

Java Best Practices & Performance

Venkata Ramana

Basic OO Principles

Abstraction

Abstraction is the process of hiding the implementation details of an object so that it can be used without understanding how it works. This allows you to create code that is easy to use and maintain.

For example, we could have a class called Vehicle with methods such as drive() and stop(). The details of how these methods work are hidden from the user, so they can simply call the methods and trust that they will work as expected.

Abstraction is important because it allows you to create code that is easy to use and understand. Abstraction allows the user to use the code without needing to know the details of how it works.

Encapsulation

Encapsulation is the process of hiding information within an object so that it cannot be accessed directly from outside the object. This allows you to control how data is used and prevents accidental modification of data.

Ex1: We could have a class called Person with attributes such as name and age. We could then create methods to get and set these attributes. This would allow you to control how the data is used, and you could add validation to ensure that the data is valid before it is set.

Ex2: The module descriptor (`module-info. java`)

Encapsulation is important because it helps to keep the data safe and secure. It also allows us to change the implementation of the code without affecting the rest of your codebase.

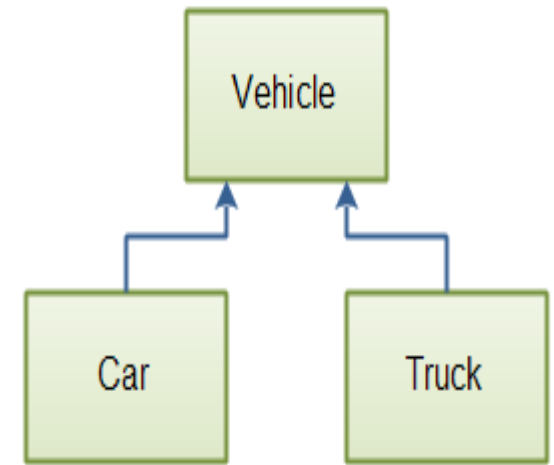
Inheritance

Inheritance is the ability of one class to inherit the attributes and methods of another class. This is useful because it allows you to create subclasses that are specialized versions of a parent class.

For example, we could have a parent class called Vehicle, with subclasses Car and Truck.

The Vehicle class would contain general attributes and methods that are common to all vehicles, such as the number of wheels and the color.

The Car and Truck subclasses would then each have their own unique attributes and methods, such as the number of doors and the size of the engine.



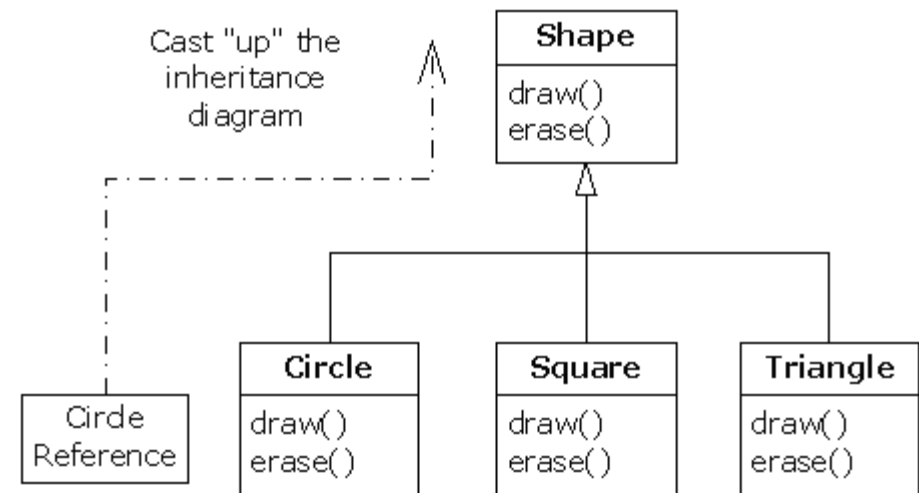
Inheritance is important in Object-Oriented Programming (OOP) because it allows for code reuse.

This means that we can write code once and then use it in multiple places, which makes your code more efficient and maintainable.

Polymorphism

Polymorphism is the ability of an object to take on multiple forms. This is useful because it allows you to create code that is more flexible and adaptable. For example, we could have a class called Shape with subclasses Circle and Rectangle. The Shape class would contain general methods such as getArea and getPerimeter. The Circle and Rectangle subclasses would then each have their own unique implementation of these methods.

Polymorphism is important because it allows you to write code that is more flexible and adaptable. Polymorphism allows you to write code that can be used with multiple types of objects.



Association in Java

Association in java, let us briefly explore the types of object relationships that can exist in OOPs. There can be two types of relationships in OOPs:

IS-A

HAS-A

IS-A (Inheritance)

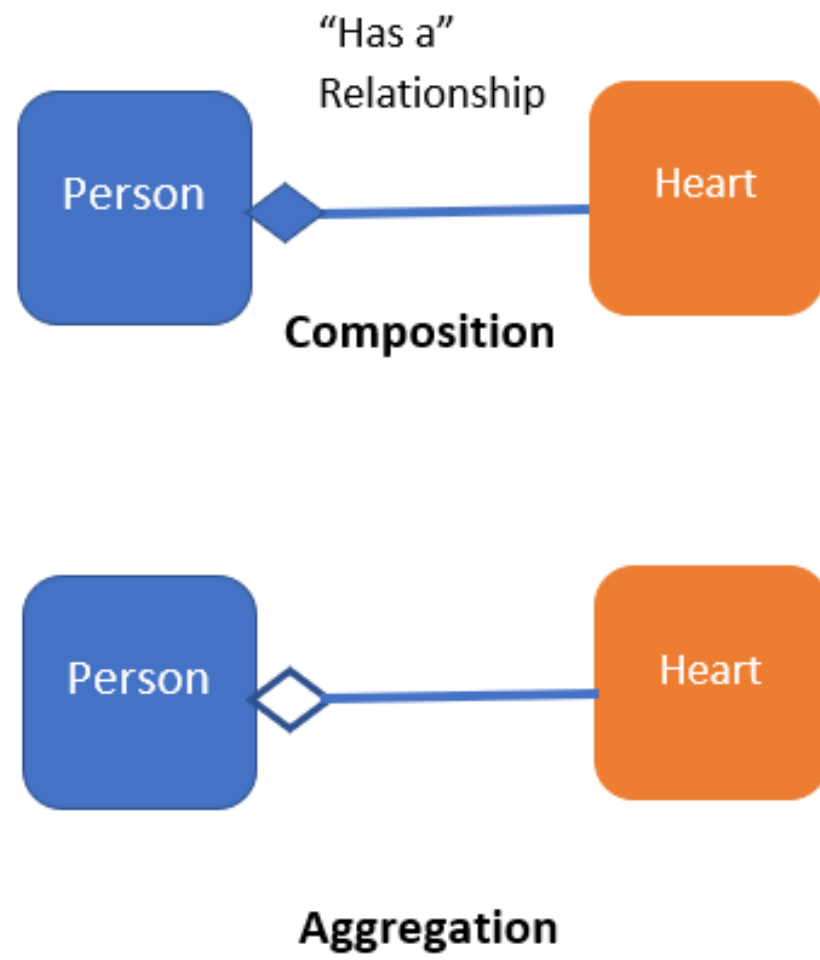
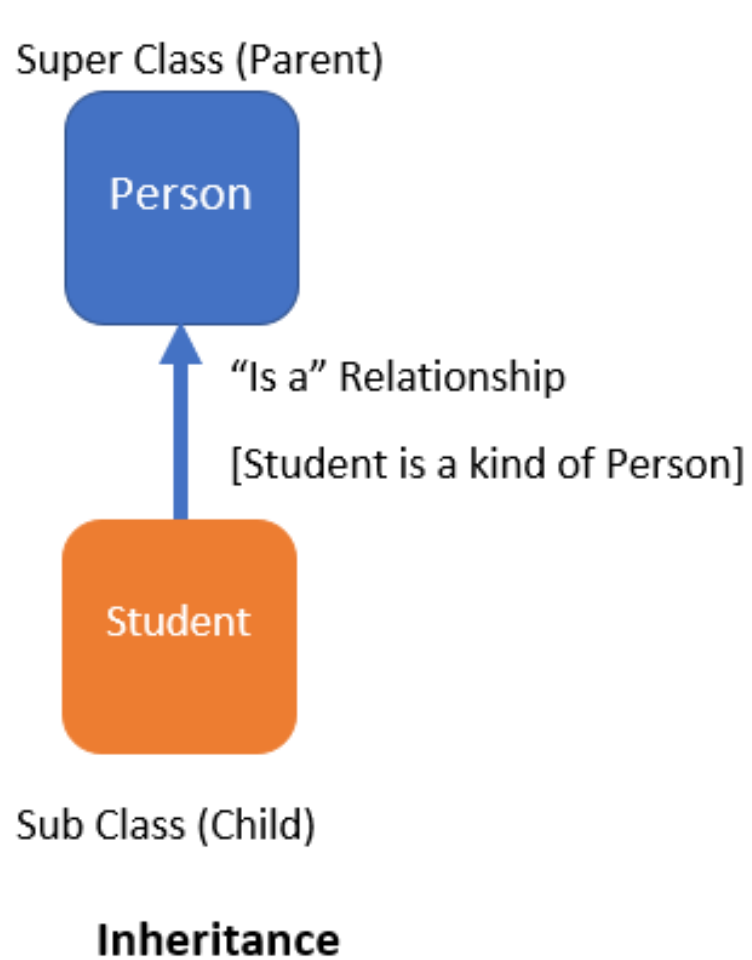
The IS-A relationship is nothing but Inheritance. The relationships that can be established between classes using the concept of inheritance are called IS-A relations.

