OOP And SOLID Principles

OOP Principles

Object Oriented Programming Concepts

- Object-oriented approach eliminates some of the pitfalls that exist in the procedural approach
- Treats data as an element in the program, not allowing it to flow around the system freely
- Ties data closely to the functions that operate on it
- Protects it from unintentional modification by other existing functions
- Allows breaking the problem into different entities called objects modeled after real world objects

Features of OOP

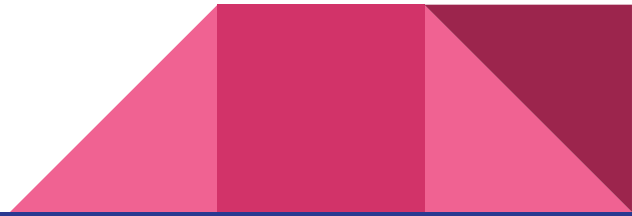
- Emphasis is on data and not on procedures
- Programs are divided into Objects
- Data structures are created to characterize objects
- Data is hidden, and external functions cannot access it
- Objects communicate with each other through methods

Classes and Objects

- Objects are runtime entities
- Any problem is analyzed and represented in terms of objects and interactions among them
- Objects communicate with each other by sending messages to one another
- Objects consist of data that represents its state and methods to manipulate the state

Encapsulation

- The wrapping up of the data and methods into the single unit
- The data is accessible only to those methods, which are wrapped in the class
- This insulation of data from being modified directly by the program is called data hiding
- Makes it possible to treat objects like black boxes
- Plan overall solution using objects without worrying about internal implementation

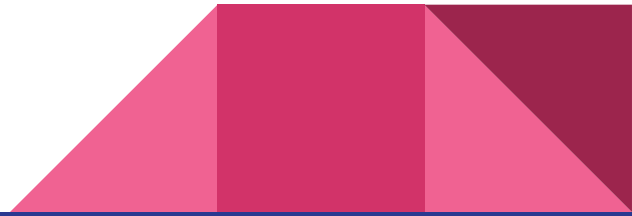


Abstraction

- The act of reducing programming complexity by representing essential features without including the background explanations or details
- An abstract class is a class with an abstract keyword.
- An abstract method is a method declared without a method body.
- An abstract class may not have all the abstract methods. Some methods are concrete.
- A method defined abstract must have its implementation in the derived class, thus making method overriding compulsory. Making a subclass abstract avoids overriding.

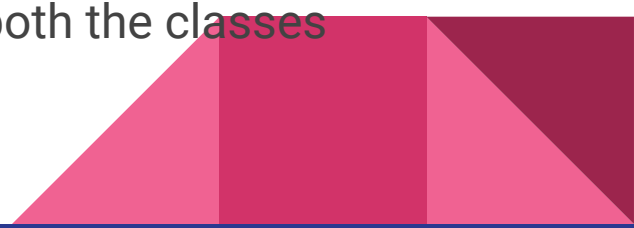
Abstraction (Cont.)

- Classes that contain abstract method(s) must be declared with abstract keyword.
- The object for an abstract class cannot be instantiated with the new operator.
- An abstract class can have parameterized constructors.
- However they can only be used in the subclass using super keyword
- Ways to achieve abstraction
    - Using abstract keyword
    - Using interface



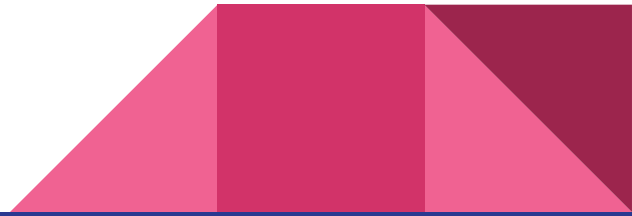
Inheritance

- The process by which objects of one class acquire some properties of objects of another class
- Each subclass shares common characteristics from its parent class
- Provides reusability
- It means that we can add additional features to parent class without modification
- New class consists of the combined features from both the classes



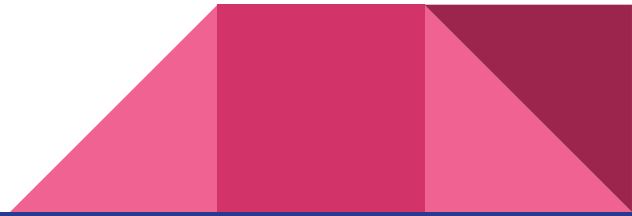
Polymorphism

- Ability to take more than one form
- Operation or method being able to exhibit different behaviors in different situations
- Objects having different internal structures share common external interface
- Achieved in two ways
  - Method Overloading
  - Method Overriding



Polymorphism (Cont.)

