OOP Recap. SOLID Principles

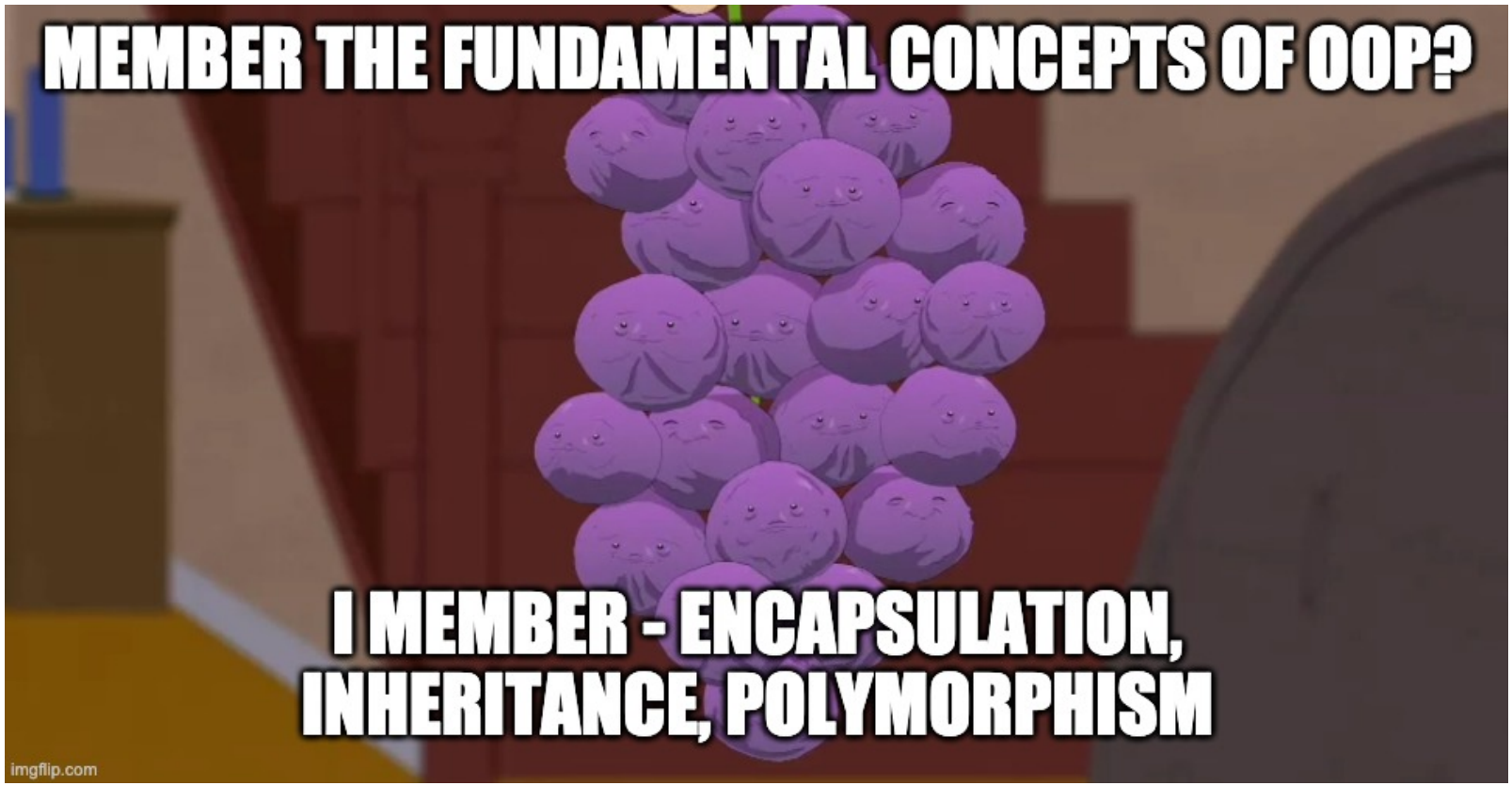
Lect. PhD. Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

Overview

Recap



Encapsulation

- Restrict direct access to an object's components
- Bundle data and methods operating on it together
- The purpose is to achieve potential for change

G. Booch - "Object-Oriented Analysis and Design with Applications"

"the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation"

Encapsulation

A few examples:

- How does it work in C++, Java, Python?
- What about SQL?
- How about a toaster or a car?

NB!

Encapsulation works at different levels, so context and semantics are always important

Encapsulation

- C++, Java, C# - **private**, **protected**, **public**
- C++ - default is **private**, while in Java default is **default** (same as protected, adding package level access).
- C# - adds the **internal** modifier, which grants access within the same assembly (*.dll* or *.exe* file)
- Underscore counting with Python
- C++ has public, protected and private inheritance

Inheritance

- Implements an **IS-A** type of relationship
- Classes vs. Interfaces
- You can inherit from interfaces (Java, C#), or other classes
- You can inherit from several interfaces and a single base class
- Particularities
  - C++, Python allow you to inherit from multiple classes
  - Java 8 adds support for default interface methods... why?
  - Diamond problem and solutions

Inheritance

Source code

git: [...]/examples/recap/inheritance

Polymorphism

- The property of an entity to react differently depending on its type
- It allows different entities to behave in different ways in response to the same action.