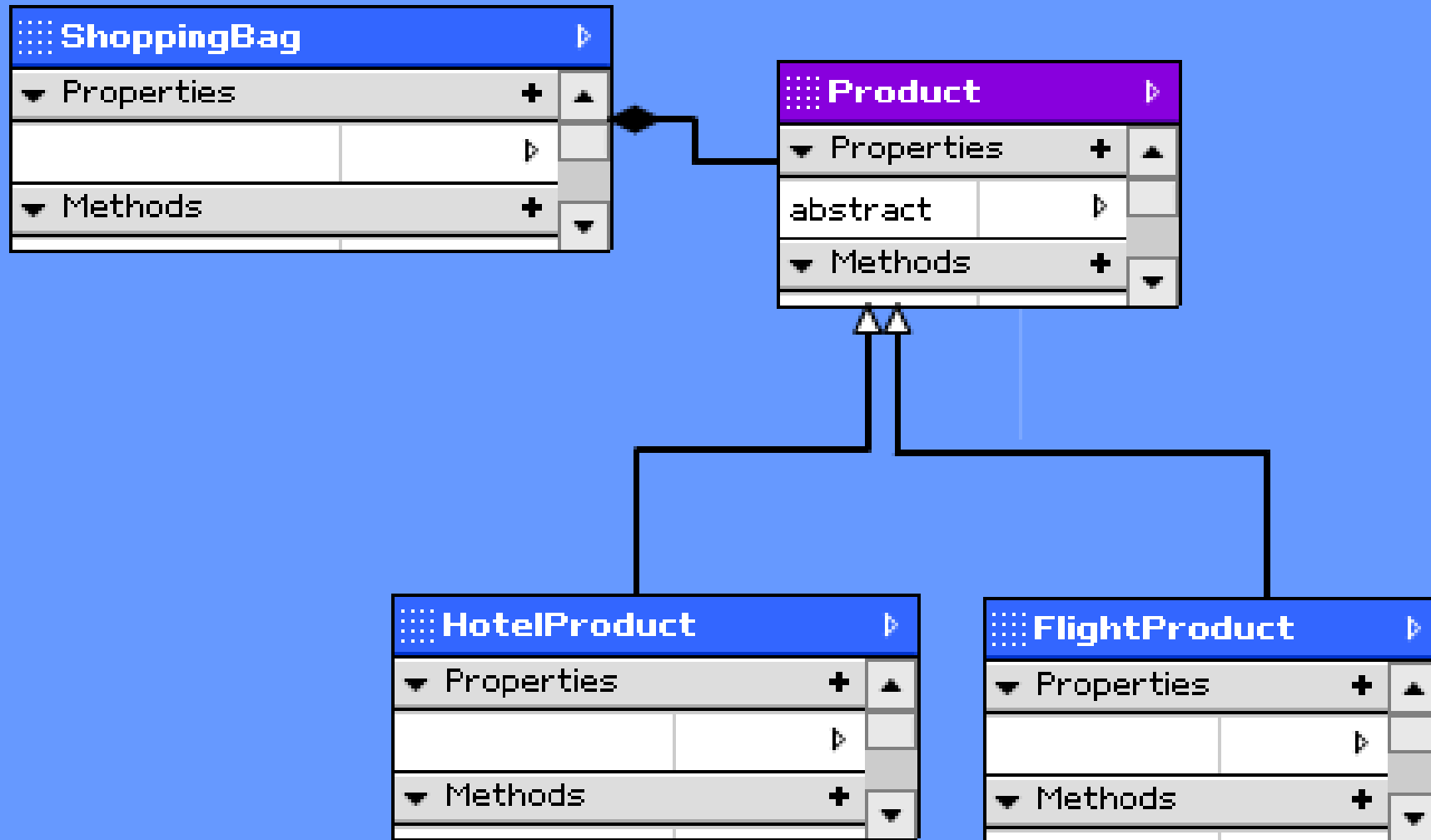
Ex: A parrot is-a Bird. Here Bird is a base class, and Parrot is a derived class, Parrot class inherits all the properties and attributes & methods (other than those that are private) of base class Bird, thus establishing inheritance(IS-A) relation between the two classes.

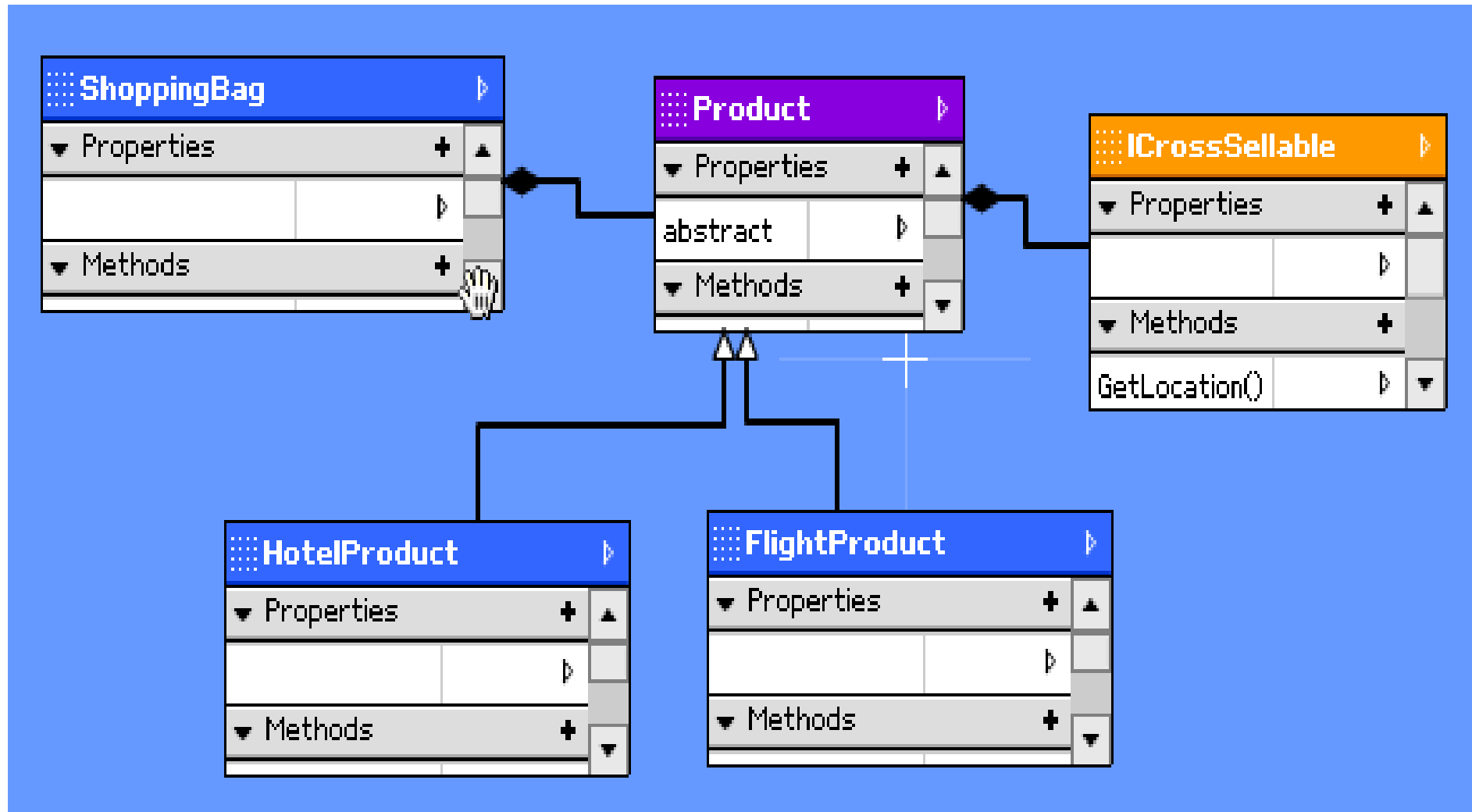
HAS-a (Association)

