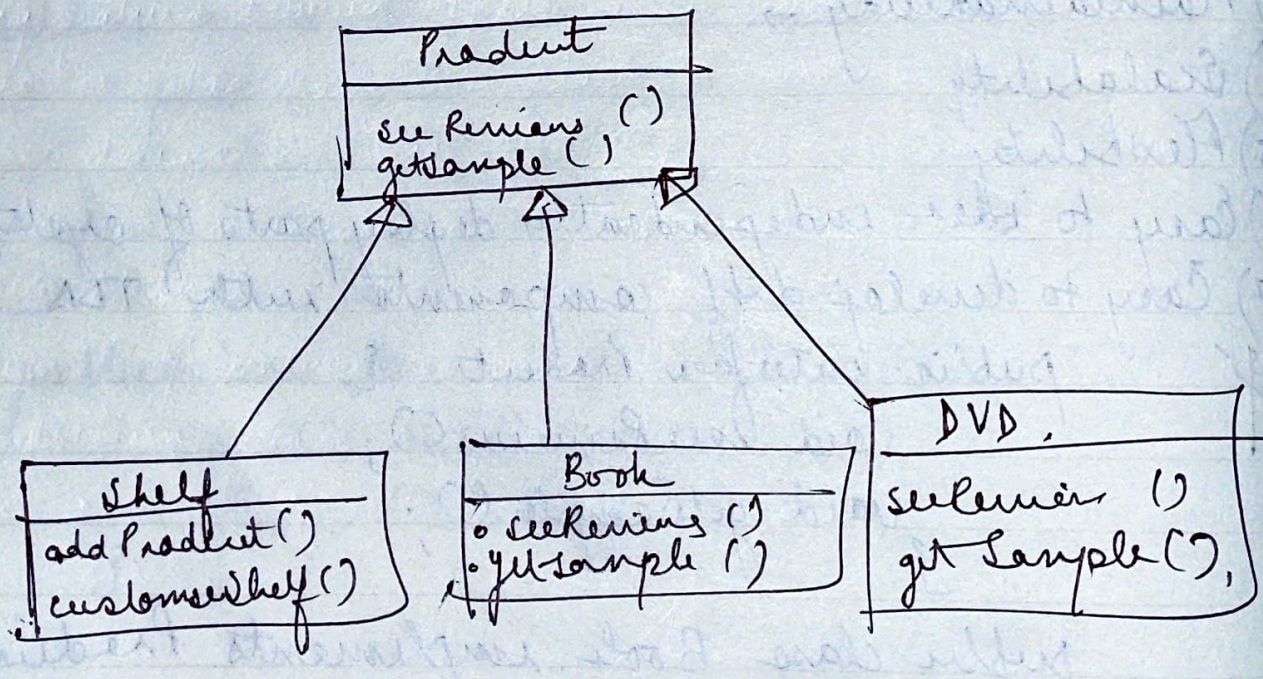
1) Loose Coupling
2) Testability → Abstraction make it easier to write unit tests
3) Maintainability →
4) Scalability
5) Flexibility
6) Easy to ~~der~~ independently deploy parts of system.
7) Easy to develop diff components with TDS

Eg:

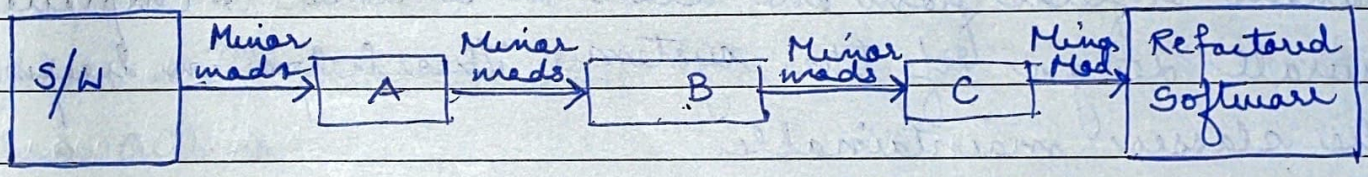
```
public interface Product {
    void bookRemains();
    void getSample();
}

public class Book implements Product {
    public void bookRemain() { }
    public void getSample() { }
}
```

```
public class DVD implement product {
    public void seeReviews () {
    }
    public void getSample () {
    }
}
public class shelf {
    Product product;
    void addProduct (Product product) {
    }
    void customiseshelf () {
    }
}
```



