**SOLID - Class Design Principles**

# Introduction

SOLID are five basic principles which help to create good software architecture. SOLID is an acronym where:-

* S stands for SRP (Single responsibility principle)
* O stands for OCP (Open closed principle)
* L stands for LSP (Liskov substitution principle)
* I stands for ISP (Interface segregation principle)
* D stands for DIP (Dependency inversion principle)

|  |  |
| --- | --- |
| Principal Name | What it says? |
| Single responsibility principle | **One class should have one and only one responsibility.** |
| Open closed principle | **Software components should be open for extension, but closed for modification. In other words, Extension should be preferred over modification.** |
| Liskov substitution principle | **Derived types must be completely substitutable for their base types. In other words, A parent class object should be able to refer child objects seamlessly during runtime polymorphism.** |
| Interface segregation principle | **Clients should not be forced to implement unnecessary methods which they will not use.** |
| Dependency inversion principle | **Depend on abstractions, not on concretions. In other words, High level modules should not depend on low level modules but should depend on abstraction.** |

# **Single responsibility principle**

The name of the principle says it all:

**"One class should have one and only one responsibility"**

**OR**

*A class should have one, and only one, reason to change.*

SRP says that a class should have only one responsibility and not multiple. In other words, you should write, change and maintain a class for only one purpose**.  There should never be more than one reason for a class to change.** This means that every class, or similar structure, in your code should have only one job to do.

Everything in the class should be related to that single purpose, i.e. be cohesive. It does not mean that your classes should only contain one method or property. There can be a lot of members as long as they relate to the single responsibility.  I would like to clarify here that one responsibility doesn’t mean that the class has only ONE method. A responsibility can be implemented by means of different methods in the class.

### Why that is this principle is required?

We all know that requirements change over time. Each of them also changes the responsibility of at least one class. The more responsibilities your class has, the more often you need to change it. If your class implements multiple responsibilities, they are no longer independent of each other.

You need to change your class as soon as one of its responsibilities changes. That is obviously more often than you would need to change it if it had only one responsibility.

Imagine designing classes with more than one responsibility/implementing more than one functionality. There’s no one stopping you to do this. But imagine the amount of dependency your class can create within itself in the due course of the development time. So when you are asked to change a certain functionality, you are not really sure how it would impact the other functionalities implemented in the class. The change might or might not impact other features, but you really can’t take risk, especially in production applications. So you end up testing all the dependent features.

You might say, we have automated tests, and the number of tests to be checked are low, but imagine the impact over time. This kind of changes get accumulates owing to the viscosity of the code making it really fragile and rigid.

In the end, you need to change your class more often, and each change is more complicated, has more side-effects, and requires a lot more work than it should have. So, it’s better to avoid these problems by making sure that each class has only one responsibility.

One way to correct the violation of SRP is to decompose the class functionalities into different classes, each of which confirms to SRP.

### ****Easier to understand****

The single responsibility principle provides another substantial benefit. Classes, software components and microservices that have only one responsibility are much easier to explain, understand and implement than the ones that provide a solution for everything. This reduces the number of bugs, improves your development speed, and makes your life as a software developer a lot easier.

An example to clarify this principle:

Suppose you are asked to implement a UserSettingService where in the user can change the settings but before that the user has to be authenticated. One way to implement this would be:

public class UserSettingService

{

public void changeEmail(User user)

{

if(checkAccess(user))

{

//Grant option to change

}

}

public boolean checkAccess(User user)

{

//Verify if the user is valid.

}

}

All looks good, until you would want to reuse the checkAccess code at some other place OR you want to make changes to the way checkAccess is being done OR you want to make change to the way email changes are being approved. In all the later 2 cases you would end up changing the same class and in the first case you would have to use UserSettingService to check for access as well, which is unnecessary.

One way to correct this is to decompose the UserSettingService into UserSettingService and SecurityService. And move the checkAccess code into SecurityService.

public class UserSettingService

{

public void changeEmail(User user)

{

if(SecurityService.checkAccess(user))

{

//Grant option to change

}

}

}

………………………………………………………………………………………………………………………………………………………………………….

public class SecurityService

{

public static boolean checkAccess(User user)

{

//check the access.

}

}

### Another example would be:

Suppose there is a requirement to download the file – may be in csv/json/xml format, parse the file and then update the contents into a database or file system. One approach would be to:

public class Task

{

public void downloadFile(location)

{

//Download the file

}

public void parseTheFile(file)

{

//Parse the contents of the file- XML/JSON/CSV

}

public void persistTheData(data)

{

//Persist the data to Database or file system.

}

}

Looks good, all in one place easy to understand. But what about the number of times this class has to be updated? What about the reusability of parser code? or download code? It’s not good design in terms of reusability of different parts of the code, in terms of cohesiveness.

One way to decompose the Task class is to create different classes for downloading the file – Downloader, for parsing the file – Parser and for persisting to the database or file system.

### ****Real-world examples of the single responsibility principle****

* **Java Persistence API (JPA):** It has one, and only one, responsibility: Defining a standardized way to manage data persisted in a relational database by using the object-relational mapping concept.

That’s a pretty huge responsibility. The specification defines lots of different interfaces for it, specifies a set of entity lifecycle states and the transitions between them, and even provides a query language, called JPQL.

But that is the only responsibility of the JPA specification. Other functionalities which you might need to implement your application, like validation, REST APIs or logging, are not the responsibility of JPA. You need to include other specifications or frameworks which provide these features.

* **JPA EntityManager :** The EntityManager interface provides a set of methods to persist, update, remove and read entities from a relational database. Its responsibility is to manage the entities that are associated with the current persistence context.