The HAS-A association on the other hand is where the Instance variables of a class refer to objects of another class. In other words, one class stores the objects of another class as its instance variables thereby establishing a HAS-A association between the two classes.

UML Diagram







There are two forms of Association that are possible in Java:

a) Aggregation

b) Composition

Aggregation:

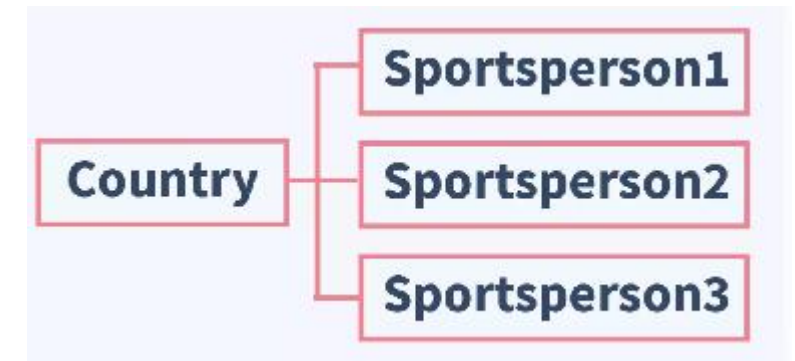
Aggregation in java is a form of HAS-A relationship between two classes. It is a relatively more loosely coupled relation than composition in that, although both classes are associated with each other, one can exist without the other independently. So Aggregation in java is also called a weak association. Let us look at a simple aggregation example to understand this better.

Example: Consider the association between a Country class and a Sportsperson class. Here's how it is defined

Country class is defined with a name and other attributes like size, population, capital, etc, and a list of all the Sportspersons that come from it.

A Sportsperson class is defined with a name and other attributes like age, height, weight, etc.

In a real-world context, we can infer an association between a country and a sports person that hails from that country. Modeling this relation to OOPs, a Country object has-a list of Sportsperson objects that are related to it. Note that a sportsperson object can exist with his own attributes and methods, alone without the association with the country object. Similarly, a country object can exist independently without any association to a sportsperson object. In, other words both Country and Sportsperson classes are independent although there is an association between them. Hence Aggregation is also known as a weak association.



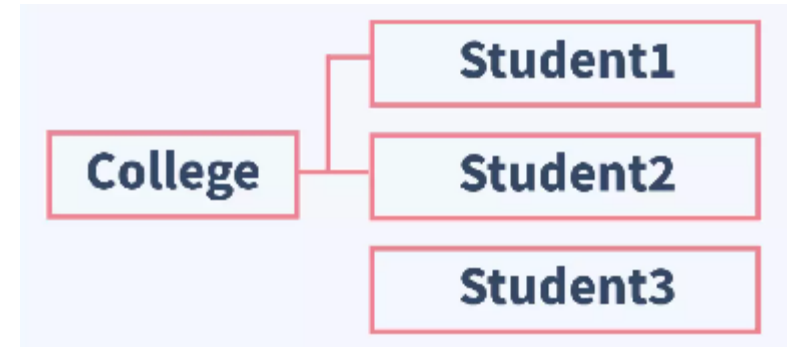
Composition:

Composition in java is a form of relation that is more tightly coupled. Composition in java is also called Strong association. This association is also known as Belongs-To association as one class, for all intents and purpose belongs to another class, and exists because of it. In a Composition association, the classes cannot exist independent of each other. If the larger class which holds the objects of the smaller class is removed, it also means logically the smaller class cannot exist. Let us explore this association clearly with an example

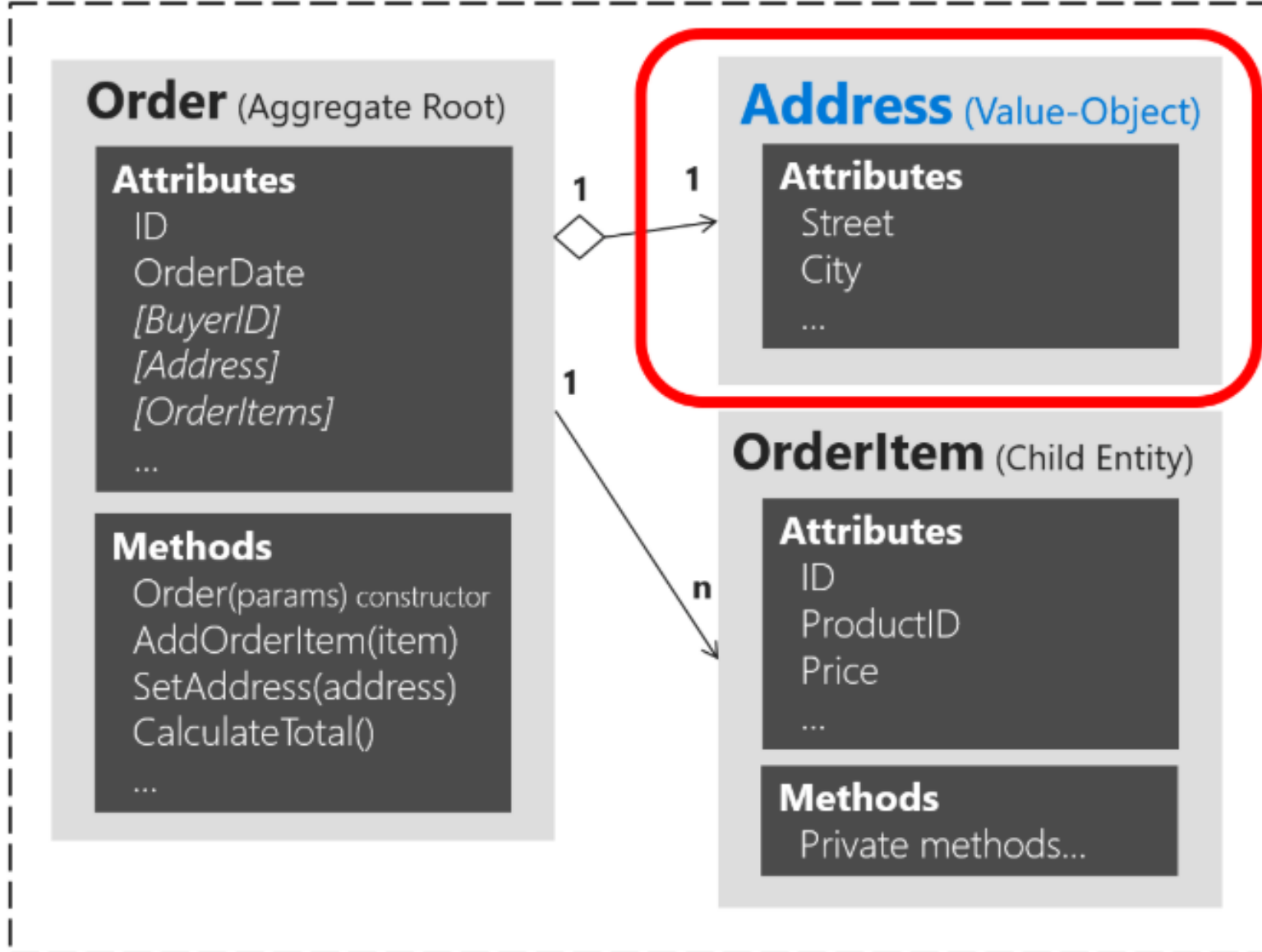
Example: The association between College and Student.
Below is how it is defined.