- Overloading retains the functionality but with different input parameters
- Difference in input parameters can be in terms of:
  - Number of parameters
  - Data type of parameters
  - Sequence of parameters



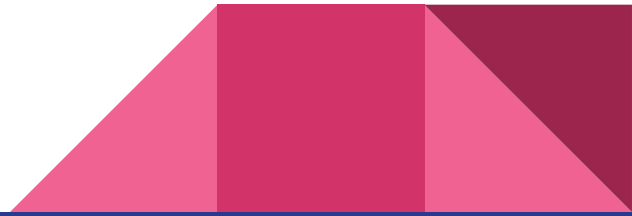
Polymorphism (Cont.)

- Overriding retains the number and type of parameters as well as the return type but provides different functionality
- Difference in input parameters can be in terms of:
  - Number of parameters
  - Data type of parameters
  - Sequence of parameters

SOLID Principles

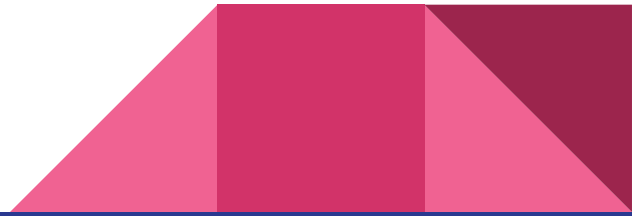
Introduction to SOLID

- SOLID principle are object-oriented design concepts
- Is an acronym made up of five other class design principles
- S - Single Responsibility
- O - Open-Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle



Single Responsibility Principle (SRP)

- Each class should be responsible for a single part or functionality of the system
- There should only be one reason to change
- Benefits
  - Ease of test
  - Loose coupling
  - Ease of maintenance



Single Responsibility Principle (Cont.)

- Customer Class
  - Responsible to store customer details
  - Includes account numbers
  - Tempting to include account balances and deposit and withdraw capabilities is against SRP
- Account Class
  - Responsible for all account related activities
  - Including account balance, deposit, withdraw functionality etc.

Single Responsibility Principle (Cont.)

- But how do you define responsibility?
- There is no precise way to define it
- Different developers can define responsibility differently
- This gives rise to conflicts
- So while trying to define SRP in terms of responsibility is logical but not practical

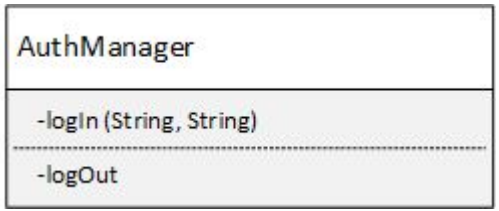
Single Responsibility Principle (Cont.)

- Another definition of SRP came up thus
- A class is said to satisfy SRP if we are able to define what the class does without using the conjunction AND
- The moment we say class A does this and this, it is a violation of SRP
- Sounds great right?
- E.g. Class AuthManager enables user to login to the system and logout from the system
- Does this class really violate SRP?
- I can always say Class AuthManager handles user authentication

Single Responsibility Principle (Cont.)

```java
public class AuthManager {

    public void logIn(String username, String password) {
        System.out.println("Login successful");
    }

    public void logOut() {
        System.out.println("Logout successful");
    }

}
```

Code



UML Diagram

Single Responsibility Principle (Cont.)

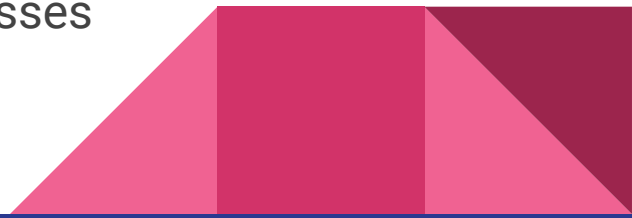
- So we still need a better definition
- We can define SRP in terms of “change”
- A class is said to satisfy SRP if it has one and only one reason to change
- Let's take an example of payroll application
- We defined Employee class as follows with the methods shown

Single Responsibility Principle (Cont.)