That is the only responsibility of the EntityManager. It doesn’t implement any business logic or validation or user authentication.

* **JPA AttributeConverter: T**he responsibility of the EntityManager might be too big to serve as an easily understandable example of the single responsibility principle. So, let’s take a look at a smaller example: an AttributeConverter as the JPA specification defines it.

The responsibility of an AttributeConverter is small and easy to understand. It converts a data type used in your domain model into one that your persistence provider can persist in the database. You can use it to persist unsupported data types, like your favorite value class, or to customize the mapping of a supported data type, like a customized mapping for enum values.

* **Spring Data Repository:** The last example to talk about is the Spring Data repository. It implements the repository pattern and provides the common functionality of create, update, remove, and read operations. The repository adds an abstraction on top of the EntityManager with the goal to make JPA easier to use and to reduce the required code for these often-used features.

You can define the repository as an interface that extends a Spring Data standard interface, e.g., Repository, CrudRepository, or PagingAndSortingRepository. Each interface provides a different level of abstraction, and Spring Data uses it to generate implementation classes that provide the required functionality.

The following code snippet shows a simple example of such a repository. The AuthorRepository extends the Spring CrudRepository interface and defines a repository for an Author entity that uses an attribute of type Long as its primary key.

interface AuthorRepository extends CrudRepository<Author, Long> {

List findByLastname(String lastname);

}

# **Open Closed Principle**

This is second important rule which you should keep in mind while designing your application. It says:

**"Software entities (classes, modules, functions, etc.)  Should be open for extension, but closed for modification"**

**“Extension should be preferred over modification.”**

“Open to extension” means that you should design your classes so that new functionality can be added as new requirements are generated. “Closed for modification” means that once you have developed a class you should never modify it, except to correct bugs.”

**“You should be able to extend a classes behavior, without modifying it.”**

In other words, Open closed design principles say that new functionality should be added by introducing new classes, methods or fields instead of modifying already tried and tested code. One of the way to achieve this is Inheritance where class is extended to introduce new functionality on top of inherited basic features.

### General Idea of this Principle

The general idea of this principle is great. It tells you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending classes. Unfortunately, Bertrand Mayer proposes to use [inheritance](https://stackify.com/oop-concept-inheritance/) to achieve this goal:

“A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.”

### Why should be open for extension?

Software entities once written shouldn’t be modified to add new functionality, instead one has to extend the same to implement the new requirement or add new functionality. So your code is less rigid and fragile and also extensible.

It means that your classes should be designed such a way that whenever fellow developers’ wants to change the flow of control in specific conditions in application, all they need to extend your class and overrides some functions and that’s it.

### Why should be closed for modification?

If we are “MODIFYING” the current existing class code for every change and every time we modify we need to ensure that all the previous functionalities and connected client are working as before.

If we are changing the existing class again and again, we need to ensure that the previous conditions with new one’s are tested again , existing client’s which are referencing this class are working properly as before.

### Open/Closed Principle to the [Polymorphic](https://stackify.com/oop-concept-polymorphism/) Open/Closed Principle

 Inheritance introduces tight coupling if the subclasses depend on implementation details of their parent class.

That’s why Robert C. Martin and others redefined the Open/Closed Principle to the [Polymorphic](https://stackify.com/oop-concept-polymorphism/) Open/Closed Principle. It uses interfaces instead of superclasses to allow different implementations which you can easily substitute without changing the code that uses them. The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software.

The main benefit of this approach is that an interface introduces an additional level of abstraction which enables loose coupling. The implementations of an interface are independent of each other and don’t need to share any code. If you consider it beneficial that two implementations of an interface share some code, you can either use [inheritance](https://stackify.com/oop-concept-inheritance/) or [composition](https://stackify.com/oop-concepts-composition/).

### Benefit or Open Closed Design Principle:

* Application will be more robust because we are not changing already tested class.
* Applying OCP to your projects limits the need to change source code once it has been written, tested and debugged. This reduces the risk of introducing new bugs to existing code, leading to more robust software.
* Flexible because we can easily accommodate new requirements.
* Easy to test and less error prone.

### How to implement this principle?

The best way to implement the open closed principle is to first start with implementing the Single Responsibility Principle: a class should have one, and only one, reason to change.   
This will separate different concerns in your code.  
The next step is representing these separate concerns by abstractions and let consumers of these concerns talk to these abstractions.

To state the open closes principle very straightforward way you can say:

* You should design modules that never change.
* When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

We achieve this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Functionality can be added by creating new classes that implement the interfaces.

Derivatives from an abstraction are closed for modification because the abstraction is fixed but behavior can be extended by creating new derivatives of the abstraction.

# **Liskov’s Substitution Principle**

This principle is a variation of previously discussed open closed principle. It says:

**"Derived types must be completely substitutable for their base types"**

**“A parent class object should be able to refer child objects seamlessly during runtime polymorphism.”**

**The Liskov Substitution Principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.**

**“The Liskov Substitution Principle (LSP) applies to inheritance hierarchies, specifying that you should design your classes so that client dependencies can be substituted with subclasses without the client knowing about the change.”**

Liskovs Substitution Principle states that any method that takes class X as a parameter must be able to work with any subclasses of X.

The principle makes sure that every class follows the contract defined by its parent class. If the class Car has a method called Break it’s vital that all subclasses breaks when the Break method is invoked. Imagine the suprise if Break() in a Ferrari only works if the switch ChickenMode is activated.

It means that the classes fellow developer created by extending your class should be able to fit in application without failure. I.e. if a fellow developer poorly extended some part of your class and injected into framework/ application then it should not break the application or should not throw fatal [exceptions](http://howtodoinjava.com/2013/04/04/java-exception-handling-best-practices/).

