MODULE 3

TOPIC 1
DESIGN PATTERNS

WHAT IS DESIGN PATTERN

- Design patterns represent the best practices used by experienced object oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

- A design pattern is a general repeatable solution to a commonly occurring problem in software design
- It is a description or template for how to solve a problem that can be used in many different situations

- A pattern is a recurring solution to a standard problem, in a context
- A Pattern has four essential elements:

  Pattern name

  Problem

  Solution

  Consequences

- **The pattern name** that describes a design problem, its solutions and consequences in a word or two.
- A pattern name provides vocabulary using which we can communicate with other people, document them and reference them in software designs. Finding a good name for a design pattern is a major issue.
- **The problem** describes when to apply the pattern. It explains the problem and its context.

- **The solution** describes the elements (classes and objects) that make up the design, their relationships, responsibilities and collaborations. The solution does not describe a concrete design or implementation, as a pattern is a template that can be applied in many situations. The pattern provides an abstract description of a design problem and how a general arrangement of elements solves the problem.
- **The consequences** are the results and trade-offs of applying the pattern. These are the costs and benefits of applying the patterns. These consequences involve space and time trade-offs, language and implementation issues as well as the effect of pattern on system's flexibility, extensibility or portability. Listing these consequences explicitly enables us to understand and evaluate them.

What are the benefits of the design pattern?

- Design patterns can speed up the development process.
- Reusing the design patterns helps to prevent subtle issues that can cause major problems and it also improves code readability
- Design pattern provides general solutions, documented in a format
- A standard solution to a common programming problem enables large scale reuse of software

Characteristics of design pattern

- **Smart:** They are elegant solutions that a novice would not think of immediately
- **Generic:** They are not normally depend on a specific system type, programming languages or application domains. They are generic in nature
- **Well proven:** They have been identified from real object oriented systems. They are not a single paper work but they have been successfully tested in several systems

- **Simple:** they are usually quite small involving small number of classes. The complex systems are built using different design patterns and their combinations
- **Reusable:** They are documented in such a manner to increase reusability
- **Object oriented:** They are built with the basic object oriented mechanisms such as classes, objects,generalization and polymorphism

What is Gang of Four (GOF)?

- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.
- These authors are collectively known as **Gang of Four (GOF)**.

- They developed 23 design patterns.It might hard to find the design pattern that solves our problem
- How to select a design pattern????

  Consider how design patterns solve design problems:

  Scan intent sections:

  Study how patterns interrelate:

  Study patterns of like purpose

  Examine a cause of redesign:

- **Consider how design patterns solve design problems**

  Considering how design patterns help you find appropriate objects, determine object granularity, specify object interfaces and several other ways in which design patterns solve the problems will let you choose the appropriate design pattern.
- **Scan intent sections**

  Looking at the intent section of each design pattern's specification we can choose the appropriate design pattern.
- **Study how patterns interrelate**

  The relationships between the patterns will direct us to choose the right patterns or group of patterns.

- **Study patterns of like purpose**

  Each design pattern specification will conclude with a comparison of that pattern with other related patterns. This will give you an insight into the similarities and differences between patterns of like purpose
- **Examine a cause of redesign**

  Look at your problem and identify if there are any causes of redesign. Then look at the catalog of patterns that will help you avoid the causes of redesign

TYPES OF DESIGN PATTERN(*****)

- Design Pattern can be classified into three types

  Creational design patterns

  Structural design patterns

  Behavioral design patterns

Design pattern catalog

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Vistor

1. Creational Design Patterns:

- Creational design patterns are concerned with **the way of creating objects.**

- These design patterns are used when a decision must be made at the time of the instantiation of a class (i.e. creating an object of a class).

- This pattern can be further divided into class-creation patterns and object-creational patterns.

- Class-creation patterns use inheritance effectively in the instantiation process.

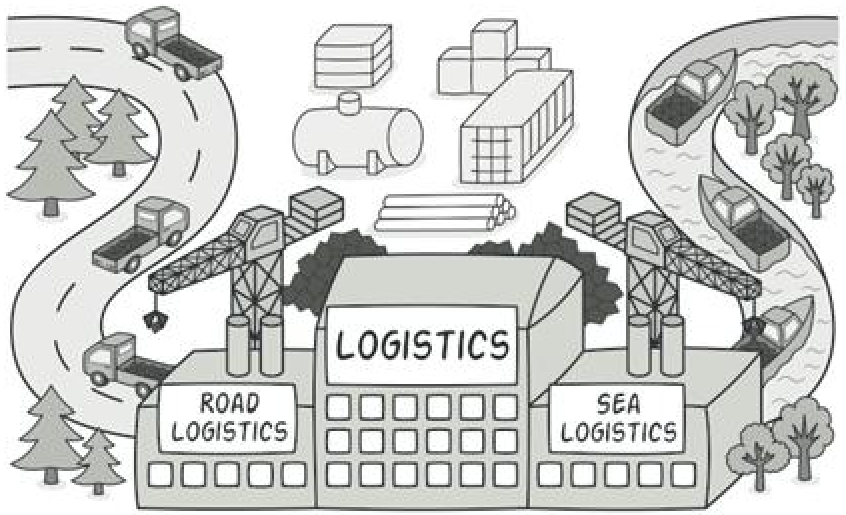
- Object-creation patterns use delegation effectively to get the job done.

a) FACTORY METHOD

- **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

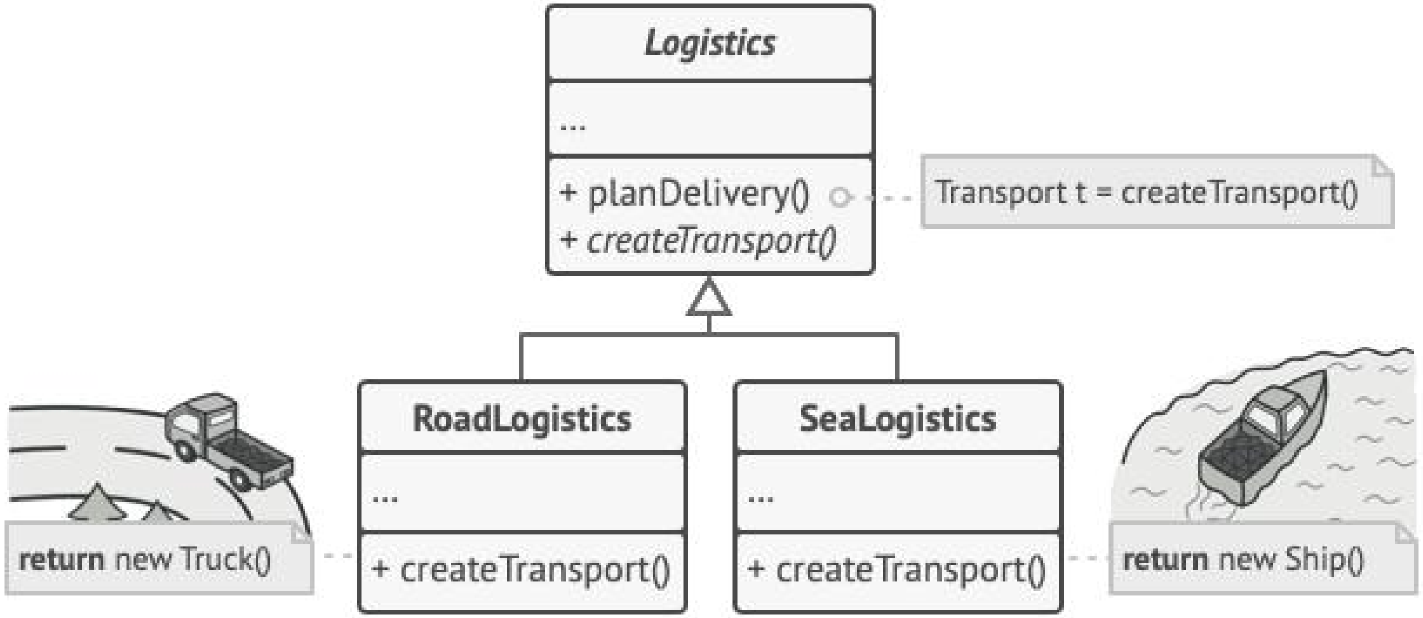
PROBLEM

Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulK of your code lives inside the Truck class.
After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.



:SOLUTION

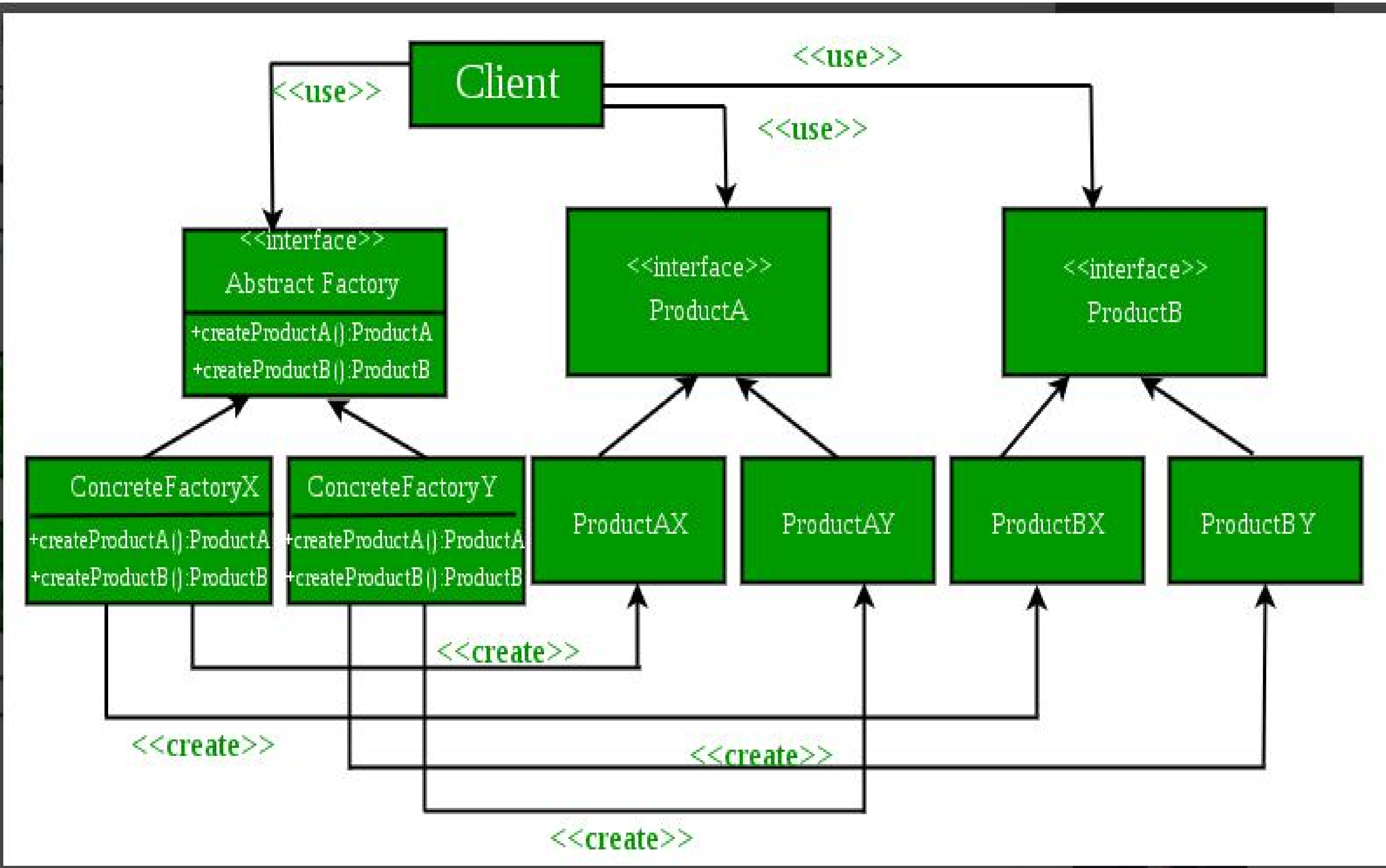
The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special *factory* method.The objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as *products*.



Subclasses can alter the class of objects being returned by the factory method.

b) Abstract Method

- *The Factory Method design pattern could be used to create objects related to a single family.*
- Abstract factory patterns work around a super factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In abstract factory pattern ,an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the factory pattern.


Client
<<use>>
<<use>>
<<use>>
<<interface>>
Abstract Factory
+createProductA():ProductA
+createProductB():ProductB
<<interface>>
ProductA
<<interface>>
ProductB
ConcreteFactoryX
+createProductA():ProductA
+createProductB():ProductB
ConcreteFactoryY
+createProductA():ProductA
+createProductB():ProductB
ProductAX
ProductAY
ProductBX
ProductBY
<<create>>
<<create>>
<<create>>
<<create>>

- **Abstract Factory** : Declares an interface for operations that create abstract product objects.
- **Concrete Factory** : Implements the operations declared in the Abstract Factory to create concrete product objects.
- **Product** : Defines a product object to be created by the corresponding concrete factory and implements the Abstract Product interface.
- **Client** : Uses only interfaces declared by Abstract Factory and Abstract Product

- The client does not know or care which concrete objects it gets from each of these concrete factories since it uses only the generic interfaces of their products

BUILDER PATTERN

- Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- A Builder class builds the final object step by step. This builder is independent of other objects.

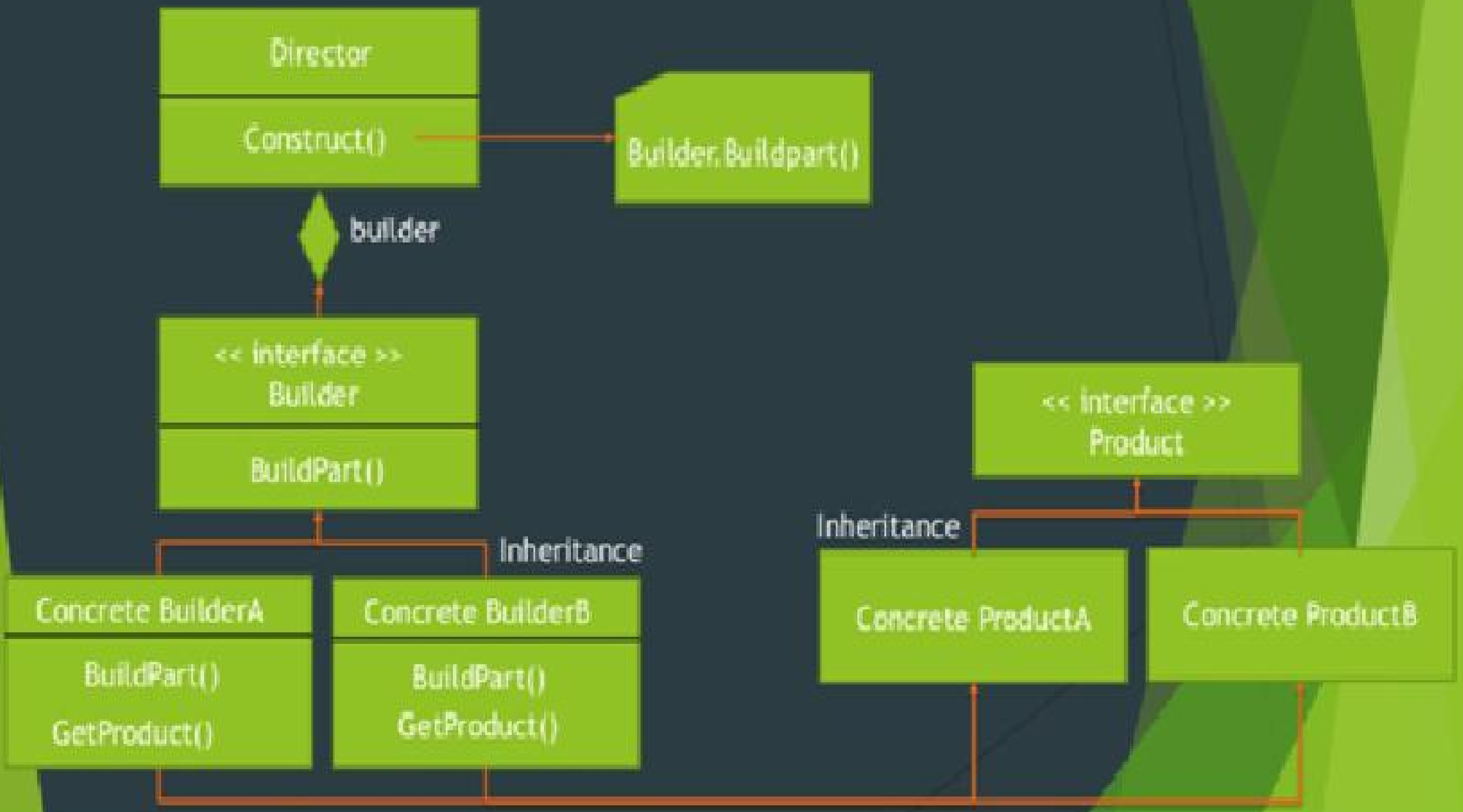
- **DIRECTOR:** Construct an object using Builder interface.Client interacts with the director
- **BUILDER**(interface): Provides an abstract interface for creating parts of the product object
- **PRODUCT**(interface):Provides an interface for product family

- Choose the builder pattern when

  Multiple constructor in your class and might expand

  Avoiding multiple constructor parameter

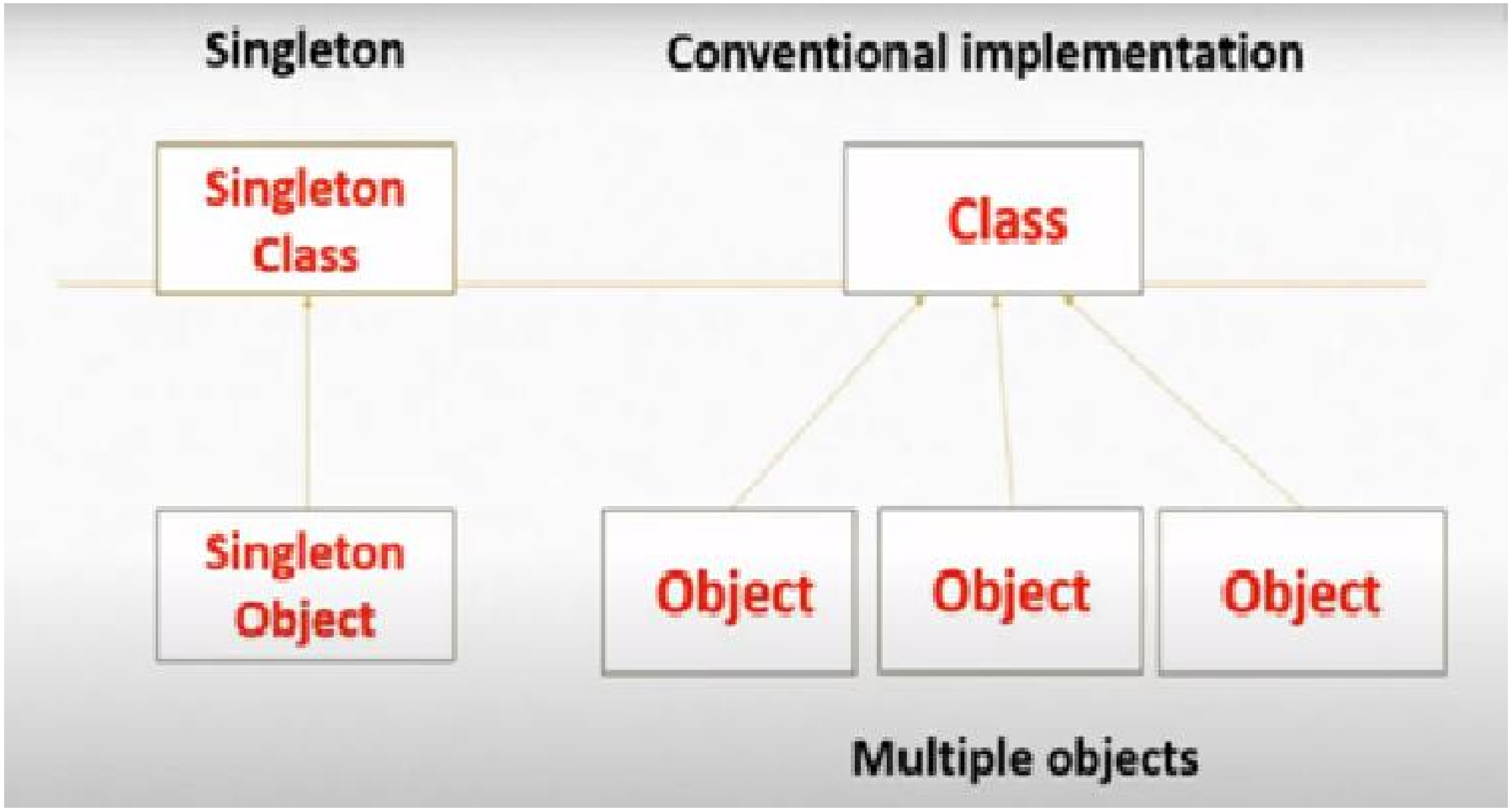
Structure



Director
Construct()
Builder.Buildpart()
builder
<< interface >> Builder
BuildPart()
Inheritance
Concrete BuilderA
BuildPart()
GetProduct()
Concrete BuilderB
BuildPart()
GetProduct()
<< Interface >> Product
Inheritance
Concrete ProductA
Concrete ProductB