In source code

Allows different objects (*depending on their type*) to respond in different ways to the same message (*a different method is called*).

Polymorphism

- Let's examine how polymorphism works:
- Java, Python, C#, C++

Source code

git: [...]/examples/recap/polymorphism

Polymorphism

- **Java**
  - Emphasis on simplicity, all methods are virtual
  - Adds a level of indirection to method calls, unless they are marked *final*
- **Python**
  - Does not make sense to *declare* variable type
  - Everything is evaluated at runtime

Polymorphism

- **C++**
  - Concerned about efficiency and space
  - *vtable* pointer overhead only for methods marked virtual
  - Other methods are bound at compile time
- **C#**
  - Shows it has roots in C++
  - Polymorphism similar to C++ implementation
  - C# adds the *override* keyword, avoiding the issue where a same-name virtual method is later added to a base class, adding unwanted polymorphism

SOLID

Introduced by Robert C. Martin in 2000 in the *Design Principles and Design Patterns* paper, they apply to any object-oriented design.

What is SOLID?

- Single responsibility principle
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Figure: Robert C. Martin

SOLID

First proponent of SOLID principles[1]

Figure: Book selection authored by Robert C. Martin

[1] This section organized according to https://stackify.com/solid-design-principles/

Importance of SOLID principles

1. The foundation of a well designed application
2. Make software designs more understandable, flexible and maintainable
3. Guidelines that can be applied while working on software to remove code smells
4. Part of an overall strategy of agile and adaptive programming

Single Responsibility Principle (SRP)

- One of the basic principles used to build software that is easy to maintain
- Can be applied at function, class, module and component level (at least)
- The answer to *What should this function / class / component do?* should not include **and**
- Entities doing only one thing are also easier to understand

What is it?

A class or module should have one, and only one, reason to change (responsibility).

Single Responsibility Principle

- Consider a module that compiles and prints a report.
- Such a module can be changed for two reasons:
  1. The content of the report could change.
  2. The format of the report could change.
- These two things change for very different causes; one substantive, and one cosmetic.
- 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.

...

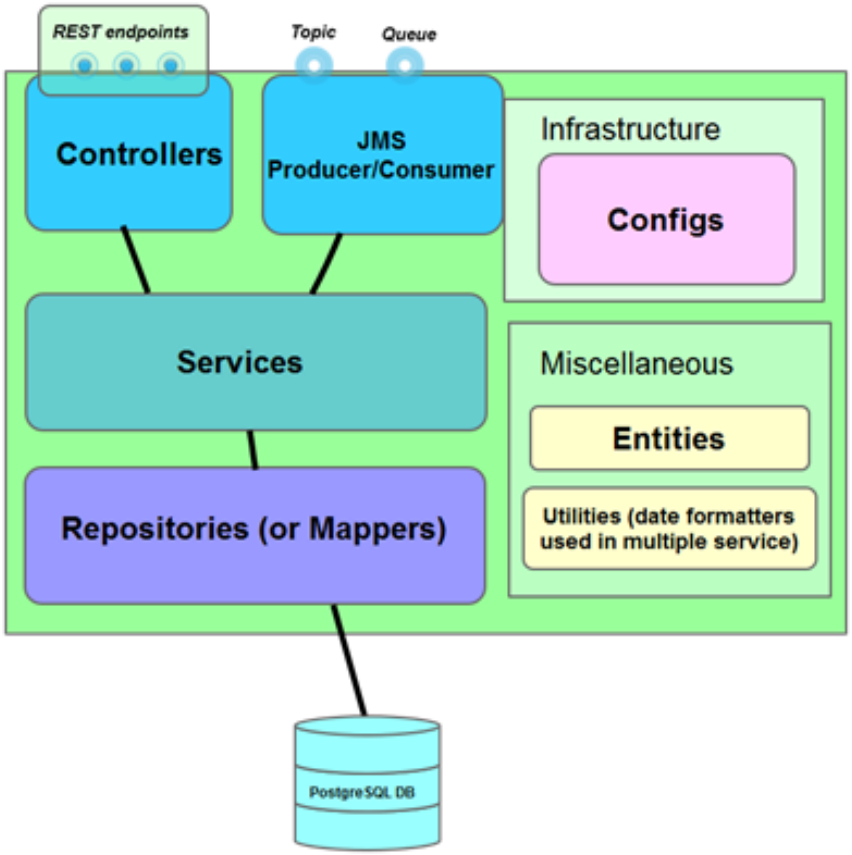
The reason it is important to keep a class focused on a single concern is that it makes the class more robust.

SRP - Separation of concerns (SoC)

- A design principle for separating a computer program into distinct sections, such that each section addresses a separate concern.
- Can be general, such as intended for modules.
- Can be specific, such as the name of a class to instantiate.
- A program that embodies SoC well is called a modular program.
- Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface.

