UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS
ILMU
KOMPUTER

Module 07: Design Patterns

Advanced Programming

Ichlasul Affan

Editor: Teaching Team (Lecturer and Teaching Assistant)

Advanced Programming

Semester Genap 2024

Fasilkom UI

Author: *Ichlasul Affan*
Email: ichlasul.affan12@cs.ui.ac.id

Table of Contents

Learning Objectives

1. Students should be able to understand the motivations of using design patterns on object-oriented design.
2. Students should be able to understand the types of behavioural design patterns and use it accordingly to relevant case studies.
3. Students should be able to understand the types of structural design patterns and use it accordingly to relevant case studies.
4. Students should be able to understand the types of creational design patterns and use it accordingly to relevant case studies.
5. Students should be able to combine and integrate various design patterns while designing program codes.

References

1. Freeman, E., et al. (2014). *Head First Design Patterns: A Brain-Friendly Guide*. O'Reilly Media, Inc.
2. Gamma, E., et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
3. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR.
4. Refactoring Guru: https://refactoring.guru

1. Motivations and Definitions of Design Pattern

What is a Design Pattern?

What is a Design Pattern?

- A (Problem, Solution) pair
- **General repeatable solution to a commonly occurring problem in software design**.
- Design patterns can speed up the development process by providing tested, proven development paradigms

Design pattern is a general repeatable solution to a **commonly occurring** problem in software **design**. Just like other things that repeat often, common problems in software design also have **patterns**. They are reproducible. So, a design pattern is a pair of a problem and its solution. A design cannot be called a “design pattern” if it is not general, not repeatable, or not common.

Design patterns in general sense, not only in Computer Science or Software Engineering, can speed up the development process because it is proven and tested by many engineers that happen to have the similar problem. So, we do not need to prove it again. We just need to make sure that our problem truly aligns with the design pattern that we want to apply.

What is a Design Pattern?

- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.
- Borrowed from Civil & Electrical Engineering domains

As in the explanation of the previous page, the concept of “design pattern” is also applied throughout all Engineering subjects, including Civil and Electrical Engineering. For example, we want to build a suspension bridge to cross a waterway or a river. Engineers are having the same “patterns of thinking” and “patterns of calculation” to determine many things, ranging from the thickness of suspension cables, the length of suspension cables, the height of the pillars, to the number of pillars. Every suspension bridge has unique needs, different maximum weight, and different land structures. However, they at least have one common way to calculate things based on those variables. Even though the result will be different, it can be tested in the same way as other suspension bridges.

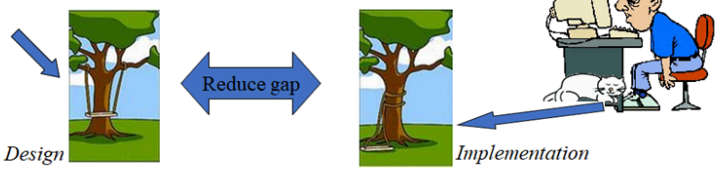
This is the same with Design Patterns in Software Engineering. We are not talking about the implementation details here. A design pattern, for example, Decorator pattern, can be implemented differently based on the programming language that we use:

- Python, by using higher-level functions → function that receives functions as parameters, and then returns a function.
- Java, by using a Decorator interface/abstract class, and its concrete implementations.

We talk about the mindset, the design process, and the structures. Although some might have different implementations, it reaches the same goal and same way to operate. By synchronising mindsets to a generic one, we can understand each other more. We can also make our code more readable because more people become familiar with such patterns.

How Patterns are Used?

- Design Problem.
- Solution.
- Implementation details.