### Enforcing the Liskov Substitution Principle

If you decide to apply this principle to your code, the behavior of your classes becomes more important than its structure. Unfortunately, there is no easy way to enforce this principle. The compiler only checks the structural rules defined by the Java language, but it can’t enforce a specific behavior.

You need to implement your own checks to ensure that your code follows the Liskov Substitution Principle. In the best case, you do this via code reviews and test cases. In your test cases, you can execute a specific part of your application with objects of all subclasses to make sure that none of them causes an error or significantly changes its performance. You can try to do similar checks during a code review. But what’s even more important is that you check that you created and executed all the required test cases.

### Example:

**Start with Violation**

Let’s use the motivator image as inspiration and define the following classes:

|  |
| --- |
| public interface IDuck  {     void Swim();  }  public class Duck : IDuck  {     public void Swim()     {        //do something to swim     }  }  public class ElectricDuck : IDuck  {     public void Swim()     {        if (!IsTurnedOn)          return;          //swim logic     }  } |

And the calling code:

|  |
| --- |
| void MakeDuckSwim(IDuck duck)  {      duck.Swim();  } |

As you can see, there are two examples of ducks. One regular duck and one electric duck.

The electric duck can only swim if it’s turned on.The MakeDuckSwim method will not work if a duck is electric and not` turned on.

**Solution**

The solution would be to automatically turn on the duck in the Swim method and by doing so make the electric duck behave exactly as defined by the IDuck interface.

|  |
| --- |
| public class ElectricDuck : IDuck  {     public void Swim()     {        if (!IsTurnedOn)          TurnOnDuck();          //swim logic     }  } |

# **Interface Segregation Principle**

This principle is my favorite one. It is applicable to interfaces as single responsibility principle holds to classes. It says:

**"Clients should not be forced to implement unnecessary methods which they will not use."**

**“Clients should not be forced to depend upon interfaces that they do not use.”**

**“Make fine grained interfaces that are client specific.”**

**How to achieve this principle:**

When we have non-cohesive interfaces, the ISP guides us to create multiple, smaller, cohesive interfaces.

### Example 1:

Take an example. Developer Alex created an interface Reportable and added two methods generateExcel() and generatedPdf (). Now client ‘A’ wants to use this interface but he intend to use reports only in PDF format and not in excel. Will he achieve the functionality easily.

NO. He will have to implement two methods, out of which one is extra burden put on him by designer of software. Either he will implement another method or leave it blank. So are not desired cases, right??

**Solution** is to create two interfaces by breaking the existing one. They should be like PdfReportable and ExcelReportable. This will give the flexibility to user to use only required functionality only.

### Example 2:

#### Starting from Violating the Interface Segregation Principle

None of us willingly ignores common design principles to write bad software. But it happens quite often that an application gets used for multiple years and that its users regularly request new features.

From a business point of view, this is a great situation. But from a technical point of view, the implementation of each change bears a risk. It’s tempting to add a new method to an existing interface even though it implements a different responsibility and would be better separated in a new interface. That’s often the beginning of interface pollution, which sooner or later leads to bloated interfaces that contain methods implementing several responsibilities.

Let’s take a look at a simple example where this happened.

In the beginning, the project used the BasicCoffeeMachine class to model a basic coffee machine. It uses ground coffee to brew a delicious filter coffee.

class BasicCoffeeMachine implements CoffeeMachine {

private Map<CoffeeSelection, Configuration> configMap;

private GroundCoffee groundCoffee;

private BrewingUnit brewingUnit;

public BasicCoffeeMachine(GroundCoffee coffee) {

this.groundCoffee = coffee;

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap<>();

this.configMap.put(CoffeeSelection.FILTER\_COFFEE, new Configuration(30, 480));

}

@Override

public CoffeeDrink brewFilterCoffee() {

Configuration config = configMap.get(CoffeeSelection.FILTER\_COFFEE);

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.FILTER\_COFFEE, this.groundCoffee, config.getQuantityWater());

}

@Override

public void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException {

if (this.groundCoffee != null) {

if (this.groundCoffee.getName().equals(newCoffee.getName())) {

this.groundCoffee.setQuantity(this.groundCoffee.getQuantity() + newCoffee.getQuantity());

} else {

throw new CoffeeException("Only one kind of coffee supported for each CoffeeSelection.");

}

} else {

this.groundCoffee = newCoffee;

}

}

}

At that time, it was perfectly fine to [extract the *CoffeeMachine* interface](https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html) with the methods *addGroundCoffee* and *brewFilterCoffee*. These are the two essential methods of a coffee machine and should be implemented by all future coffee machines.

public interface CoffeeMachine {

CoffeeDrink brewFilterCoffee() throws CoffeeException;

void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException;

}

#### Polluting the interface with a new method

But then somebody decided that the application also needs to support espresso machines. The development team modeled it as the *EspressoMachine* class that you can see in the following code snippet. It’s pretty similar to the *BasicCoffeeMachine*class.

The developer decided that an espresso machine is just a different kind of coffee machine. So, it has to implement the *CoffeeMachine* interface.

The only difference is the *brewEspresso* method, which the *EspressoMachine* class implements instead of the *brewFilterCoffee* method. Let’s ignore the Interface Segregation Principle for now and perform the following three changes:

1. The EspressoMachine class implements the CoffeeMachine interface and its brewFilterCoffee method.

public CoffeeDrink brewFilterCoffee() throws CoffeeException {

throw new CoffeeException("This machine only brews espresso.");

}

2. We add the brewEspresso method to the CoffeeMachine interface so that the interface allows you to brew an espresso.

public interface CoffeeMachine {

CoffeeDrink brewFilterCoffee() throws CoffeeException;

void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException;

CoffeeDrink brewEspresso() throws CoffeeException;

}

3. You need to implement the brewEspresso method on the BasicCoffeeMachine class because it’s defined by the CoffeeMachine interface. You can also provide the same implementation as a [default method](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html) on the CoffeeMachine interface.

