OOP Design Principles

SOLID

COMP3607
Object Oriented Programming II

Week 2

Outline

- SOLID Design Principles
  - Single Responsibility Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion

SOLID Design Principles

SOLID is one of the most popular sets of design principles in object-oriented software development. It’s a mnemonic acronym for the following five design principles:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion

Single Responsibility Principle

https://deviq.com/single-responsibility-principle/

Single Responsibility Principle

“A class should have one, and only one, reason to change.” ” - Robert C. Martin

The SRP makes your software easier to implement and prevents unexpected side-effects of future changes.

It also makes your software easier to understand.

https://stackify.com/solid-design-principles/

Single Responsibility Principle

To follow this principle, your class isn’t allowed to have more than one responsibility, e.g., the management of entities or the conversion of data types. This avoids any unnecessary, technical coupling between responsibilities and reduces the probability that you need to change your class. It also lowers the complexity of each change because it reduces the number of dependent classes that are affected by it.

Frequency and effects of changes

Requirements change over time. Each requirement 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.

This affects all classes or components that depend on the changed class.

Depending on your change, you might need to update the dependencies or recompile the dependent classes even though they are not directly affected by your change. They only use one of the other responsibilities implemented by your class, but you need to update them anyway.

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.

Avoiding these problems

You can avoid these problems by asking a simple question before you make any changes: What is the responsibility of your class/component/microservice?

If your answer includes the word “and”, you’re most likely breaking the single responsibility principle. Then it’s better take a step back and rethink your current approach. There is most likely a better way to implement it.

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.

2 min

Exercise

Single Responsibilities

Some examples of responsibilities to consider that may need to be separated include:

- Persistence
- Validation
- Notification
- Error Handling
- Logging
- Class Selection / Instantiation
- Formatting
- Parsing
- Mapping

Example: JPA
Java Persistence API

The Java Persistence API (JPA) specification* 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.

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.

*Renamed in 2019 to Jakarta Persistence 2.2: https://jakarta.ee/specifications/persistence/

Example: JPA AttributeConverter

The responsibility of an AttributeConverter in the JPA 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..

Example: JPA AttributeConverter

```
@Converter(autoApply = true)
public class DurationConverter implements AttributeConverter<Duration, Long
> {
    @Override
    public Long convertToDatabaseColumn(Duration attribute) {
      return attribute.toNanos();
    }

    @Override
    public Duration convertToEntityAttribute(Long duration) {
        return Duration.of(duration, ChronoUnit.NANOS);
    }
}
```

2 min

Exercise

Open/Closed Principle

https://deviq.com/open-closed-principle/

Open/Closed Principle

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” - Robert C. Martin

The Issue with Inheritance

Write your code so that you will be able to add new functionality without changing the existing code.

This prevents situations in which a change to one of your classes also requires you to adapt all depending classes.

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

Use Interfaces

Using interfaces instead of superclasses allows 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.

Use Interfaces

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.

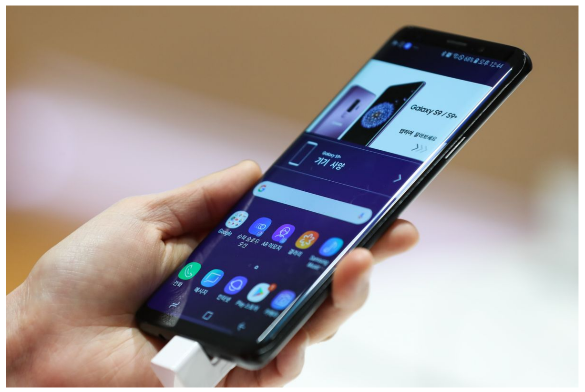
If you consider it beneficial that two implementations of an interface share some code, you can either use inheritance or composition.

2 min

Exercise

Example: Coffee Brewing

A simple application that controls a basic coffee machine to brew you a delicious filtered coffee in the morning.



https://stackify.com/solid-design-open-closed-principle/

Example: Coffee Brewing -Classes-

The BasicCoffeeMachine class

The implementation of the *BasicCoffeeMachine* class is relatively simple. It just has a constructor, a public method to add ground coffee, and a method, *brewCoffee*, that brews a filtered coffee.