College class is defined with name and the list of students that are studying in it

A Student class is defined with name and the college he is studying at. Here a student must be studying in at least one college if he is to be called Student. If the college class is removed, Student class cannot exist alone logically, because if a person is not studying in any college then he is not a student.



Order Aggregate (Multiple entities and Value-Object)



❑ Specification

❑ Framework

❑ Pattern

Specification

Provides API , standards, recommended practices, codes
And technical publications, reports and studies.

JCP - Java Community Process

JSR - Java Specification Request

JSRs directly relate to one or more of the Java platforms.

There are 3 collections of standards that comprise the three Java editions:

Standard, Enterprise and Micro

Java EE (47 JSRs)

The Java Enterprise Edition offers APIs and tools for developing multitier enterprise applications.

Java SE (48 JSRs)

The Java Standard Edition offers APIs and tools for developing desktop and server-side enterprise applications.

Java ME (85 JSRs)

Java ME technology, Java Micro Edition, designed for embedded systems (mobile devices)

JSR 168,286,301 - Portlet Applications

JSR 127,254,314 - JSF

JSR 220 - Ejb3.0 & Jpa JSR 318 – EJB 3.1

JSR 340: Java Servlet 3.1 Specification

JSR 250 - Common Annotations for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws

JSR 311,370 - Jax-RS

JSR 299 - Context & DI

JSR 330 – DI

JSR-303 - Bean Validations

JSR208,312 – JBI (Java Business Integration)

A **framework** is a body of pre-written code that acts as a template or skeleton, which a developer can then use to create an application by filling in their own code as needed to get the app to work as they intend it to.

A framework is created to be used over and over so that developers can program their application without the manual overhead of creating every line of code from scratch.

Java frameworks are bodies of prewritten code used by developers to create apps using the Java programming language.

A Java framework is a type of framework specific to the Java programming language, used as a platform for developing software applications and Java programs.

Ex: Spring, Hibernate, Spring Boot, apache cxf/ eclipse jersey etc.,

Design Pattern

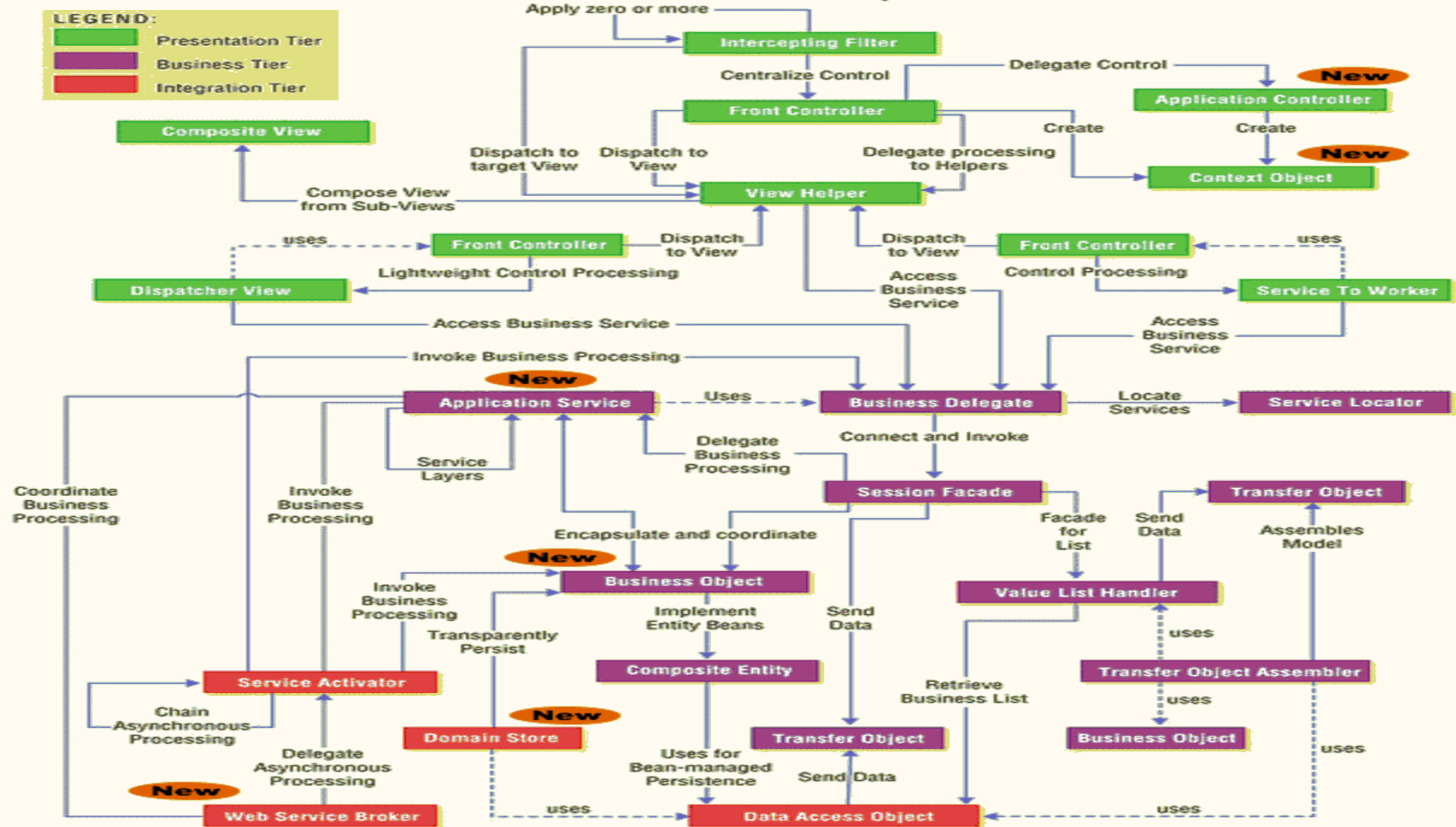
In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design

GOF Design Patterns

		Purpose		
		Creational	Structural	Behavioral
SCOPE	Class	Factory Method	Class Adapter	Interpreter Template Method
	OBJECT	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