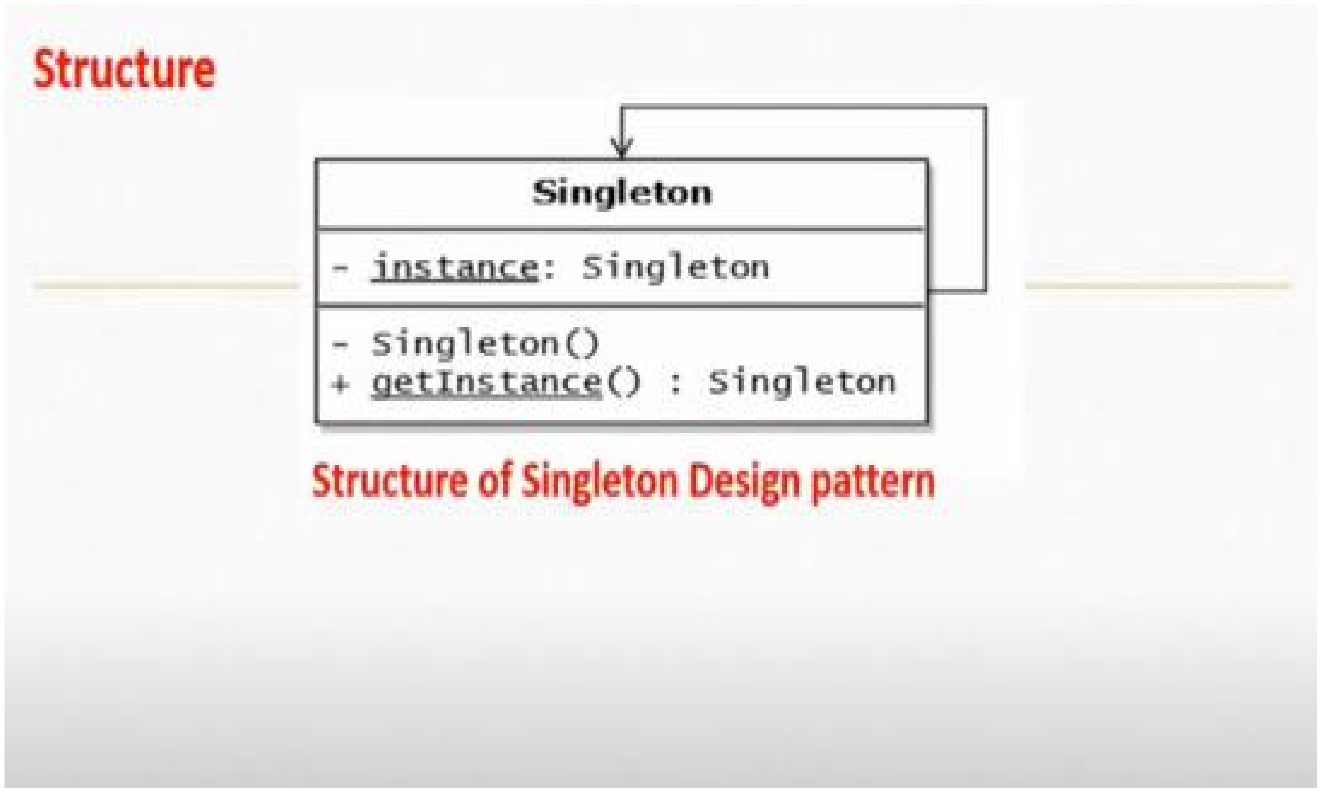
SINGLETON DESIGN PATTERN

- Singleton pattern is one of the simplest design pattern.
- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- Define a class that has only one instance and provides a global point of access to it
- A class must ensure that only single instance should be created and single object can be used by all other classes



Eg: Prime Minister or President of our country

- Mostly used in multi-threaded and database applications
- It saves memory because object is not created at each request. Only single instance is reused again and again


Structure
Singleton
- instance: Singleton
- Singleton()
+ getInstance() : Singleton
Structure of Singleton Design pattern

Instance: It defines instance operation that lets clients to access its unique instance

Get instance(): It is responsible for accessing unique instance

PROTOTYPE DESIGN PATTERN

- The prototype pattern is a creational design pattern.
- Prototype patterns is required, when object creation is time consuming, and costly operation, so we create object with existing object itself.
- One of the best available way to create object from existing objects are **clone() method**.
- Clone is the simplest approach to implement prototype pattern.

- Participants for performing the prototype design pattern are

  Prototype

    Declares an instance for cloning itself

  Concrete prototype

    Implement an operation of cloning itself

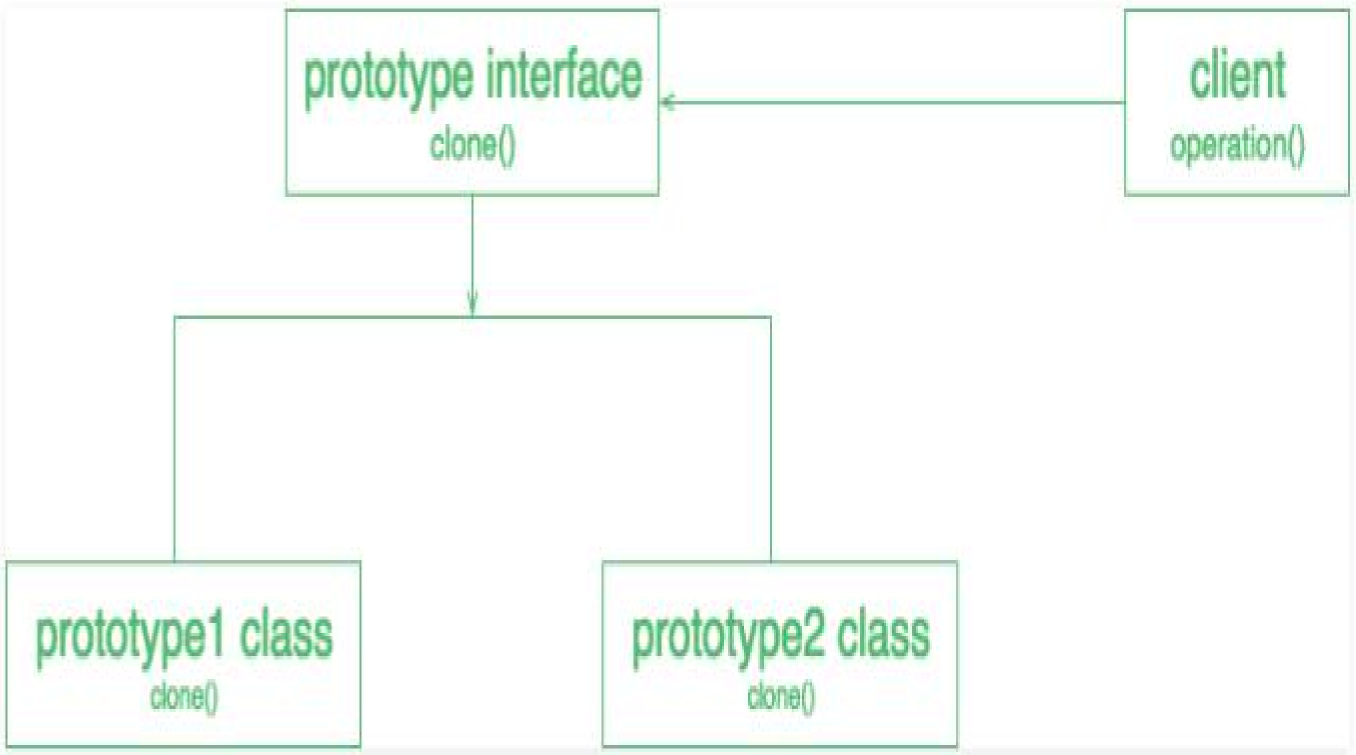
  Client

    Creates an object by asking the prototype to clone itself

Applicability

- When a system should be independent of how its product are created, composed,and represented
- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

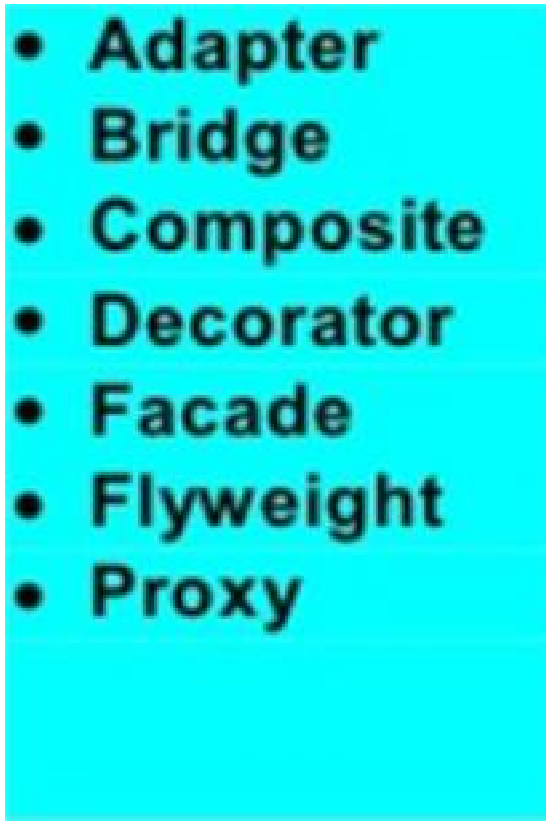
Structure


prototype interface
clone()
client
operation()
prototype1 class
clone()
prototype2 class
clone()

2.STRUCTURAL DESIGN PATTERN

- Structural design patterns are concerned with how classes and objects are composed to form larger structures.
- Structural class patterns use inheritance to compose interfaces or implementations.
- Eg: Multiple inheritances mixes two or more classes into one

Structural design pattern



- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Adapter

- Adapter pattern works as a bridge between two incompatible interfaces.
- This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
- This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

  Example : A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

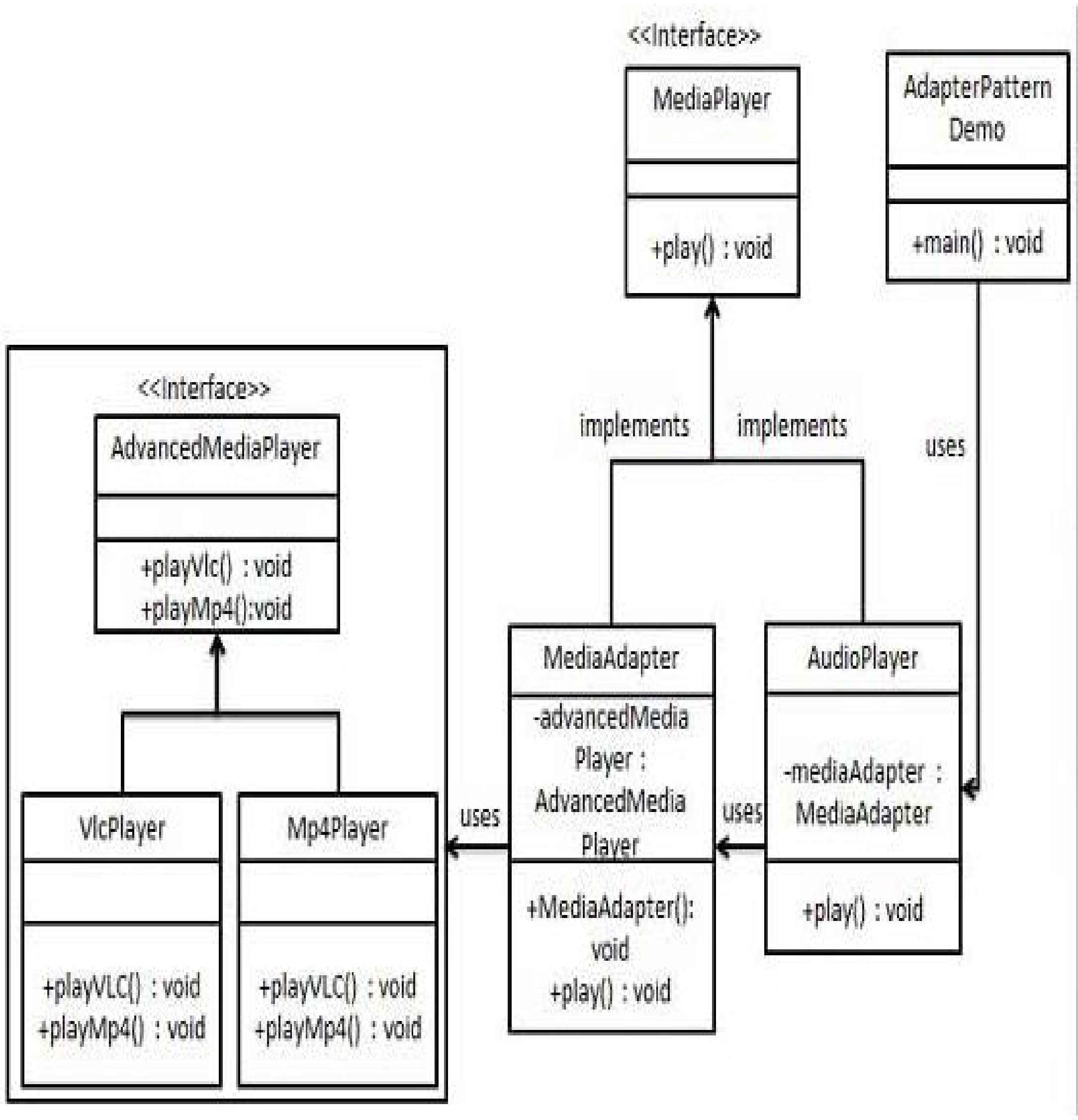
Step 1:Create interfaces for Media Player and Advanced Media Player.

Step 2: Create concrete classes implementing the *AdvancedMediaPlayer* interface.

Step 3:Create adapter class implementing the *MediaPlayer* interface.

Step 4:Create concrete class implementing the *MediaPlayer* interface.

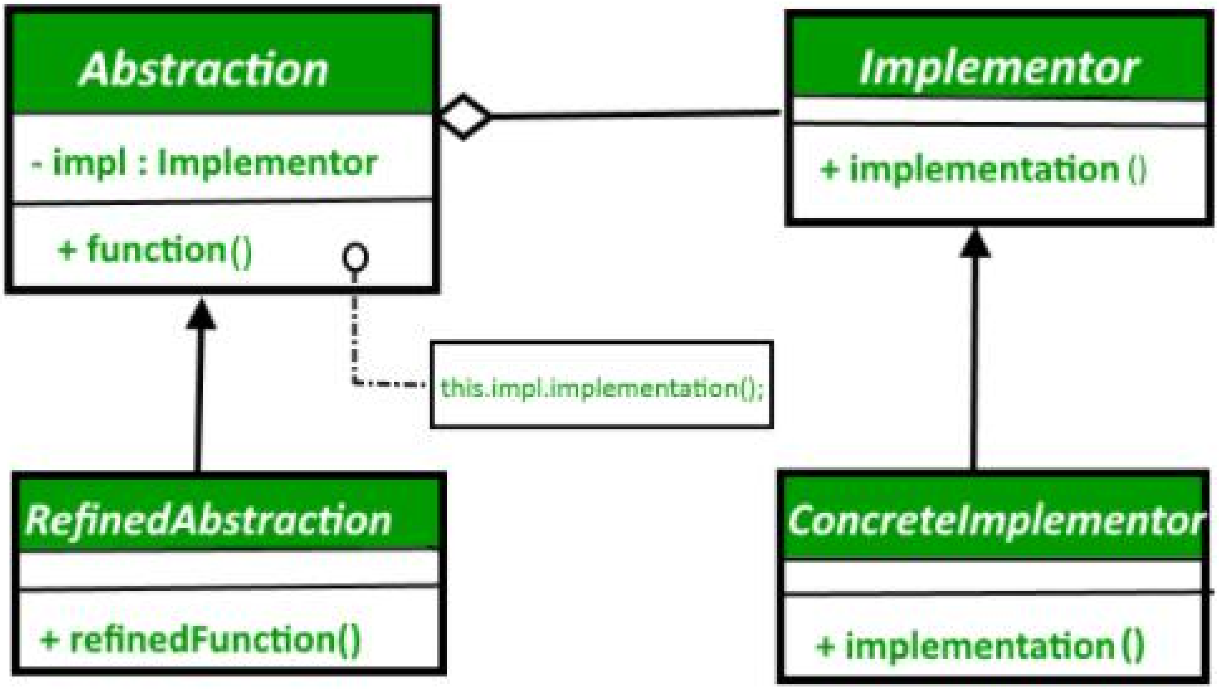
Step 5: Use the AudioPlayer to play different types of audio formats.



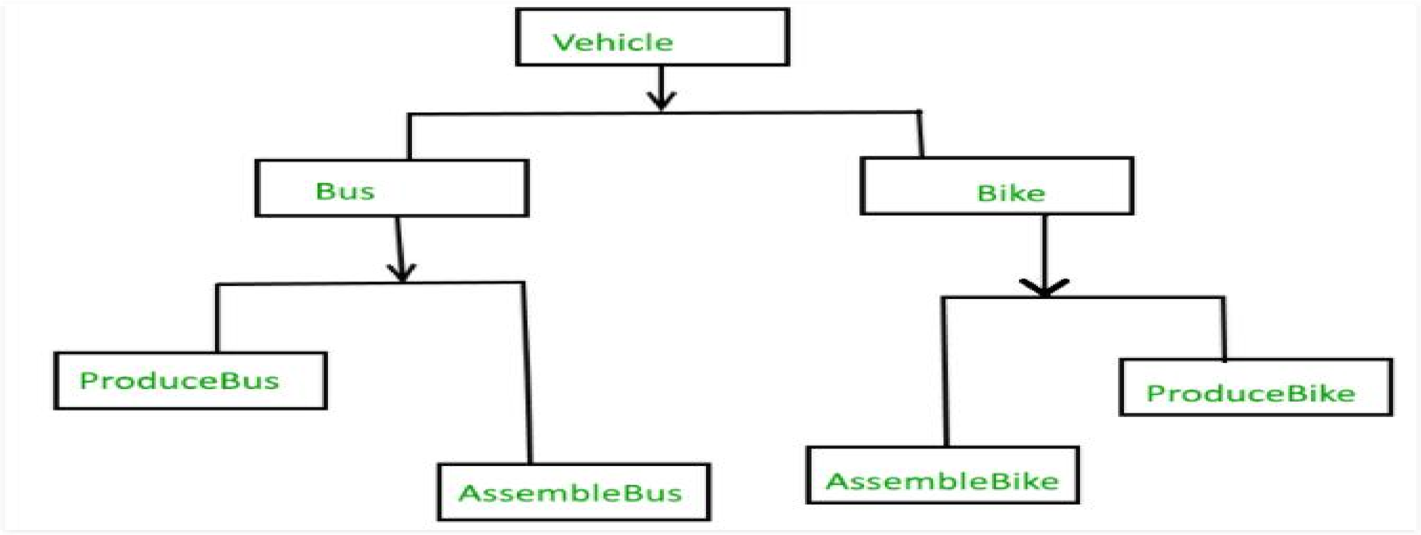
Bridge

- “Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. “
- This type of design pattern comes under structural pattern
- This pattern decouples implementation class and abstract class by providing a bridge structure between them.
- This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes.
- Both types of classes can be altered structurally without affecting each other.
- The abstraction is an interface or abstract class and the implementor is also an interface or abstract class.