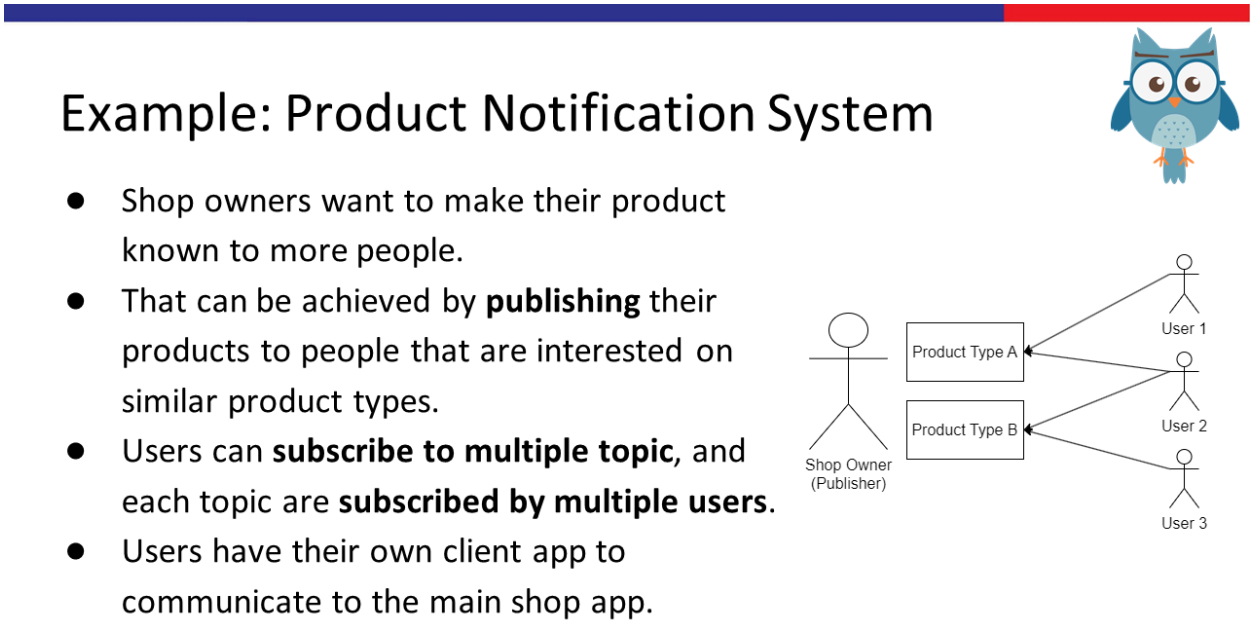
Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. 1995.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-oriented software architecture: a system of patterns*. 2002.

Design patterns will reduce the gap between the understanding of users/clients, analysts, and developers. Developers will be less likely to be confused. Catalogue of design patterns can also include implementation examples, such as this beautiful catalogue from Refactoring.Guru: https://refactoring.guru/design-patterns/examples. You can look at different examples of design patterns from different programming languages there. Everyone can look at it, including analysts and system testers.

Motivation, Explained by Case Study

Product Notification System



Imagine you have a shop. Of course, you want to increase the publicity of your shop and/or your products, right? To increase the publicity of things, we need to... obviously communicate! It can be anything, from advertisements, to notifications and mailing lists.

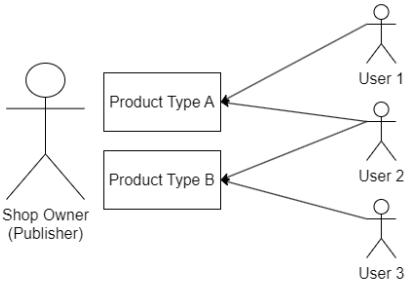
In this case study, shop owners want to publish their products to a set of people that like a certain type of product. Users will be given free choice of one or more topics they want to subscribe to. A topic can be subscribed by multiple users.

In this case study, the notification is not happening only in one single application, but through other apps. Let us imagine about your favourite e-commerce store: they have their own back-end server, and they have a mobile app that can be used via your own phone. You did not access the server directly. You access features of the shop via the app on your phone. In this case study, the notifications of products will be given from **server** to a **Receiver app**.

This is the topic of this week's Tutorial. Look at the bottom part of this module to check the tutorial.

Publishers + Subscribers = Observer Pattern

- Treat **Shop Owner** as the Subject/Publisher
- Treat **Users** as the Subscribers/Observers
- Subscriber/Observer **can subscribe or unsubscribe** from the Subject/Publisher
- Changes in Subject/Publisher are "broadcasted" to every Subscribers/Observers, by calling a method.



