### **What does TDD mean?**

TDD is Test Driven Development. This means writing a test that fails because the specified functionality doesn't exist, then writing the simplest code that can make the test pass, then refactoring to remove duplication, etc. You repeat this Red-Green-Refactor loop over and over until you have a complete feature.

### **What does BDD mean?**

BDD is Behavior Driven Development. This means creating an executable specification that fails because the feature doesn't exist, then writing the simplest code that can make the spec pass. You repeat this until a release candidate is ready to ship.

The payload of an API is **the data you are interested in transporting to the server when you make an API request**. Simply put, it is the body of your HTTP request and response message.

The SOLID principle helps in reducing tight coupling. Tight coupling means a group of classes are highly dependent on one another which you should avoid in your code. Opposite of tight coupling is loose coupling and your code is considered as a good code when it has loosely-coupled classes. Loosely coupled classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable. Now let’s discuss one by one these principles…

**1. Single Responsibility Principle:**This principle states that “*a class should have only one reason to change*” which means every class should have a single responsibility or single job or single purpose. Take the example of developing software. The task is divided into different members doing different things as front-end designers do design, the tester does testing and backend developer takes care of backend development part then we can say that everyone has a single job or responsibility.  
Most of the time it happens that when programmers have to add features or new behavior they implement everything into the existing class which is completely wrong. It makes their code lengthy, complex and consumes time when later something needs to be modified. Use *layers* in your application and break God classes into smaller classes or modules.

**2. Open/Closed Principle:**This principle states that “*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*” which means you should be able to extend a class behavior, without modifying it.  
Suppose developer A needs to release an update for a library or framework and developer B wants some modification or add some feature on that then developer B is allowed to extend the existing class created by developer A but developer B is not supposed to modify the class directly. Using this principle separates the existing code from the modified code so it provides better stability, maintainability and minimizes changes as in your code.

**3. Liskov’s Substitution Principle:**The principle was introduced by Barbara Liskov in 1987 and according to this principle “*Derived or child classes must be substitutable for their base or parent classes*“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.  
You can understand it in a way that a farmer’s son should inherit farming skills from his father and should be able to replace his father if needed. If the son wants to become a farmer then he can replace his father but if he wants to become a cricketer then definitely the son can’t replace his father even though they both belong to the same family hierarchy.  
One of the classic examples of this principle is a rectangle having four sides. A rectangle’s height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class. In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov Substitution Principle. Check the link [Liskov Substitution Principle](https://www.youtube.com/watch?v=Jecou7B3nhc" \t "_blank) for better understanding.

**4. Interface Segregation Principle:**This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “*do not force any client to implement an interface which is irrelevant to them*“. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.  
Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets. In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don’t eat in your food. Here the menu should be different for different types of customers. The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

**5. Dependency Inversion Principle:**Before we discuss this topic keep in mind that Dependency Inversion and [Dependency Injection](https://en.wikipedia.org/wiki/Dependency_injection) both are different concepts. Most of the people get confused about it and consider both are the same. Now two key points are here to keep in mind about this principle

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

The above lines simply state that if a high module or class will be dependent more on low-level modules or class then your code would have tight coupling and if you will try to make a change in one class it can break another class which is risky at the production level. So always try to make classes loosely coupled as much as you can and you can achieve this through *abstraction*. The main motive of this principle is decoupling the dependencies so if class A changes the class B doesn’t need to care or know about the changes.  
You can consider the real-life example of a TV remote battery. Your remote needs a battery but it’s not dependent on the battery brand. You can use any XYZ brand that you want and it will work. So we can say that the TV remote is loosely coupled with the brand name. Dependency Inversion makes your code more reusable.