#Refactoring → process of restructing code while not changing its original functionality. The goal of refactoring is to improve internal code by making many small changes without altering code's external behaviour.
→SD refactor code to improve design, structure & implementation of s/w.
→ It improves code readability & reduces complexities
→ helps to find bugs & vulnerabilities.
Purpose → as well as advantages
→ Makes code more efficient
→ Makes code maintainable & reusable
→ Makes code cleaner & easy to understand
→ Easier for SDev. to find bugs & vulnerabilities



Challenges →
1) Takes extra time
2) Can lead to extra work
Refactoring Techniques →
1) Composing Methods → This technique involves breaking down large methods into smaller ones. By doing this we make code more easier to understand, maintain & reuse.

- Extract Method →

Problem → You have a code fragment that can be grouped together.

Solution → Move this code to a seperate new method & replace old code with a call to the method.

- Inline Method →

Problem → When a method body is more obvious than method itself use this technique.

Solution → Replace calls to method with its content. Eg delete method itself.

- Extract Variable →

Problem → You have an expression that's hard to understand.

Solution → Place the result of expression in seperate variables that are self explanatory.

2) Moving features between objects → It involves relocating methods or fields from one class to another to improve overall design of the system. This helps in keeping the classes maintainable.

- Move Method

Problem → If method is used more in another class than in its own class.

Solution → Create a new method in class that uses the method most, then move code from old method to there. Turn the code of original method into a reference to the new method in other class.

3) Organising Data → It focuses on restructing the data structure used in code to make them more understandable. This involves grouping related data together in classes eliminating redundant data & ensuring data is stored acc. to design goals.

eg:- Change Value to Reference
Problem → If you have identical instances of single class then you need to replace it with single object.

Solution → Convert identical objects into single object.

4) Simplifying Conditional Expressions → It involves techniques like extracting conditional logic into separate methods using polymorphism or simplifying complex boolean expressions using logical operators.

eg Consolidate conditional exp:
Problem → You have multiple conditionals that lead to same result

Solution → Consolidate all these conditionals in single expression.

5) Simplifying method calls → It involves techniques like reducing no. of parameters by grouping related data into objects, creating helper methods.

eg: Add Parameters
Problem → If method doesn't have enough data to perform certain actions.

Solution → Create a new parameter to pass data

6) Dealing with Generalizations → It refers to process of abstracting common functionality into a interface or high level abstraction. This technique involves identifying areas where generalization can simplify code & improve flexibility & extensibility.

eg Pull up Field
Problem → Two classes have same Field

Solution → Remove field from subclasses & move it to superclass

~~Continuous Integration~~

Test Driven Development → It is the process in which test cases are written before code that validates those cases.

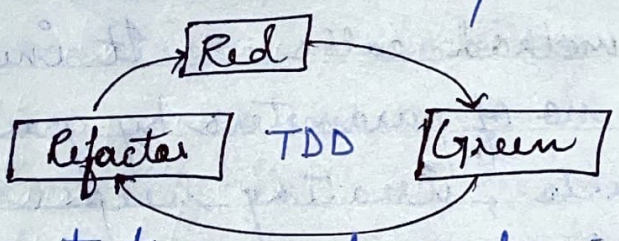
→ It is iterative in nature. Also known 'Test Driven Design'

↳ It is a technique in which automated unit tests are used to drive the design and free decoupling of dependencies. Coding, Testing, Designing → TDD is a blend of these 3 activities

Steps →

1) Write a test cases that describes the fun^c completely.
2) Run the test cases & make sure new test case fails.
3) Write the code that passes the failed test case.
4) Run the test cases.
5) Refactor code → to remove duplication of code.

Motto of TDD



i) Red → Create a test case & make it fail.
ii) Green → Make the test case pass by any means.
iii) Refactor → Change the code to remove duplicacy.