SRP - Separation of concerns (SoC)

Layered designs in information systems are another embodiment of separation of concerns (e.g., presentation layer, business logic layer, data access layer, persistence layer).



SRP - Separation of concerns (SoC)

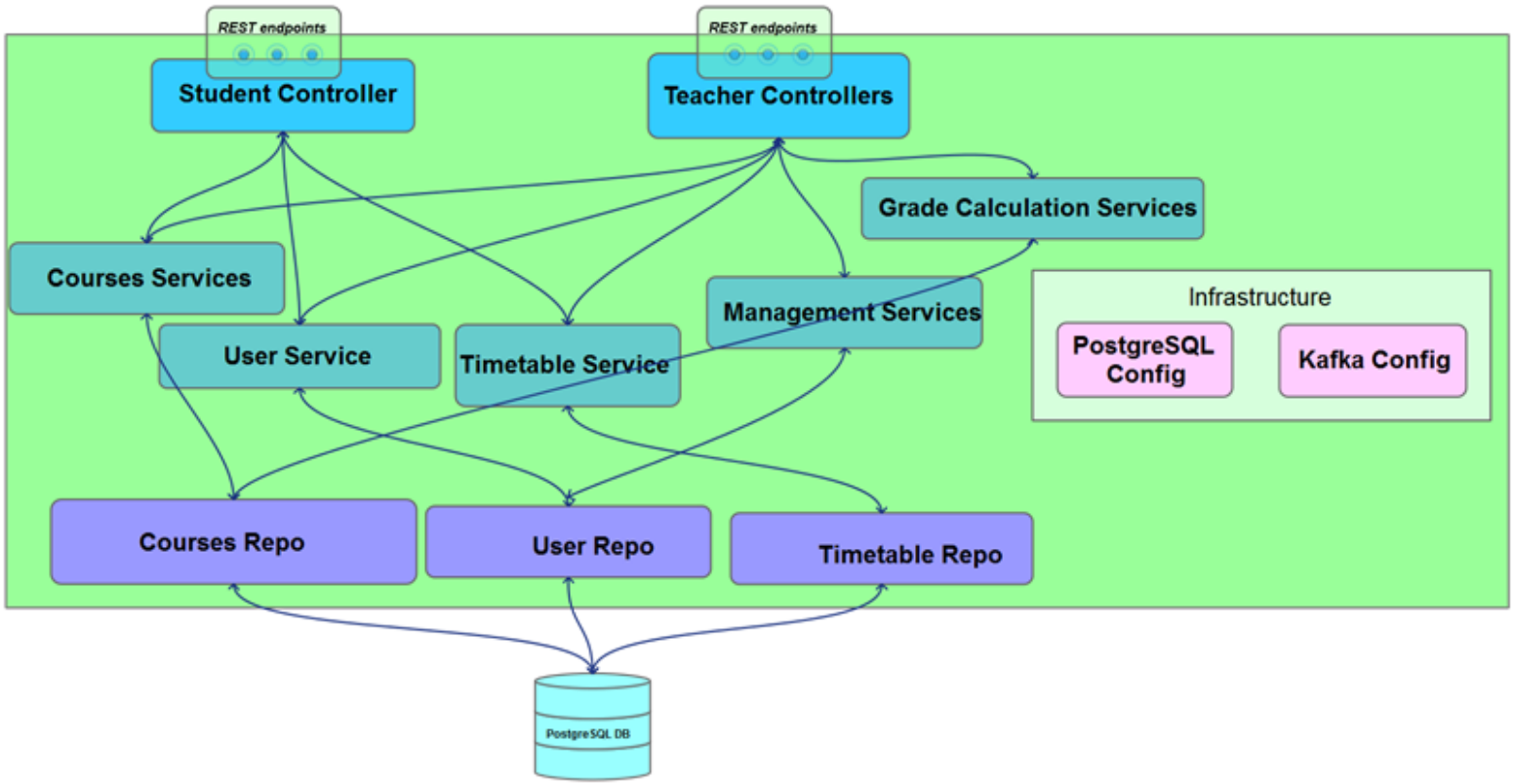


Figure: Separation of concerns

Open/Closed Principle

Bertrand Mayer

Software entities (functions, classes, modules, components) should be open for extension, but closed for modification.

- Idea is to enable adding functionality without changing existing code
- It should prevent changes in one place from requiring changes in many other places
- How to achieve this?
  - **Bertrand Mayer** - Inheritance
  - **Robert C. Martin** - Polymorphism

Open/Closed Principle

Bertrand Mayer - Inheritance

Bertrand Mayer

"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.

- Inheritance opens the issue of derived classes using implementation details of the parent
- Tension between *inheritance* and *encapsulation*.

Open/Closed Principle

Robert C. Martin - Polymorphism

- Replace inheritance with programming to interfaces
- Interfaces are *closed* to modification, but *open* for new implementations
- Interfaces add an additional abstraction level, facilitating loose coupling

Open/Closed Principle

Coffee Machine example

- We have a simple coffee machine that brews filter coffee
- We have an app to control it

Problem

How does the app change when we buy a fancy coffee machine, which can brew both filter coffee (using ground coffee) and espresso (using coffee beans)?