The BasicCoffeeApp class

The main method of the *BasicCoffeeApp* prepares a Map with ground coffee, instantiates a *BasicCoffeeMachine* object, and calls the *brewCoffee* method to make the coffee.

https://stackify.com/solid-design-open-closed-principle/

2 min

Exercise

Example: Coffee Brewing -Accommodating Changes-

What happens when you replace your *BasicCoffeeMachine*? You might get a better one with an integrated grinder, which can brew more than just filter coffee. Unfortunately, the *CoffeeApp* doesn’t support this kind of coffee machine.

It would be great if your app could control both types of coffee machines. But that will require a few code changes. And as you’re already on it, why not change it so that you will not need to adapt it to future coffee machines.

https://stackify.com/solid-design-open-closed-principle/

Example: Coffee Brewing
Extracting the *CoffeeMachine* interface

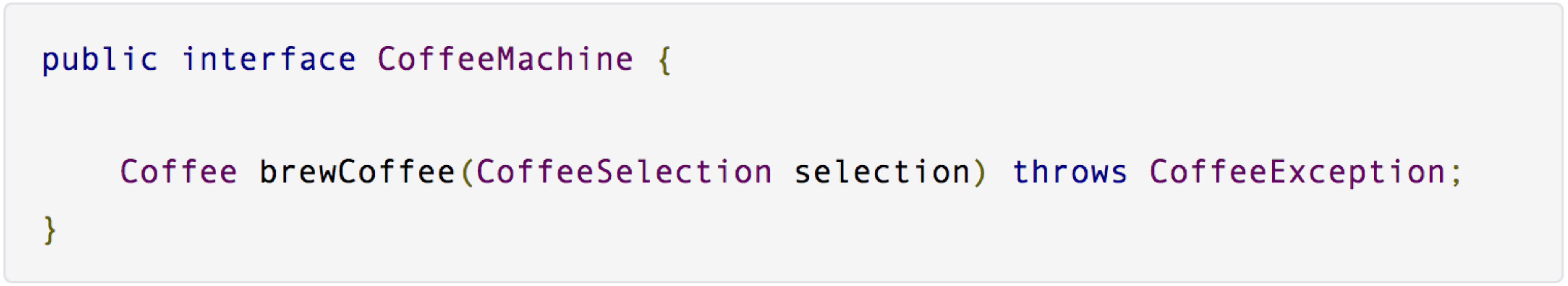
Following the Open/Closed Principle, you need to extract an **interface** that enables you to control the coffee machine. That's often the critical part of the refactoring.

You need to include the methods that are **mandatory** for controlling the coffee machine, but none of the optional methods which would limit the flexibility of the implementations

https://stackify.com/solid-design-open-closed-principle/

Example: Coffee Brewing
Extracting the *CoffeeMachine* interface

In this example, that's only the *brewCoffee* method. So, the *CoffeeMachine* interface specifies only one method, which needs to be implemented by all classes that implement it.



```
public interface CoffeeMachine {

    Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException;
}
```

https://stackify.com/solid-design-open-closed-principle/

Example: Coffee Brewing
Adapting the *BasicCoffeeMachine* class

In the next step, you need to adapt the *BasicCoffeeMachine* class.

It already implements the *brewCoffee* method and provides all the functionality it needs.

So, you just need to declare that the *BasicCoffeeMachine* class implements the *CoffeeMachine* interface.

```
public class BasicCoffeeMachine implements CoffeeMachine { ... }
```

2 min

Exercise

Example: Coffee Brewing
Add More Implementations

You can now add new implementations of the *CoffeeMachine* interface.