There are two roles from this case: the **Shop Owner** and the **Users**. To make the vocabulary more generic, we can call the **Shop Owner** as the **Publisher** and a **User** as a **Subscriber**. A Subscriber can subscribe or unsubscribe from the Publisher. In this case, the Publisher offers some product types to subscribe to, or we can call it the **Topic**.

With the role of **Publisher** and **Subscriber**, there is a design pattern for the communication process called **Observer** design pattern.

The (Problem, Solution) of Observer Pattern

*"Different kinds of **subscriber objects are interested in the state changes or events of a publisher object**, and want to react in their own unique way when the publisher generates an event. Moreover, **the publisher wants to maintain low coupling** to the subscribers. What to do?"* (Larman, 2001)

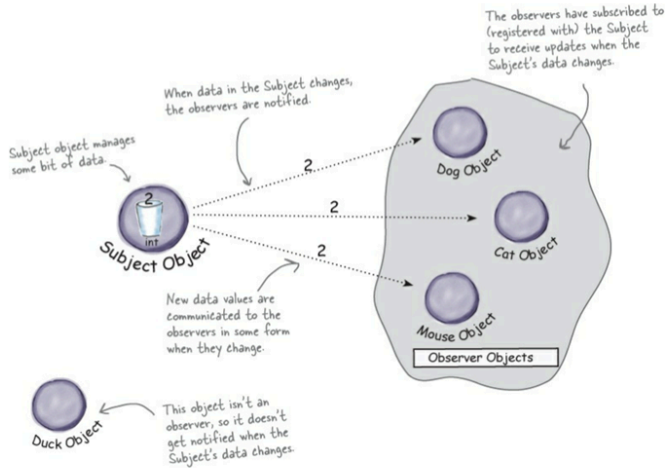
*"**Define a "subscriber" or "listener" interface**. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event, and notify them when an event occurs"* (Larman, 2001)

Source: Larman, Craig. "Applying UML and pattern: an introduction to object oriented analysis and design and the unified process." (2001).

Larman (2001) from his book explaining Observer design pattern by a paragraph of Problem and a paragraph of Solution mentioned above. Here is the explanation:

- **Problem**
  Subscriber objects are interested in the events of a publisher object, but the publisher **does not want to be coupled** to the concrete classes of subscribers. It does not want to know much about each concrete class of subscribers.
- **Solution**
  We will abstract the **Subscriber** term as an interface. The Publisher then will depend on the Subscriber interface to dynamically register subscribers and notify each of them when an event occurs.
  Remember **"Dependency Inversion Principle"** back in Module 3, that the code should only depend on abstractions, not concrete implementations. Thus, if we make alternatives of the concrete implementations, we do not need to modify the dependent code, which then conforms with **"Open-Closed Principle"**.

Publishers + Subscribers = Observer Pattern

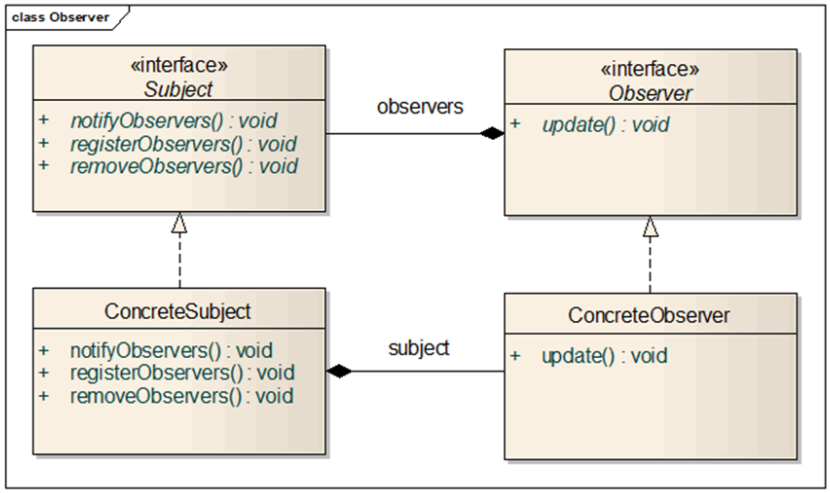


Taken from Head First Design Patterns pg. 45

This is the data flow illustration of the pattern. To read the explanation, you can either zoom in, or read this:

1. Let us say for example Dog, Cat, Mouse, and Duck objects are implementing an Observer interface. Dog, Cat, and Mouse are subscribed to (registered with) the Subject (Publisher), while Duck does not.
2. When there is a change in the data, let us say Subject changed the integer to 2. Subject then will be notifying the observers in the same way for each observer.
3. The observers, which will receive the data from the Subject, will need to extract or process the data.
4. Observer design pattern defines a **one-to-many** dependency between Subject/Publisher to Subscriber/Observer.

Observer Pattern

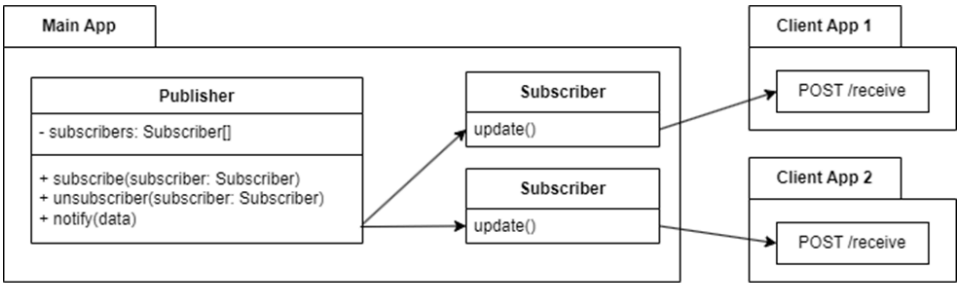


Here is the class diagram for the Observer design pattern. There are two abstractions: **`Subject`** (or Publisher) and **`Observer`** (or Subscriber). Concrete objects that represent a role will implement the respective interface for that role. **`Subject`** then will call each **`Observer`**'s `update()` method to give the data and let the Observer process it.

Previously, we did not see the **`Subject`** interface yet, so why does this interface show up here? Because by also abstracting the **`Subject`**, we can implement multiple subjects with different things to share/notify and diverse ways to build the data. This further improves the compliance to **"Open-Closed Principle"**, where the code will be easy to extend without any modifications to existing code.

Observer Pattern for Different App Instances

- The class diagram shown in previous slide are for designing Publisher and Subscriber relationship inside a single app.
- What about the case of when the subscriber are a different instance?
- Use **“push notifications” via a network protocol** (such as: HTTP).



Then you wonder, what about when the **`Subscriber`** is not in the same application instance as the main app? Then we will still have the same thing as the usual **`Observer`** pattern, but this time, the **`Subscriber`** might just be a class that saves a URL and/or an identifier of the other instance. When the Publisher calls the **`update()`** method in the **`Subscriber`** object, the **`Subscriber`** object will then call (or make a request) to the respective **`Client`** app via a network protocol (HTTP for example). This is often called by **push notifications**.