Open/Closed Principle

A tale of two coffee makers...

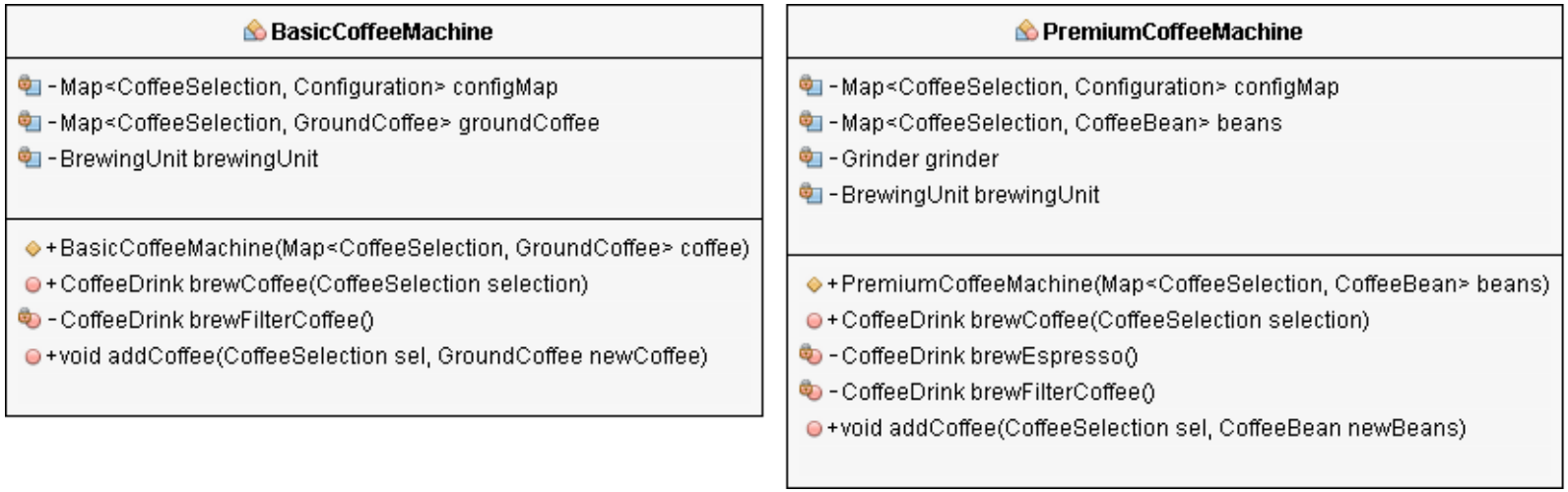


Figure:
https://stackify.com/solid-design-liskov-substitution-principle/

Open/Closed Principle

Solution

- Extract the common functionalities of coffee machines to an interface
- The app talks to the machine through the interface

Source code

git: [...]/examples/solid/openclosed

Liskov Substitution principle

Barbara Liskov - "Data Abstraction"

Let $\Theta(x)$ be a property provable about objects **x** of type **T**. Then $\Theta(y)$ should be true for objects **y** of type **S** where **S** is a subtype of **T**.

Liskov Substitution Principle

- If **S** is a subtype of **T**, then objects of type **T** may be replaced with objects of type **S** without breaking program behaviour
- Derived classes must be usable through the base class interface, without the need for the user of the class to know the difference
- Think Java method overwriting!
  - Overriden methods can have more lax requirements, but not stricter ones!
  - Care with input parameters, return values (covariant return types in Java 5+), thrown exceptions!

Liskov Substitution Principle

Example 1

Basic example for Liskov Substitution Principle

Liskov Substitution Principle

Liskov Substitution at work

- Say we have two coffee machines, a basic and a premium one
- A common base class or interface could make the code of the coffee app using it simpler
- What issues might we run into, if any?

Liskov Substitution Principle

A tale of two coffee makers...