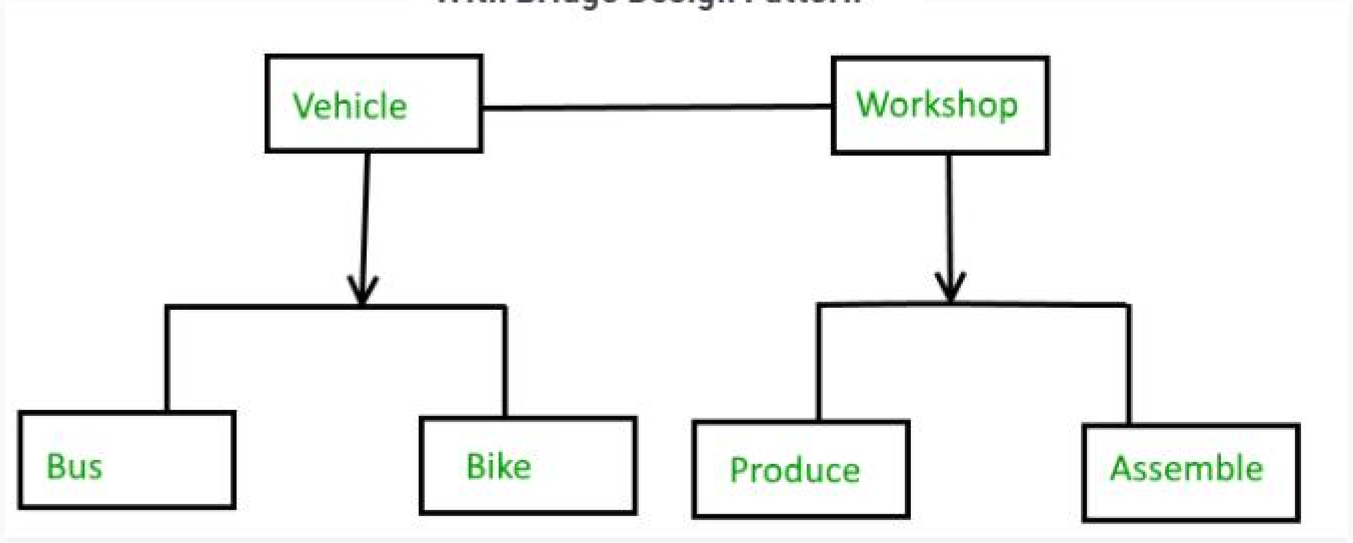
Structure


Abstraction
- impl : Implementor
+ function()
this.impl.implementation();
Implementor
+ implementation ()
RefinedAbstraction
+ refinedFunction()
ConcreteImplementor
+ implementation ()

Without Bridge Design Pattern

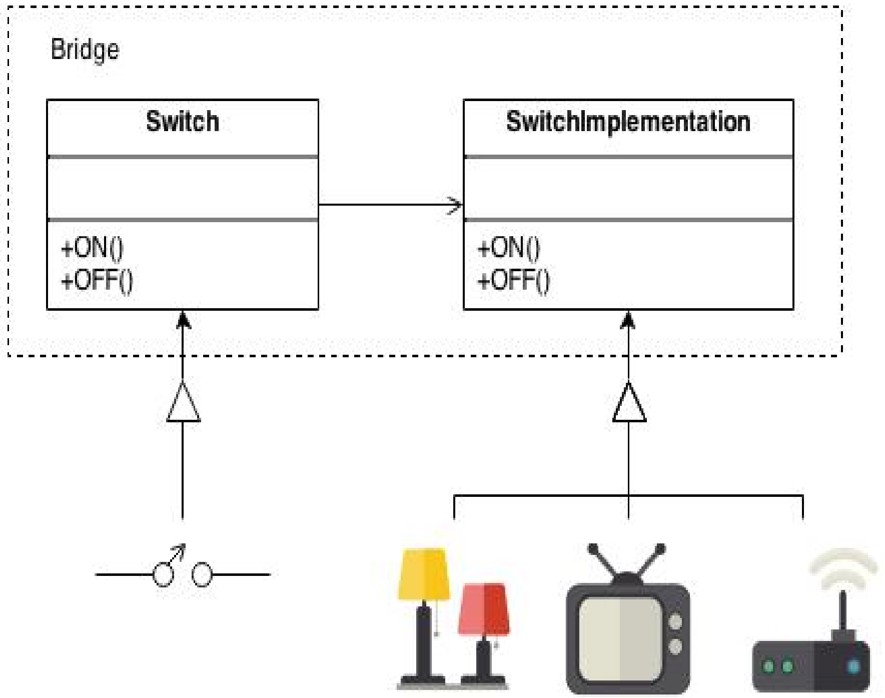


With Bridge Design Pattern



- The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.
- A household switch controlling lights, ceiling fans, etc. is an example of the Bridge.
- The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.

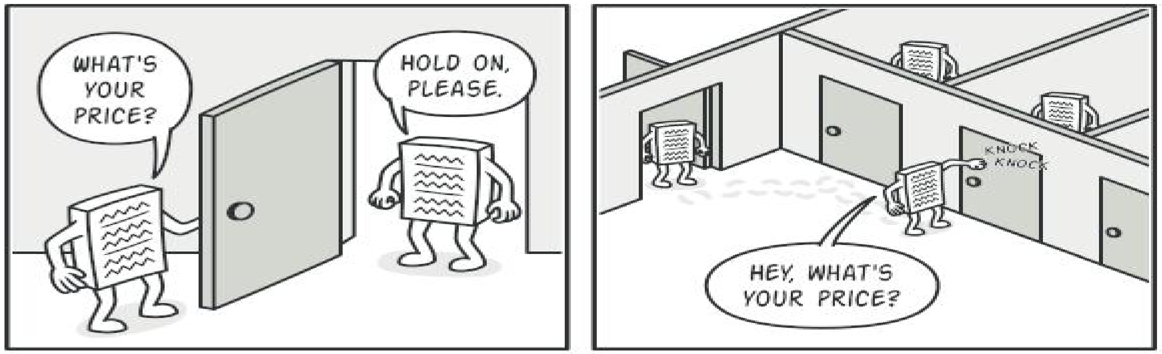
Eg:



Composite

(Thing made up of several parts or elements)

- Compose objects into tree structure to represent part-whole hierarchy
- Allows you to treat individual objects and compositions object uniformly
- This is applicable when we want to create an object and represent in tree struc



Component is the abstraction for all components

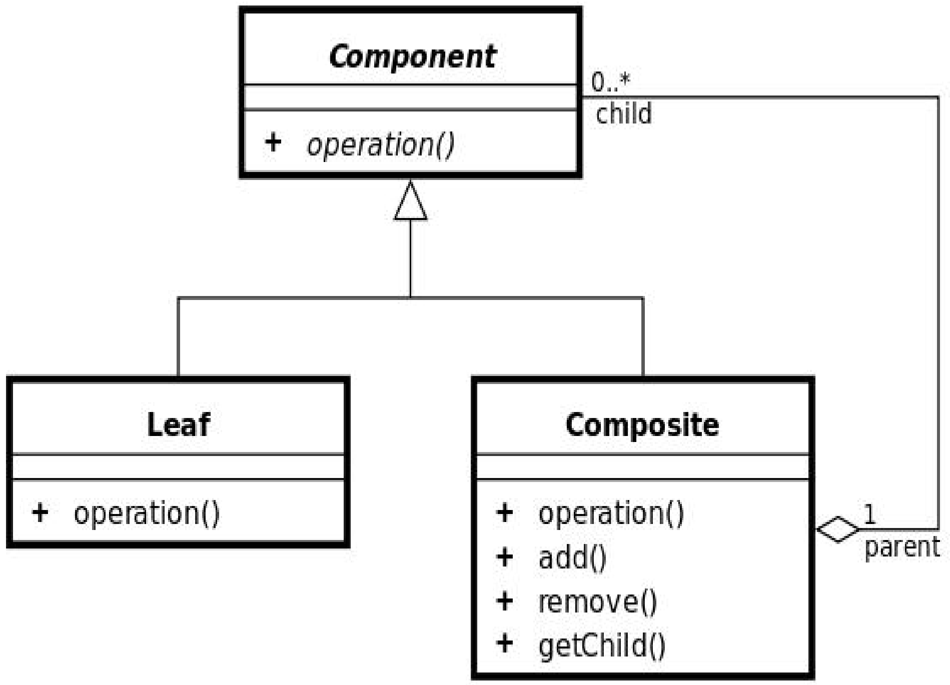
- declares the interface for objects in the composition

Leaf represents leaf objects in the composition

- implements all Component methods

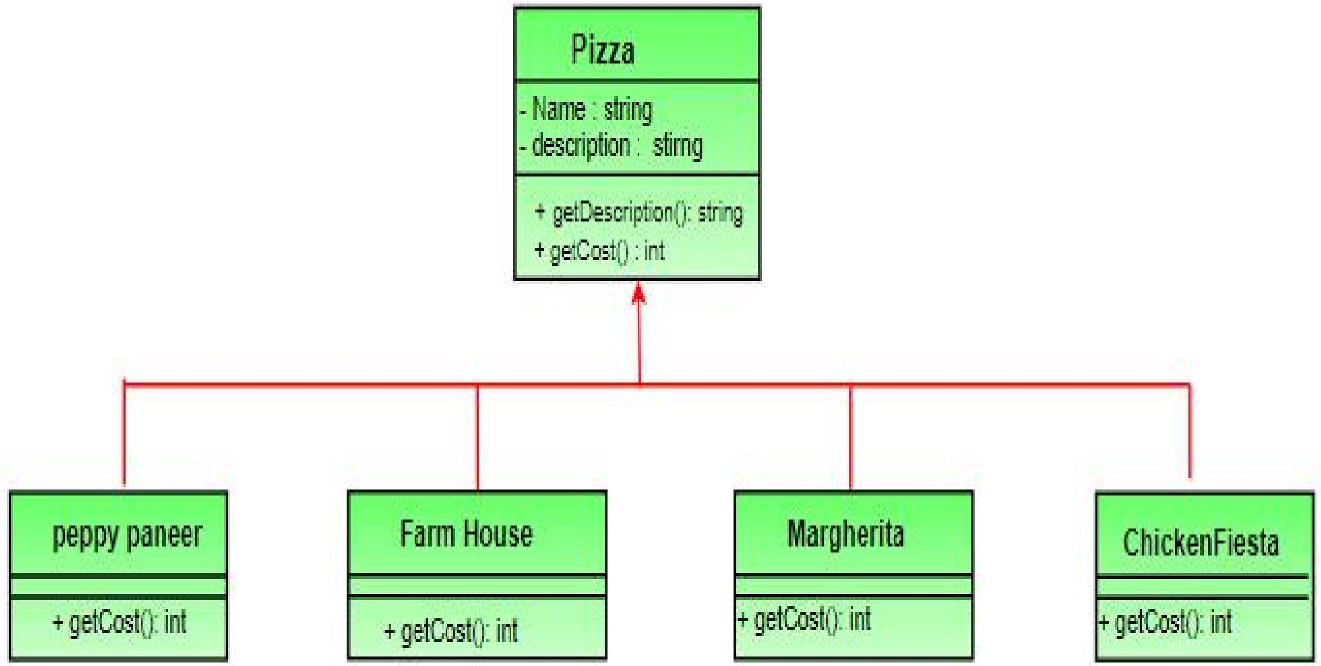
Composite represents a composite Component (component having children)

- implements methods to manipulate children
- implements all Component methods, generally by delegating them to its children

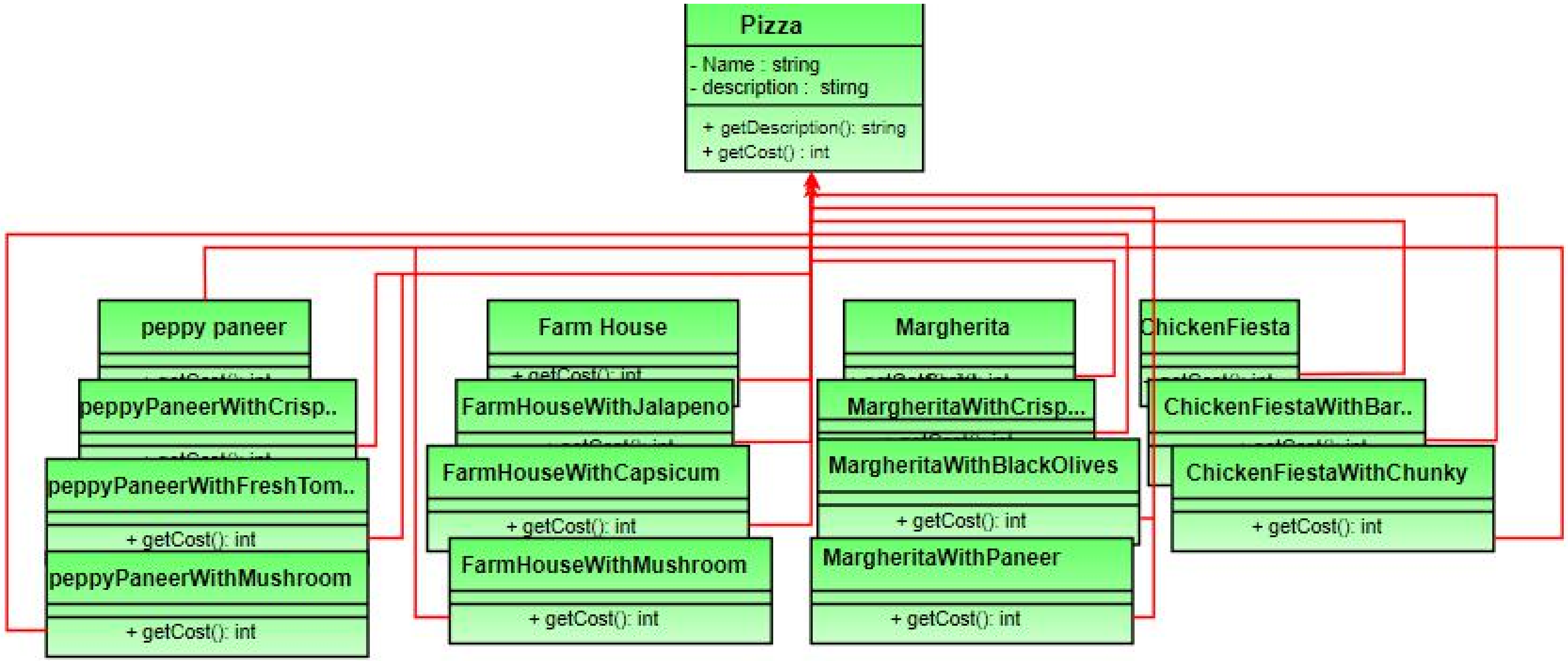


Decorator

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure during dynamically.
- This design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.
- This pattern creates a decorator class which wraps the original class and provides additional functionality
- More flexible than inheritance(static)
- It is also known as wrapper pattern

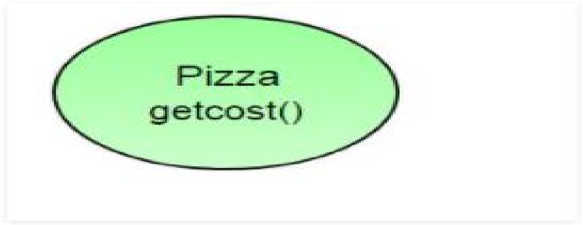


•Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a **Pizza** class.
•Now suppose a new requirement, in addition to a pizza, customer can also ask for several toppings such as Fresh Tomato, Paneer, Jalapeno, Capsicum, Barbeque, etc

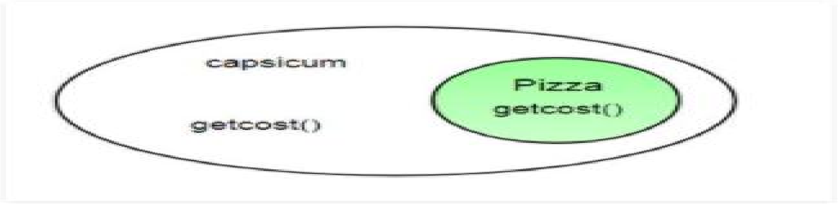


- This looks very complex. There are way too many classes and is a maintenance nightmare. Also if we want to add a new topping or pizza we have to add so many classes

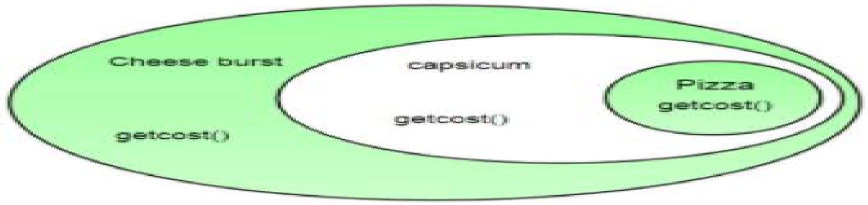
1. Take a pizza object.



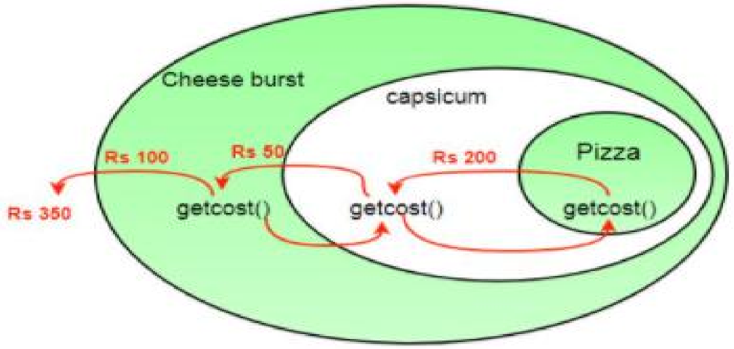
2. "Decorate" it with a Capsicum object.



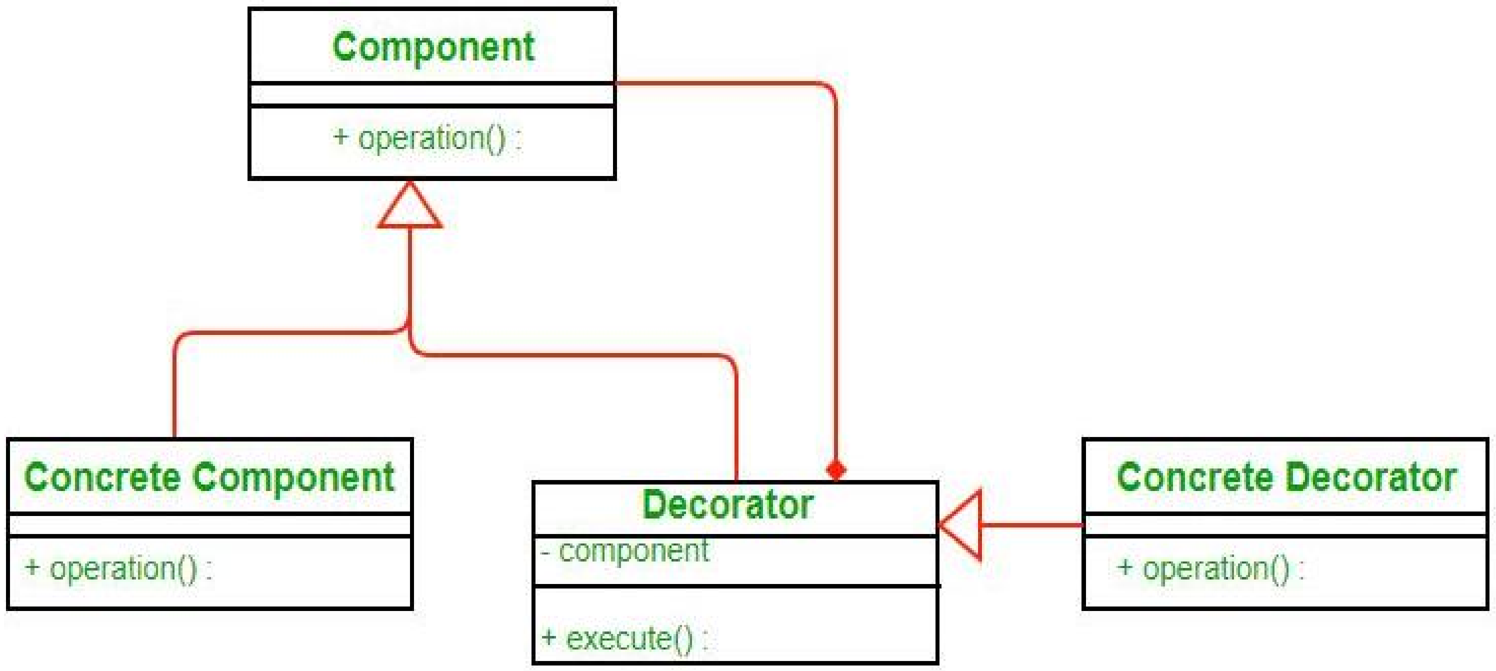
3. "Decorate" it with a CheeseBurst object.



4. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.



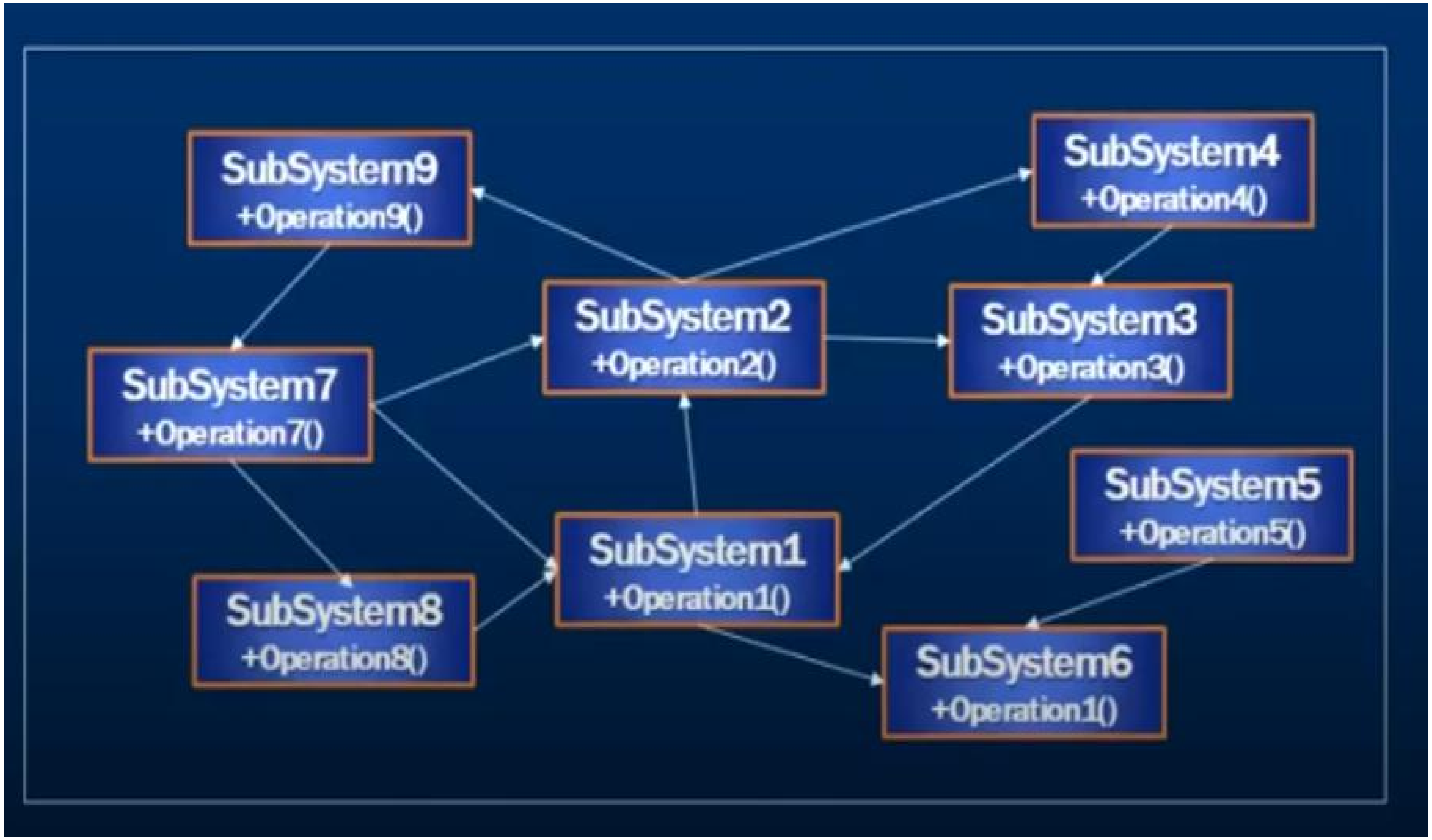
Structure



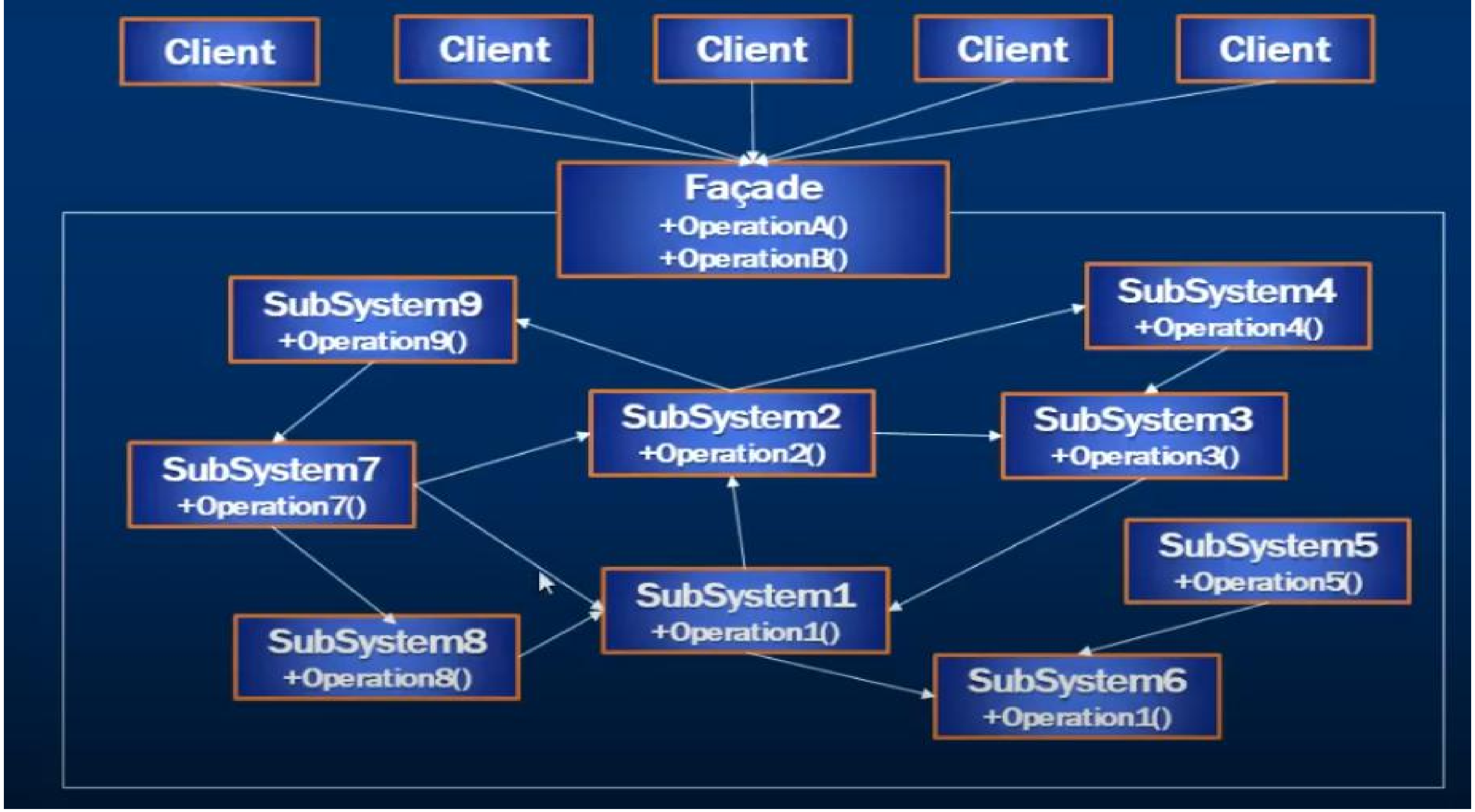
• The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime.
•The decorator pattern is an alternative to subclassing. Subclassing adds behaviour at compile time, and the change affects all instances of the original class; decorating can provide new behaviour at runtime for individual objects.

Facade

- Provides an unified and simplified interface in a subsystem,therfore it hides the complexities of the subsystem from the client
- Falls under the category of structural design pattern
- Facade means face or mask
- We want to provide a simple interface to a complex subsystem.subsystems ofetn get more complex as they evolve
- There are many dependencies between clients and the implementation classes of an abstraction
- We want to layer the subsystems.Use a facade to define an entry point to each subsystem level

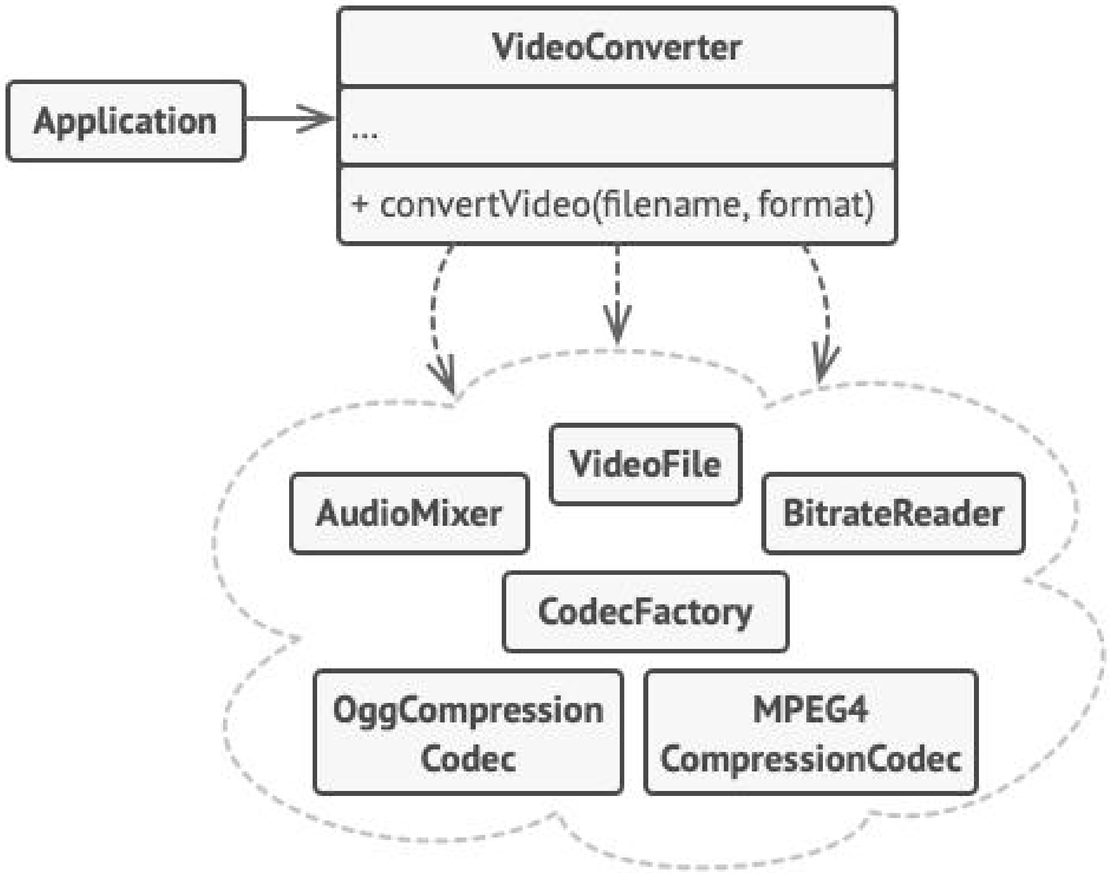


There are many subsystems interacting with each other and each of them having their own implementations.This is more complex.To simplify this,making use of facade design pattern



- •It provides single simplified interface with more general facilities of a subsystem.
- •Clients will interact with this facade object to get their expected subsystem details
- •Facade knows which system or which subsystem classes are responsible for a request

Eg:


VideoConverter
...
+ convertVideo(filename, format)
Application
VideoFile
AudioMixer
BitrateReader
CodecFactory
OggCompression Codec
MPEG4 CompressionCodec

Flyweight

- Mostly used when we want to create large number of similar patterns
- Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory requirements
- **Intrinsic and Extrinsic States**

  Each "**flyweight**" object is divided into two pieces

  The state-dependent (**extrinsic**) part, and the state-independent (**intrinsic**) part

Intrinsic:

Intrinsic state is shared in the **Flyweight** object.

Suppose in a text editor when we enter a character, an object of Character class is created, the attributes of the Character class are {name, font, size}. We do not need to create an object every time client enters a character since letter ‘B’ is no different from another ‘B’ . If client again types a ‘B’ we simply return the object which we have already created before. Now all these are intrinsic states (name, font, size), since they can be shared among the different objects as they are similar to each other.

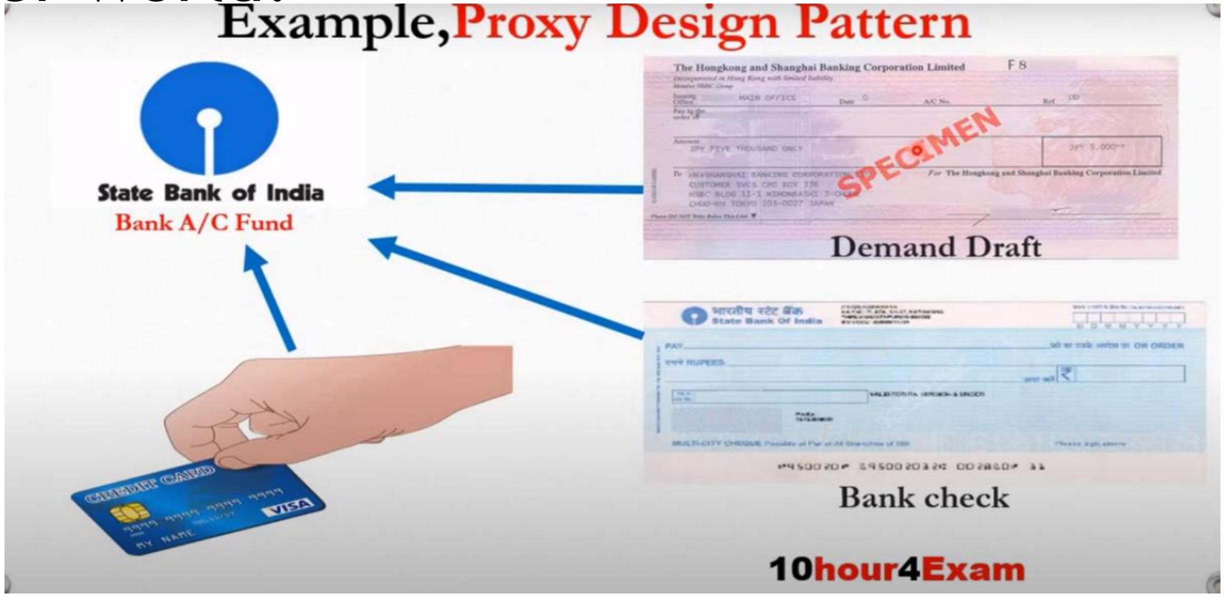
- **Extrinsic** state is stored or computed by client objects, and passed to the **Flyweight** when its operations are invoked.

Eg:

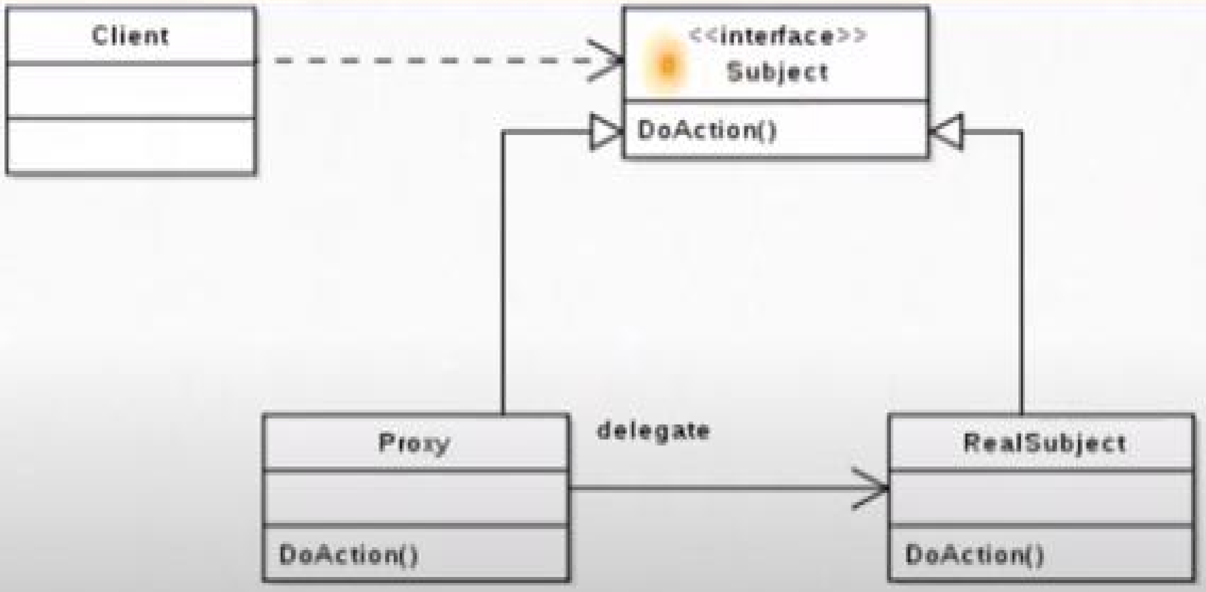
Suppose we need to add more attributes to the Character class, they are row and column. They specify the position of a character in the document. Now these attributes will not be similar even for same characters, since no two characters will have the same position in a document, these states are termed as extrinsic states, and they can’t be shared among objects.

Proxy

- In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.
- In proxy pattern, we create object having original object to interface its functionality to outer world.