Core J2EE Patterns, 2nd Edition

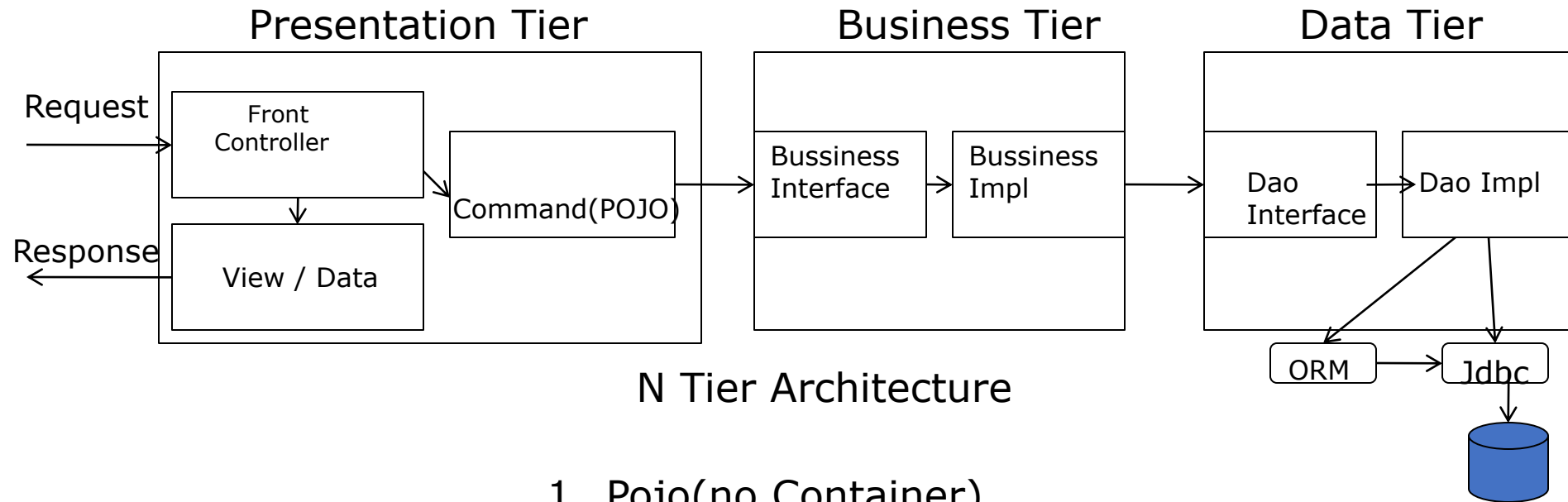


RESTful API Patterns

- ❑ Statelessness
- ❑ Content Negotiation
- ❑ URI Templates
- ❑ Pagination
- ❑ Versioning
- ❑ Authorization
- ❑ API facade
- ❑ Discoverability
- ❑ Idempotent
- ❑ Circuit breaker

Microservice Patterns:

- ☐ API gateway
- ☐ Service registry
- ☐ Circuit breaker
- ☐ Messaging
- ☐ Database per Service
- ☐ Access Token
- ☐ Saga
- ☐ Event Sourcing & CQRS




1. Servlet/jsp
2. MVC
 - Struts
 - JSF
 - Flex
 - Gwt
 - Spring MVC
 - ...





1. Pojo(no Container)
2. Ejb 2.x(HW Container)
 - Session Bean
 - Mdb
3. Pojo + LW Container
 - Spring
 - Microcontainer
 - Xwork
4. Ejb3.0

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
 - Hibernate
 - Kodo
 - Toplink
 - MyBatis
5. JPA

+ Spring Templates

← → ↻ spicejet.com

 **BOOK** ADD-ONS DEALS GIFT CARD SP

 **Flights**  Hotels  Holiday Packages  Flight S


☒ One Way ☐ Round Trip ☐ Multicity

*FROM *TO *DEPART DATE

← → ↻ yatra.com

yatra Flights Hotels

Book Flights, Hotels and Holiday Packages

Depart From
New Delhi
DEL 

<html>

Web MVC

Web API

<xml />

{JSON}



Web Application:

It is an end-to-end solution for a user.

Which means, User can:

- ☐ Open it using a browser Interact with it.
- ☐ User can click on something and after some processing, its result will be reflected in the browser screen. Human-System interaction.

Web API / Web service

With Web APIs alone, a user can not interact with it, because it only returns data, not views.

- ❑ It is a system which interacts with another system
- ❑ It does not return views, it returns data
- ❑ It has an endpoint set, which can be hit by other systems to get data which it provides.

Single Responsibility

Each software module or a class should have one and only one reason to change

Liskov Substitution

You should be able to use any derived class instead of a base class without modification

Dependency Inversion

High level classes should not depend on low level classes instead both should depend upon abstraction

SOLID

```
graph TD; S((S)) --> SR[Single Responsibility]; O((O)) --> LS[Liskov Substitution]; L((L)) --> DI[Dependency Inversion]; I((I)) --> OC[Open/Closed]; D((D)) --> IS[Interface Segregation];
```

Design Principles

Open/Closed

A Software Class or module should be open for extension but closed for modification

Interface Segregation

Client should not be forced to use an interface which is not relevant to it

Open/Closed

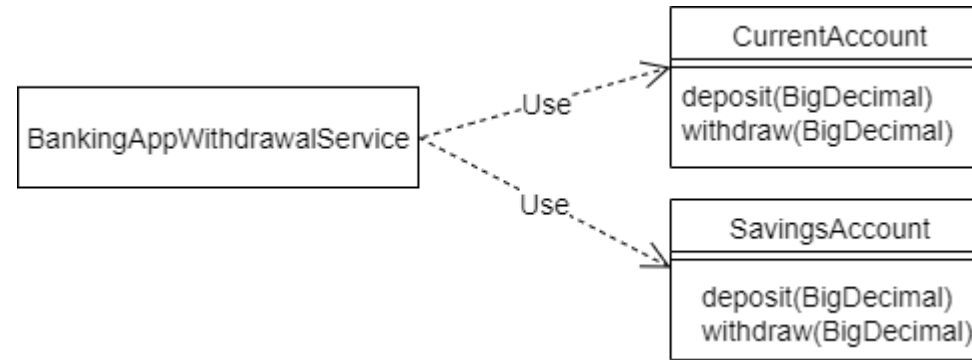
A Software Class or module
should be open for extension
but closed for modification

The goal of the Open/Closed principle encourages us to design our software so we add new features only by adding new code. When this is possible, we have loosely coupled, and thus easily maintainable applications.

Without the Open/Closed Principle

Our banking application supports two account types – “current” and “savings”. These are represented by the classes CurrentAccount and SavingsAccount respectively.

The BankingAppWithdrawalService serves the withdrawal functionality to its users:



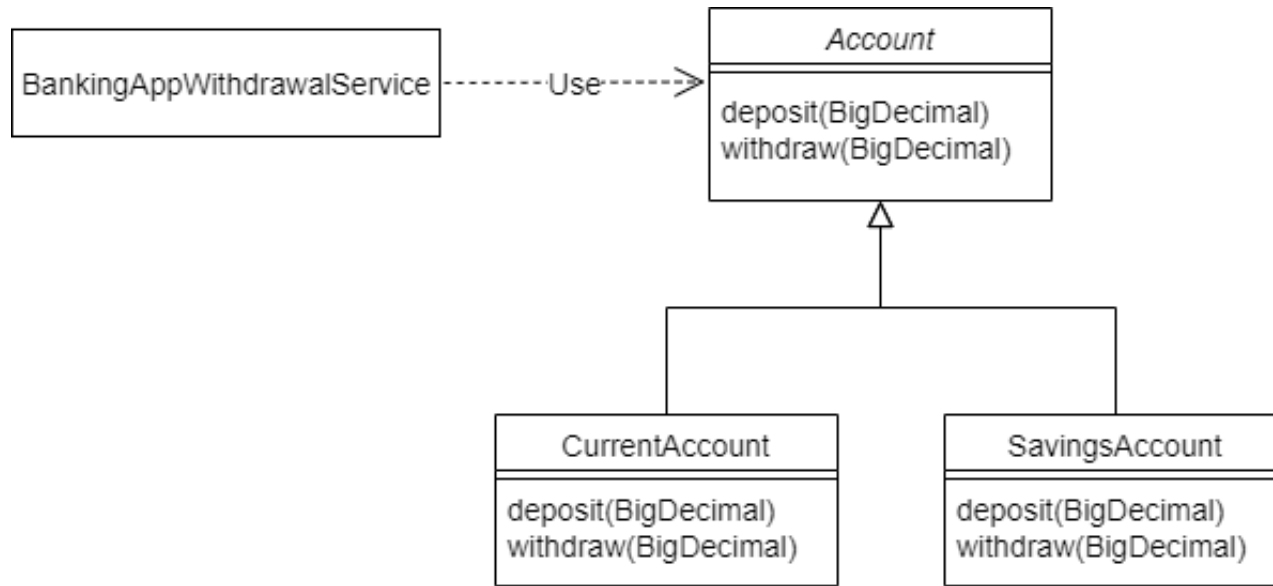
Unfortunately, there is a problem with extending this design.

The `BankingAppWithdrawalService` is aware of the two concrete implementations of account.