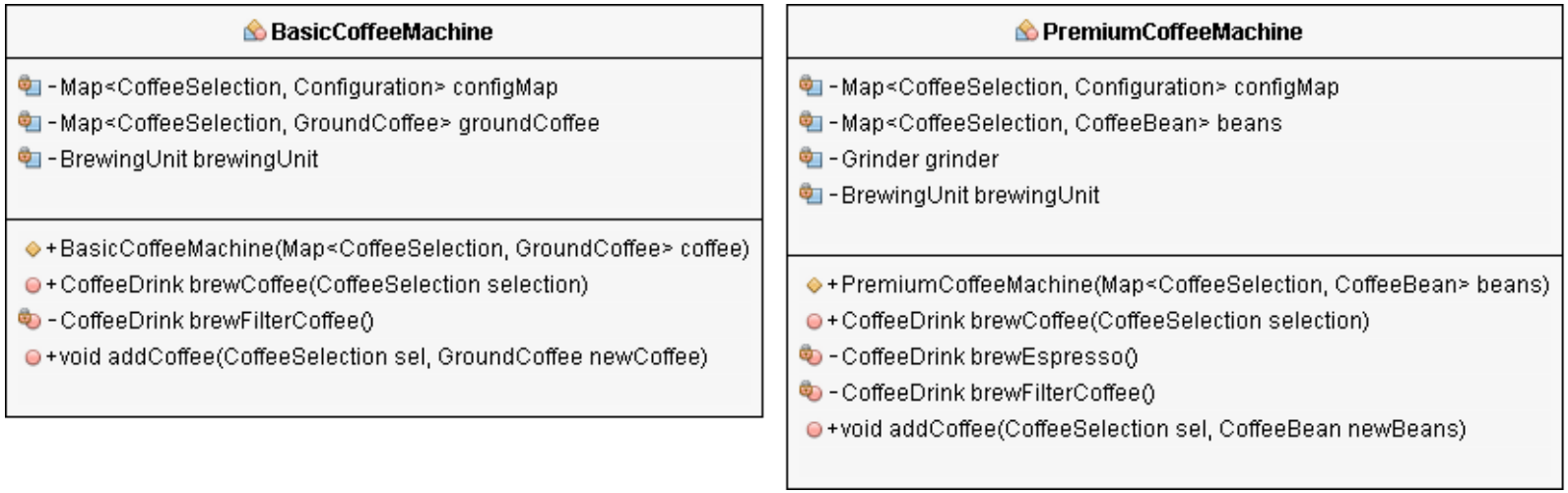


Figure: `https://stackify.com/solid-design-liskov-substitution-principle/`

Liskov Substitution Principle

- A common parent could unify only the *brewCoffee()* and *addCoffee()* methods
- The *brewCoffee()* methods can both make filter coffee, so the base class or interface method has to at least support that
- Parameters for *addCoffee()* differ!?
- A common base class for *GroundCoffe* and *CoffeeBean* (maybe *Coffee*?) is possible, but requires additional check in both machines
- Common interface should only required what is supported in both machines - *brewCoffee()* method that makes filter coffee