# **Factory Method Pattern**

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

#### **Advantage of Factory Design Pattern**

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### **Usage of Factory Design Pattern**

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

Singleton Pattern says that just**"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

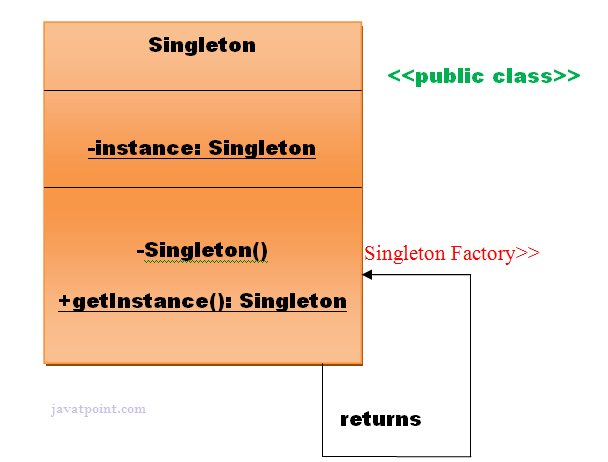
#### **Advantage of Singleton design pattern**

* Saves memory because object is not created at each request. Only single instance is reused again and again.

#### **Usage of Singleton design pattern**

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

#### **Uml of Singleton design pattern**



#### **How to create Singleton design pattern?**

To create the singleton class, we need to have static member of class, private constructor and static factory method.

* **Static member:** It gets memory only once because of static, itcontains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

### **Understanding early Instantiation of Singleton Pattern**

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of classloading.

Let's see the example of singleton design pattern using early instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj=**new** A();//Early, instance will be created at load time
3. **private** A(){}
5. **public** **static** A getA(){
6. **return** obj;
7. }
9. **public** **void** doSomething(){
10. //write your code
11. }
12. }

### **Understanding lazy Instantiation of Singleton Pattern**

In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.

Let's see the simple example of singleton design pattern using lazy instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj;
3. **private** A(){}
5. **public** **static** A getA(){
6. **if** (obj == **null**){
7. **synchronized**(Singleton.**class**){
8. **if** (obj == **null**){
9. obj = **new** Singleton();//instance will be created at request time
10. }
11. }
12. }
13. **return** obj;
14. }
16. **public** **void** doSomething(){
17. //write your code
18. }
19. }

### **Significance of Classloader in Singleton Pattern**

#### **If singleton class is loaded by two classloaders, two instance of singleton class will be created, one for each classloader.**

### **Significance of Serialization in Singleton Pattern**

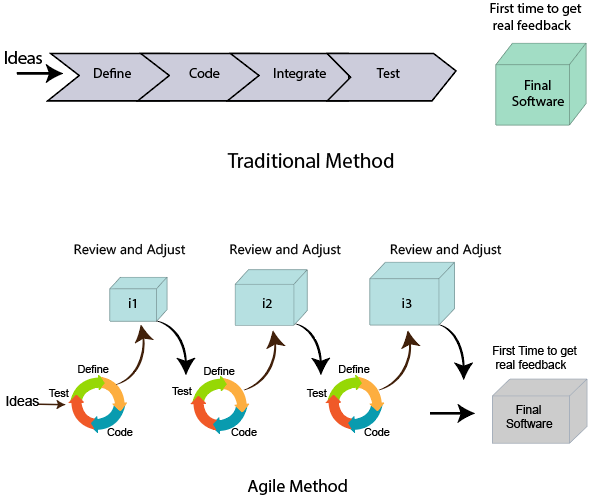
If singleton class is Serializable, you can serialize the singleton instance. Once it is serialized, you can deserialize it but it will not return the singleton object.

To resolve this issue, you need to override the **readResolve() method** that enforces the singleton. It is called just after the object is deserialized. It returns the singleton object.

What is Agile Methodology?

An agile methodology is an iterative approach to software development. Each iteration of agile methodology takes a short time interval of 1 to 4 weeks. The agile development process is aligned to deliver the changing business requirement. It distributes the software with faster and fewer changes.

The single-phase software development takes 6 to 18 months. In single-phase development, all the requirement gathering and risks management factors are predicted initially.



## Roles in Agile

There are two different roles in a Agile methodology. These are the Scrum Master and Product Owner.