Therefore, the `BankingAppWithdrawalService` would need to be changed every time a new account type is introduced.

Using the Open/Closed Principle to Make the Code Extensible

Let's redesign the solution to comply with the Open/Closed principle. We'll close BankingAppWithdrawalService from modification when new account types are needed, by using an Account base class instead:



Here, we introduced a new abstract `Account` class that `CurrentAccount` and `SavingsAccount` extend.

The `BankingAppWithdrawalService` no longer depends on concrete account classes. Because it now depends only on the abstract class, it need not be changed when a new account type is introduced.

Consequently, the `BankingAppWithdrawalService` is open for the extension with new account types, but closed for modification, in that the new types don't require it to change in order to integrate.

A New Account Type

The bank now wants to offer a high interest-earning fixed-term deposit account to its customers.

To support this, let's introduce a new `FixedTermDepositAccount` class. A fixed-term deposit account in the real world “is a” type of account. This implies inheritance in our object-oriented design.

So, let's make `FixedTermDepositAccount` a subclass of `Account`:

```
public class FixedTermDepositAccount extends Account {  
    // Overridden methods...  
}
```

However, the bank doesn't want to allow withdrawals for the fixed-term deposit accounts.

This means that the new `FixedTermDepositAccount` class can't meaningfully provide the `withdraw` method that `Account` defines. One common workaround for this is to make `FixedTermDepositAccount` throw an `UnsupportedOperationException` in the method it cannot fulfill:

```
public class FixedTermDepositAccount extends Account {  
    @Override  
    protected void deposit(BigDecimal amount) {  
        // Deposit into this account  
    }  
  
    @Override  
    protected void withdraw(BigDecimal amount) {  
        throw new UnsupportedOperationException("Withdrawals are not supported by  
FixedTermDepositAccount!!");  
    }  
}
```

Testing Using the New Account Type

While the new class works fine, let's try to use it with the BankingAppWithdrawalService:

```
Account myFixedTermDepositAccount = new  
FixedTermDepositAccount();  
myFixedTermDepositAccount.deposit(new BigDecimal(1000.00));
```

```
BankingAppWithdrawalService withdrawalService = new  
BankingAppWithdrawalService(myFixedTermDepositAccount);  
withdrawalService.withdraw(new BigDecimal(100.00));
```

Unsurprisingly, the banking application crashes with the error:

Withdrawals are not supported by FixedTermDepositAccount!!
There's clearly something wrong with this design if a valid combination of objects results in an error.

What Went Wrong?

The BankingAppWithdrawalService is a client of the Account class. It expects that both Account and its subtypes guarantee the behavior that the Account class has specified for its withdraw method:

```
/**  
 * Reduces the account balance by the specified amount  
 * provided given amount > 0 and account meets minimum available  
 * balance criteria.  
 *  
 * @param amount  
 */  
protected abstract void withdraw(BigDecimal amount);
```

However, by not supporting the withdraw method, the FixedTermDepositAccount violates this method specification. Therefore, we cannot reliably substitute FixedTermDepositAccount for Account.

In other words, the FixedTermDepositAccount has violated the Liskov Substitution Principle.

Liskov Substitution

You should be able to use any derived class instead of a base class without modification

When Is a Subtype Substitutable for Its Supertype?

A subtype doesn't automatically become substitutable for its supertype. To be substitutable, the subtype must behave like its supertype.

An object's behavior is the contract that its clients can rely on. The behavior is specified by the public methods, any constraints placed on their inputs, any state changes that the object goes through, and the side effects from the execution of methods.

Subtyping in Java requires the base class's properties and methods are available in the subclass.

Refactoring

To fix the problems we found in the banking example, let's start by understanding the root cause.

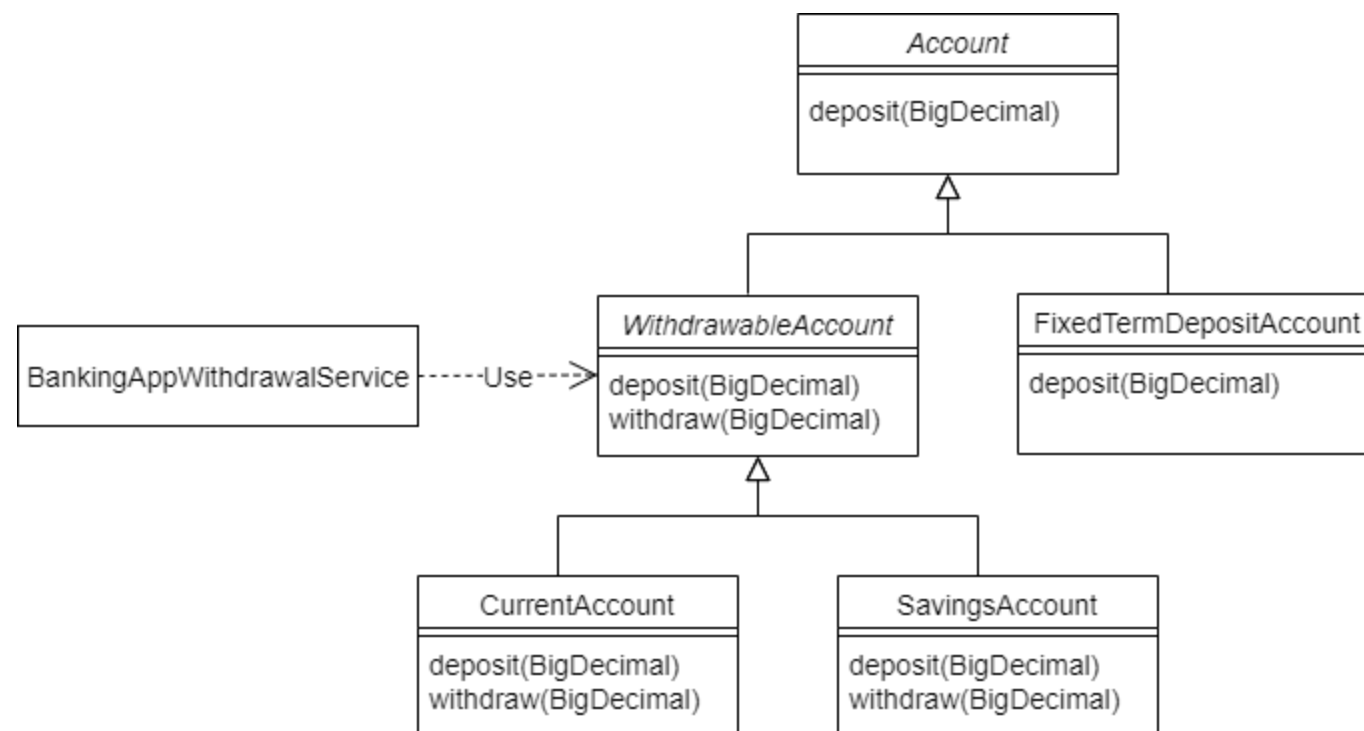
The Root Cause

In the example, our `FixedTermDepositAccount` was not a behavioral subtype of `Account`.

The design of `Account` incorrectly assumed that all `Account` types allow withdrawals. Consequently, all subtypes of `Account`, including `FixedTermDepositAccount` which doesn't support withdrawals, inherited the `withdraw` method.

Though we could work around this by extending the contract of `Account`, there are alternative solutions.

Revised Class Diagram



Because all accounts do not support withdrawals, we moved the withdraw method from the Account class to a new abstract subclass WithdrawableAccount.

Both CurrentAccount and SavingsAccount allow withdrawals. So they've now been made subclasses of the new WithdrawableAccount.

This means BankingAppWithdrawalService can trust the right type of account to provide the withdraw function.

Refactored BankingAppWithdrawalService

BankingAppWithdrawalService now needs to use the WithdrawableAccount:

```
public class BankingAppWithdrawalService {  
    private WithdrawableAccount withdrawableAccount;  
  
    public BankingAppWithdrawalService(WithdrawableAccount  
withdrawableAccount) {  
        this.withdrawableAccount = withdrawableAccount;  
    }  
  
    public void withdraw(BigDecimal amount) {  
        withdrawableAccount.withdraw(amount);  
    }  
}
```

As for FixedTermDepositAccount, we retain Account as its parent class. Consequently, it inherits only the deposit behavior that it can reliably fulfill and no longer inherits the withdraw method that it doesn't want.