Advantages

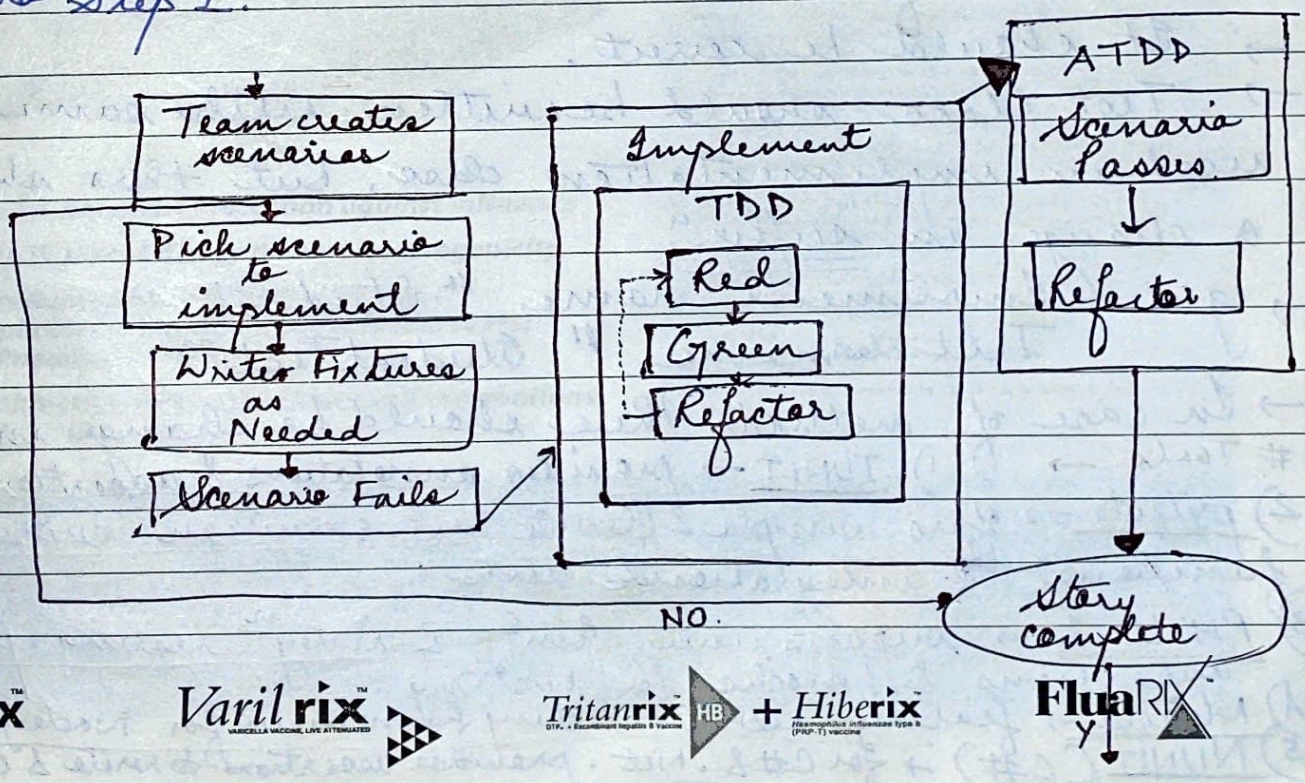
1) You only write code that's needed.
2) More Modular design → Considering on micro feature at a time. So writing the test case, the code automatically becomes easy to check.
3) Easy to maintain → As the app is decoupled & has a clear interface; the code becomes easier to maintain.
4) Easy to refactor →
5) Small development cycle
6) Less debugging
7) High test coverage → There's a test for each feature.

Disadvantages of TDD →

1) No silver bullet → Test helps to find bugs but it can't help you when you have misunderstood a problem.
2) Slow Process → First write test code, then run it & finally start writing it so it takes a lot of Time.
3) Whenever requirements change, tests got to be maintained.
4) If a team is deiding to follow TDD, everyone has to follow it, else it can create problems.

Test Driven Workflow →

1) Write a concise, minimal test case for a new functionality
→ This test will fail since functionality isn't implemented.
2) Implement the new functionality and run all tests, both new & pre existing ones. Repeat until all test passes.
3) Clean up code and ensure all tests still pass, If return to Step 1.



Need for TDD →

1) Ensures Quality → by focusing on requirements before code
2) Keeps code neat → whole process into smaller steps.
3) Maintains documentation → provides how system works
4) Repeatable Test & Rapid Changes.
5) Improves Flexibility & extensibility & maintainability

Road Map of TDD →

1) Clear out thought & break it in test case
Follow red green approach to build test case
2) 1st create red test & after exposing all problems make changes & make it a green test.