### **1. Scrum Master**

The Scrum Master is a team leader and facility provider who helps the team member to follow agile practices, so that the team member meets their commitments and customers requirements. The scrum master plays the following responsibilities:

* They enable the close co-operation between all the roles and functions.
* They remove all the blocks which occur.
* They safeguard the team from any disturbances.
* They work with the organization to track the progress and processes of the company.
* They ensure that Agile Inspect & Adapt processes are leveraged correctly which includes
  + Planned meetings
  + Daily stand-ups
  + Demo
  + Review
  + Retrospective meetings, and
  + Facilitate team meetings and decision-making process.

Advantages of Agile Methodology

1. Customer satisfaction is rapid, continuous development and delivery of useful software.
2. Customer, Developer, and Product Owner interact regularly to emphasize rather than processes and tools.
3. Product is developed fast and frequently delivered (weeks rather than months.)
4. A face-to-face conversation is the best form of communication.
5. It continuously gave attention to technical excellence and good design.
6. Daily and close cooperation between business people and developers.
7. Regular adaptation to changing circumstances.
8. Even late changes in requirements are welcomed.

|  |  |
| --- | --- |
| **Agile methodology** | **Waterfall model** |
| It follows the incremental approach. | It is a sequential design process. |
| It divides the project development lifecycle into a sprint. | The software development process is divided into distinct phases. |
| Agile methodology is a flexible methodology. | The Waterfall is a structured software development methodology. |
| Agile is the collection of many different projects. | It is completed as one single project. |
| The test plan is reviewed after each sprint | Test plan is reviewed after complete development. |
| Testing team can take part in the requirements change phase without problems. | It is difficult for the test to initiate any change in needs. |

The Twelve Principle of Agile Manifesto

1. **Customer Satisfaction:** Manifesto provides high priority to satisfy the costumer's requirements. This is done through early and continuous delivery of valuable software.
2. **Welcome Change:** Making changes during software development is common and inevitable. Every changingrequirement should be welcome, evenin the late development phase. Agile process works to increase the customers' competitive advantage.
3. **Deliver the Working Software:** Deliver the working software frequently, ranging from a few weeks to a few months with considering the shortest timeperiod.
4. **Collaboration:** Business people (Scrum Master and Project Owner) and developers must work together during the entire life of a project development phase.
5. **Motivation:** Projects should be build around motivated team members. Provide such environment that supportsindividual team members and trust them. It makes them feel responsible for gettingthe job donethoroughly.
6. **Face-to-face Conversation:** Face-to-face conversation betweenScrum Master anddevelopment team and between the Scrum Master and customers for the most efficient and effective method of conveying information to and within a development team.
7. **Measure the Progress as per the Working Software:** The working software is the key and primary measure of the progress.
8. **Maintain Constant Pace:** The aim of agile development is sustainable development. All the businesses and users should be able to maintain a constant pace with the project.
9. **Monitoring:** Pay regular attention to technical excellence and good design to maximize agility.
10. **Simplicity:** Keep things simple and use simple terms to measure the work that is not completed.
11. **Self-organized Teams:** The Agile team should be self-organized. They should not be depending heavily on other teams because the best architectures, requirements, and designs emerge from self-organized teams.
12. **Review the Work Regularly:** The work should be reviewed at regular intervals, so that the team canreflect on how to become more productive and adjust its behavior accordingly.

# **Agile Software Development Life Cycle (SDLC)**

**Software development life cycle (SDLC)** is a phenomenon to **design**, **develop** and, **test** high-quality software. The primary aim of SDLC is to produce high-quality software that fulfills the customer requirement within times and cost estimates.

**Each iteration of agile SDLC consists of cross-functional teams working on various phases:**

1. Requirement gathering and analysis
2. Design the requirements
3. Construction/ iteration
4. Deployment
5. Testing
6. Feedback

# **Agile Project Management**

Agile project management is an interactive approach to manage software development. The agile project management focuses on continuous releases and covers customer feedback with every iteration.

Traditionally the agile project management is classified into two frameworks: **scrum** and **kanban**. The [scrum framework](https://www.javatpoint.com/agile-scrum) focused fixed-length project iterations, whereas [kanban framework](https://www.javatpoint.com/agile-kanban) focused on continuous releases. After competition of project first iteration (or steps) project management activity immediately moves on to the next.