Single Responsibility

Each software module or a class should have one and only one reason to change

“single responsibility principle” which states “gather together those things that change for the same reason, and separate those things that change for different reasons.”

A microservices architecture takes this same approach and extends it to the loosely coupled services which can be developed, deployed, and maintained independently. Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a larger complex business problem.

Example

A class or module should have one, and only one, reason to be changed (i.e. rewritten).

As an example, consider a module that compiles and prints a report.

Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change.

The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

Dependency Inversion

High level classes should not depend on low level classes instead both should depend upon abstraction

Definition of the Dependency Inversion Principle

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features.

To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

Dependency Inversion Principle consists of two parts:

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.

An important detail of this definition is, that high-level and low-level modules depend on the abstraction. The design principle does not just change the direction of the dependency, as you might have expected when you read its name for the first time. It splits the dependency between the high-level and low-level modules by introducing an abstraction between them. So in the end, you get two dependencies:




- ✓ the high-level module depends on the abstraction, and
- ✓ the low-level depends on the same abstraction.




Brewing coffee with the Dependency Inversion Principle

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





BasicCoffeeMachine

-  - Configuration config
-  - Map<CoffeeSelection, GroundCoffee> groundCoffee
-  - BrewingUnit brewingUnit

-  + BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee)
-  + Coffee brewFilterCoffee()
-  + void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee)

PremiumCoffeeMachine

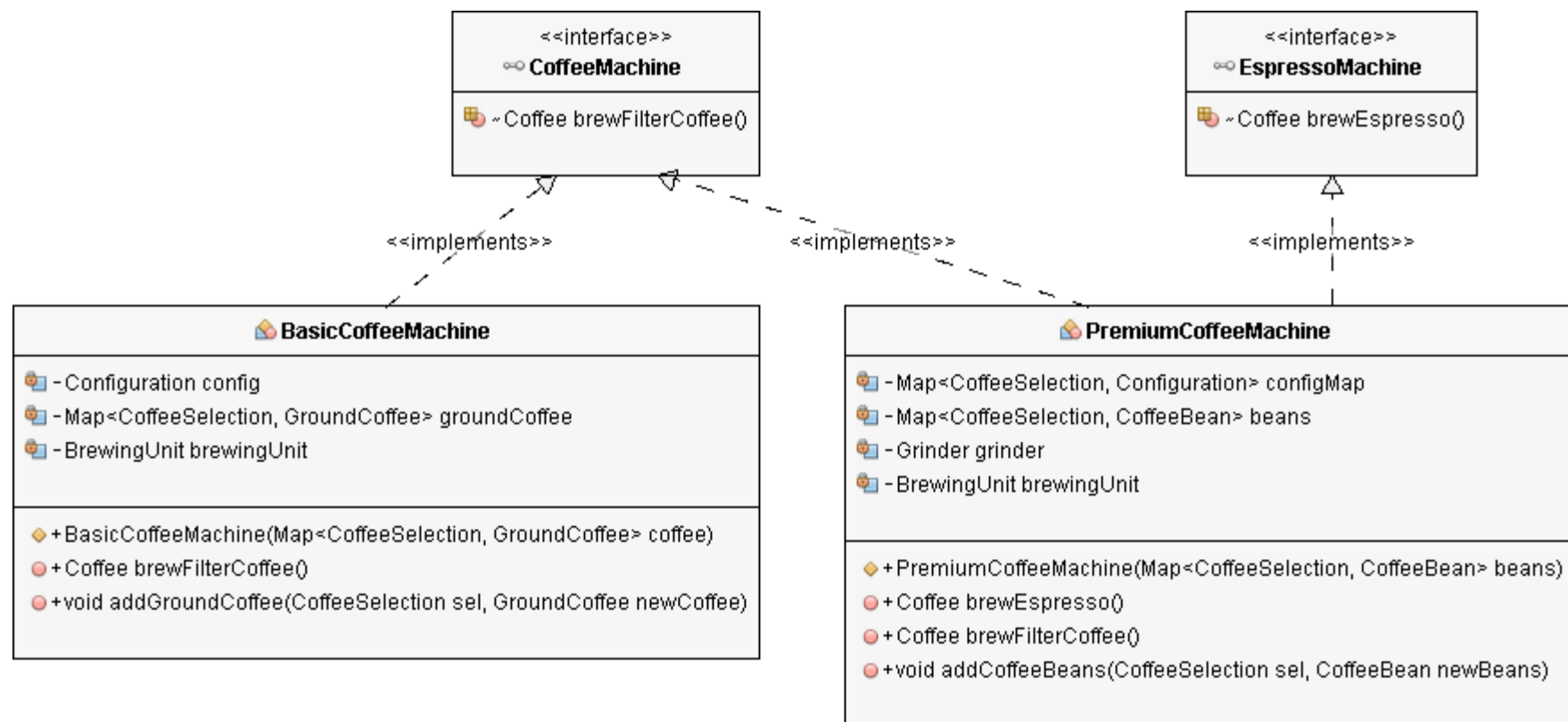
-  - Map<CoffeeSelection, Configuration> configMap
-  - Map<CoffeeSelection, CoffeeBean> beans
-  - Grinder grinder
-  - BrewingUnit brewingUnit

-  + PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans)
-  + Coffee brewEspresso()
-  + Coffee brewFilterCoffee()
-  + void addCoffeeBeans(CoffeeSelection sel, CoffeeBean 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, we 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.

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

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



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 EspressoMachine implementations are interchangeable.

Implementing the coffee machine application

We 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 {
```

```
    public class CoffeeApp {  
        private CoffeeMachine coffeeMachine;
```

```
        public CoffeeApp(CoffeeMachine coffeeMachine) {  
            this.coffeeMachine = coffeeMachine  
        }  
    }
```