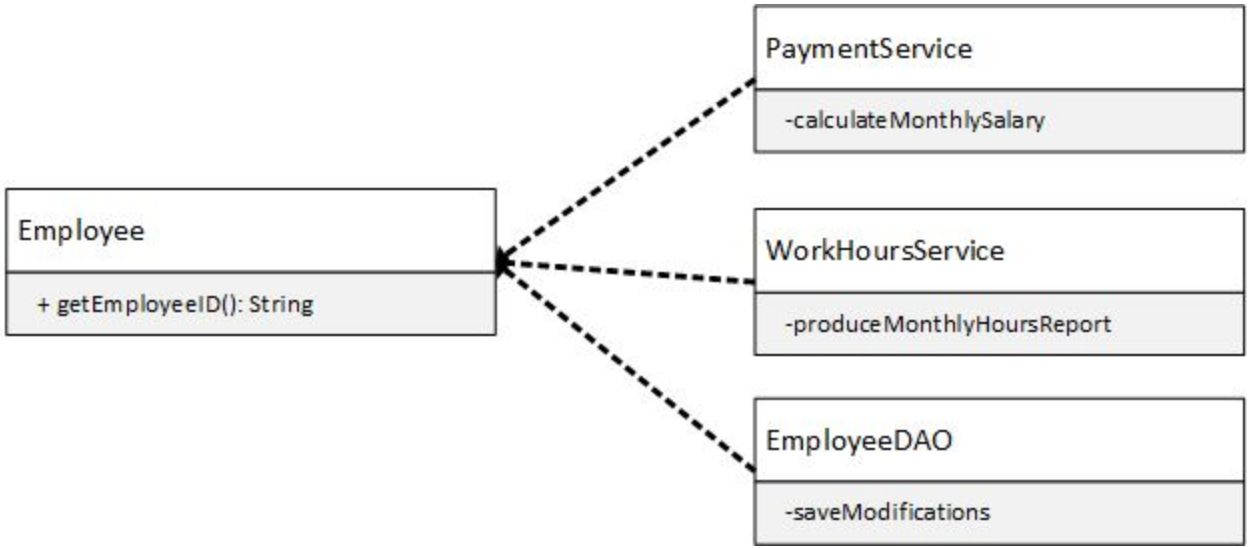
- What are the probable reasons for this class to change?
  - Finance department wants to introduce new benefit like internet allowance for WFH
  - HR department wants to change report format to comply with new management request
  - Engineering department can decide to migrate to a new DB
- A clear violation of SRP as the class has more than one reason to change

Single Responsibility Principle (Cont.)

- What can go wrong?
- Say HR department asked you to implement a new feature to identify employees working extra hours
- So we modified a method called calculateMonthlyHours which returns monthly hours along with over time
- This was also being used in calculateSalaries, which stopped working
- The solution is decouple and split it into multiple classes



Single Responsibility Principle (Cont.)



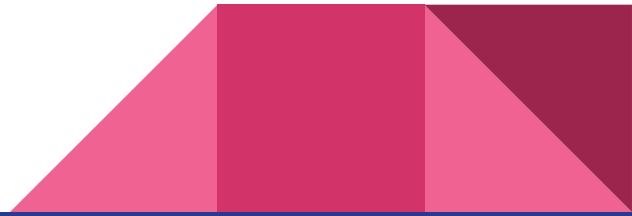
Single Responsibility Principle (Cont.)

- It may not always be as evident as we have seen to see such dependencies or reasons for change
- Take another example AuthManager that we saw previously

```
public class AuthManager_V2 {
    private String loggedInUser = "";

    public void logIn(String username, String password) {
        String hashPwd = hashPassword(password);
        if(userExistsInDB(username, hashPwd)) {
            loggedInUser = username;
        }
    }

    private String hashPassword(String pwd) {
        return pwd+pwd;
    }

    private boolean userExistsInDB(String username, String hPwd) {
        return true;
    }
}
```

Single Responsibility Principle (Cont.)