@Override

public CoffeeDrink brewEspresso() throws CoffeeException {

throw new CoffeeException("This machine only brews filter coffee.");

}

After you’ve done these changes, your class and class diagram should look like this:

public class EspressoMachine implements CoffeeMachine {

private Map configMap;

private GroundCoffee groundCoffee;

private BrewingUnit brewingUnit;

public EspressoMachine(GroundCoffee coffee) {

this.groundCoffee = coffee;

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap();

this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

}

@Override

public CoffeeDrink brewEspresso() {

Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.ESPRESSO,

this.groundCoffee, config.getQuantityWater());

}

@Override

public void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException {

if (this.groundCoffee != null) {

if (this.groundCoffee.getName().equals(newCoffee.getName())) {

this.groundCoffee.setQuantity(this.groundCoffee.getQuantity()

+ newCoffee.getQuantity());

} else {

throw new CoffeeException(

"Only one kind of coffee supported for each CoffeeSelection.");

}

} else {

this.groundCoffee = newCoffee;

}

}

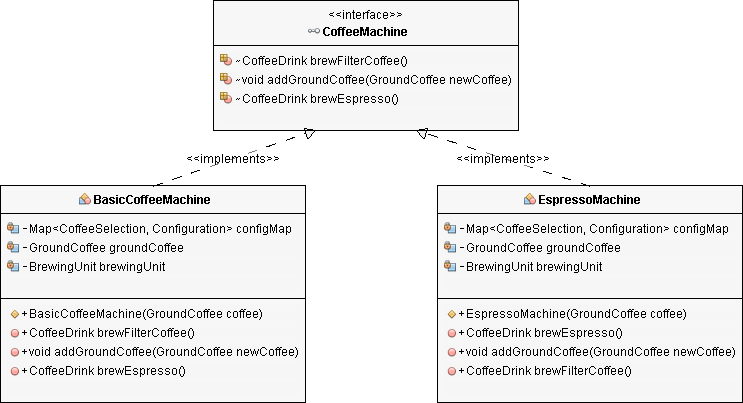
@Override

public CoffeeDrink brewFilterCoffee() throws CoffeeException {

throw new CoffeeException("This machine only brew espresso.");

}

}



Especially the 2nd and 3rd change should show you that the *CoffeeMachine* interface is not a good fit for these two coffee machines. The *brewEspresso* method of the *BasicCoffeeMachine* class and the *brewFilterCoffee* method of the *EspressoMachine*class throw a *CoffeeException* because these operations are not supported by these kinds of machines. You only had to implement them because they are required by the *CoffeeMachine* interface.

But the implementation of these two methods isn’t the real issue. The problem is that the *CoffeeMachine* interface will change if the signature of the *brewFilterCoffee*method of the *BasicCoffeeMachine* method changes. That will also require a change in the *EspressoMachine* class and all other classes that use the *EspressoMachine*, even so, the *brewFilterCoffee* method doesn’t provide any functionality and they don’t call it.

#### Follow the Interface Segregation Principle

OK, so how can you fix the CoffeMachine interface and its implementations BasicCoffeeMachine and EspressoMachine?

You need to split the CoffeeMachine interface into multiple interfaces for the different kinds of coffee machines. All known implementations of the interface implement the addGroundCoffee method. So, there is no reason to remove it.

public interface CoffeeMachine {

void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException;

}

That’s not the case for the brewFilterCoffee and brewEspresso methods. You should create two new interfaces to segregate them from each other. And in this example, these two interfaces should also extend the CoffeeMachine interface. But that doesn’t have to be the case if you refactor your own application. Please check carefully if an interface hierarchy is the right approach, or if you should define a set of interfaces.

After you’ve done that, the FilterCoffeeMachine interface extends the CoffeeMachineinterface, and defines the brewFilterCoffee method.

public interface FilterCoffeeMachine extends CoffeeMachine {

CoffeeDrink brewFilterCoffee() throws CoffeeException;

}

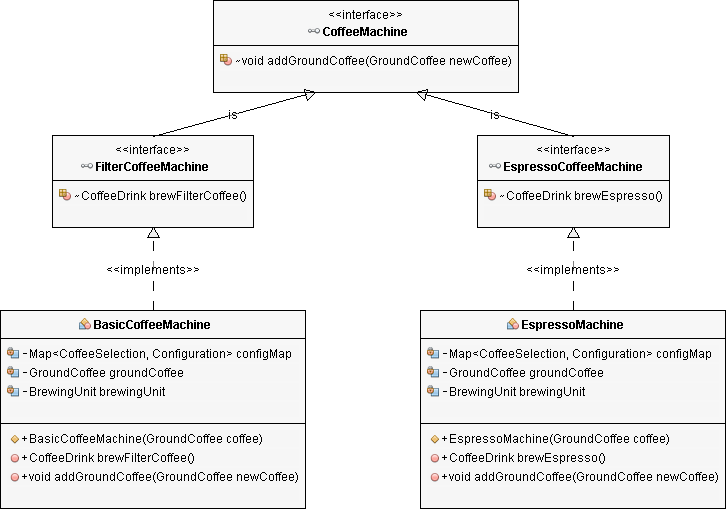
And the EspressoCoffeeMachine interface also extends the CoffeeMachine interface, and defines the brewEspresso method.

public interface EspressoCoffeeMachine extends CoffeeMachine {

CoffeeDrink brewEspresso() throws CoffeeException;

}

Congratulation, you segregated the interfaces so that the functionalities of the different coffee machines are independent of each other. As a result, the BasicCoffeeMachine and the EspressoMachine class no longer need to provide empty method implementations and are independent of each other.