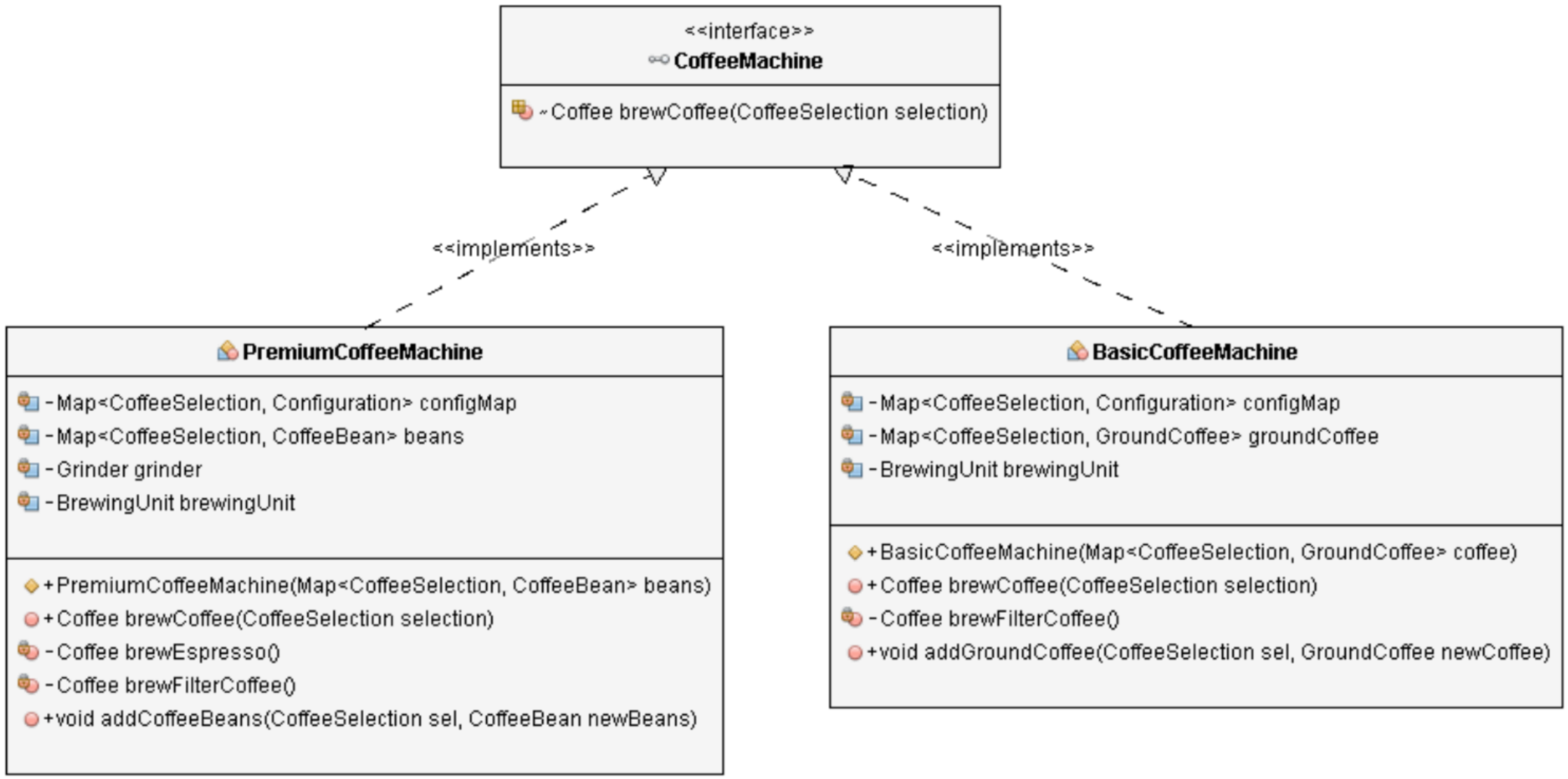
The implementation of the *PremiumCoffeeMachine* class is more complex than the *BasicCoffeeMachine* class. Its *brewCoffee* method, which is defined by the *CoffeeMachine* interface, supports two different *CoffeeSelections*.

Based on the provided *CoffeeSelection*, the method calls a separate, private method that brews the selected coffee.

In the implementation of these methods, the class also uses composition to reference a *Grinder*, which grinds the coffee beans before brewing the coffee.

https://stackify.com/solid-design-open-closed-principle/

Example: Coffee Brewing
Add More Implementations



<<interface>>
CoffeeMachine
~Coffee brewCoffee(CoffeeSelection selection)
<<implements>>
<<implements>>
PremiumCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, CoffeeBean> beans
-Grinder grinder
-BrewingUnit brewingUnit
+PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans)
+Coffee brewCoffee(CoffeeSelection selection)
-Coffee brewEspresso()
-Coffee brewFilterCoffee()
+void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans)
BasicCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, GroundCoffee> groundCoffee
-BrewingUnit brewingUnit
+BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee)
+Coffee brewCoffee(CoffeeSelection selection)
-Coffee brewFilterCoffee()
+void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee)

https://stackify.com/solid-design-open-closed-principle/

2 min

Exercise

References

- https://stackify.com/solid-design-principles/
- https://stackify.com/dependency-inversion-principle/
- https://deviq.com/solid/