- This clearly violates SRP, though the class apparently has single responsibility
- There are 3 reasons for this class to change
  - Change in login flow itself
  - Change in hash algorithm or some other encrypting mechanism
  - Change in DB to store user credentials
- To resolve this the class must be refactored
- There are many ways this can be done
- One way is as below



Single Responsibility Principle (Cont.)

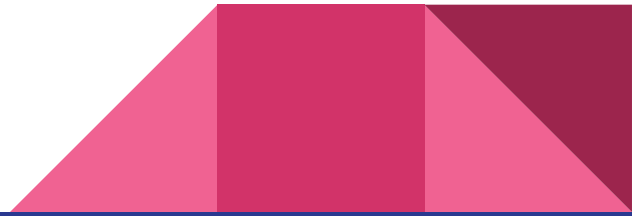
```
public class AuthManager_V3 {
    private PasswordHasher pwdHasher;
    private UserDAO userDao;
    private String loggedInUser = "";

    public void logIn(String username, String password) {
        String hashPwd = pwdHasher.hashPassword(password);
        if(userDao.userExistsInDB(username, hashPwd)) {
            loggedInUser = username;
        }
    }

}
```

```
public class UserDAO {

    public boolean userExistsInDB(String username, String hPwd) {
        return true;
    }

}
```

```
public class PasswordHasher {

    public String hashPassword(String pwd) {
        return pwd+pwd;
    }

}
```

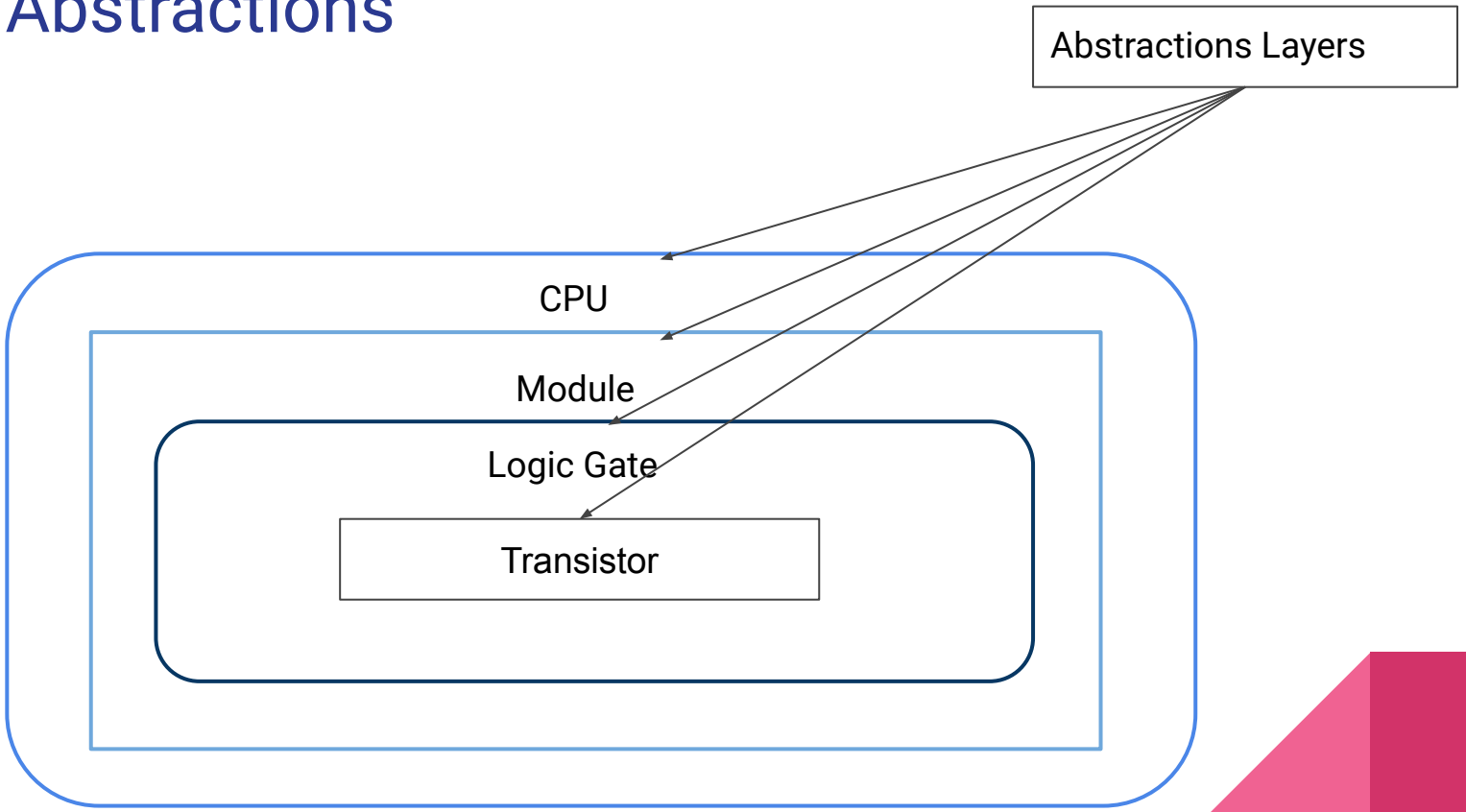
Single Responsibility Principle (Cont.)