The BasicCoffeeMachine class now implements the FilterCoffeeMachine interface, which only defines the addGroundCoffee and the brewFilterCoffee methods.

public class BasicCoffeeMachine implements FilterCoffeeMachine {

private Map<CoffeeSelection, Configuration> configMap;

private GroundCoffee groundCoffee;

private BrewingUnit brewingUnit;

public BasicCoffeeMachine(GroundCoffee coffee) {

this.groundCoffee = coffee;

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap<>();

this.configMap.put(CoffeeSelection.FILTER\_COFFEE, new Configuration(30,

480));

}

@Override

public CoffeeDrink brewFilterCoffee() {

Configuration config = configMap.get(CoffeeSelection.FILTER\_COFFEE);

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.FILTER\_COFFEE,

this.groundCoffee, config.getQuantityWater());

}

@Override

public void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException {

if (this.groundCoffee != null) {

if (this.groundCoffee.getName().equals(newCoffee.getName())) {

this.groundCoffee.setQuantity(this.groundCoffee.getQuantity()

+ newCoffee.getQuantity());

} else {

throw new CoffeeException(

"Only one kind of coffee supported for each CoffeeSelection.");

}

} else {

this.groundCoffee = newCoffee;

}

}

}

And the EspressoMachine class implements the EspressoCoffeeMachine interface with its methods addGroundCoffee and brewEspresso.

public class EspressoMachine implements EspressoCoffeeMachine {

private Map configMap;

private GroundCoffee groundCoffee;

private BrewingUnit brewingUnit;

public EspressoMachine(GroundCoffee coffee) {

this.groundCoffee = coffee;

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap();

this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

}

@Override

public CoffeeDrink brewEspresso() throws CoffeeException {

Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.ESPRESSO,

this.groundCoffee, config.getQuantityWater());

}

@Override

public void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException {

if (this.groundCoffee != null) {

if (this.groundCoffee.getName().equals(newCoffee.getName())) {

this.groundCoffee.setQuantity(this.groundCoffee.getQuantity()

+ newCoffee.getQuantity());

} else {

throw new CoffeeException(

"Only one kind of coffee supported for each CoffeeSelection.");

}

} else {

this.groundCoffee = newCoffee;

}

}

}

#### Extending the application

After you segregated the interfaces so that you can evolve the two coffee machine implementations independently of each other, you might be wondering how you can add different kinds of coffee machines to your applications. In general, there are four options for that:

1. The new coffee machine is a FilterCoffeeMachine or an EspressoCoffeeMachine. In this case, you only need to implement the corresponding interface.
2. The new coffee machine brews filter coffee and espresso. This situation is similar to the first one. The only difference is that your class now implements both interfaces; the FilterCoffeeMachine and the EspressoCoffeeMachine.
3. The new coffee machine is completely different to the other two. Maybe it’s one of these pad machines that you can also use to make tea or other hot drinks. In this case, you need to create a new interface and decide if you want to extend the CoffeeMachine interface. In the example of the pad machine, you shouldn’t do that because you can’t add ground coffee to a pad machine. So, your PadMachineclass shouldn’t need to implement an addGroundCoffee method.
4. The new coffee machine provides new functionality, but you can also use it to brew a filter coffee or an espresso. In that case, you should define a new interface for the new functionality. Your implementation class can then implement this new interface and one or more of the existing interfaces. But please make sure to segregate the new interface from the existing ones, as you did for the FilterCoffeeMachine and the EspressoCoffeeMachine interfaces.

### Example 3:

There are few common methods to manage the master records exists in AdminPersistenceService<T> which is implemented by DefaultGuestPersistenceService, DefaultMemberPersistenceService etc.

**public** **interface** AdminPersistenceService<T> {

**public** T saveRecord(T model);

**public** T updateRecord(T model);

**public** List<T> fetchAllRecords();

**public** **void** deleteRecordById(String id);

**public** T findRecordByKey(String id);

**public** List<T> saveRecords(List<T> records);

**public** **void** deleteRecords(List<T> records);

}

Now there is a class UserPersistenceService which requires few new methods plus the method exists AdminPersistenceService.

So I add those methods in AdminPersistenceService and not introducing UserPersistenceService interface then it would be compulsory to implement those methods or leave it blank for DefaultGuestPersistenceService, DefaultMemberPersistenceService etc. and In that case Clients which are using DefaultGuestPersistenceService , DefaultMemberPersistenceService etc. are not interested in those methods. So that’s why we introduced new Interface UserPersistenceService.

**public** **interface** UserPersistenceService<T> **extends** AdminPersistenceService<T>{

**public** User findUserDetailsByUserId(String userid);

**public** **void** changePassword(String userId,String password);

}

# Dependency Inversion Principle

Most of us are already familiar with the words used in principle’s name. It says:

**" Dependency Inversion Principle says that Depend on abstractions, not on concretions"**

**“Dependency Inversion Principle states that High level modules should not depend on low level modules. Both should depend on abstraction.”**

**“Dependency Inversion Principle states that instead of lower module defines an interface, higher level module defines an interface which is implemented by low level modules. Hence inverting the dependency control (from letting the class control them to letting the caller control them).”**

**“Abstractions should be owned by higher-level modules and implemented by lower-level modules.”**

**“Dependency Inversion Principle introduce an abstraction that decouples the high-level and low-level modules from each other.”**

**So, in the end, you get two dependencies:**

1. **the high-level module depends on the abstraction, and**
2. **the low-level depends on the same abstraction.**

The idea is that we isolate our class behind a boundary formed by the abstractions it depends on. If all the details behind those abstractions change, then our class is still safe. This helps keep coupling low and makes our design easier to change. DIP also allows us to test things in isolation; details like database are plugins to our system.