Implementation → Essential to implement both test case & source code seperately. There should be two directories. In every programming lang. there should be a diff. package for both.

→ eg:- "src/main/java" → implementation
"src/test/java" → testing

Structure of TDD →

→ It should be correct.
→ Test class should be written with same name used in implementation class, but there should be a change in "suffix."
→ eg:- Implement name "Student"
Test class name "StudentTest"
→ In case of methods there should be change in "prefix"

Tools → 1) JUNIT → provides annotations & assertions.
2) pytest → offers simple & flexible test execution: supports unit, functional & integrational tests.
3) PHPUnit → supports unit func^{n} & integ^{n} testing. provides assertions & mocks for testing.
4) Mocha → feature rich JS testing framework for Node.js & browser
5) NUNIT (C#) → for C# & .Net. provides assertions to write & execute tests

UNIT-3
(contd)

Versions

Continuous Integration → It is a s/w dev. process where developers integrate new code they've written more frequently adding/integrating it to the shared repository atleast once a day. Automated testing is done against each iteration of build to identify integration issues. Overall, CI helps streamline the process, resulting in higher quality s/w.

Practices of CI

1) Code reviews & collaboration → review code before merged into main codebase.
2) Continuous monitoring → for errors & bugs & a log is ~~log is~~ logs are automatically collected & analyzed.
3) Security scan → performed on code to identify vulnerabilities
4) Ability to rollback → to a previous version in case of failure.
5) Automated testing & of code changes → Every code is automatically built & tested to ensure it doesn't break existing functionality.

→ CI is practice of automating the integration of code changes from multiple contributors to a single s/w project.

↳ Automated tools are used to assert new code's correctness before integrating.

UNIT TESTS → narrow in scope & verify behaviour of individual methods or functions.

Integration Test → makes sure that multiple components behave correctly.

Acceptance Test → Similar to IT as above but they focus on business case rather than components.

~~UI Test →~~

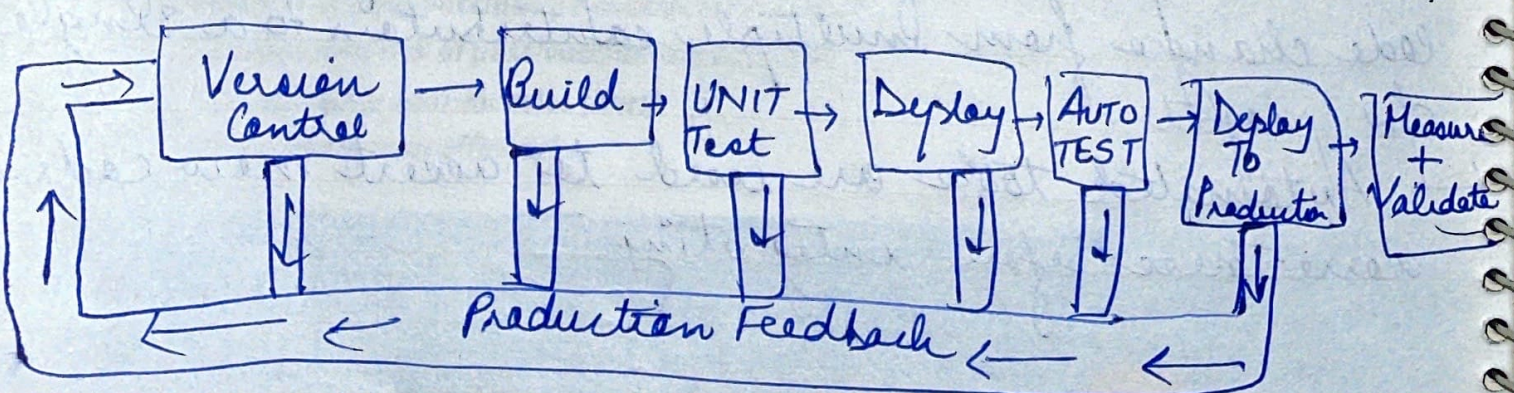
CI in 5 steps →

1) Start writing tests for critical parts of code
2) Get a CI service to run tests automatically on every push to main repo.
3) Make sure your team integrates their changes everyday.
4) Fix the build as soon as it's broken.
5) Write tests for every new story that you implement.