```
        public Coffee prepareCoffee() throws CoffeeException {  
            Coffee coffee = this.coffeeMachine.brewFilterCoffee();  
            System.out.println("Coffee is ready!");  
            return coffee;  
        }  
    }  
}
```

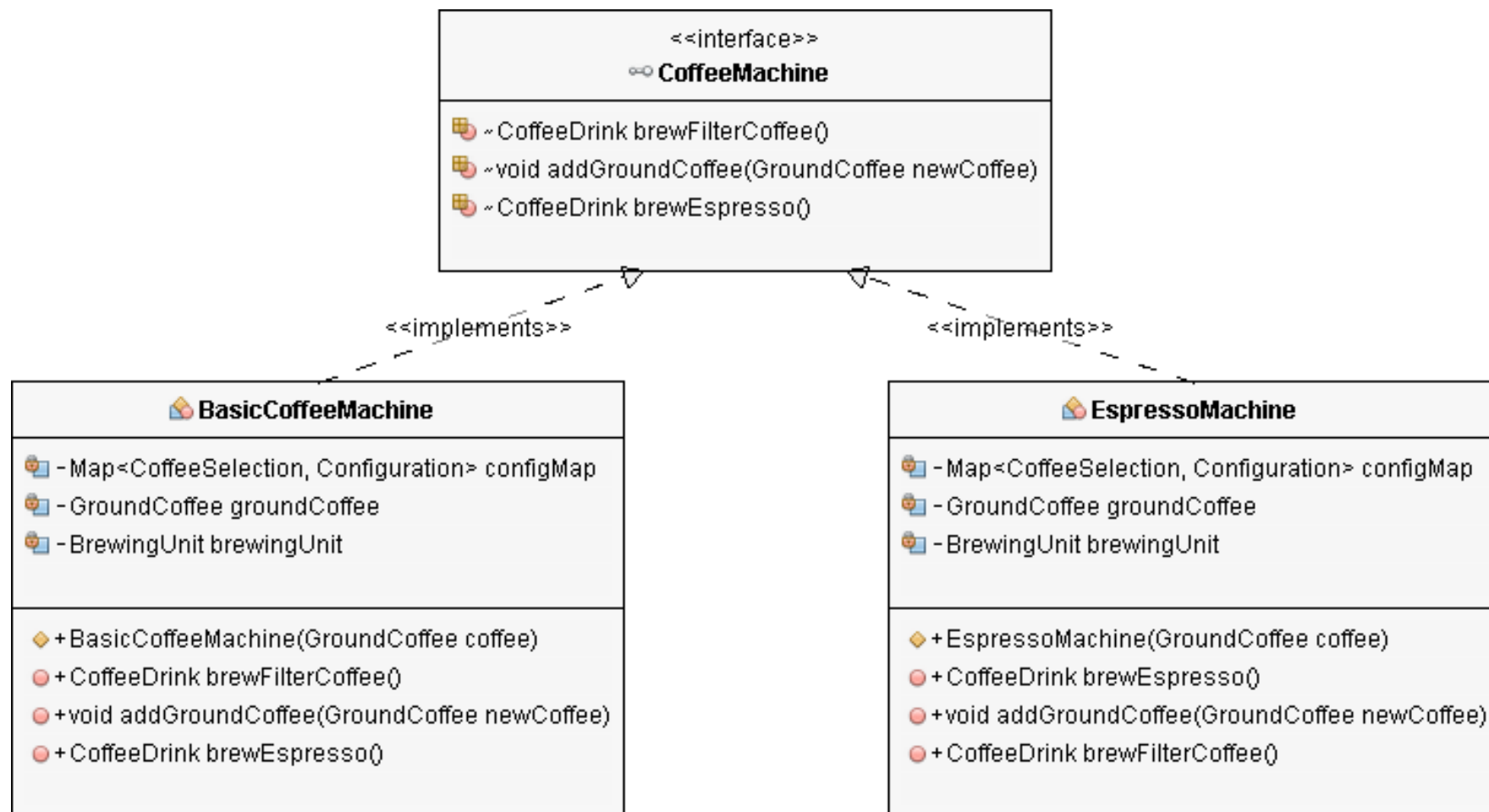
The only code that directly depends on one of the implementation classes is the `CoffeeAppStarter` class, which instantiates a `CoffeeApp` object and provides an implementation of the `CoffeeMachine` interface.

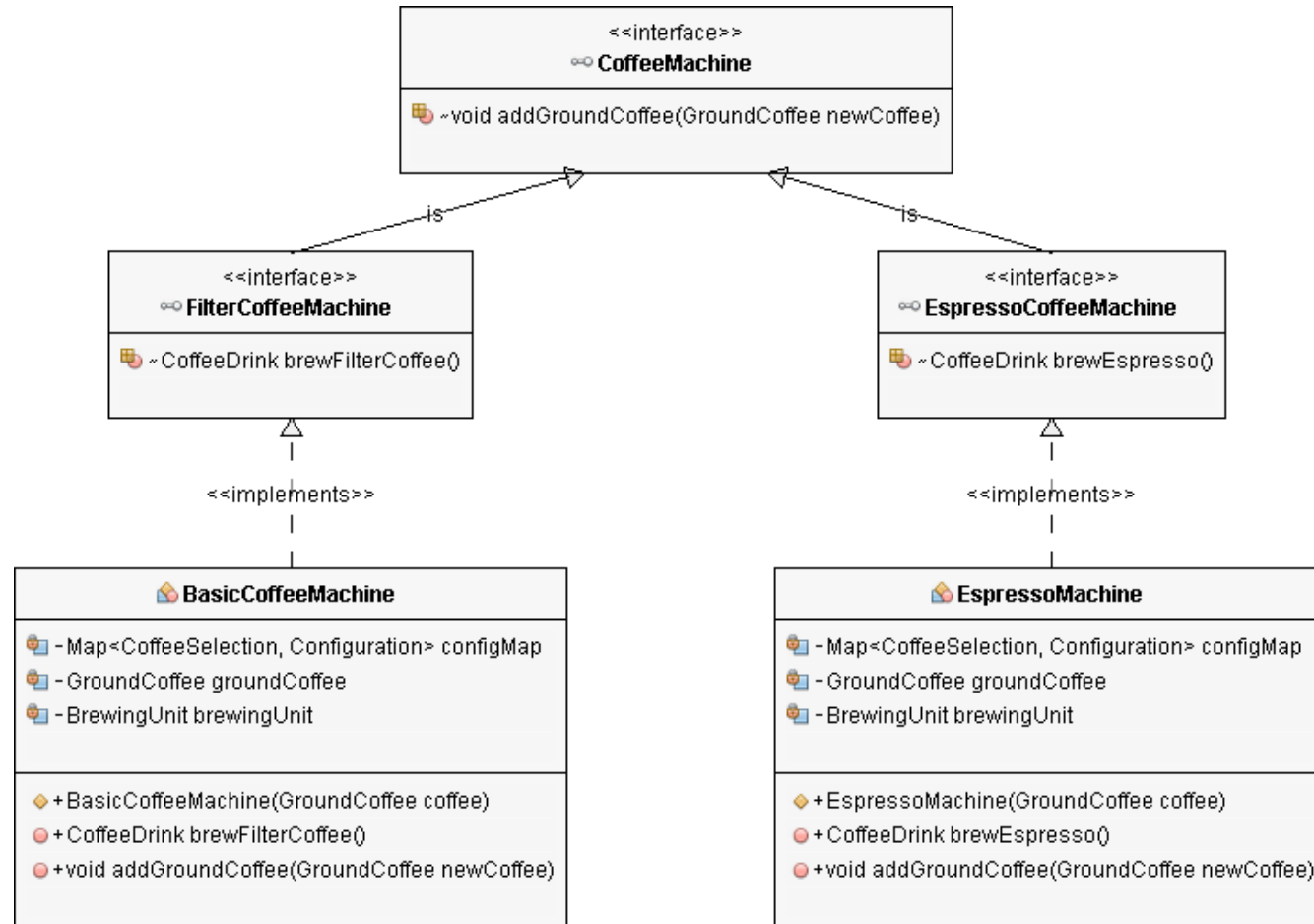
We could avoid this compile-time dependency entirely by using a dependency injection framework, like Spring or CDI, to resolve the dependency at runtime.

```
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();  
        } catch (CoffeeException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Interface Segregation

Client should not be forced to use an interface which is not relevant to it





We 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 Java Shell Tool or in short JShell is a major addition in Java 9.

A REPL (Read-Evaluate-Print-Loop) is a simple, interactive programming environment that takes user input (Read), executes the user command (Evaluate), returns the result to the user (Print) and waits for the next user input (Loop).

```
JDK\bin>jshell
| Welcome to JShell -- Version 12.0.1
| For an introduction type: /help intro

jshell>
```

Printing Hello World

Use var keyword(java v10) to create a new variable as shown below:

```
jshell> var message = "Hello World"
```

```
message ==> "Hello World"
```

Convert message into uppercase

```
message.toUpperCase()
```

Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples
Packages	<p>The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.</p> <p>Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.</p>	<p>com.sun.eng</p> <p>com.apple.quicktime.v2</p> <p>edu.cmu.cs.bovik.cheese</p>
Classes	<p>Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).</p>	<p>class Raster;</p> <p>class ImageSprite;</p>
Interfaces	<p>Interface names should be capitalized like class names.</p>	<p>interface RasterDelegate;</p> <p>interface Storing;</p>

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	<code>run();</code> <code>runFast();</code> <code>getBackground();</code>
Variables	<p>Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore <code>_</code> or dollar sign <code>\$</code> characters, even though both are allowed.</p> <p>Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are <code>i</code>, <code>j</code>, <code>k</code>, <code>m</code>, and <code>n</code> for integers; <code>c</code>, <code>d</code>, and <code>e</code> for characters.</p>	<pre>int i; char c; float myWidth;</pre>
Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores (" <code>_</code> "). (ANSI constants should be avoided, for ease of debugging.)	<code>static final int MIN_WIDTH = 4;</code> <code>static final int MAX_WIDTH = 999;</code> <code>static final int GET_THE_CPU = 1;</code>