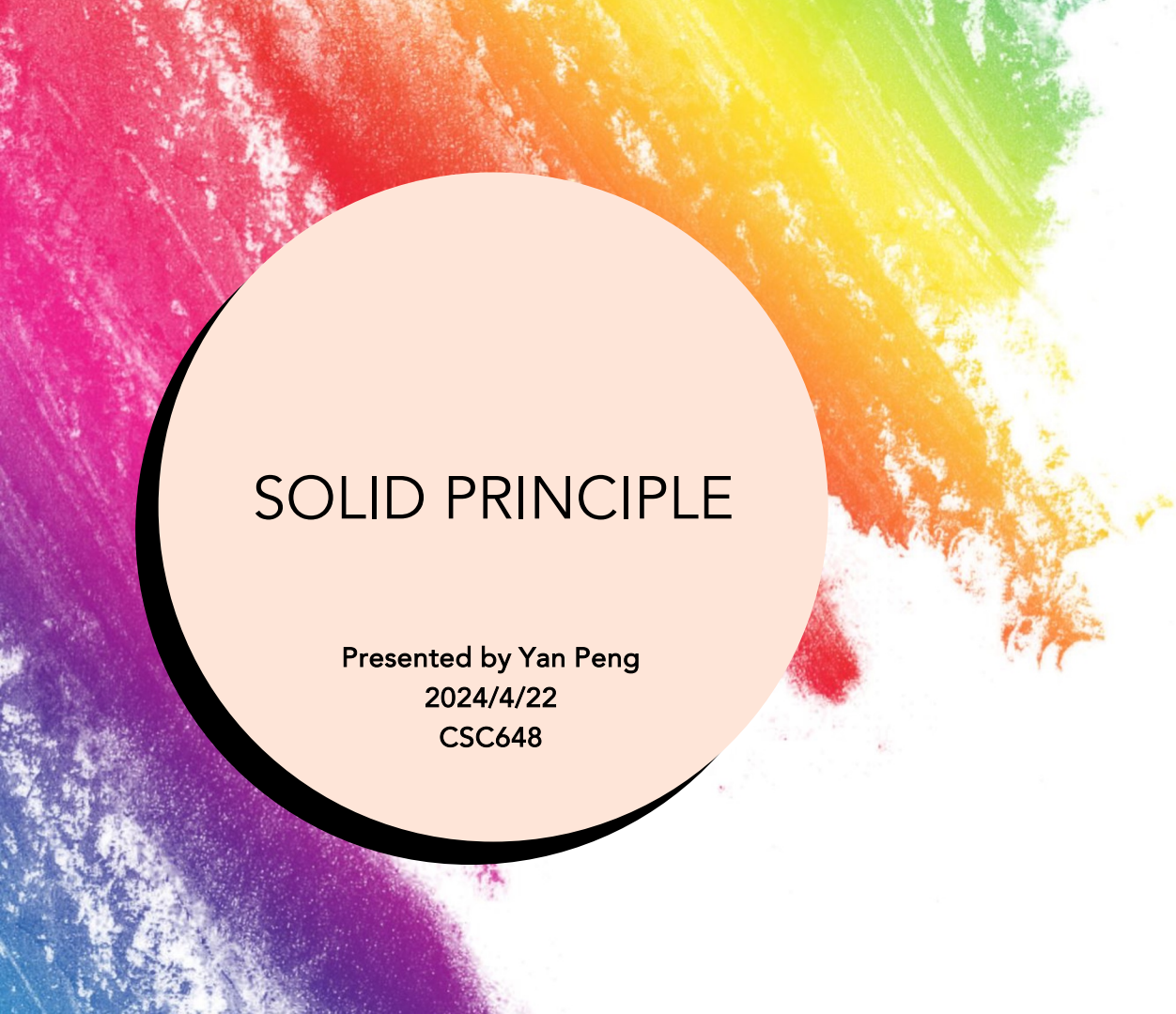
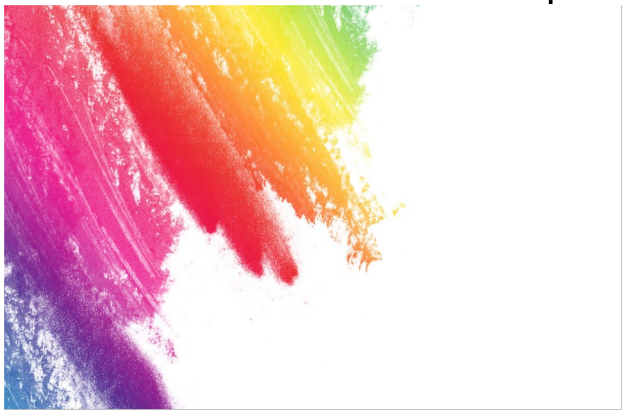


SOLID PRINCIPLE

Presented by Yan Peng
2024/4/22
CSC648

What is SOLID?
& How can SOLID help?



Drives good design

- Maintainable
- readable
- testable
- flexible
- extendable

- **Single Responsibility Principle**
  A module should have one and only one responsibility.
- **Open Closed Principle**
  Objects or entities should be open for extension but closed for modification.
- **Liskov Substitution Principle**
  Derived types must be completely substitutable for their base types.
- **Interface Segregation Principle**
  clients should not be forced to depend upon interfaces that they don't use.
- **Dependency Inversion Principle**
  Entities must depend on abstractions not on concretions.