In other words, you should design your software in such a way that various modules can be separated from each other using an abstract layer to bind them together.

### Example 1:

The classical use of this principle of [BeanFactory](http://howtodoinjava.com/2013/03/20/different-spring-3-ioc-containers-with-example/) in [spring framework](http://howtodoinjava.com/java-spring-framework-tutorials/). In spring framework, all modules are provided as separate components which can work together by simply injected dependencies in other module. They are so well closed in their boundaries that you can use them in other software modules apart from spring with same ease.

This has been achieved by **dependency inversion** and **open closed principles**. All modules expose only abstraction which is useful in extending the functionality or plugin in another module.

### Example 2:

#### Based on the other SOLID principles

If you consequently apply the Open/Closed Principle and the Liskov Substitution Principle to your code, it will also follow the Dependency Inversion Principle.

The Open/Closed Principle required a software component to be open for extension, but closed for modification. You can achieve that by introducing interfaces for which you can provide different implementations. The interface itself is closed for modification, and you can easily extend it by providing a new interface implementation.

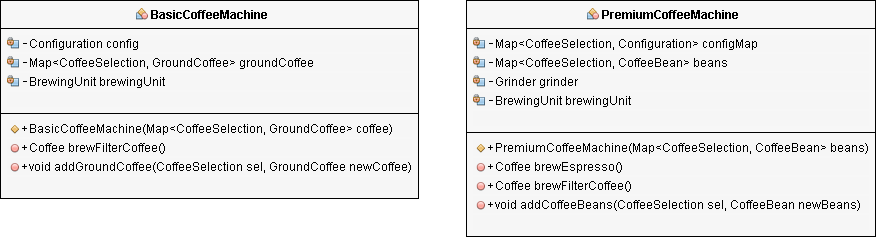
Your implementations should follow the Liskov Substitution Principle so that you can replace them with other implementations of the same interface without breaking your application.

Let’s take a look at the CoffeeMachine project in which I will apply all three of these design principles.

#### Brewing coffee with the Dependency Inversion Principle

You can buy lots of different coffee machines. Rather simple ones that use water and ground coffee to brew filter coffee, and premium ones that include a grinder to freshly grind the required amount of coffee beans and which you can use to brew different kinds of coffee.

If you build a coffee machine application that automatically brews you a fresh cup of coffee in the morning, you can model these machines as a *BasicCoffeeMachine* and a *PremiumCoffeeMachine* class.



#### Implementing the *BasicCoffeeMachine*

The implementation of the *BasicCoffeeMachine* is quite simple. It only implements a constructor and two public methods. You can call the *addGroundCoffee* method to refill ground coffee, and the *brewFilterCoffee* method to brew a cup of filter coffee.

import java.util.Map;

public class BasicCoffeeMachine implements CoffeeMachine {

private Configuration config;

private Map<CoffeeSelection, GroundCoffee> groundCoffee;

private BrewingUnit brewingUnit;

public BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee).

this.groundCoffee = coffee;

this.brewingUnit = new BrewingUnit();

this.config = new Configuration(30, 480);

}

@Override

public Coffee brewFilterCoffee() {

// get the coffee

GroundCoffee groundCoffee = this.groundCoffee.get(CoffeeSelection.FILTER\_COFFEE);

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.FILTER\_COFFEE, groundCoffee, this.config.getQuantityWater());

}

public void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee) throws CoffeeException {

GroundCoffee existingCoffee = this.groundCoffee.get(sel);

if (existingCoffee != null) {

if (existingCoffee.getName().equals(newCoffee.getName())) {

existingCoffee.setQuantity(existingCoffee.getQuantity() + newCoffee.getQuantity())

} else {

throw new CoffeeException("Only one kind of coffee supported for each CoffeeSelection.")

}

} else {

this.groundCoffee.put(sel, newCoffee)

}

}

}

#### Implementing the *PremiumCoffeeMachine*

The implementation of the PremiumCoffeeMachine class looks very similar. The main differences are:

* It implements the addCoffeeBeans method instead of the addGroundCoffeemethod.
* It implements the additional brewEspresso method.

The brewFilterCoffee method is identical to the one provided by the BasicCoffeeMachine.

import java.util.HashMap;

import java.util.Map;

public class PremiumCoffeeMachine {

private Map<CoffeeSelection, Configuration> configMap;

private Map<CoffeeSelection, CoffeeBean> beans;

private Grinder grinder

private BrewingUnit brewingUnit;

public PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {

this.beans = beans;

this.grinder = new Grinder();

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap<>();

this.configMap.put(CoffeeSelection.FILTER\_COFFEE, new Configuration(30, 480));

this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

}

public Coffee brewEspresso() {

Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.ESPRESSO),

config.getQuantityCoffee())

// brew an espresso

return this.brewingUnit.brew(CoffeeSelection.ESPRESSO, groundCoffee,

config.getQuantityWater());

}

public Coffee brewFilterCoffee() {

Configuration config = configMap.get(CoffeeSelection.FILTER\_COFFEE);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.FILTER\_COFFEE),

config.getQuantityCoffee());

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.FILTER\_COFFEE, groundCoffee,

config.getQuantityWater());

}

public void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans) throws CoffeeException {

CoffeeBean existingBeans = this.beans.get(sel);

if (existingBeans != null) {

if (existingBeans.getName().equals(newBeans.getName())) {

existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity());

} else {

throw new CoffeeException("Only one kind of coffee supported for each CoffeeSelection.");

}

} else {

this.beans.put(sel, newBeans);

}

}

}

To implement a class that follows the Dependency Inversion Principle and can use the BasicCoffeeMachine or the PremiumCoffeeMachine class to brew a cup of coffee, you need to apply the Open/Closed and the Liskov Substitution Principle. That requires a small refactoring during which you introduce interface abstractions for both classes.