Liskov Substitution Principle

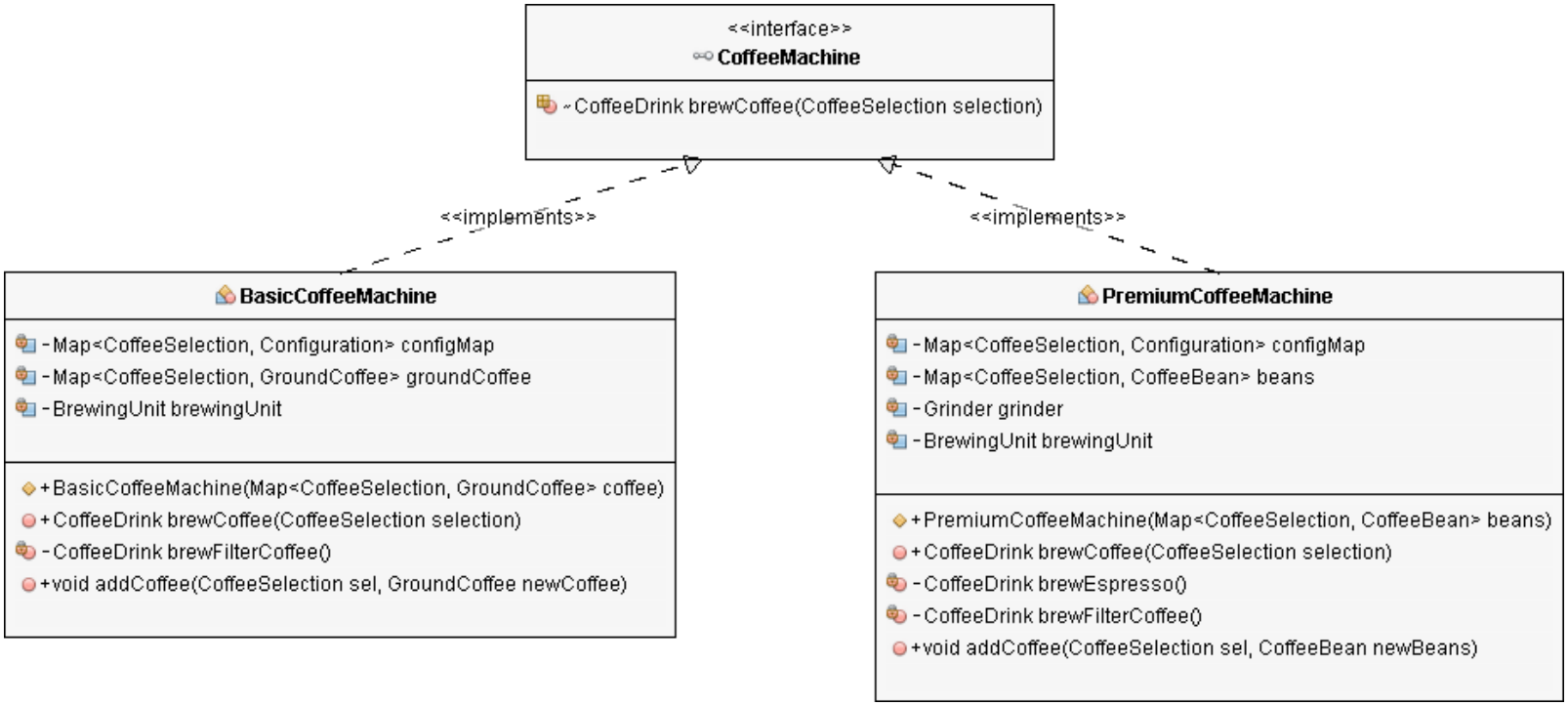


Figure: `https://stackify.com/solid-design-liskov-substitution-principle/`

Interface Segregation principle

Robert C. Martin

"Clients should not be forced to depend upon interfaces that they do not use."

- Split large interfaces into smaller and more specific ones; clients will only know about those in which they are directly interested
- Keeps a system decoupled - easier to refactor, change, and redeploy
- The contents of an interface should be decided upon depending on the needs of the client

Interface Segregation principle

- No one writes bad software because they want to
- Clients wanting new functionalities (yesterday) is great for business, but can be a technological *nightmare*
- *Interface pollution* - forcing clients to depend on methods they should not care about
- A tale of two coffee machines...

Interface Segregation principle

- Class *BasicCoffeeMachine* models a basic, filter coffee maker
- Reading that it's better to program behind an interface, we extract the *CoffeeMachine* interface, with methods *addGroundCoffee()* and *brewFilterCoffee()*
- Wouldn't it be great if we also support esspresso machines? (modeled in the *EsspressoMachine* class)
- Of course, the espresso machine has the *brewEspresso()* method, which is a different type of coffee

Interface Segregation principle

What to do, what to do?

1. Refactor under the *CoffeeMachine* interface
2. Use the interface segregation principle

Interface Segregation principle

Refactor under the *CoffeeMachine* interface

1. Change *EsspressoMachine* so that it implements the *CoffeeMachine* interface $->$ also implement *brewFilterCoffee()*
2. Add the *brewEspresso()* method to the *CoffeeMachine* interface
3. Add the *brewEspresso()* method to the *BasicCoffeeMachine*
4. **Hint:** maybe use a default interface method?