Structure



Client
<<interface>>
Subject
DoAction()
Proxy
DoAction()
delegate
RealSubject
DoAction()

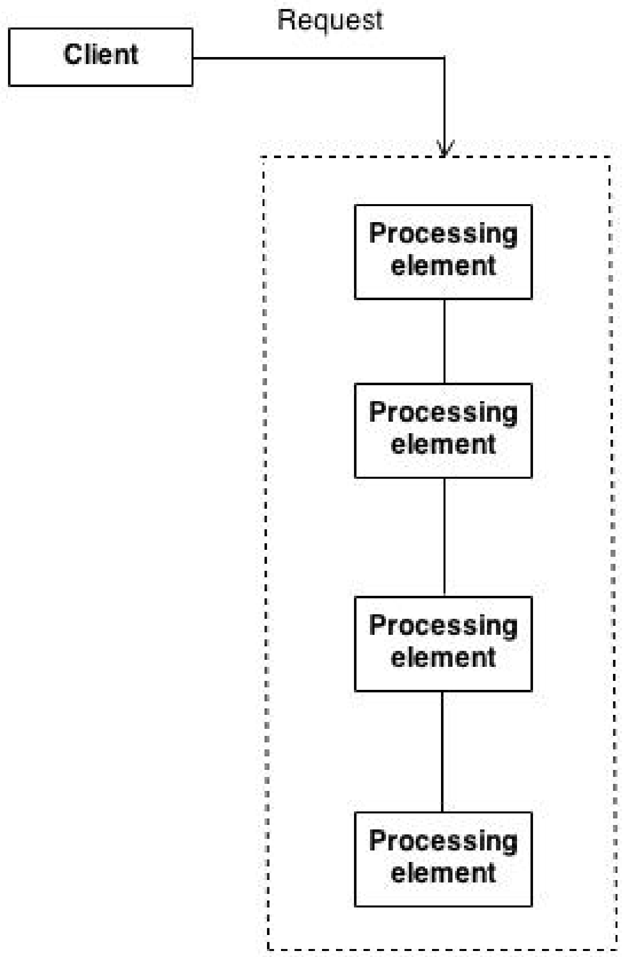
Behavioral Design Pattern

- Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns.By doing so,these patterns increase flexibility in carrying out this communication.
- Behavioral patterns are those which are concerned with interaction between the objects.
- A behavioral pattern explains how objects interact. It describes how objects and classes send messages to each other to make things happen and how the steps of a task are divided among different objects.

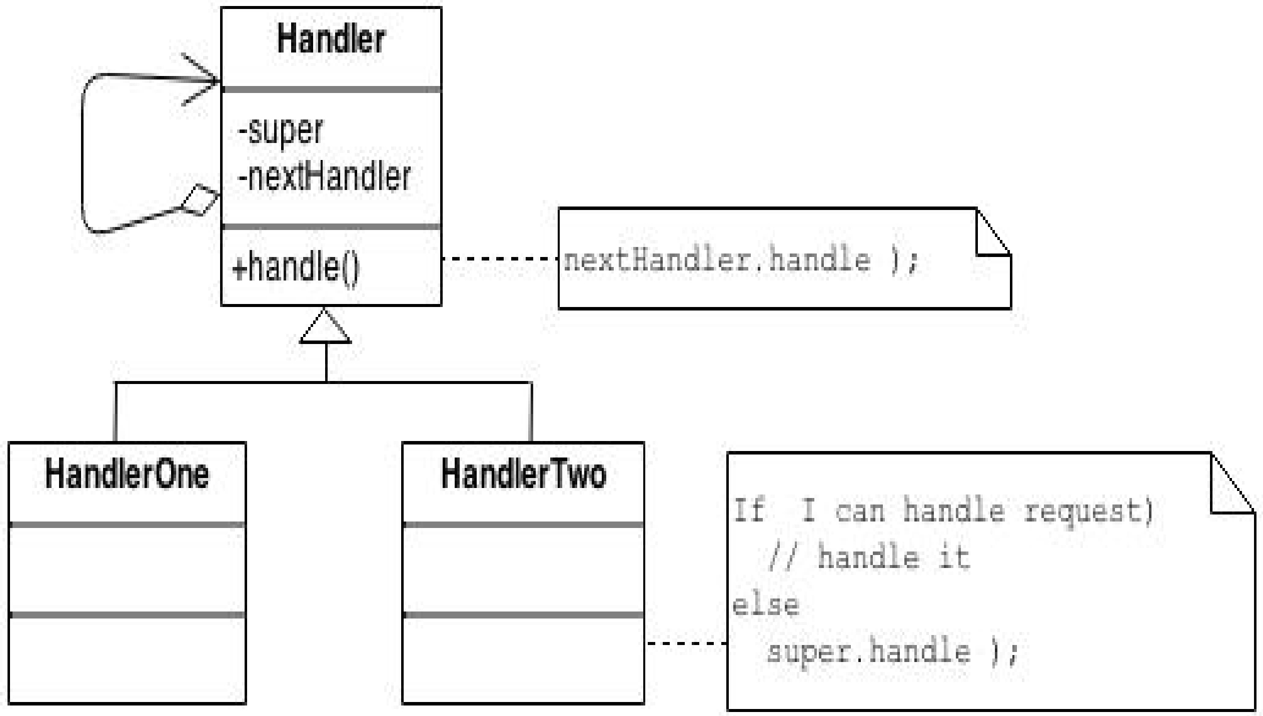
- **Chain of responsibility**
  A way of passing a request between a chain of objects
- **Command**
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program
- **Iterator**
  Sequentially access the elements of a collection
- **Mediator**
  Defines simplified communication between classes
- **Memento**
  Capture and restore an object's internal state
- **Null Object**
  Designed to act as a default value of an object
- **Observer**
  A way of notifying change to a number of classes
- **State**
  Alter an object's behavior when its state changes
- **Strategy**
  Encapsulates an algorithm inside a class
- **Template method**
  Defer the exact steps of an algorithm to a subclass
- **Visitor**
  Defines a new operation to a class without change

Chain of responsibility

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.



Processing elements are also called handlers



```
nextHandler.handle );
```

```
If  I can handle request)
  // handle it
else
  super.handle );
```

Concept of Anti patterns

- An **anti-pattern** is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive
- AntiPatterns highlight the most common problems that face the software industry and provide the tools to enable you to recognize these problems and to determine their underlying causes
- When a pattern becomes an Anti pattern, it is useful to have an approach for evolving the solution into a better one.this process of change ,migration,or evolution is called refactoring.In refactoring,we change one solution to another

- Antipatterns are present in almost any organisation and software project.
- They can be categorised into three groups

Development AntiPatterns: mainly concern the software developer

"situations encountered by the programmer when solving programming problems"

Architectural AntiPatterns:: are important for the software architect

"Common problems in system structure, their consequences and solutions"

Management Antipatterns are relevant for the software project manager

"common problems and solutions due to the software organisation"

TOPICS 2

- Unit testing and Unit Testing frameworks,
- The xUnit Architecture,
- Writing Unit Tests using at least one of Junit (for Java), unittest (for Python) or phpdbg (PHP).
- Writing tests with Assertions, defining and using Custom Assertions, single condition tests, testing for expected errors, Abstract test.

UNIT TESTING AND UNIT TESTING FRAMEWORK

What is Unit Testing?

- **UNIT TESTING** is a type of software testing where individual units or components of a software are tested.
- The purpose is to validate that each unit of the software code performs as expected.
- Unit Testing is done during the development (coding phase) of an application by the developers.
- Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

- Unit tests help to fix bugs early in the development cycle and save costs.
- It helps the developers to understand the testing code base and enables them to make changes quickly
- Unit testing can be done in the following two ways

  Manual Testing

  Automated Testing

Manual Testing	Automated Testing
Executing the test cases manually without any tool support is known as manual testing. ▪ Since test cases are executed by human resources so it is very **time consuming and tedious**. ▪ As test cases need to be executed manually so more testers are required in manual testing. ▪ It is less reliable as tests may not be performed with precision each time because of human errors. ▪ No programming can be done to write sophisticated tests which fetch hidden information.	Taking tool support and executing the test cases by using automation tool is known as automation testing. ▪ Fast Automation runs test cases significantly faster than human resources. ▪ The **investment over human resources is less** as test cases are executed by using automation tool. ▪ Automation tests perform precisely same operation each time they are run and **are more reliable**. ▪ Testers **can program sophisticated tests** to bring out hidden information.

Unit test framework

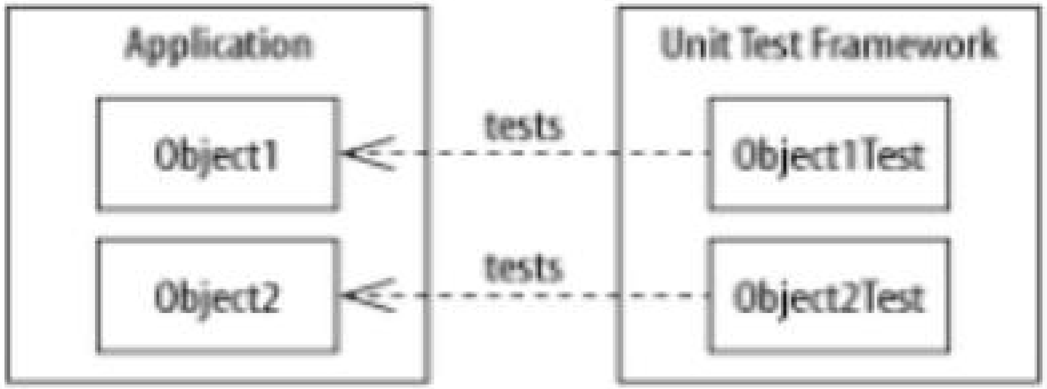
- Unit test frameworks are software tools to support writing and running unit tests, including a foundation on which to build tests and the functionality to execute the tests and report their results
- They can also be used as development tools on a par with preprocessors and debuggers.

- Unit test frameworks can contribute to almost every stage of software development, including software architecture and design, code implementation and debugging, performance optimization, and quality assurance

- Unit tests usually are developed concurrently with production code, but are not built into the final software product.

- The relationship of unit tests to production code :

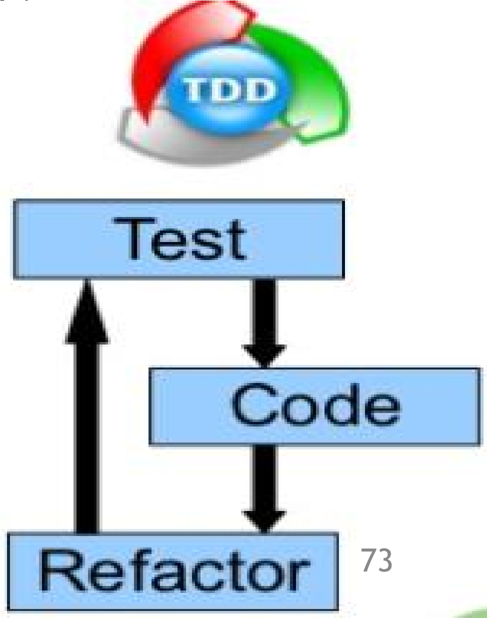
- Unit tests usually are developed concurrently with production code, but are not built into the final software product.
- The relationship of unit tests to production code :



The unit tests use the application's objects, but exist inside the unit test framework.

- Tests are categorized as *black box* or *white box*, depending on the amount of access to the internal workings.
- Black box (Functional Test)　　　　White box (Structural test)

- **<u>Test Driven Development</u>**
- Unit test frameworks are a key element of Test Driven Development (TDD), also known as "test-first programming."
- The key rule of TDD can be summarized as "test twice, code once,"
- "Test twice, code once" refers to the three-step procedure involved
- Write a test of the new code and see it fail.
- Write the new code, doing "the simplest thing that could possibly work."
- See the test succeed, and refactor the code.
- These three basic steps are the *TDD cycle* .



Examples for unit test frameworks:

- Junit,TestNG: Unit testing framework for java
- Nunit: Unit testing framework for c#
- Embunit: Unit testing framework for C and C++
- HtmlUnit, Junit,TestNG: Unit testing framework for javascript

XUnit

- **xUnit** is the collective name for several unit **testing** frameworks that derive their structure and functionality from Smalltalk's Sunit.
- Free and open source software

xUnit Family Members

Junit: Java

CppUnit: C++

Nunit:. NET

PyUnit: python

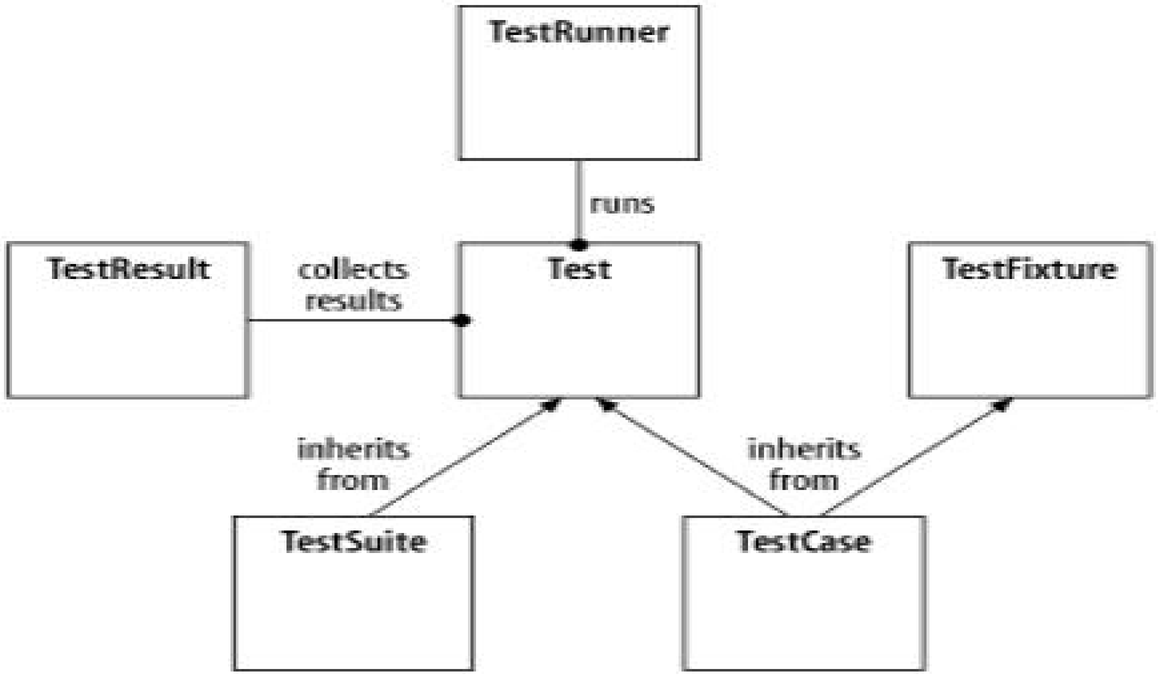
Sunit: also known as smalltalk unit, basis of xunit architecture

vbUnit: Visual Basic

utPLSQL: oracle's PL/SQL

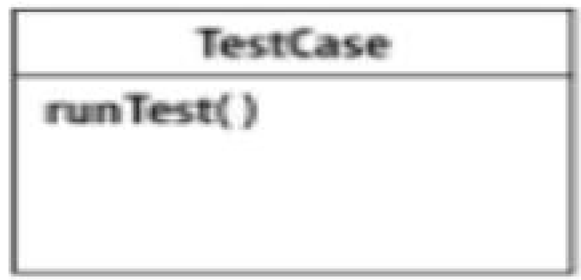
Xunit Architecture

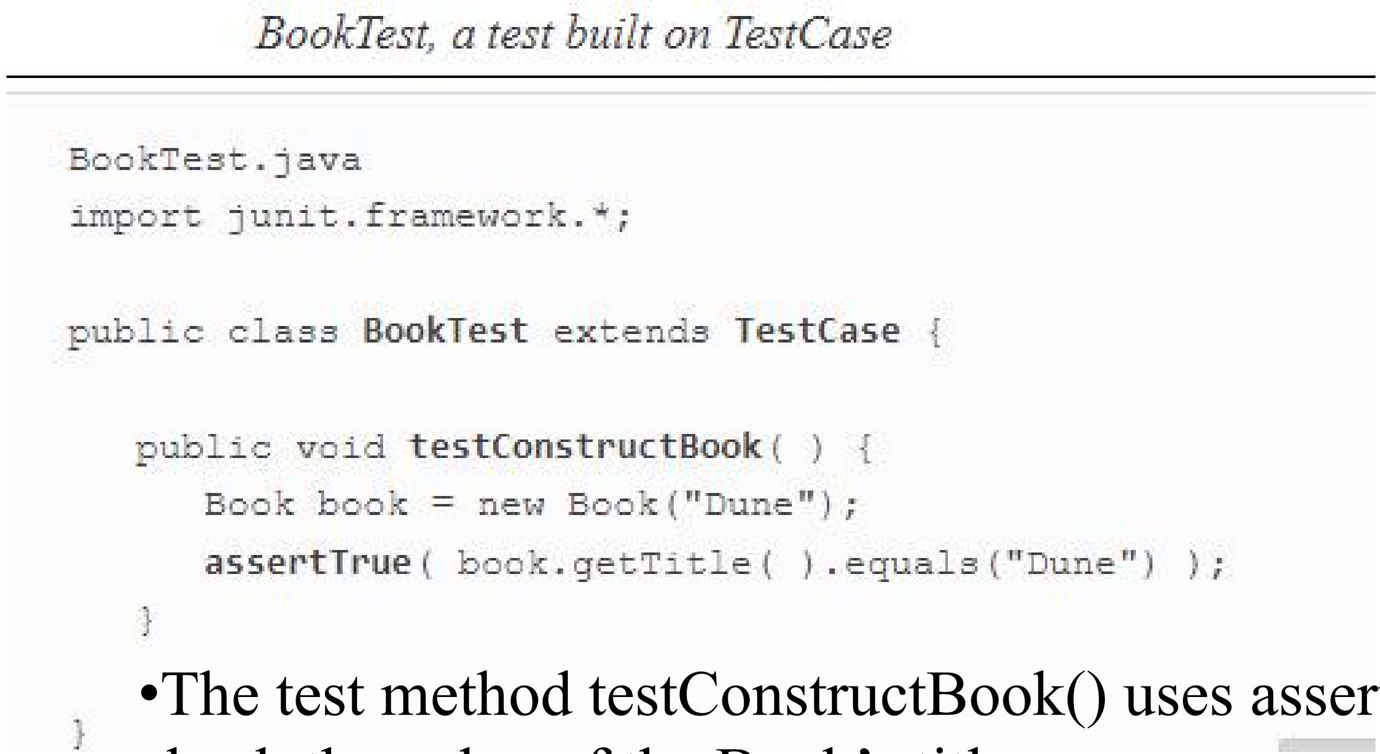
- The xUnits all have the same basic architecture
- The key classes are TestCase, TestRunner TestFixture, TestSuite and



- **<u>TestCase</u>**

  This is the smallest unit of testing. This checks for a specific response to a particular set of inputs. Unittest provides a base class, **TestCase**, which may be used to create new test cases
- All unit tests are inherited from TestCase.
- To create a unit test, define a test class that is descended from TestCase and add a test method to it