- Framework
  - List all requirements and group them into classes
  - Analyze each item in list to identify potential reasons to change
  - Extract such functionalities that change due to different reasons into stand alone classes
- The above approach works well for simple as well as complex problems
- If you are thinking that breaking down classes into multiple classes might not be efficient
- Remember lego blocks



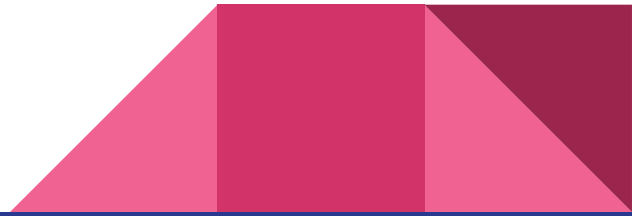
Single Responsibility Principle (Cont.)

Abstractions

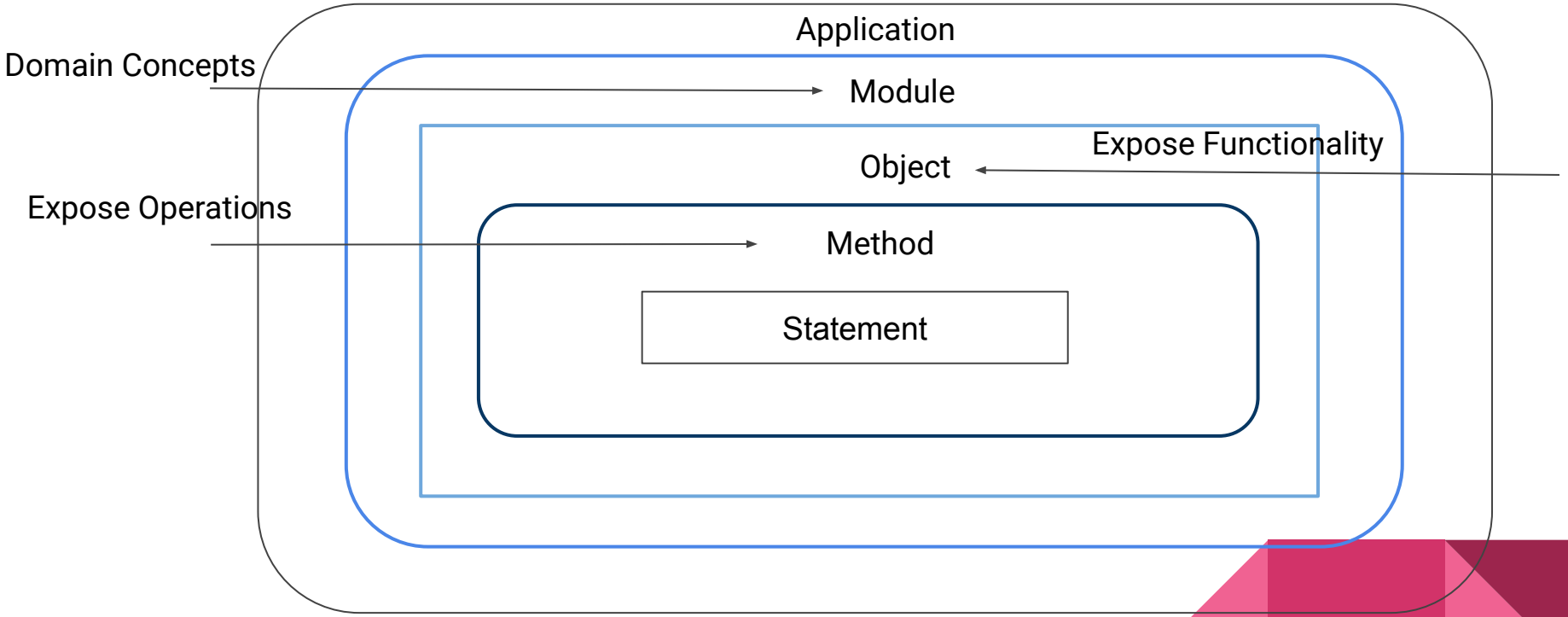


Abstractions (Cont.)

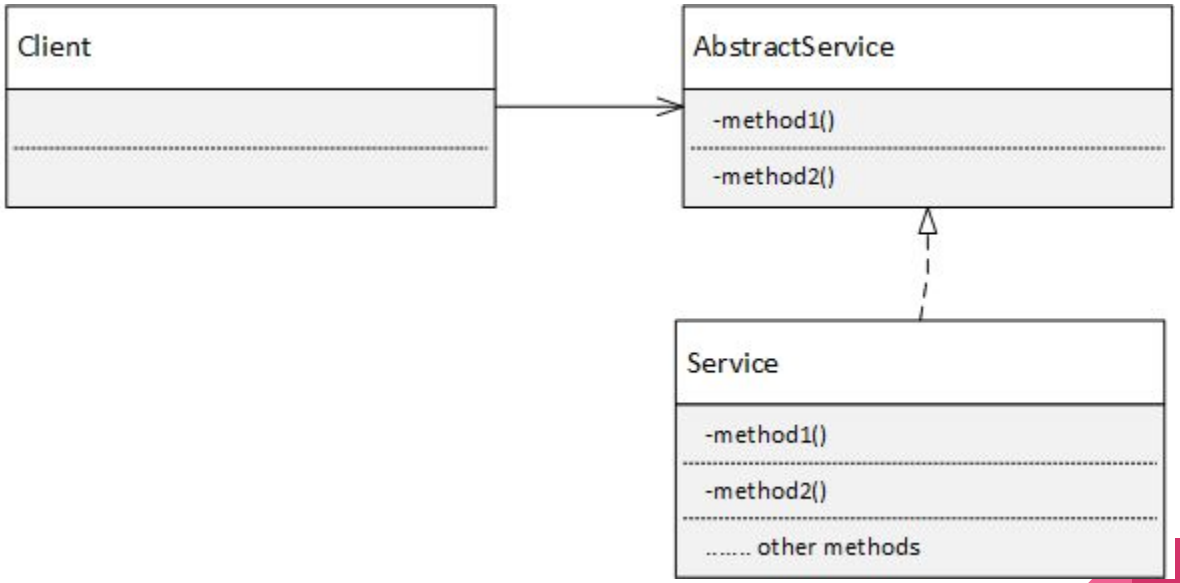
- Hide the internal details and present what is of interest to the outside world
- Can be done in multiple layers as shown above