Single Responsibility Principle

Is the code below correct? Why?

```
Public class Employee
{
        public int CalculateSalary()
          { return 10000;}
        public void SaveToDatabase(){
                                        ….}
}
```

Single Responsibility Principle

Separate the responsibility into two
Classes. We have separated the responsibilities based on the core employee responsibility.

```
// Following SRP
Public class Employee
{
      public int CalculateSalary()
        { return 10000;}
 }

Public class EmployeeRepository
{
     Public void SaveToDatabase( Employee employee)
       {……}
}
```

Employee class only deals with employee-related operations (CalculateSalary() while the EmployeeRepository class is responsible for saving employees to the database (SaveToDatabase()).

Single Responsibility Principle

Advantage of SRP:

When a class has only one responsibility, it becomes easier to change and test. If a class has multiple responsibilities, changing one responsibility may impact others and more testing efforts will be required then.

Open/Close Principle

If you want to add other account type, you don't need to change the class, you can just add new class.

Classes should be open for extension but closed for modification: Polymorphism / abstraction

```
// Base Account class
// This class is closed for modification because its implementation does not change when new types of accounts are added.

public class Account{
    public string Name {get;set;}
    public string Address {get;set;}
    public double Balance {get;set;}
}

//By using an interface, the Account class can be extended to support different types of interest calculations without modifying the Account class itself.
interface InterestAccount
{ double CalculateInterest(Account account);
}

public class SavingAccount implements InterestAccount{
    public double CalculateInterest(Account account){
        return account.Balance *0.3;
    }
}

public class CheckingAccount implements InterestAccount{
    public double CalculateInterest(Account account){
```

Open Close Principle

Advantage of Open/Close Principle:
The benefit is simple testing is required to test individual classes, but if you will keep on adding and modifying in one class, even for the smallest modification, the whole class needs to be tested.

Liskov Substitution Principle

LSP states that an object of a child class must be able to replace an object of the parent class without breaking the application.

Tips: We should never design our classes like this where we have to throw the error like this. Therefore, those methods must be applicable for derived classes. Also, derived classes can override the base method. ***Generally speaking***, most parent classes are abstract classes, which are regarded as a framework, and specific functions need to be implemented by subclasses. Prevent subclasses from overriding methods already implemented by parent classes.

```
public class Employee {
    public int calculateSalary() {
        return 10000;
    }

    public int calculateBonus() {
        return 5000;
    }
}

// Derived/child/subclass
public class PermanentEmployee extends Employee {
    @Override
    public int calculateSalary() {
        return 20000;
    }
    @Override
    public int calculateBonus() {
            return 6000;
    }
}

public class ContractEmployee extends Employee {
    @Override
    public int calculateSalary() {
        return 20000;
    }

    @Override // violate the LSP
because bonus is not applicable for contractual employee

    public int calculateBonus() {
        throw new UnsupportedOperationException("Not implemented");
    }
}
```

Interface Segregation Principle

ISP states that a class should not be forced to implement interfaces that it does not use.

Tips:

It is better to have smaller interfaces than larger interfaces.

Solution for this case:

Split them and implement based on it.

```
public interface Idrive{
    void Drive();
}
public interface IFly{
    void Fly();
}

public class NormalCar implements Idrive{
    public void Drive(){
        System.print.out("Drive car");
    }

public class FlyingCar implements Idrive, IFly{
    public void Drive(){.....}
    public void Faly(){....}
}
```

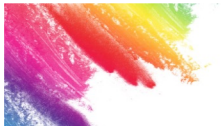
Bad code:

```
public interface IVehicle{
    void Drive();
    void Fly();
}

// suppose manager  ask you to create a class for just normal car.
public class NormalCar implements IVehicle{
    public void Drive(){
        System.print.out("Drive car");
    }

// fly method is not applicable for normal car, then you have to implement it.
// we are forcing on normal car class to implement the  ivehical interface here
    public void Fly(){
        throw new NotImplementedException();
    }
}
```

Dependency Inversion Principle:



DIP states that high-level class must not depend on a lower-level class. Both should depend on abstractions, and abstractions should not depend on details.

In this example, the NotificationService is a high-level module that depends on the MessageService abstraction.

It doesn't directly depend on the EmailService (a low-level module),
but instead, it receives an implementation of MessageService
(which could be EmailService or any other class implementing MessageService) through its constructor.

This way, the NotificationService is **decoupled** from the specific implementation of the message sending mechanism, adhering to the Dependency Inversion Principle.

```
// Abstraction
interface MessageService {
    void sendMessage(String message);
}

// Low-level module
class EmailService implements MessageService {
    public void sendMessage(String message) {
        // Send email
        System.out.println("Email sent: " + message);
    }
}

// High-level module
class NotificationService {
    private final MessageService messageService;
// constructor injection
    public NotificationService(MessageService messageService) {
        this.messageService = messageService;
    }

    public void sendNotification(String message) {
        messageService.sendMessage(message);
    }
}
public class Main {
    public static void main(String[] args) {
        MessageService emailService = new EmailService();
// Create an instance of NotificationService and inject the
EmailService
        NotificationService notificationService = new
NotificationService(emailService);
        notificationService.sendNotification("Hello, this is a notification!");
    }
}
```

Advantage of Dependency Inversion Principle

Flexibility: By depending on abstractions (interfaces), classes become more flexible. You can easily switch the implementation of MessageService from EmailService to SMSService or any other service without modifying the NotificationService class. This makes your code more adaptable to changes in requirements or technology.

Following DIP often leads to adhering to the SRP as well. Each class has a single responsibility, which is important for code maintainability and readability.

Conclusion

SOLID principles are still the foundation for modern software architecture.

keeping these principles in mind while designing, writing, and refactoring your code so that your code will be much more clean, extendable, and testable.

- Don't surprise the people who use your code.
- Don't overwhelm the people who read your code.
- Use proper boundaries for your code.
- Use the right level of coupling—keep things together that belong together and keep them apart if they belong apart.

Thank YOU