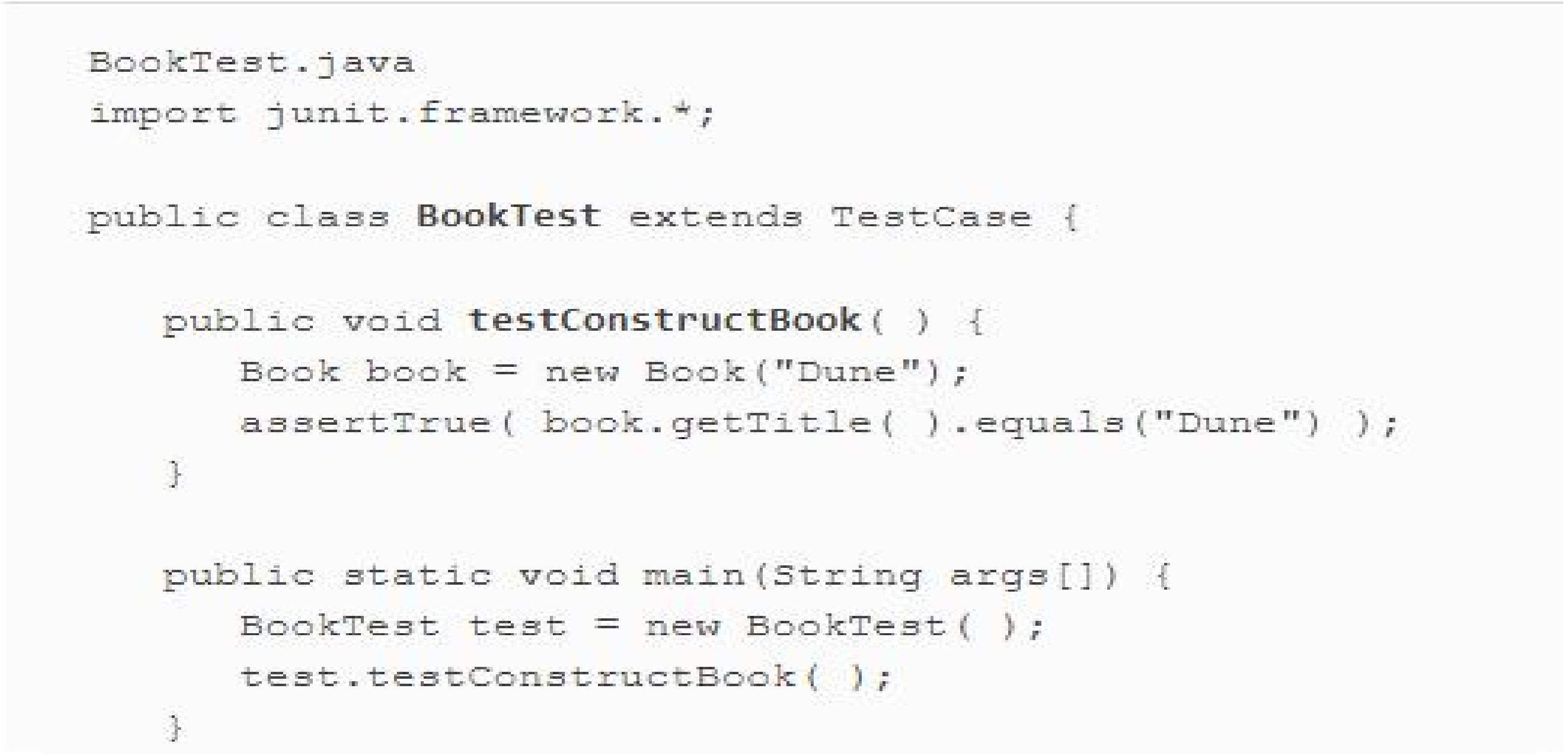


BookTest, a test built on TestCase

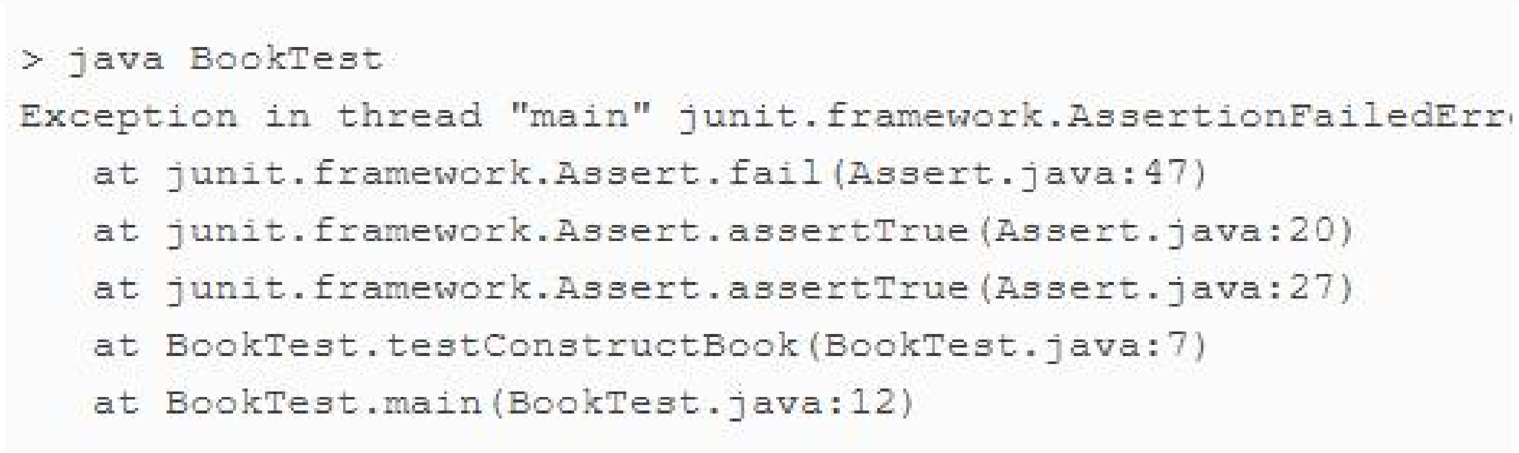
```
BookTest.java
import junit.framework.*;

public class BookTest extends TestCase {

   public void testConstructBook( ) {
      Book book = new Book("Dune");
      assertTrue( book.getTitle( ).equals("Dune") );
   }

}
```

- The test method testConstructBook() uses assertTrue() to check the value of the Book's title
- Test conditions always are evaluated by the framework's assert methods.
- If a condition evaluates to TRUE, the framework increments the successful test counter.
- If it is FALSE, a test failure has occurred and the framework records the details, including the failure's location in the code. After a failure, the framework skips the rest of the code in the test method

BookTest with changes allowing it to be run



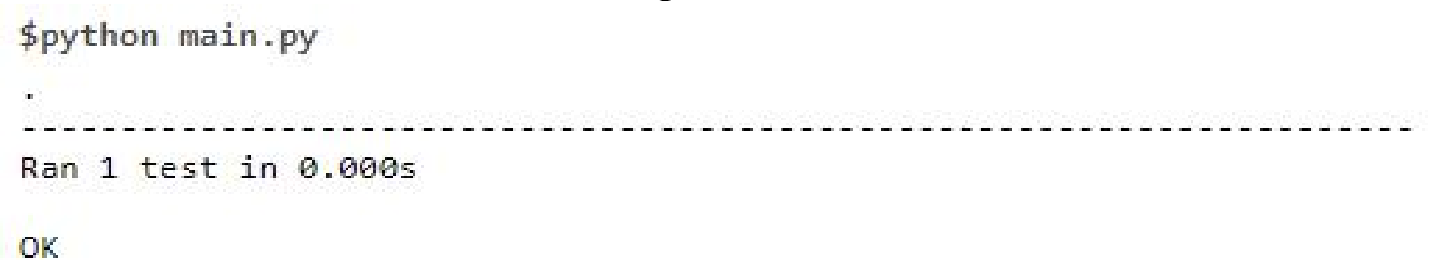
```
BookTest.java
import junit.framework.*;

public class BookTest extends TestCase {

    public void testConstructBook( ) {
        Book book = new Book("Dune");
        assertTrue( book.getTitle( ).equals("Dune") );
    }

    public static void main(String args[]) {
        BookTest test = new BookTest( );
        test.testConstructBook( );
    }
```



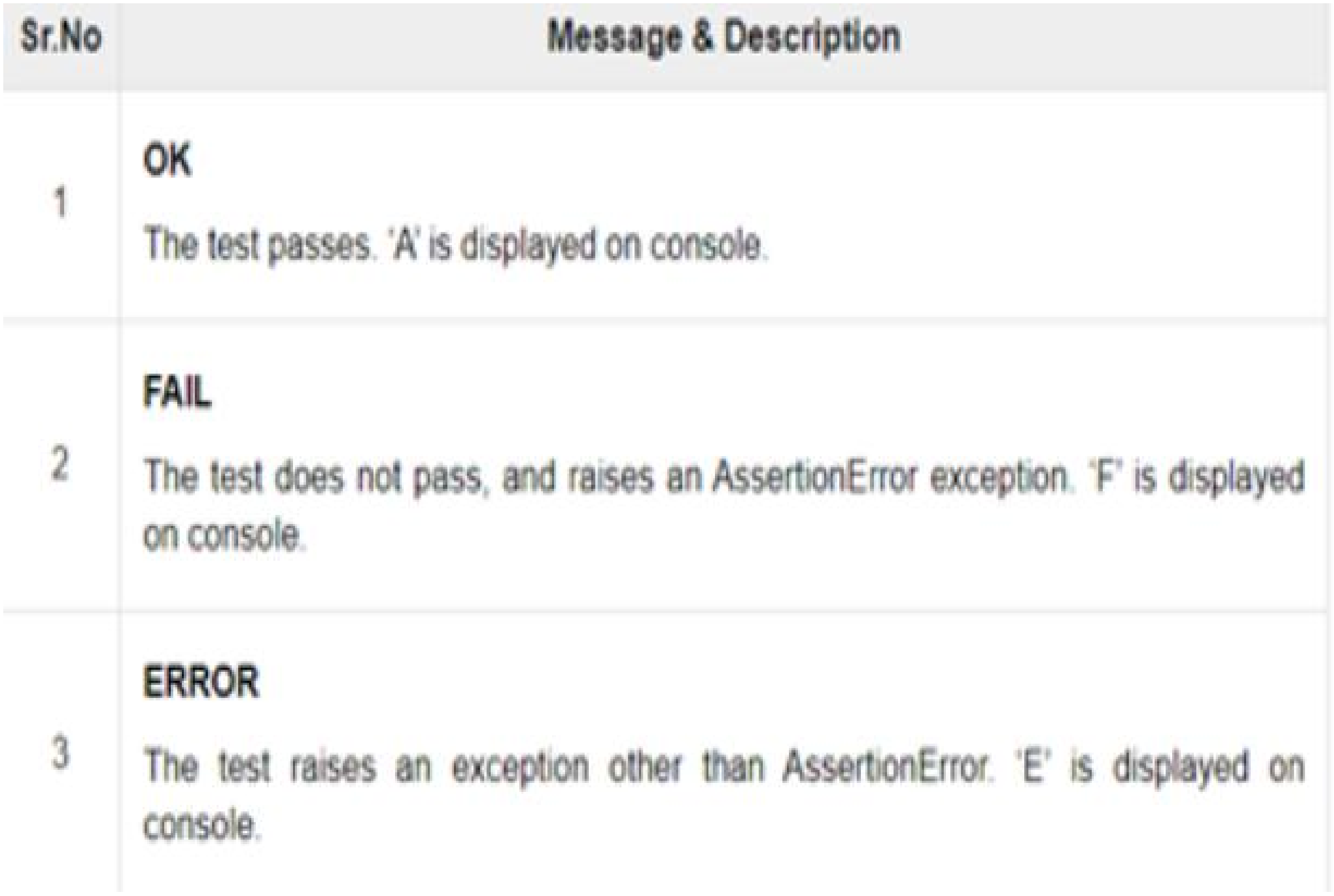
```
> java BookTest
Exception in thread "main" junit.framework.AssertionFailedErr
   at junit.framework.Assert.fail(Assert.java:47)
   at junit.framework.Assert.assertTrue(Assert.java:20)
   at junit.framework.Assert.assertTrue(Assert.java:27)
   at BookTest.testConstructBook(BookTest.java:7)
   at BookTest.main(BookTest.java:12)
```

TestRunner

- This is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.



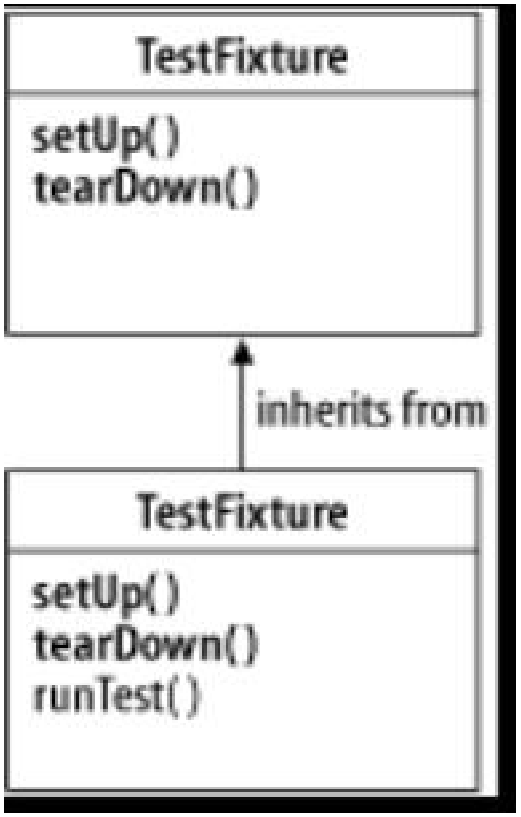
```
$python main.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```



Sr.No	Message & Description
1	**OK** The test passes. 'A' is displayed on console.
2	**FAIL** The test does not pass, and raises an AssertionError exception. 'F' is displayed on console.
3	**ERROR** The test raises an exception other than AssertionError. 'E' is displayed on console.

TESTFIXTURE

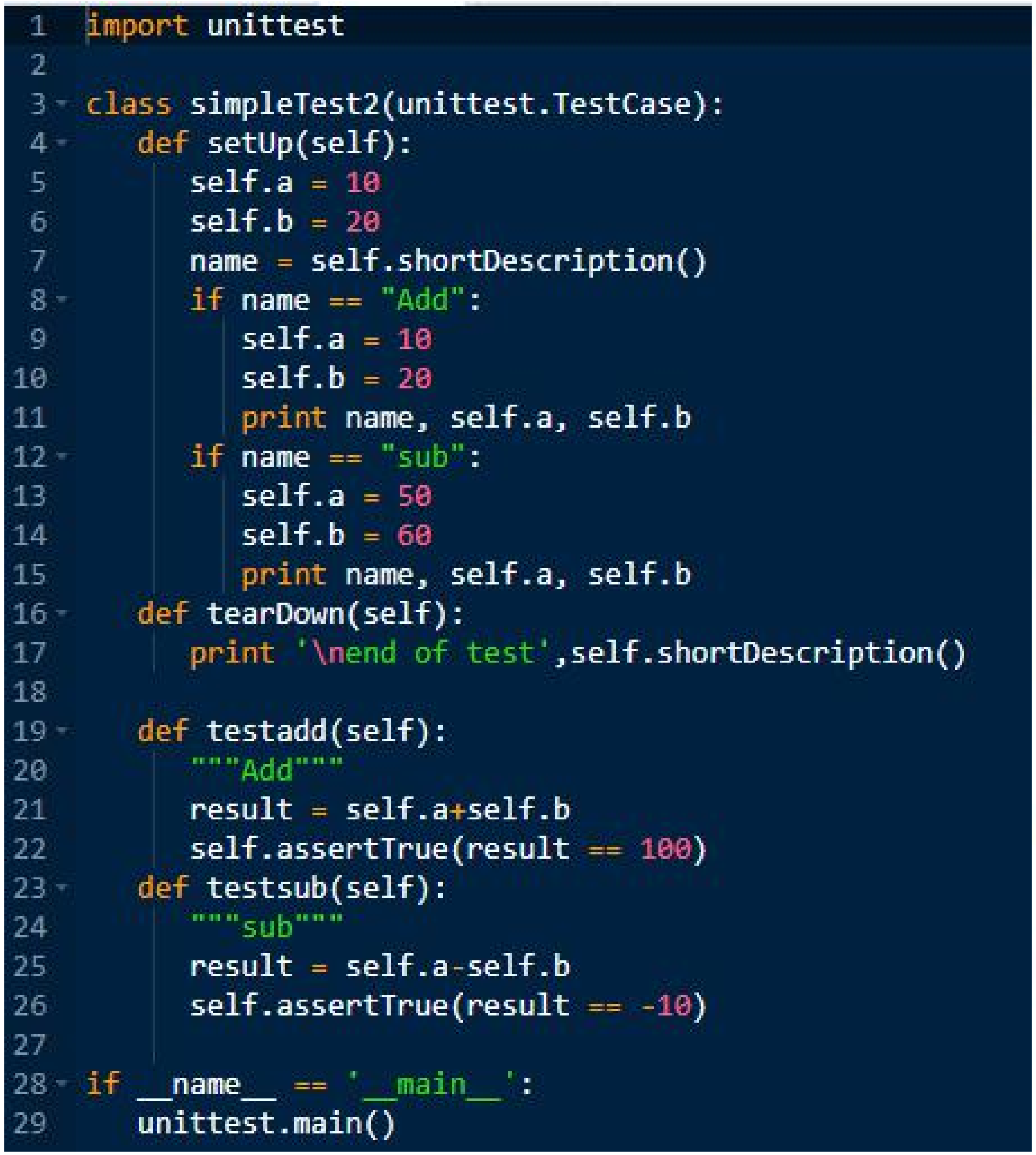
- There can be numerous tests written inside a TestCase class.
- These test methods may need database connection, temporary files or other resources to be initialized. These are called fixtures.
- TestCase includes a special hook to configure and clean up any fixtures needed by your tests. To configure the fixtures, override **setUp().** To clean up, override **tearDown()**.
- **The setUp( ) method** is called prior to each test method, establishing the initial environment for the test.
- **The tearDown( ) method** is always called after each test method to clean up the test environment, even if there is a failure.



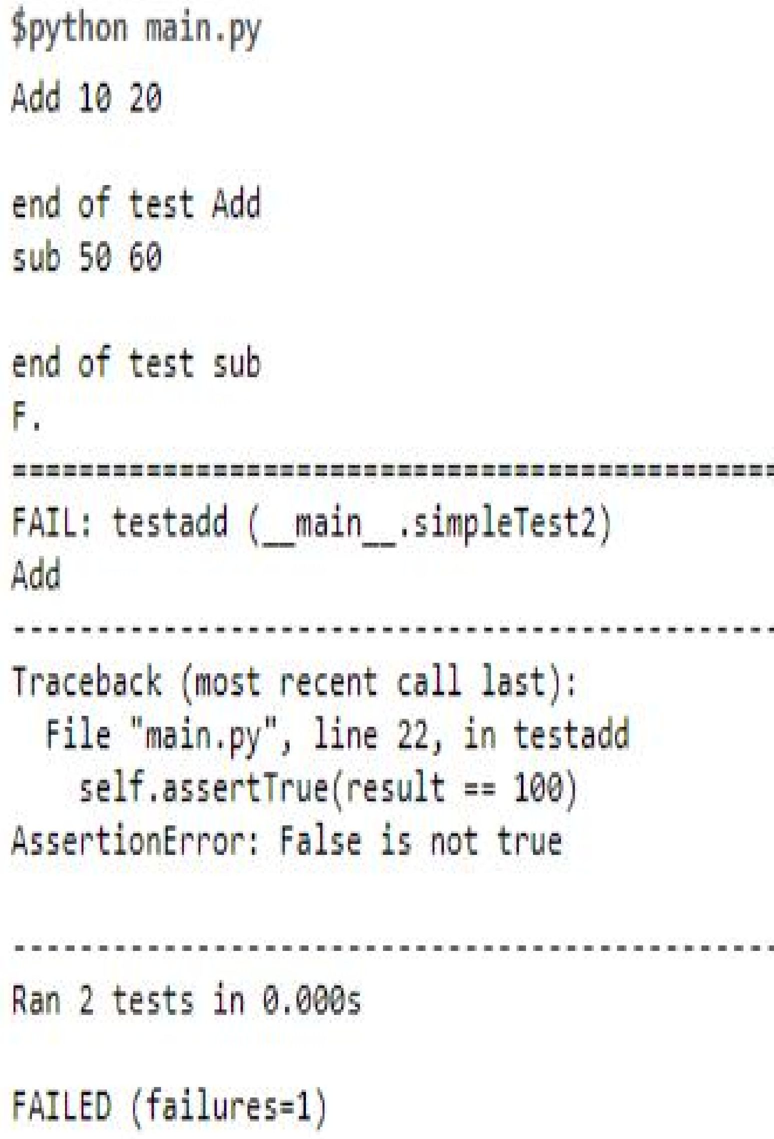
• The TestFixture behavior comes into play when multiple test methods have objects in common
•Although the tests use the same objects, they can make changes without the possibility of affecting the next test.

Advantages

•Test methods can share objects but still run in isolation
•Test coupling is minimized.
•Code duplication between tests is reduced
•the test methods can be run in any order, since they are isolated



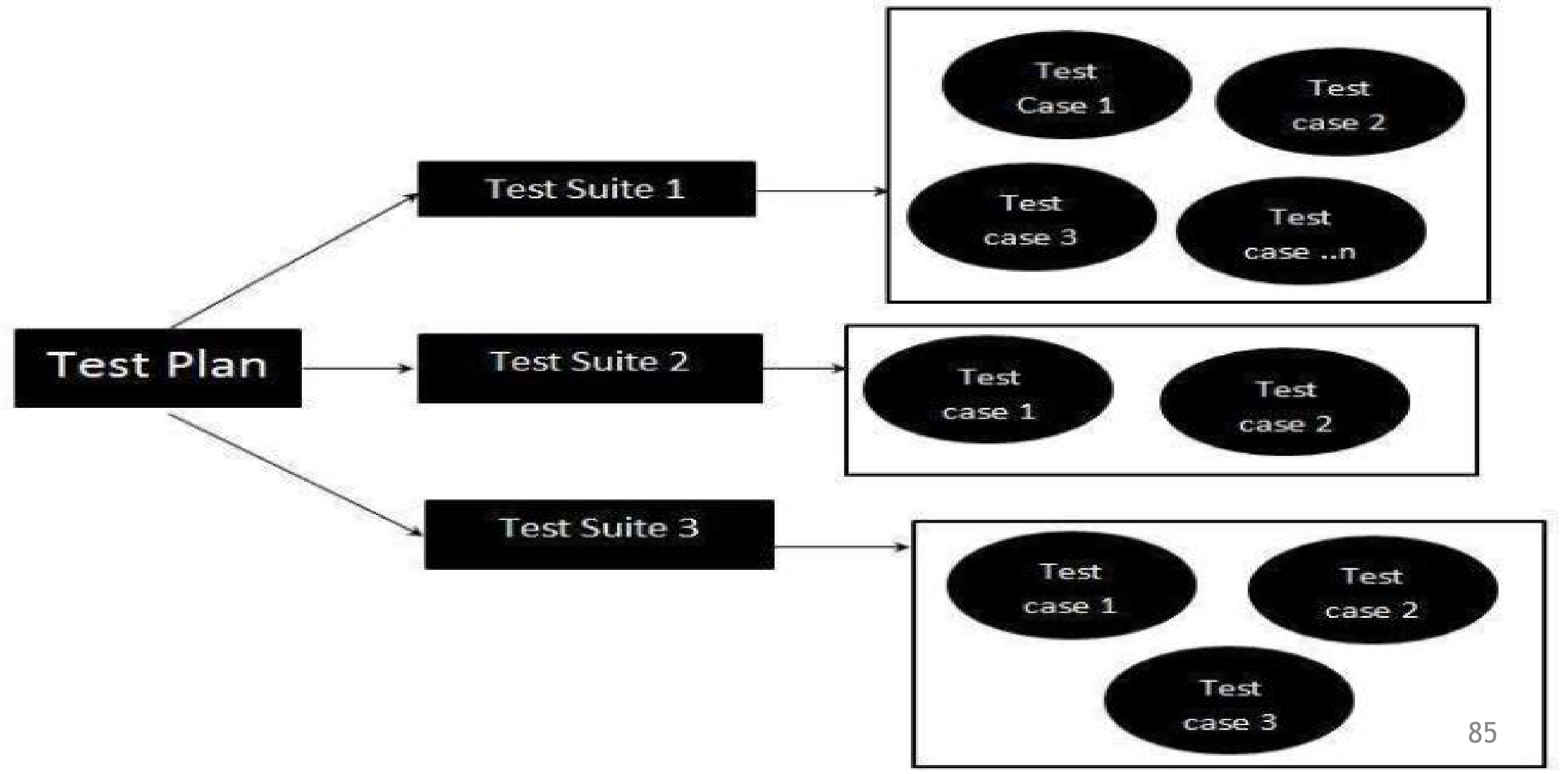
```python
import unittest