Interface Segregation principle

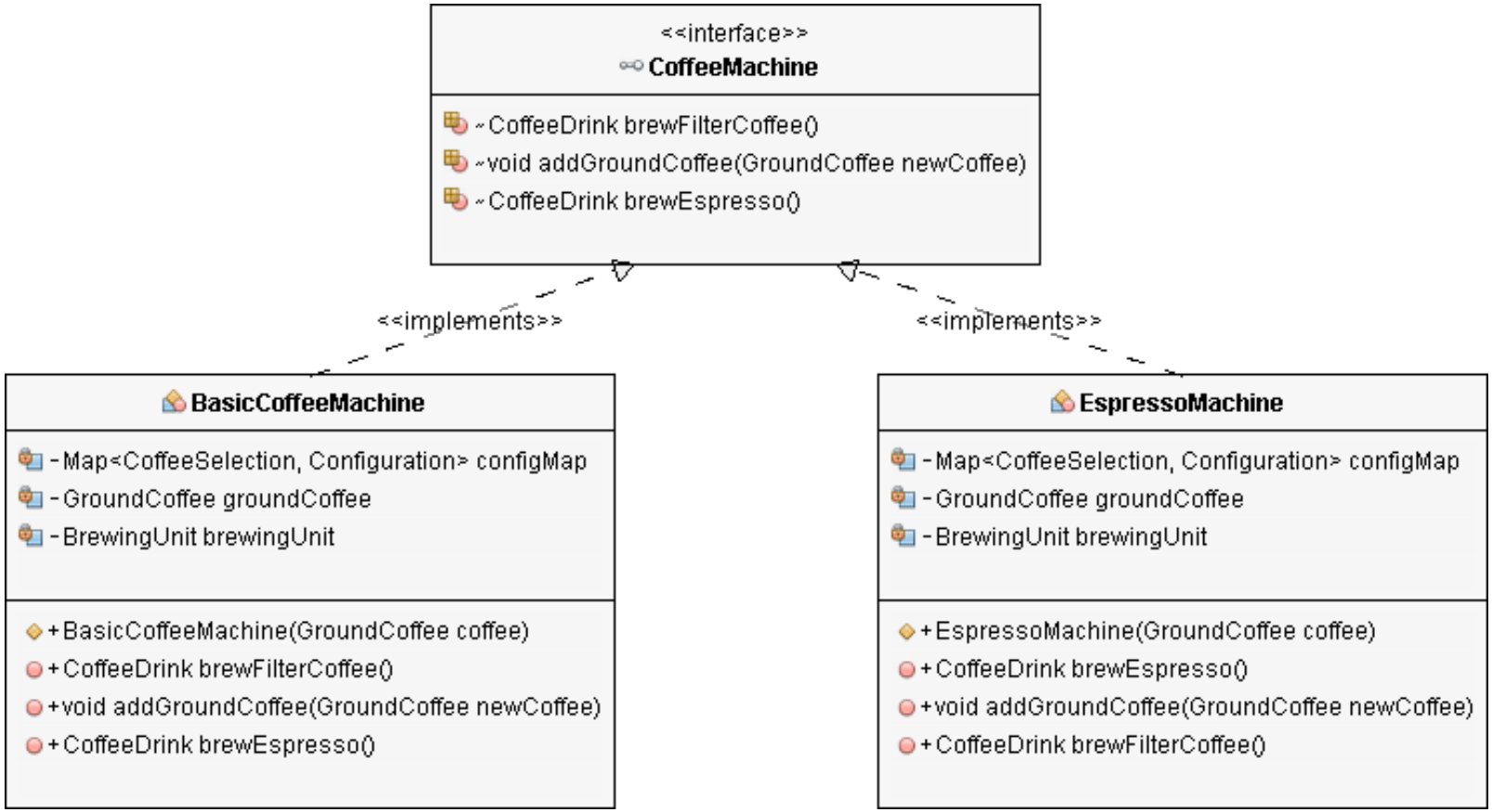


Figure:
https://stackify.com/interface-segregation-principle/

Interface Segregation principle

Problems?

1. Classes must implement a contract they cannot provide
2. Programming through the interface might result in an *Exception* - no coffee for you...
3. The interface and classess depend on things they have no control of (e.g. change in *BasicCoffeeMachine* affects the interface and the *EspressoMachine* class)

Interface Segregation principle

Use the interface segregation principle

1. Identify and group common functionalities in a base interface - *CoffeeMachine*
2. Have separate interfaces for different types of coffee makers

Interface Segregation principle

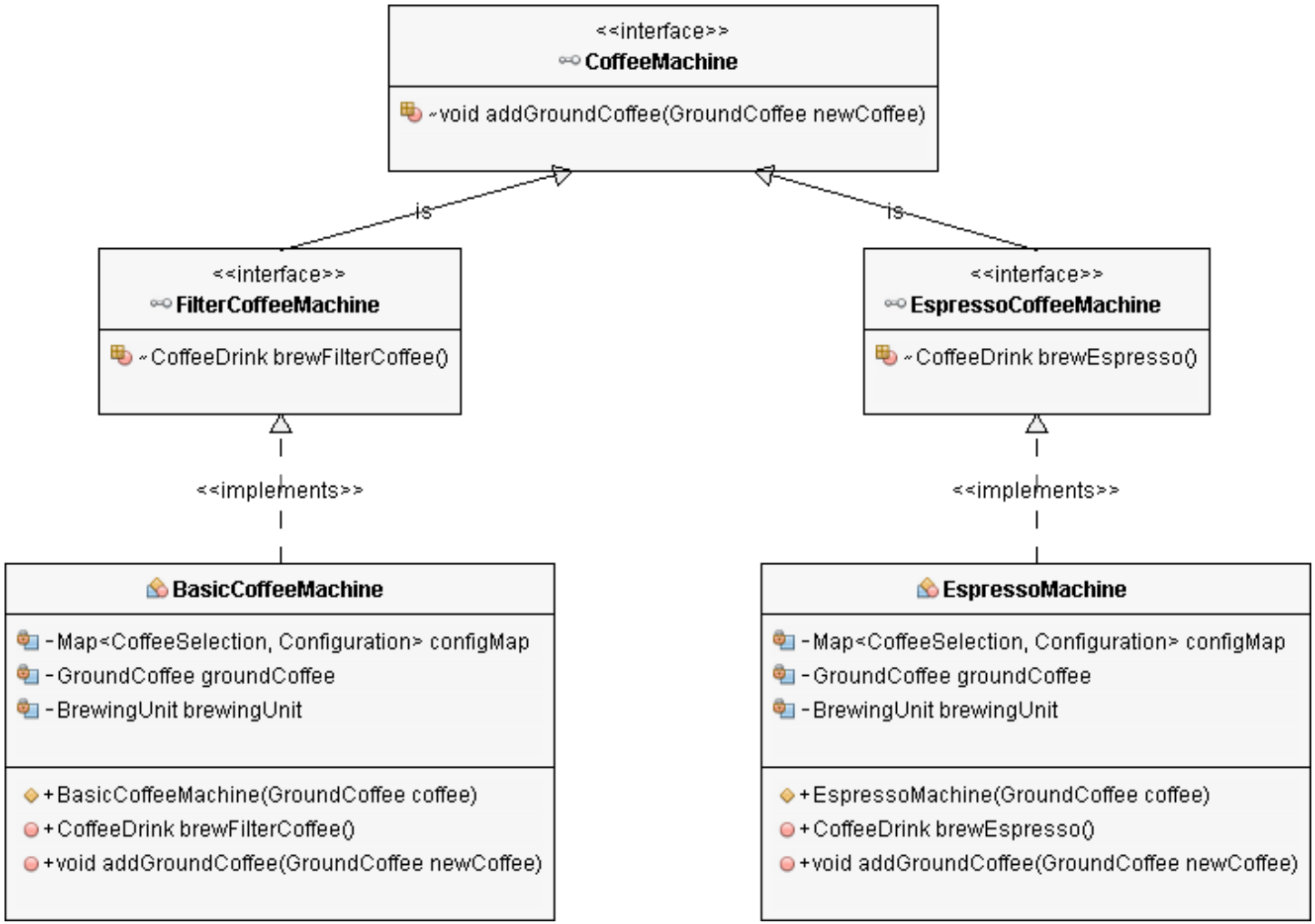


Figure:
https://stackify.com/interface-segregation-principle/

Interface Segregation principle

Follow-up question

'Member the *PremiumCoffeeMachine* that can make both filter and espresso?

