Introduction:

Robert C. Martin defined five object-oriented design principles:

* **S**ingle-Responsibility Principle
* **O**pen-Closed Principle
* **L**iskov’s Substitution Principle
* **I**nterface Segregation Principle, and
* **D**ependency Inversion Principle

**These together are popularly known as the SOLID principles.** When designing an object-oriented system, we should try to stick to these principles wherever possible.**These principles help us design a system that’s more extensible, understandable and maintainable.**

Using these principles can help us save a lot of efforts down the road as our application size grows.

1. Single-Responsibility Principle:

As the name suggests, the Single-Responsibility Principle(SRP) states that **every class must exactly do just one thing. In other words, there should not be more than one reason for us to modify a class.**

As we know, large systems usually have thousands of classes. If for any new requirement, multiple classes need to be touched then there are more chances of us introducing bugs by breaking another functionality.

The Single-Responsibility Principle provides us with the following benefits:

* **Less coupling:** Since every class would be doing just one thing, there’ll be far fewer dependencies
* **Easier to test:** the code will more likely be easier to test with far fewer test cases covering the system in entirety

The model classes of our system usually always follow the SRP principle. So say, we need to modify the state of users in our system, we’ll only touch the *User* class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class User {        private int id;      private String name;      private List<Address> addresses;        //constructors, getters, setters  } |

And so it follows the SRP principle.

1. Open-Closed Principle:

The Open-Closed Principle states that**the software components must be open for extension but closed for modification.**The intention here is to avoid introducing bugs in the system by breaking some existing working functionality due to code modifications. We should rather extend the existing class to support any additional functionality.

**This rule applies to the more stable classes of our system which have passed through the testing phases and is working well in production**. We’ll want to avoid breaking anything in that existing code and so we should rather extend its supported functionality to cater to new requirements.

Let’s say we have an *EventPlanner* class in our system which is running well on our production servers for long:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | public class EventPlanner {        private List<String> participants;      private String organizer;        public void planEvent() {          System.out.println("Planning a simple traditional event");          ...      }        ...  } |

But now, we’re planning to have a *ThemeEventPlanner* instead, which will plan events using a random theme to make them more interesting. Instead of directly jumping into the existing code and adding the logic to select an event theme and use it, it’s better to extend our production-stable class:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class ThemeEventPlanner extends EventPlanner {      private String theme;        ...  } |

For large-systems, it’ll be not very straight-forward to identify for what all purposes a class might have been used. And so by only extending the functionality, **we’re reducing chances of us dealing with the unknowns of the system.**

ADVERTISEMENT



1. Liskov’s Substitution Principle:

The Liskov’s Substitution Principle says that **a derived type must be able to complete substitute its base type without altering the existing behavior.**So, if we have two classes*A* and *B* such that *B extends A,*we should be able to replace*A* with*B* in our entire code base without impacting the system’s behavior.

For us to be able to achieve this, **the objects of our subclasses must behave exactly in the same way as that of the superclass objects.**

This principle**helps us avoid incorrect relationships between types as they can cause unexpected bugs or side effects.**

Let’s see the below example:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | public class Bird {      public void fly() {          System.out.println("Bird is now flying");      }  }    public class Ostrich extends Bird {      @Override      public void fly() {         throw new IllegalStateException("Ostrich can't fly");      }  } |

Though *Ostrich* is a *Bird*, still it can’t fly and so this is a clear violation of the Liskov substitution principle(LSP). **Also, the codes involving the logic on type checks are a clear indication that the incorrect relationships have been established.**

There’re two ways to refactor code to follow LSP:

* Eliminate incorrect relationships between objects
* Use the “*Tell, don’t ask*” principle to eliminate type checking and casting

Let’s say we have some code involving type-checks:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //main method code  for(User user : listOfUsers) {      if(user instanceof SubscribedUser) {          user.offerDiscounts();      }      user.makePurchases();  } |

**Using *“Tell, don’t ask”* principle, we’ll refactor the above code to look like:**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14 | public class SubscribedUser extends User {      @Override      public void makePurchases() {          this.offerDiscounts();          super.makePurchases();      }        public void offerDiscounts() {...}  }    //main method code  for(User user : listOfUsers) {      user.makePurchases();  } |

1. Interface Segregation Principle:

As per the Interface Segregation Principle, **the clients should not be forced to deal with the methods that they don’t use. We should split the larger interface into smaller ones, wherever needed.**

Let’s say we have a *ShoppingCart* interface:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public interface ShoppingCart {        void addItem(Item item);      void removeItem(Item item);      void makePayment();      boolean checkItemAvailability(Item item);    } |

Making payments and checking an item’s availability is not what a shopping cart is intended to do. There’s a high probability of us encountering implementations of this interface which won’t use those methods.

So, it’s a good idea to break the above interface as:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | public interface BaseShoppingCart {      void addItem(Item item);      void removeItem(Item item);  }    public interface PaymentProcessor {      void makePayment();  }    public interface StockVerifier {      boolean checkItemAvailability(Item item);  } |

The Interface Segregation Principle(ISP) also reinforces other principles:

* **Single Responsibility Principle:** Classes that implement smaller interfaces are usually more focused and usually have a single purpose
* **Liskov Substitution Principle:** With smaller interfaces, there are more chances of us having classes implementing them to fully substitute the interface

1. Dependency Inversion:

It is one of the most popular and useful design principles as it promotes loose coupling among objects. The Dependency Inversion Principle states that **the high-level modules should not depend on low-level modules; both should depend on the abstractions.**

**High-level modules tell us what the software should do**. User Authorization and Payment are examples of high-level modules.

On the other hand,**the low-level modules tell us how the software should do various tasks** i.e. it involves implementation details. Some examples of low-level modules include security(OAuth), networking, database access, IO, etc.

Let’s write a *UserRepository* interface and its implementation class:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | public interface UserRepository {      List<User> findAllUsers();  }  public class UserRepository implements UserRepository {        public List<User> findAllUsers() {          //queries database and returns a list of users          ...      }  } |

We have here extracted out the abstraction of the module in an interface.

Now say we have high-level module *UserAuthorization*which checks whether a user is authorized to access a system or not. We’ll only use the reference of the *UserRepository* interface:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class UserAuthorization {        ...        public boolean isValidUser(User user) {          UserRepository repo = UserRepositoryFactory.create();          return repo.getAllUsers().stream().anyMatch(u -> u.equals(user));      }  } |

Additionally, we’re using a factory class to instantiate a *UserRepository*.

Note that**we are only relying on the abstraction and not concretion.**And so, we can easily add more implementations of *UserRepository* without much impact on our high-level module.

How elegant it is!

Conclusion:

In this tutorial, we discussed the SOLID design principles. We also looked at the code examples in Java for each of these principles.