class simpleTest2(unittest.TestCase):
    def setUp(self):
        self.a = 10
        self.b = 20
        name = self.shortDescription()
        if name == "Add":
            self.a = 10
            self.b = 20
            print name, self.a, self.b
        if name == "sub":
            self.a = 50
            self.b = 60
            print name, self.a, self.b
    def tearDown(self):
        print '\nend of test',self.shortDescription()

    def testadd(self):
        """Add"""
        result = self.a+self.b
        self.assertTrue(result == 100)
    def testsub(self):
        """sub"""
        result = self.a-self.b
        self.assertTrue(result == -10)

if __name__ == '__main__':
    unittest.main()
```



```
$python main.py
Add 10 20

end of test Add
sub 50 60

end of test sub
F.
==================================================
FAIL: testadd (__main__.simpleTest2)
Add
--------------------------------------------------
Traceback (most recent call last):
  File "main.py", line 22, in testadd
    self.assertTrue(result == 100)
AssertionError: False is not true


--------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

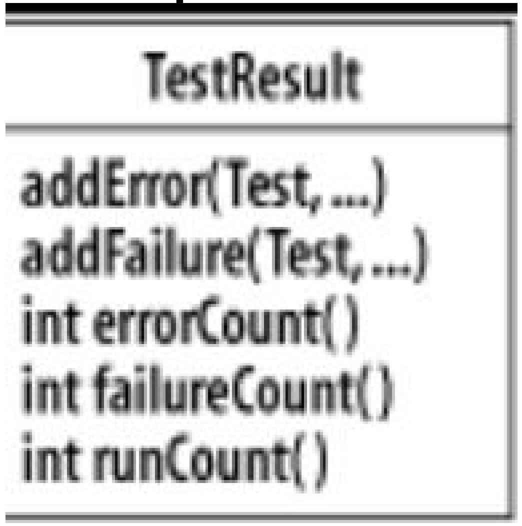
Testsuit e

- Xunit contains a class for aggregating unit tests called TESTSUITE
- Test suite is a container that has a set of tests which helps testers in executing and reporting the test execution status.
- It can take any of the three states namely Active,



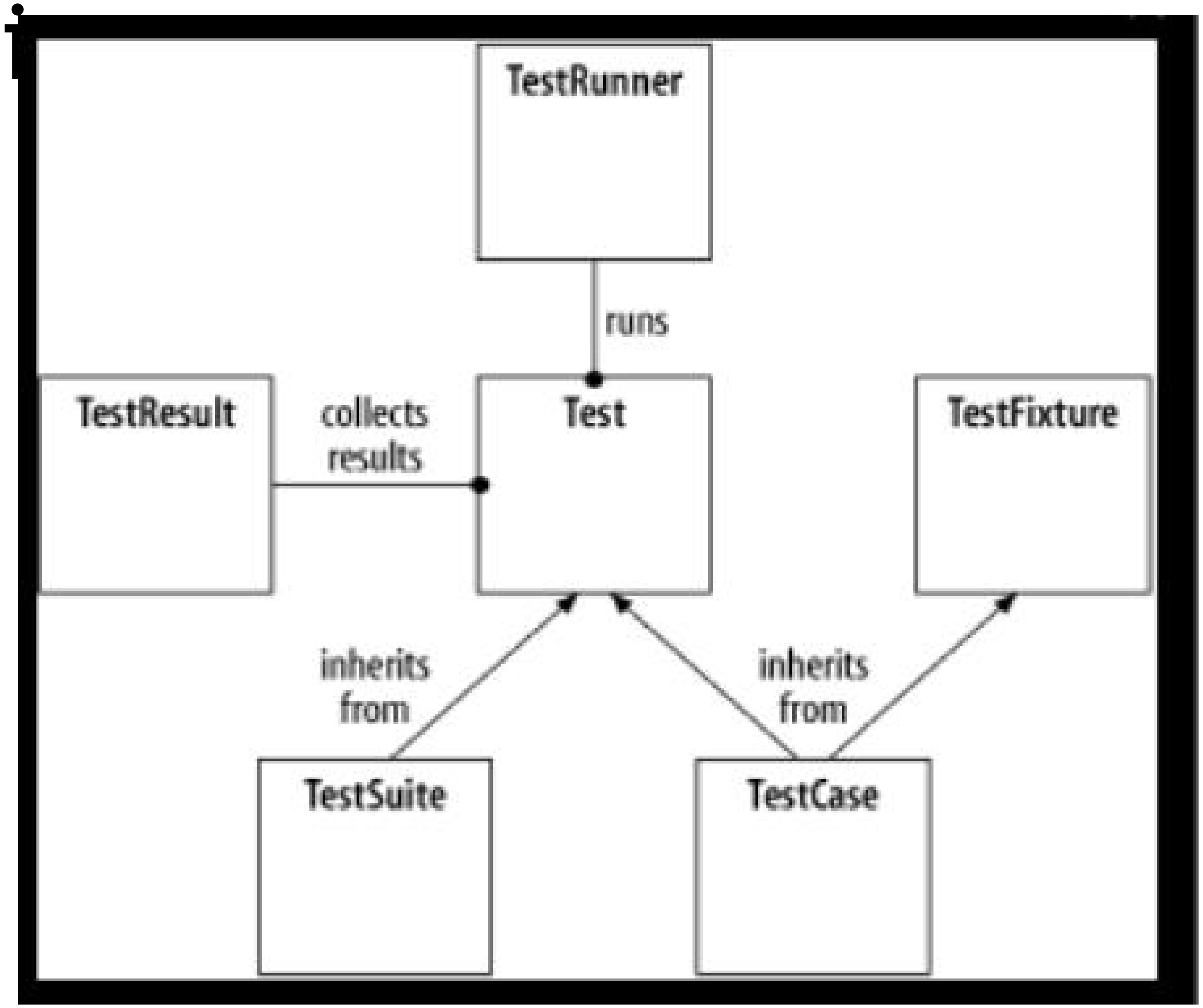
Test result

- A test case result is a record of the outcome of a test case.
- Teams often execute a test case multiple times during the development cycle with the goal of getting the test case to a passing state.
- We can track and document test case results associated with respective test cases.



- TestResult is a simple object.
- It counts the tests run and collects test failures and errors so the framework can report them.
- The failures and errors include details about the location in the code where they occurred and any associated test descriptions

Overall vi


TestRunner
runs
TestResult
collects results
Test
TestFixture
inherits from
inherits from
TestSuite
TestCase

What is Assertion?

- An assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program. A test assertion is defined as an expression, which encapsulates some testable logic specified about a target under test.
- If the assertion fails, an Assertion Error will be raised
- Plain Assertions are the most generic type of test assertion, which take a Boolean condition that must evaluate to TRUE for the test to succeed
- Some of the assertion functions are

  Basic Boolean Asserts

  Comparative Asserts

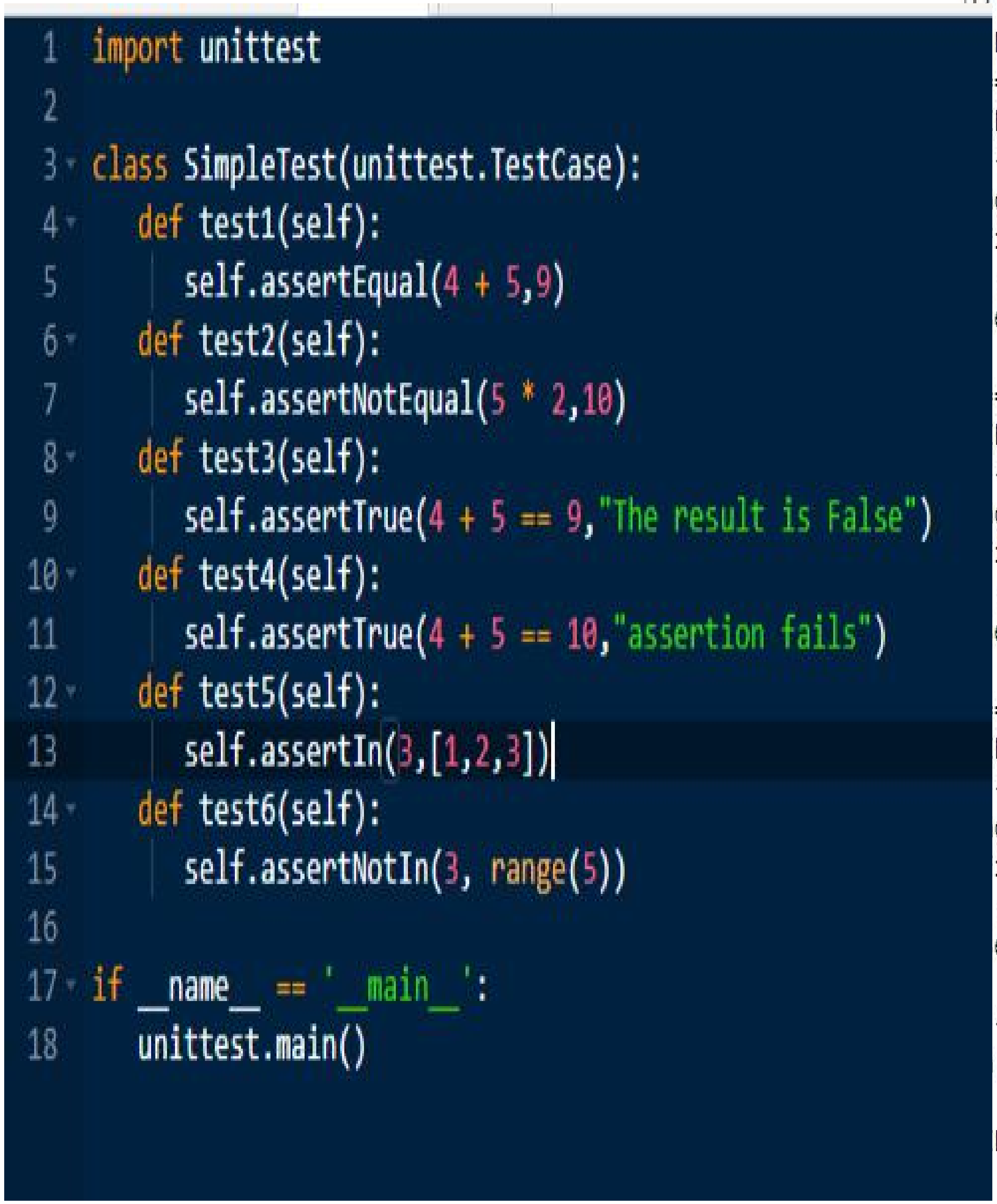
Basic Boolean Asserts

- Basic assert functions evaluate whether the result of an operation is True or False. All the assert methods accept a **msg** argument that, if specified, is used as the error message on failure
- Eg:**assertEqual(arg1, arg2, msg = None)**

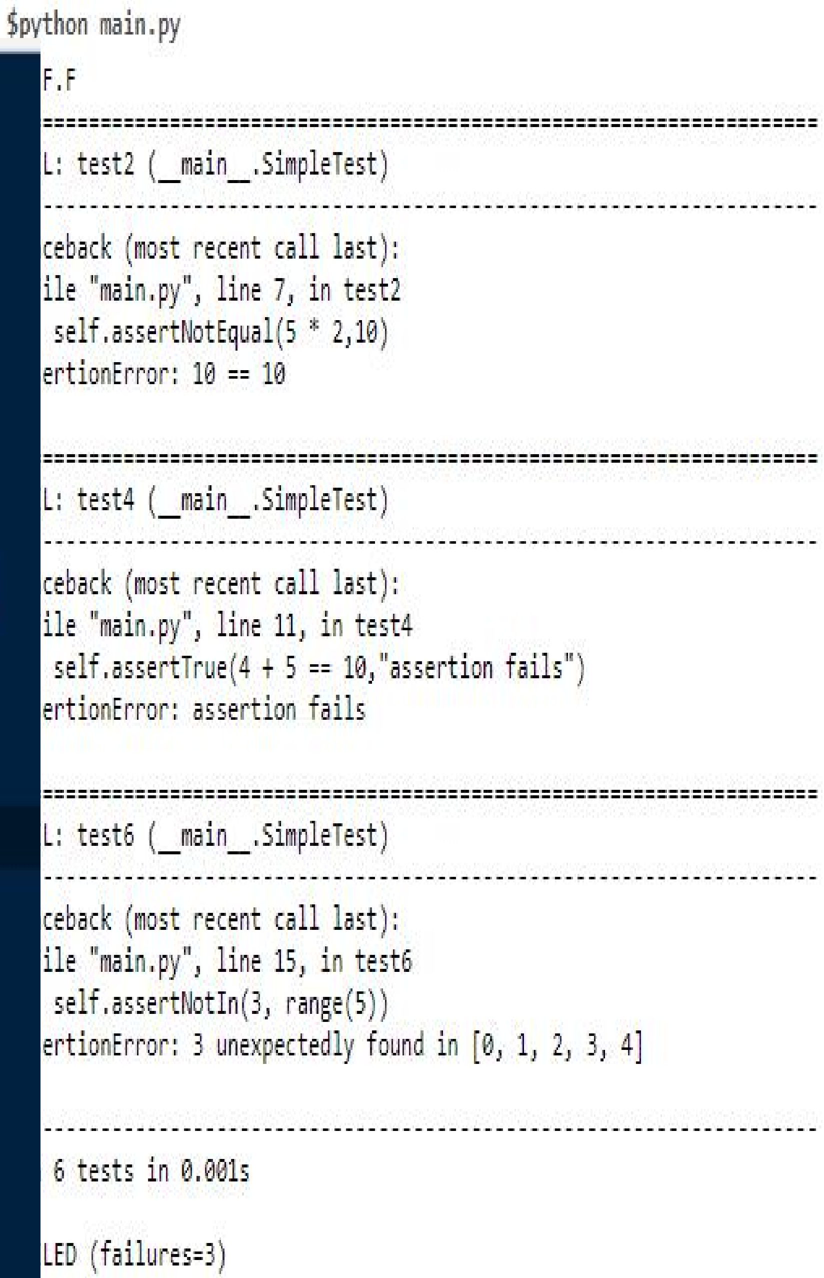
  Test that *arg1* and *arg2* are equal. If the values do not compare equal, the test will fail.
- **assertNotEqual(arg1, arg2, msg = None)**

  Test that *arg1* and *arg2* are not equal. If the values do compare equal, the test will fail.