Build Deployment & CI process → To implement CI/CD deployment, adopting Build Tool is 1st step.

→ Build tools provide the features of an extensive library of plugins, build & source code management functionalities, parallel testing.

→ Process is shown below.



Challenges of Build Automation

1) Longer Builds reduce productivity.
2) Large no. of builds resulting in limited build server access.
3) Complex builds may reduce flexibility.

~~Benefits of Automation Build Tools~~ →

Advantages of CI →

1) Streamlined Deployment → enables quick & reliable deployment
2) Early Issue Detection → helps catches bugs early.
3) Quick Feedback → to developer about quality of code change
4) Reduced Integration Risk
5) Improved Collaboration → encourages teamwork

Disadvantages of CI →

→ Dependency Management → can be challenging
→ False Results → Automated Testing may produce false o/p
→ Infrastructure Overhead → Requires servers & testing environ.
→ Integration Overhead → more time & effort

Automated Build Tools → It is a s/w that compiles source code to machine code.

→ Tools are used to automate the process
→ Tools can be categorized into 2 types:

a) Build Automation Utilities → performs task of generating build artifacts. Maven & Gradle.
b) Build Automation Server → i) On demand Automation
ii) Scheduled Automation
iii) Triggered Automation

Eg. Jenkins, Maven, Gradle, Travis CI, Circle CI, Apache Ant

Advantages of ABT →

1) Saves Time
2) Saves Money.
3) Keeping a history of builds & releases
4) accelerates the process.
5) dependencies will be eliminated

Examples of Automated Build Tools →

1) Jenkins →
   a) Free of Cost
   b) best for small to large businesses

→ It is an open source tool. It performs testing, building & deploying s/w. The platform is easy to install.

→ It works as an CI server & as a Continuous delivery hub.

Features →

1) It automates repetitive tasks in SD.
2) It has security features like authentication, encryption
3) It is highly scalable.
4) It has user friendly UI
5) It is highly customizable as it offers wide range of plugins.
6) distributes work across multiple machines.
7) it has good community support

2) Maven → a) Free of cost b) small → large business

→ It provides functionalities for project management. It has functions like project building, reporting & documentation.

It is extensible through plugins.

#Features → 1) it support working on multiple projects simultaneously.

2) it has features for dependency management

3) provides a large & growing repo of libraries.

4) No additional configuration required.

5) Highly flexible.

6) For dependency management it provides support to central repo of JARs.

3) Apache Ant → For individual business; free

→ It is used to assemble, compile, test & run JAVA applications. It has feature for combining build & dependency management.

→ allows to develop Antlibs

→ Antlibs will include Ant tasks & types.

→ No forcing of coding conventions

→ Highly flexible.

→ Availability of ready made & open source Antlibs

→ open source command line tool.

Version Control

→ They are a category of software tools that helps in recording changes made to files & keeping track of modifications done to code.

Importance →

→ helps dev. team to efficiently communicate & track all changes made to source code along with info. like who made and what changes have been made.

→ seperate branch for every contributor who made changes & changes aren't merged unless they are analyzed.

→ keeps source code organized

→ improves productivity

Benifits →

1) enhances project development
2) provides efficient collaboration
3) increases productivity
4) Reduces possibility of errors / conflicts
5) provides mobility (can be contributed from anywhere).
6) Helps in recovery in case of a disaster
7) Informs about, who, what, when, why changes have been made. of project
8) For each contributor, a diff copy of ~~the~~ is maintained & not merged unless it is validated. eg:- Git

2 things required to make changes visible
0) Commit 0) Update

repository is like "database of changes". It contains all edits & versions

Checkout (copy work) → It is personal copy of all files in a project. You can edit to this copy w/o affecting other's work & finally commit to the repository

Types of Version Control System

1) Local VCS → most simplest form. Developers manage their project files without a central repo. Developers create multiple copies / backups of their files manually.
eg:- RCS (Revision Control System)

2) Centralized → There is a single central repo that stores entire history of project. Developer check out files from central repository, make changes & commit back.
eg:- SVN (Subversion)

3) Distributed → Each developer has their own local copy of entire repo including its history. Developers can work independently, make changes & commit back to their local repository
eg:- Git