SOLID
Dependency Inversion Principle

Upcode Software
Engineer Team

-2023-

CONTENT

1. What is DIP?
2. Why needs to DIP?
3. How to use DIP?
4. Where to use DIP?
5. SOURCE code (github)

1. What is Dependency Inversion Principle(DIP) ?

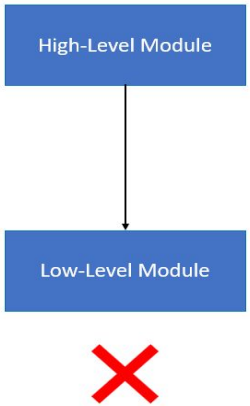
- **Robert Martin** introduced them in the book Agile Software Development, Principles, Patterns, and Practices
- **SOLID** is a mnemonic for five design principles intended to make software designs more understandable, flexible and maintainable.

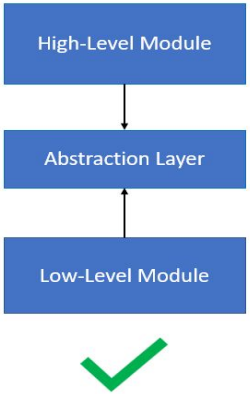
Dependency Inversion principle states:

- High-level modules should **not depend on low-level modules**. Both should depend on **abstractions**. (**First defined by Robert C. Martin**)

2. Why needs to DIP ?

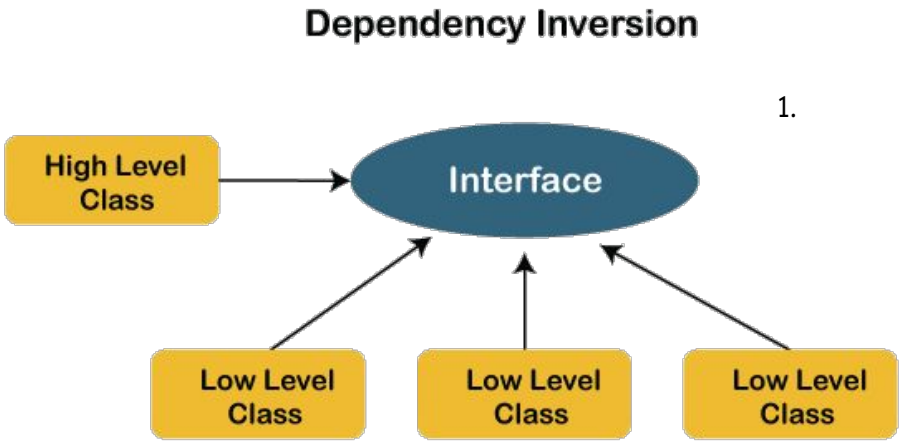
- High-level modules should **not depend on low-level modules**. Both should depend on **abstractions**.
- **Abstractions** should not depend on details. **Details** should depend on abstractions.





3. How to use DIS ? (1/n)

- **High-level classes** shouldn't depend on low-level classes.
- Both should depend **on abstractions.**
- **Abstractions** shouldn't depend on details. Details should depend on abstractions.

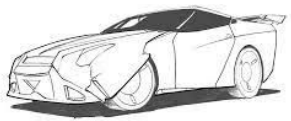


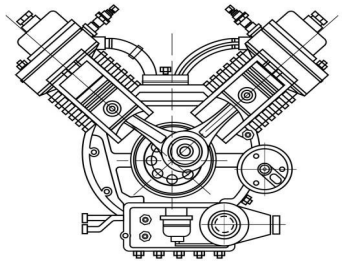
3. How to use DIS ? (2/n)

- We have a Car class that depends on the concrete Engine class; therefore, it is not obeying DIP.

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
```

```
public class Engine {
    public void start() {...}
}
```





4. Where to use DIP ? (1/n)

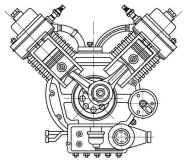
- What if we wanted to add another engine type, let's say a diesel engine? This will require refactoring the Car class.
- However, we can solve this by introducing a layer of abstraction. Instead of Car depending directly on **Engine**, let's add an interface
- We can connect any type of **Engine** that implements the Engine interface to the Car class

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
```

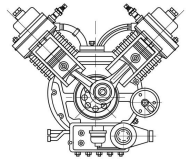
```
public class PetrolEngine implements Engine {
    public void start() {...}
}
```

```
public class DieselEngine implements Engine {
    public void start() {...}
}
```