- Each layer hides the specific details about all the layers under it and presents high level summary to the outside



Abstractions In OOP



Abstractions In OOP

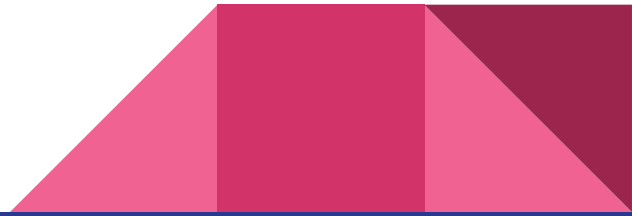


Open-Closed Principle (OCP)

- Software components (classes, modules, functions) should be open for extension, but closed for modification
- E.g. Bank has introduced privilege customer feature
- If a customer is meeting certain criteria like certain amount of FD, certain amount of average balance, then he can be upgraded to privilege customer with additional benefits
- Customer class should be extended and not modified to accommodate this feature

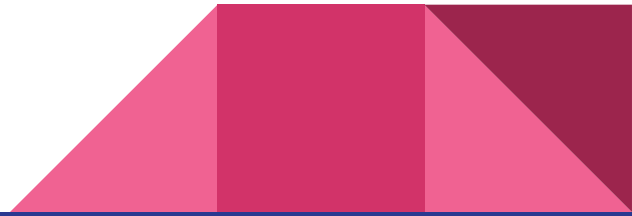
Open-Closed Principle (Cont.)