Dependency Inversion principle

- Refers to decoupling software modules.
- The principle states:
  - 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.
- When designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them.

Dependency Inversion principle

- Traditional layers pattern
  - *Lower-level* components are designed to be consumed by higher-level components which enable increasingly complex systems to be built
  - *Higher-level* components depend directly upon lower-level components to achieve some task

Follow-up question

Where have I heard this before?

Dependency Inversion principle

The tale of coffee machines - *BasicCoffeeMachine* and *PremiumCoffeeMachine*

- Abstract available functionalities behind interfaces
- Create suitable interfaces - fewer classes/interfaces do not necessarily improve design

Dependency Inversion principle

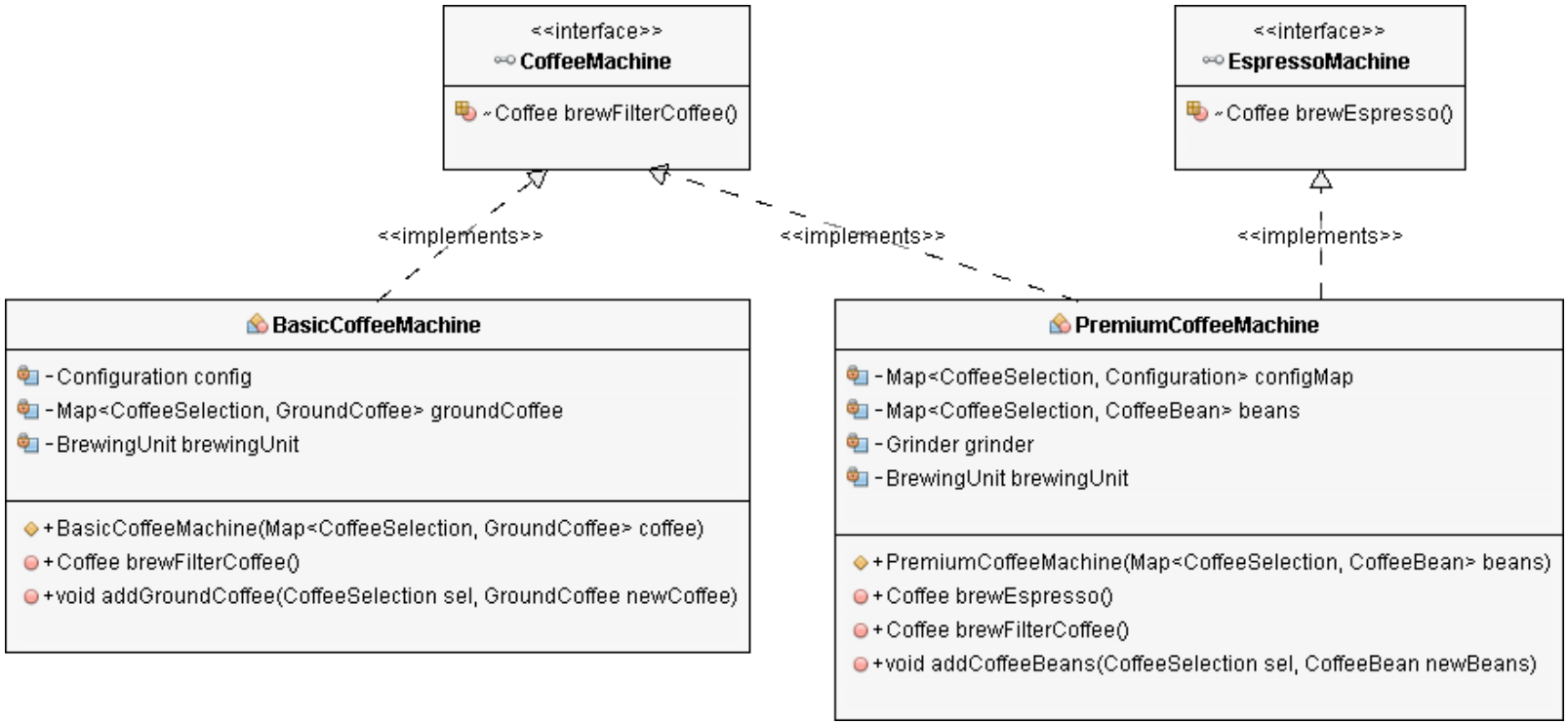


Figure:
https://stackify.com/dependency-inversion-principle/