Petrol Engine

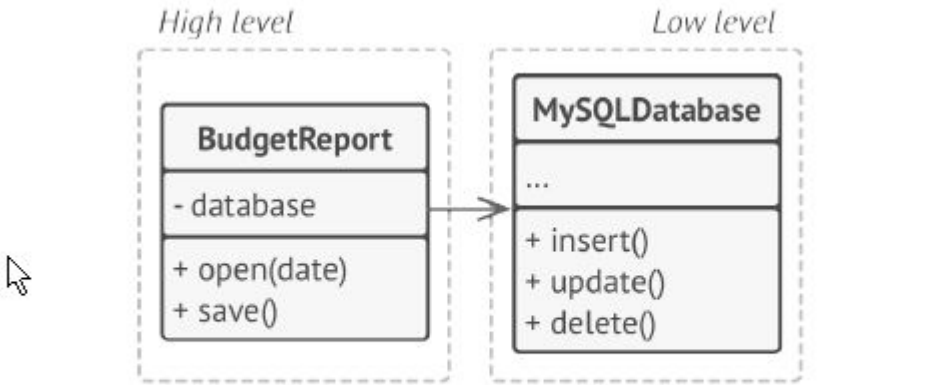


Diesel Engine



4. Where to use DIP ? (2/n)

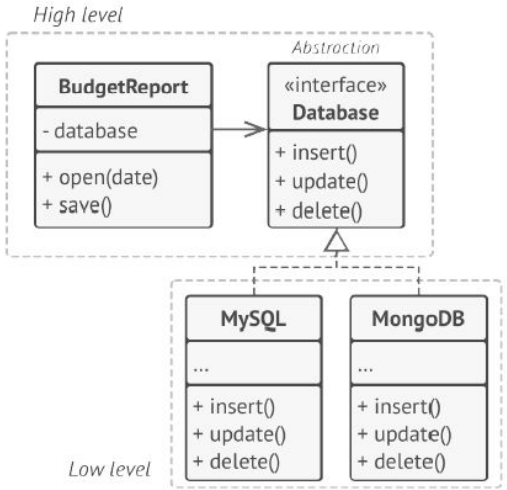
- The high-level budget reporting class uses a low-level database class for reading and persisting its data.



BEFORE: a high-level class depends on a low-level class.

4. Where to use DIP ? (3/n)

- You can fix this problem by creating a high-level interface that describes read/write operations and making the reporting
- class use that interface instead of the low-level class.

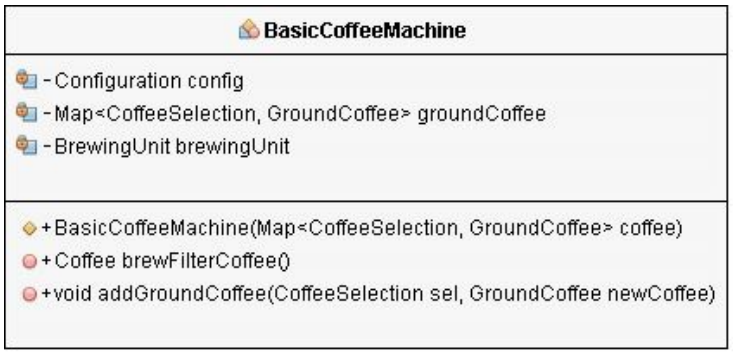


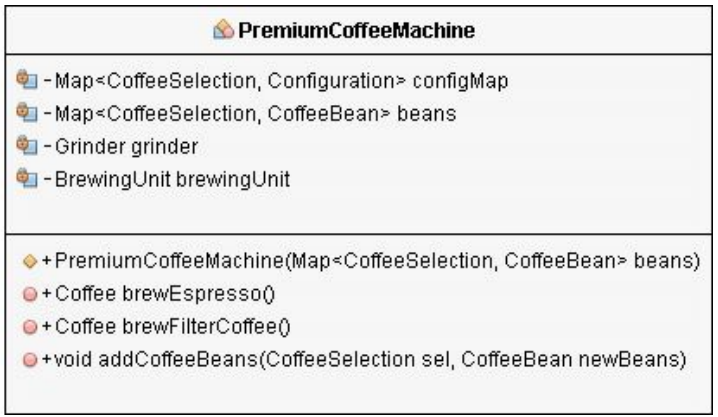
AFTER: low-level classes depend on a high-level abstraction.

5. SOURCE code (1/n)

- **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.

UML Diagram





5. SOURCE code (2/n) - Basic Coffee Machine

```
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.getNa
me())) {

existingCoffee.setQuantity(existingCoffee.getQuanti
ty() + newCoffee.getQuantity())
            } else {
                throw new CoffeeException("Only one
kind of coffee supported for each CoffeeSelection.")
            }
        } else {
            this.groundCoffee.put(sel, newCoffee)
        }
    }
```

5. SOURCE code (3/n) - Premium Coffee Machine

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

- It implements the ***addCoffeeBeans*** method instead of the *addGroundCoffee* method.
- It implements the additional ***brewEspresso*** method.