- Sounds confusing?
- What do we mean by Open for extension?
  - Inheritance
  - Polymorphism
  - Adding new functions without touching existing ones
- But what do we mean by closed for modification?
  - How to refactor code?
  - What about bug fixes?
  - How to add new features?



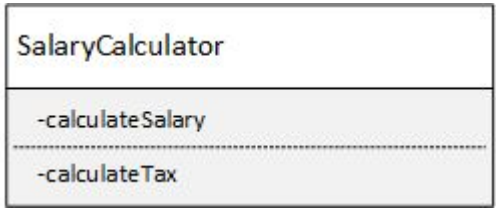
Open-Closed Principle (Cont.)

- So how do we modify the behavior?
- Answer is using abstractions
- Create abstract base classes that are fixed
- Create as many derived classes as we need or demanded to create modified behaviors
- Hence the definition - Open for extension but closed for modification
- This is precisely the principle of polymorphism
- So we can say polymorphism is at the heart of OCP



Open-Closed Principle (Cont.)

- Lets us say we have a class shown as below
- What is wrong with it?
- Violates SRP
- Does not confirm to OCP

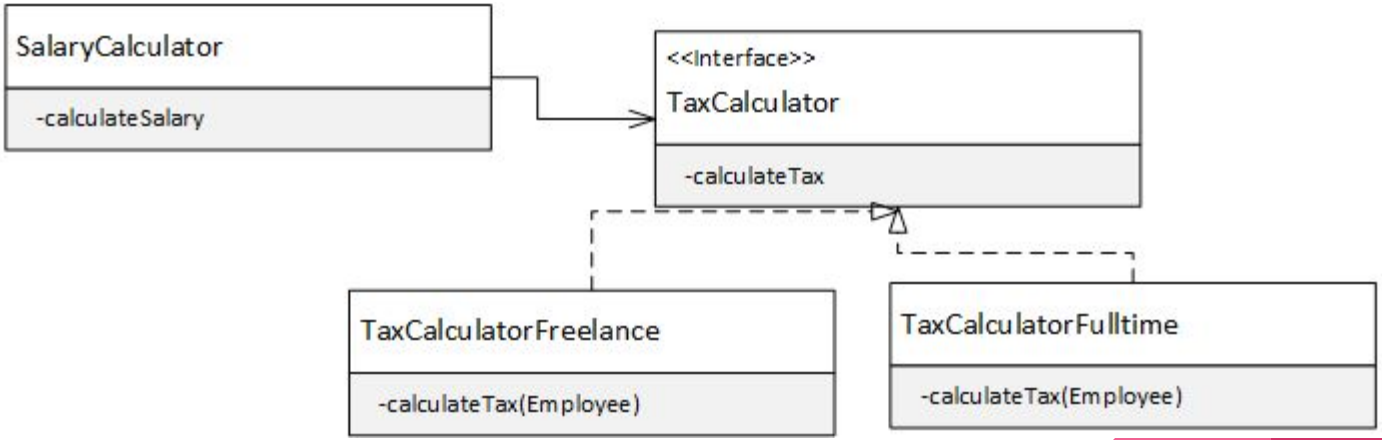


Open-Closed Principle (Cont.)