#### Introducing abstractions

The main task of both coffee machine classes is to brew coffee. But they enable you to brew different kinds of coffee. If you use a *BasicCoffeeMachine*, you can only brew filter coffee, but with a *PremiumCoffeeMachine*, you can brew filter coffee or espresso. So, which interface abstraction would be a good fit for both classes?

As all coffee lovers will agree, there are huge [differences between filter coffee and espresso](https://www.perfectdailygrind.com/2017/07/espresso-vs-filter-whats-difference/). That’s why we are using different machines to brew them, even so, some machines can do both. I, therefore, suggest to create two independent abstractions:

* The FilterCoffeeMachine interface defines the Coffee brewFilterCoffee() method and gets implemented by all coffee machine classes that can brew a filter coffee.
* All classes that you can use to brew an espresso, implement the EspressoMachineinterface, which defines the Coffee brewEspresso() method.

As you can see in the following code snippets, the definition of both interface is pretty simple.

public interface CoffeeMachine {

Coffee brewFilterCoffee();

}

public interface EspressoMachine {

Coffee brewEspresso();

}

In the next step, you need to refactor both coffee machine classes so that they implement one or both of these interfaces.

#### Refactoring the BasicCoffeeMachine class

Let’s start with the BasicCoffeeMachine class. You can use it to brew a filter coffee, so it should implement the CoffeeMachine interface. The class already implements the brewFilterCoffee() method. You only need to add implements CoffeeMachine to the class definition.

public class BasicCoffeeMachine implements CoffeeMachine {

private Configuration config;

private Map<CoffeeSelection, GroundCoffee> groundCoffee;

private BrewingUnit brewingUnit;

public BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee) {

this.groundCoffee = coffee;

this.brewingUnit = new BrewingUnit();

this.config = new Configuration(30, 480);

}

@Override

public Coffee brewFilterCoffee() {

// get the coffee

GroundCoffee groundCoffee = this.groundCoffee.get(CoffeeSelection.FILTER\_COFFEE);

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.FILTER\_COFFEE, groundCoffee, this.config.getQuantityWater());

}

public void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee) throws CoffeeException {

GroundCoffee existingCoffee = this.groundCoffee.get(sel);

if (existingCoffee != null) {

if (existingCoffee.getName().equals(newCoffee.getName())) {

existingCoffee.setQuantity(existingCoffee.getQuantity() + newCoffee.getQuantity());

} else {

throw new CoffeeException("Only one kind of coffee supported for each CoffeeSelection.");

}

} else {

this.groundCoffee.put(sel, newCoffee);

}

}

#### **Refactoring the***PremiumCoffeeMachine***class**

The refactoring of the PremiumCoffeeMachine also doesn’t require a lot of work. You can use the coffee machine to brew filter coffee and espresso, so the PremiumCoffeeMachine class should implement the CoffeeMachine and the EspressoMachine interfaces. The class already implements the methods defined by both interfaces. You just need to declare that it implements the interfaces.

import java.util.HashMap;

import java.util.Map;

public class PremiumCoffeeMachine implements CoffeeMachine, EspressoMachine {

private Map<CoffeeSelection, Configuration> configMap;

private Map<CoffeeSelection, CoffeeBean> beans;

private Grinder grinder;

private BrewingUnit brewingUnit;

public PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {

this.beans = beans;

this.grinder = new Grinder();

this.brewingUnit = new BrewingUnit();

this.configMap = new HashMap<>();

this.configMap.put(CoffeeSelection.FILTER\_COFFEE, new Configuration(30, 480));

this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));

}

@Override

public Coffee brewEspresso() {

Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.ESPRESSO),

config.getQuantityCoffee());

// brew an espresso

return this.brewingUnit.brew(CoffeeSelection.ESPRESSO, groundCoffee,

config.getQuantityWater());

}

@Override

public Coffee brewFilterCoffee() {

Configuration config = configMap.get(CoffeeSelection.FILTER\_COFFEE);

// grind the coffee beans

GroundCoffee groundCoffee = this.grinder.grind(

this.beans.get(CoffeeSelection.FILTER\_COFFEE),

config.getQuantityCoffee());

// brew a filter coffee

return this.brewingUnit.brew(CoffeeSelection.FILTER\_COFFEE,

groundCoffee,config.getQuantityWater());

}

public void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans) throws CoffeeException {

CoffeeBean existingBeans = this.beans.get(sel);

if (existingBeans != null) {

if (existingBeans.getName().equals(newBeans.getName())) {

existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity());

} else {

throw new CoffeeException("Only one kind of coffee supported for each CoffeeSelection.");