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

5. SOURCE code (4/n) - Premium Coffee Machine

```
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());
    }
```

5. SOURCE code (5/n) - Premium Coffee Machine

```
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);
        }
    }
```

5. SOURCE code (6/n) - Abstraction

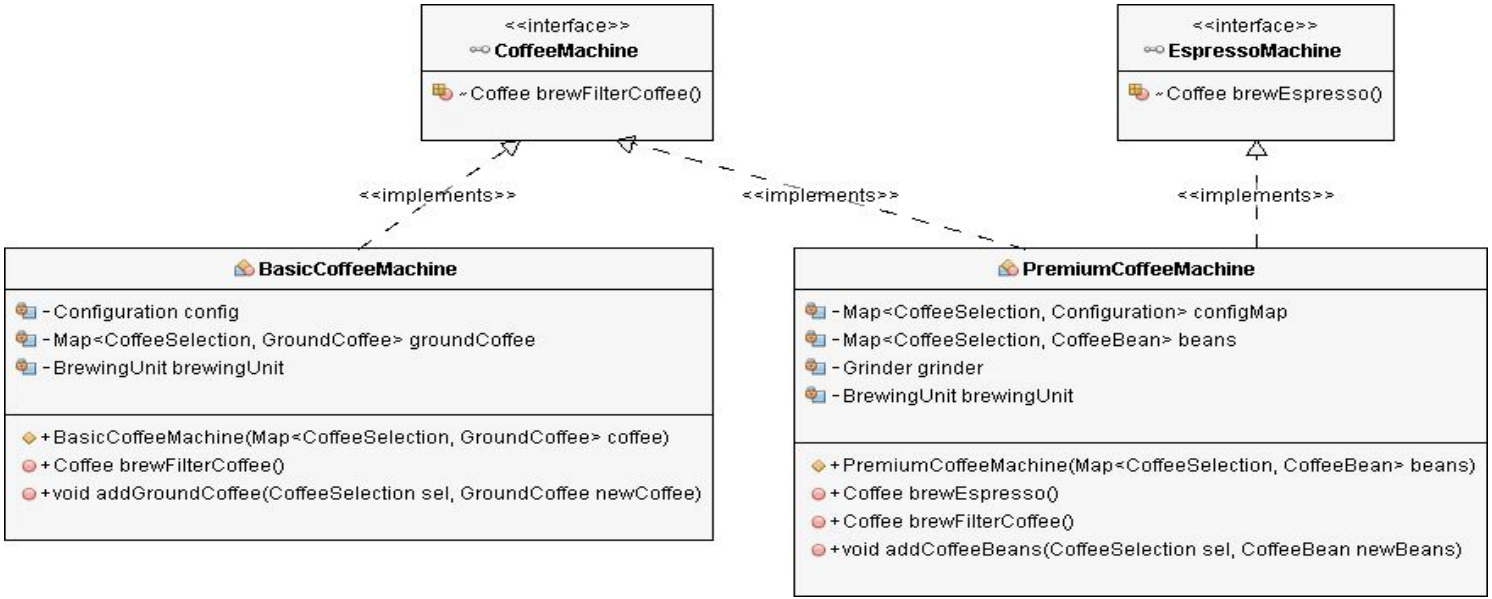
- The main task of both coffee machine classes is to brew coffee.
- But they enable you to brew different kinds of coffee. If you use a *B**asicCoffeeMachine***, you can only brew filter coffee, but with a ***PremiumCoffeeMachine***, you can brew filter coffee or espresso. So, which interface abstraction would be a good fit for both classes?
- As all coffee lovers will agree, there are huge differences between filter coffee and espresso.
- 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:

5. SOURCE code (7/n) - Abstraction

```
public interface CoffeeMachine {
    Coffee brewFilterCoffee();
}

public interface EspressoMachine {
    Coffee brewEspresso();
}
```

5. SOURCE code (8/n) - Abstraction



Summary

- **The Dependency Inversion Principle** is the fifth and final design principle that

  It introduces an interface abstraction between **higher-level and lower-level software components to remove the dependencies between t**hem.

- As you have seen in the example project, you only need to consequently apply the **Open/Closed and the Liskov Substitution principles** to your code base.
- After you have done that, your classes also comply with the Dependency Inversion Principle.
- This enables you to change **higher-level and lower-level components** without affecting any other classes, as long as you don’t change any interface abstractions.

Reference Resources

1. **Dive into design pattern compression (book)**
2. SOLID Design Principles Explained: Dependency Inversion Principle with
3. What is SOLID? Principles for Better Software Design
4. Code Examples

Thank you!

Presented by **Hamdamboy Urunov**

(hamdamboy.urunov@gmail.com)