- As a first step we can break it as below to ensure it does not violate SRP

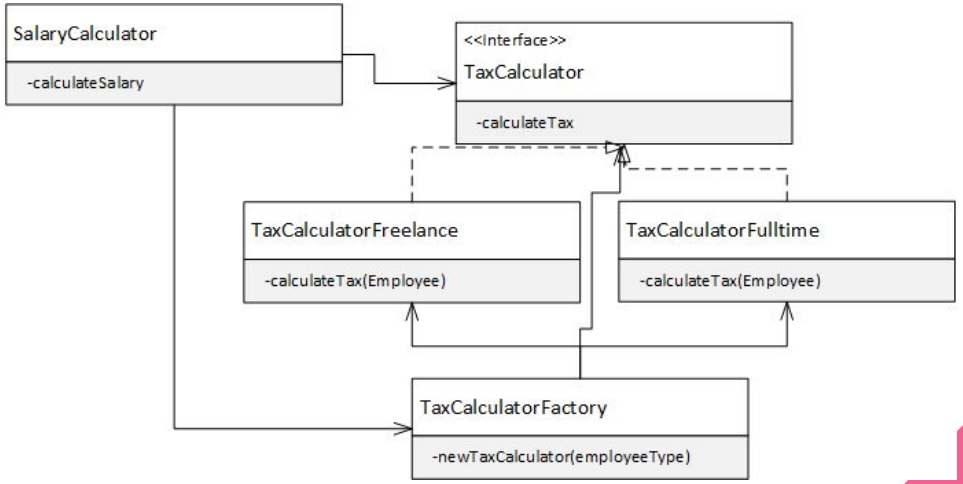
Open-Closed Principle (Cont.)

- We then bring in abstraction to take care of OCP
- TaxCalculator is open for extension but closed for modification



Open-Closed Principle (Cont.)

- Now all we need to do is implement a factory class that checks the employee type and accordingly returns TaxCalculator object for that type of employee

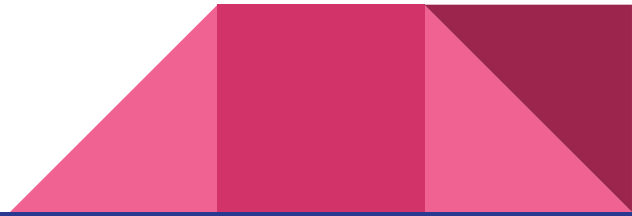


Liskov Substitution Principle

- This extends open - closed principle by focusing on the behavior or the super class and its subclasses
- Defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application
- Which means that objects of the subclasses should behave the same way as objects of the superclass

Liskov Substitution Principle (Cont.)

- You create a class
- Your fellow developer creates a class by extending your class
- The extended class should fit into the application without major issues
- An overridden method of the subclass should accept same input parameters as the method of the superclass
- Methods in the subclass should not implore stricter restrictions than that of superclass

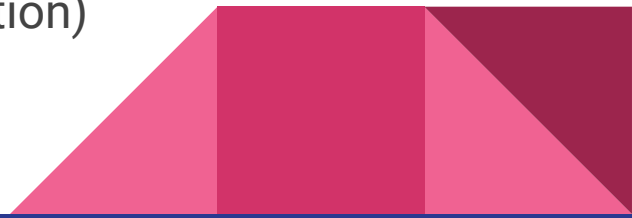


Liskov Substitution Principle (Cont.)

- Similarly the output value of the subclass methods should follow same rules as the output value of the superclass methods
- This is tough to be enforced to implement as this is more to do with behavior than the structure of classes
- Best way to ensure this principle is followed is via code reviews and test cases

Liskov Substitution Principle (Cont.)

- E.g. Basic Customer and Premium Customer
- Basic customer class accepts loan request from the customer and processes it and tells whether loan is accepted or rejected
- Premium customer class has access to customer bank account, so based on customer spending pattern, it suggests loan suitable for the customer
- Both these classes implement a shared interface LoanCustomer which has an abstract method called loanProcessing(loanApplication)



Interface Segregation Principle

- No client should be forced to depend on the methods that it does not intend to use
- Clients should not be forced to implement methods which they will not use
- An extension of Single Responsibility Principle for interfaces
- E.g. ReportGenerator interface
  - generateExcelReport
  - generatePDFReport
- Someone needing either pdf or excel will be forced to implement other

Dependency Inversion Principle

- High-level modules should not depend on low-level modules, both should depend on abstractions
- In other words design should depend on abstractions and not concrete methods
- E.g. Laptop design which supports standard as well as on-screen keyboard
- Laptop objects should not instantiate the keyboard classes using the new keyword
- Instead it should get necessary keyboard from an interface