}

} else {

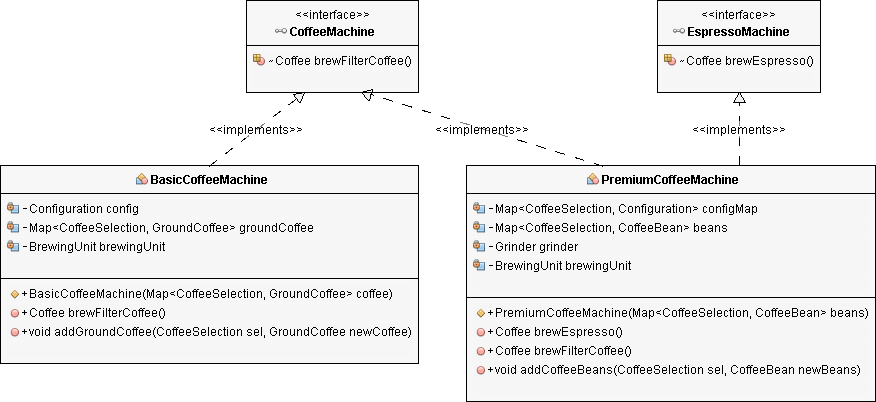
this.beans.put(sel, newBeans);

}

}

}

The BasicCoffeeMachine and the PremiumCoffeeMachine classes now follow the Open/Closed and the Liskov Substitution principles. The interfaces enable you to add new functionality without changing any existing code by adding new interface implementations. And by splitting the interfaces into CoffeeMachine and EspressoMachine, you separate the two kinds of coffee machines and ensure that all CoffeeMachine and EspressMachine implementations are interchangeable.



#### **Implementing the coffee machine application**

You can now create additional, higher-level classes that use one or both of these interfaces to manage coffee machines without directly depending on any specific coffee machine implementation.

As you can see in the following code snippet, due to the abstraction of the CoffeeMachine interface and its provided functionality, the implementation of the CoffeeApp is very simple. It requires a CoffeeMachine object as a constructor parameter and uses it in the prepareCoffee method to brew a cup of filter coffee.

public class CoffeeApp {

private CoffeeMachine coffeeMachine;

public CoffeeApp(CoffeeMachine coffeeMachine) {

this.coffeeMachine = coffeeMachine

}

public Coffee prepareCoffee(CoffeeSelection selection

throws CoffeeException {

Coffee coffee = this.coffeeMachine.brewFilterCoffee();

System.out.println("Coffee is ready!");

return coffee;

}

}

The only code that directly depends on one of the implementation classes is the CoffeeAppStarter class, which instantiates a CoffeeApp object and provides an implementation of the CoffeeMachine interface. You could avoid this compile-time dependency entirely by using a dependency injection framework, like [Spring](https://spring.io/) or [CDI](http://cdi-spec.org/), to resolve the dependency at runtime.

import java.util.HashMap;

import java.util.Map;

public class CoffeeAppStarter {

public static void main(String[] args) {

// create a Map of available coffee beans

Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelection, CoffeeBean>();

beans.put(CoffeeSelection.ESPRESSO, new CoffeeBean(

"My favorite espresso bean", 1000));

beans.put(CoffeeSelection.FILTER\_COFFEE, new CoffeeBean(

"My favorite filter coffee bean", 1000))

// get a new CoffeeMachine object

PremiumCoffeeMachine machine = new PremiumCoffeeMachine(beans);

// Instantiate CoffeeApp

CoffeeApp app = new CoffeeApp(machine);

// brew a fresh coffee

try {

app.prepareCoffee(CoffeeSelection.ESPRESSO);

} catch (CoffeeException e) {

e.printStackTrace();

}

}

}

### Example 3:

In our customer class if you remember we had created a logger class to satisfy SRP. Down the line let’s say new Logger flavor classes are created.

class Customer

{

private FileLogger obj = new FileLogger();

public virtual void Add()

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

obj.Handle(ex.ToString());

}

}

}

Just to control things we create a common interface and using this common interface new logger flavors will be created.

interface ILogger

{

void Handle(string error);

}

Below are three logger flavors and more can be added down the line.

class FileLogger : ILogger

{

public void Handle(string error)

{

System.IO.File.WriteAllText(@"c:\Error.txt", error);

}

}

class EverViewerLogger : ILogger

{

public void Handle(string error)

{

*// log errors to event viewer*

}

}

class EmailLogger : ILogger

{

public void Handle(string error)

{

*// send errors in email*

}

}

Now depending on configuration settings different logger classes will used at given moment. So to achieve the same we have kept a simple IF condition which decides which logger class to be used, see the below code.

class Customer : IDiscount, IDatabase

{

private IException obj;

public virtual void Add(int Exhandle)

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

if (Exhandle == 1)

{

obj = new MyException();

}

else

{

obj = new EmailException();

}

obj.Handle(ex.Message.ToString());

}

}

The above code is again violating SRP but this time the aspect is different ,its about deciding which objects should be created. Now it’s not the work of “Customer” object to decide which instances to be created, he should be concentrating only on Customer class related functionalities.

If you watch closely the biggest problem is the “NEW” keyword. He is taking extra responsibilities of which object needs to be created.

So if we INVERT / DELEGATE this responsibility to someone else rather the customer class doing it that would really solve the problem to a certain extent.

So here’s the modified code with INVERSION implemented. We have opened the constructor mouth and we expect someone else to pass the object rather than the customer class doing it. So now it’s the responsibility of the client who is consuming the customer object to decide which Logger class to inject.

class Customer : IDiscount, IDatabase

{

private Ilogger obj;

public Customer(ILogger i)

{

obj = i;

}

}

So now the client will inject the Logger object and the customer object is now free from those IF condition which decide which logger class to inject. This is the Last principle in SOLID Dependency Inversion principle.

Customer class has delegated the dependent object creation to client consuming it thus making the customer class concentrate on his work.

IDatabase i = new Customer(new EmailLogger());