- **assertTrue(expr, msg = None)**
  Test that *expr* is true. If false, test fails
- **assertFalse(expr, msg = None)**
  Test that *expr* is false. If true, test fails
- **assertIn(arg1, arg2, msg = None)**
  Test that *arg1* is in *arg2*.
- **assertNotIn(arg1, arg2, msg = None)**
  Test that *arg1* is not in *arg2*.



```python
import unittest

class SimpleTest(unittest.TestCase):
    def test1(self):
        self.assertEqual(4 + 5,9)
    def test2(self):
        self.assertNotEqual(5 * 2,10)
    def test3(self):
        self.assertTrue(4 + 5 == 9,"The result is False")
    def test4(self):
        self.assertTrue(4 + 5 == 10,"assertion fails")
    def test5(self):
        self.assertIn(3,[1,2,3])
    def test6(self):
        self.assertNotIn(3, range(5))

if __name__ == '__main__':
    unittest.main()
```



```
$python main.py
F.F
======================================================================
L: test2 (__main__.SimpleTest)
----------------------------------------------------------------------
ceback (most recent call last):
ile "main.py", line 7, in test2
 self.assertNotEqual(5 * 2,10)
ertionError: 10 == 10

======================================================================
L: test4 (__main__.SimpleTest)
----------------------------------------------------------------------
ceback (most recent call last):
ile "main.py", line 11, in test4
 self.assertTrue(4 + 5 == 10,"assertion fails")
ertionError: assertion fails

======================================================================
L: test6 (__main__.SimpleTest)
----------------------------------------------------------------------
ceback (most recent call last):
ile "main.py", line 15, in test6
 self.assertNotIn(3, range(5))
ertionError: 3 unexpectedly found in [0, 1, 2, 3, 4]

----------------------------------------------------------------------
 6 tests in 0.001s

LED (failures=3)
```

Comparative Asserts

- **assertGreater** (first, second, msg = None)

  Test that *first* is greater than *second* depending on the method name. If not, the test will fail.
- **assertGreaterEqual** (first, second, msg = None)

  Test that *first* is greater than or equal to *second* depending on the method name. If not, the test will fail
- **assertLess** (first, second, msg = None)

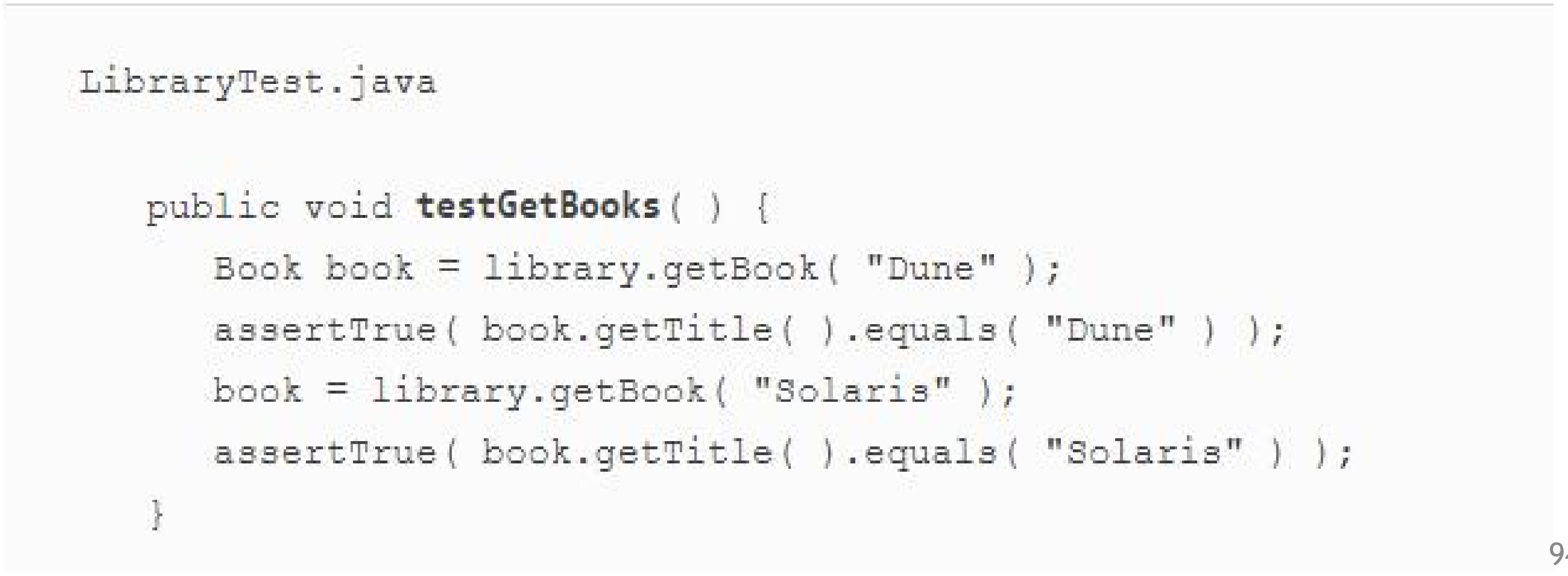
  Test that *first* is less than *second* depending on the method name. If not, the test will fail
- **assertLessEqual** (first, second, msg = None)

  Test that *first* is less than or equal to *second* depending upon the method name. If not, the test will fail.

CUSTOM ASSERTION

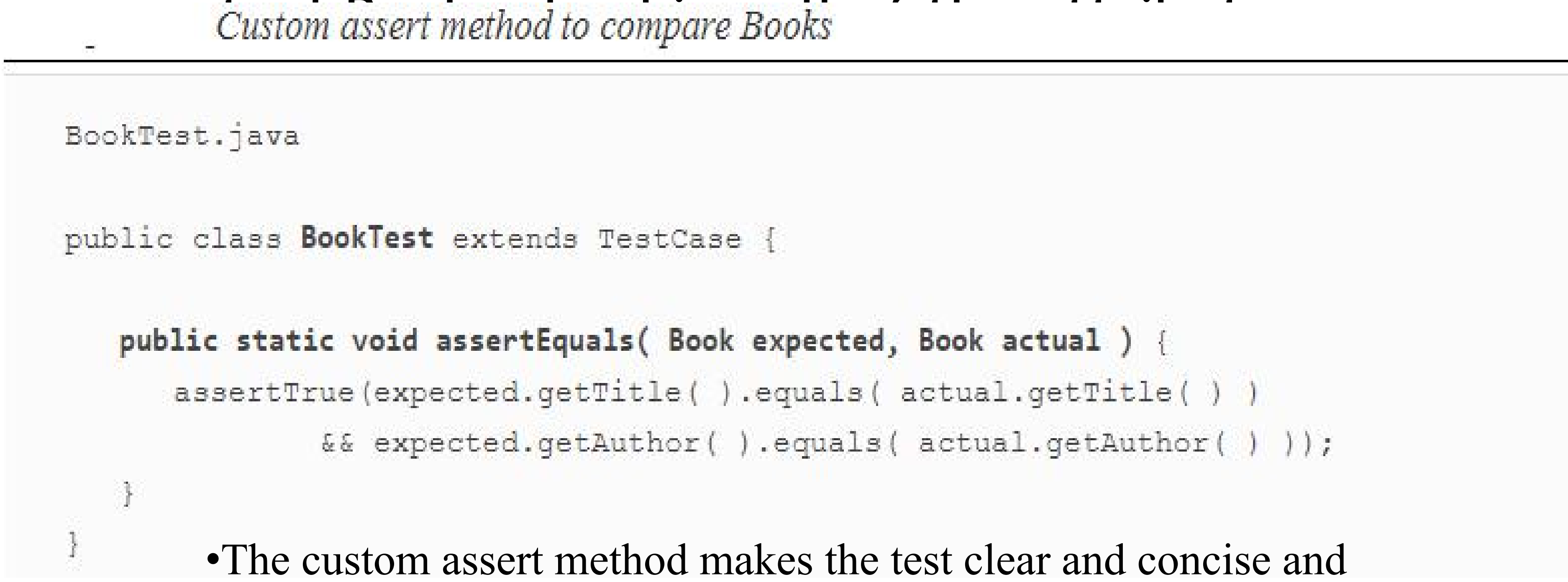
- The basic assert methods cover only a few common cases.
- It’s often useful to extend them to cover additional test conditions and data types.
- Custom assert methods save test coding effort and make the test code more readable.

Test comparing two Books using their title attributes



```
LibraryTest.java

  public void testGetBooks( ) {
      Book book = library.getBook( "Dune" );
      assertTrue( book.getTitle( ).equals( "Dune" ) );
      book = library.getBook( "Solaris" );
      assertTrue( book.getTitle( ).equals( "Solaris" ) );
  }
```

- To be really sure that the test Book is correct, the tests should also check the Book's author, but this means adding extra asserts to each test.
- It's clearly useful to have an assert method that compares an expected Book to the

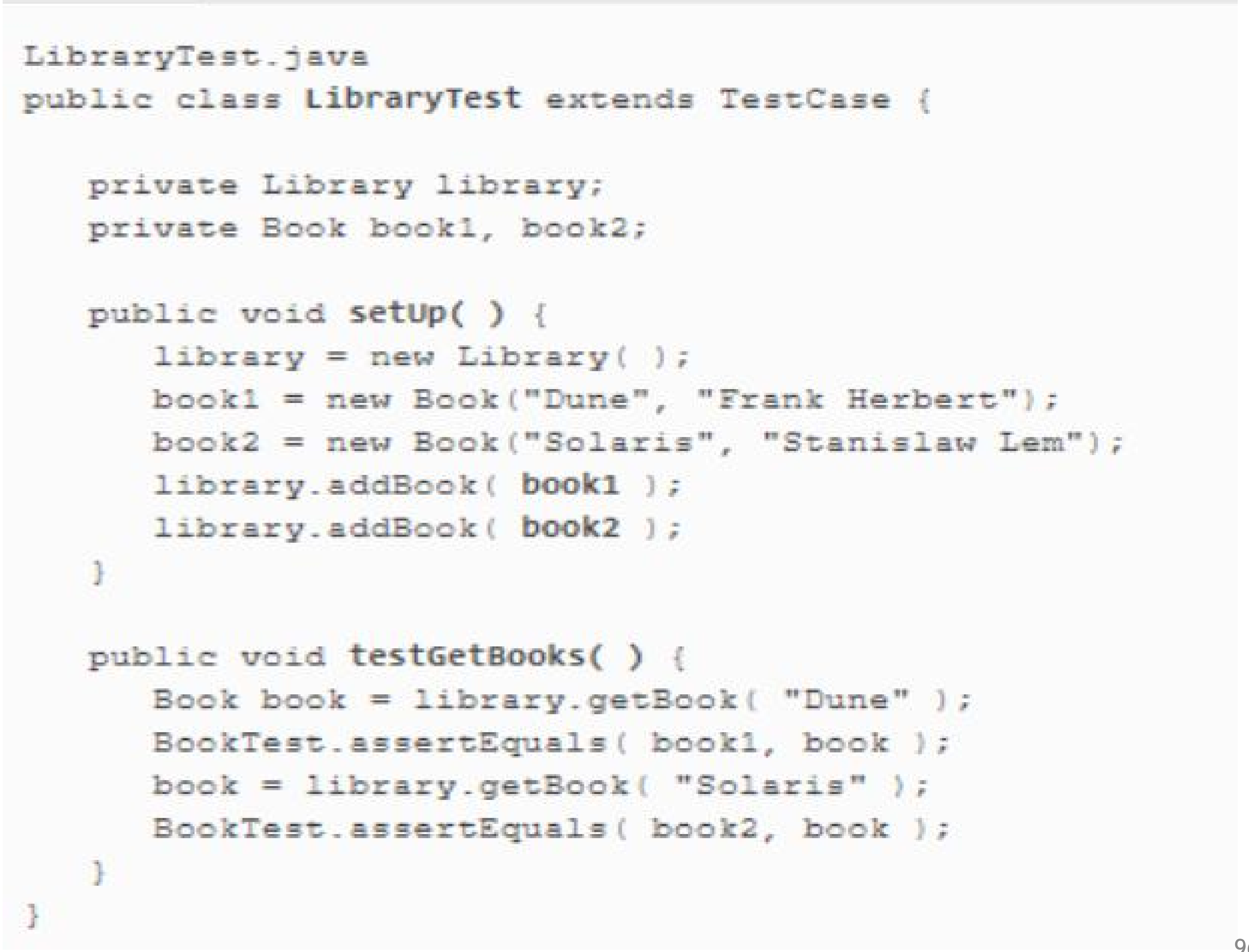


Custom assert method to compare Books

```
BookTest.java

public class BookTest extends TestCase {

  public static void assertEquals( Book expected, Book actual ) {
     assertTrue(expected.getTitle( ).equals( actual.getTitle( ) )
            && expected.getAuthor( ).equals( actual.getAuthor( ) ));
  }
}
```

- The custom assert method makes the test clear and concise and improves it by comparing all the Book attributes, not just the title

Using the custom assert method



```
LibraryTest.java
public class LibraryTest extends TestCase {

    private Library library;
    private Book book1, book2;

    public void setUp( ) {
        library = new Library( );
        book1 = new Book("Dune", "Frank Herbert");
        book2 = new Book("Solaris", "Stanislaw Lem");
        library.addBook( book1 );
        library.addBook( book2 );
    }

    public void testGetBooks( ) {
        Book book = library.getBook( "Dune" );
        BookTest.assertEquals( book1, book );
        book = library.getBook( "Solaris" );
        BookTest.assertEquals( book2, book );
    }
}
```

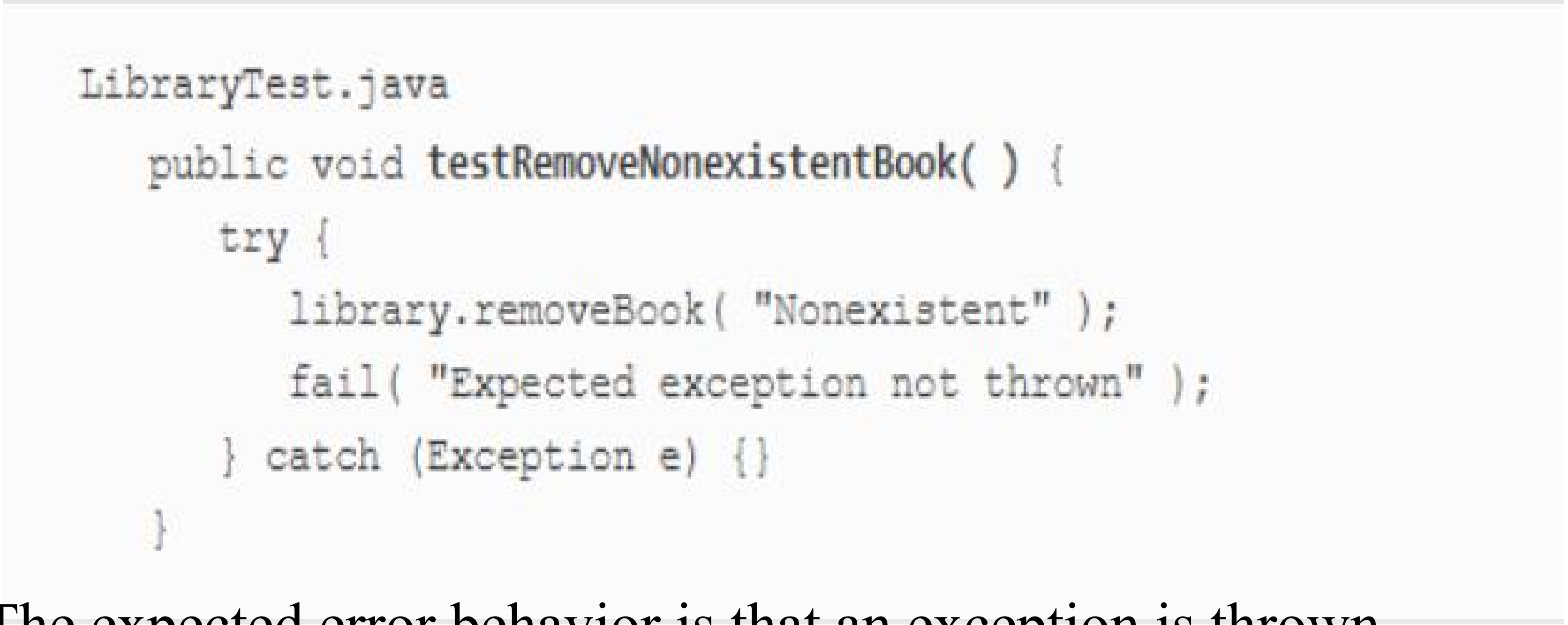
Single Condition Tests

- A useful rule of thumb is that a test method should only contain a single test assertion.
- The idea is that a test method should only test one behavior.
- If there is more than one assert condition, multiple things are being tested.
- When there is more than one condition to test, then a test fixture should be set up, and each condition placed in a separate test method.

Testing Expected Errors

- It is important to test the error-handling behavior of production code in addition to its normal behavior.
- Such tests generate an error and assert that the error is handled as expected.
- In other words, an expected error produces a unit test success.

Unit test for expected exception



```
LibraryTest.java
   public void testRemoveNonexistentBook( ) {
      try {
         library.removeBook( "Nonexistent" );
         fail( "Expected exception not thrown" );
      } catch (Exception e) {}
   }
```

•The expected error behavior is that an exception is thrown when the removeBook( ) method is called for a nonexistent Book.

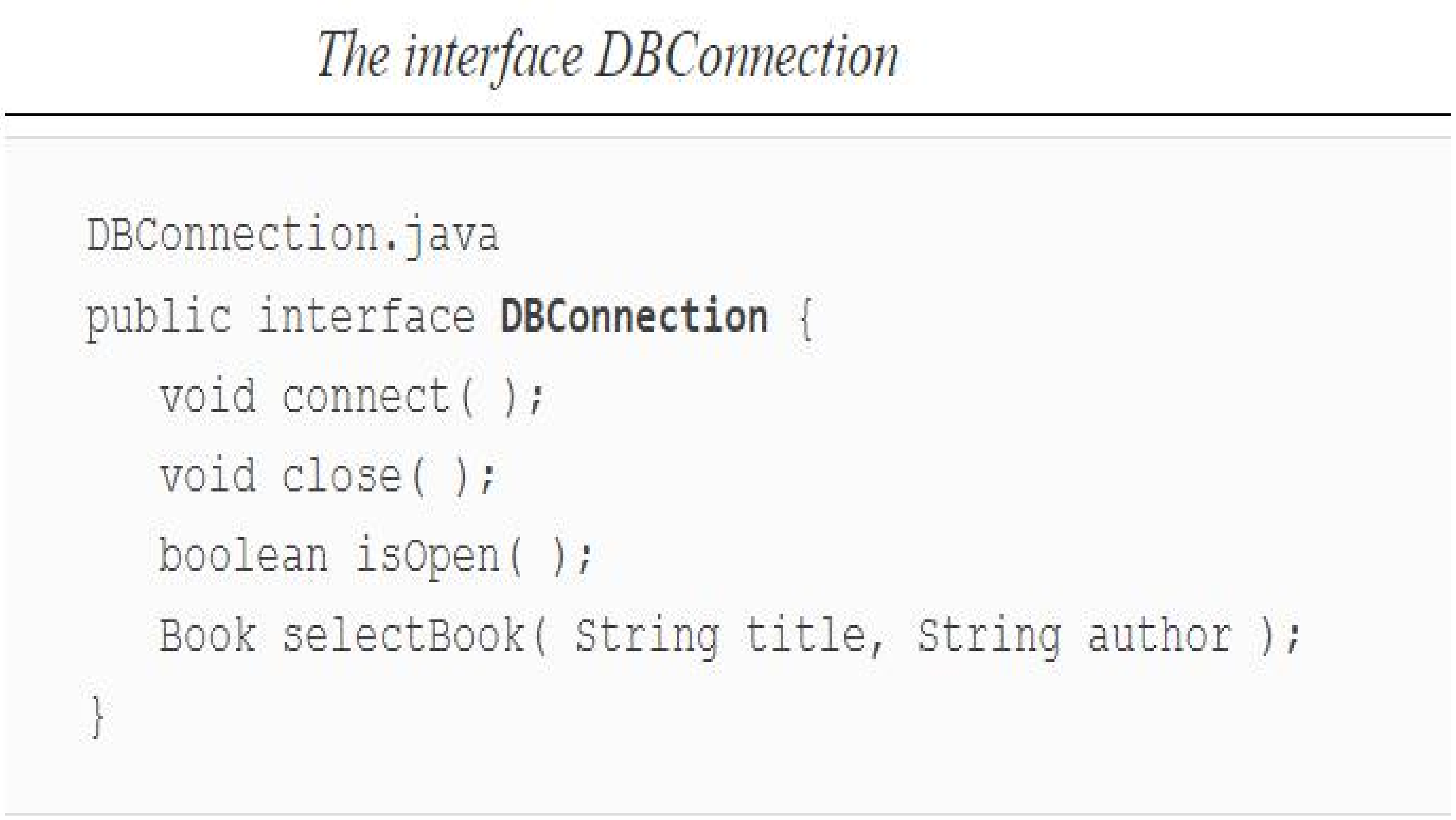
• If the exception is thrown, the unit test succeeds. If it is not thrown, fail() is called.

•The fail( ) method is another useful variation on the basic assert method. It is equivalent to assertTrue(false), but it reads better.

ABSTRACT TEST

- An Abstract TestCase is a TestCase for an AbstractClass that ensures that concrete implementations of the abstract class behave as expected.
- The Abstract TestCase will have abstract methods to enable it to obtain concrete subclasses of the Abstract class under test, to obtain appropriate arguments for tests and expected results.

Create an AbstractTest for the interface DBConnection



The interface DBConnection

```java
DBConnection.java
public interface DBConnection {
   void connect( );
   void close( );
   boolean isOpen( );
   Book selectBook( String title, String author );
}
```

The AbstractTest class AbstractDBConnectionTestCase

```
AbstractDBConnectionTestCase.java
import junit.framework.*;

public abstract class AbstractDBConnectionTestCase extends TestCase {

   public abstract DBConnection getConnection( );

   public void testIsOpen( ) {
      DBConnection connection = getConnection( );
      connection.connect( );
      assertTrue( connection.isOpen( ) );
   }

   public void testClose( ) {
      DBConnection connection = getConnection( );
      connection.connect( );
      connection.close( );
      assertTrue( !connection.isOpen( ) );
   }
}
```