Observer Pattern

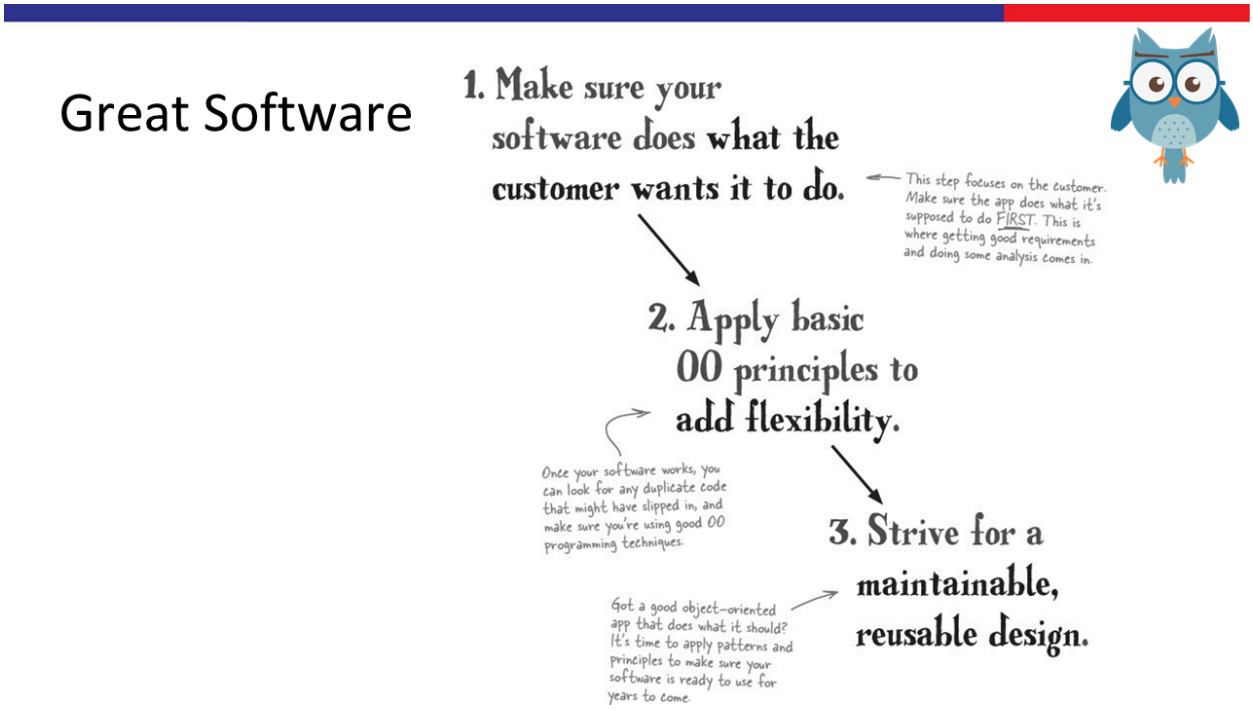
- Pros
  - Abstracts coupling between Subject and Observer
  - Supports broadcast communication
  - Supports unexpected updates
  - Enables reusability of subjects and observers independently
- Cons
  - Exposes the Observer to the Subject (push model)
  - Exposes the Subject to the Observer (pull model)

Observer design pattern allows us to add more **`Subject`**s and **`Observer`**s without modifying existing code. This is because we already abstract **`Subject`** and **`Observer`** into interfaces, thus the dependencies (or coupling) happen only to the interfaces.

Observer design pattern enables new possibilities on message passing, such as unexpected or emergency updates, and broadcast communication (via mailing list or push notification). This pattern can also be used to communicate between threads (see **Module 6: Concurrency** on Message Passing using **`std::sync::mpsc`** in Rust).

Observer pattern has some disadvantages though. It would expose **`Observer`** objects to the **`Publisher`**, if we use the Push model (**`Publisher`** sends data by calling **`Observer`**'s `update()` method). On the other side of the coin, if we use the Pull model (**`Observer`** actively calls **`Publisher`** to send data), the **`Publisher`** object will be exposed to the **`Observer`**.

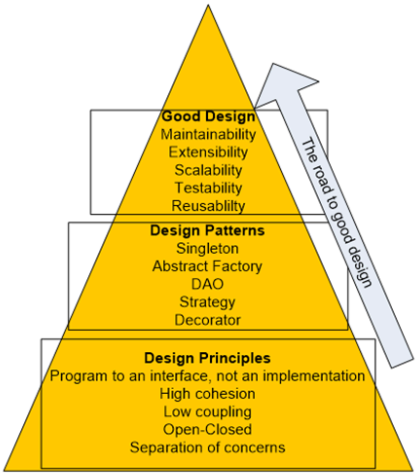
Why Use Design Patterns?



Three steps towards a **Great Software**:

1. **Make sure your software does what the customer wants it to do**.
   This can be achieved by **proper requirement gatherings and analysis**. We must do this **FIRST** before we can achieve anything. Although things about requirement gathering and analysis will be learned in a different course, you will learn about that by practice, via the Group Project.
   After the project kicks off, you also need to make **good tests** to achieve this. We have learned that in **Module 4: Test-Driven Development and Refactoring**.
2. **Apply basic OO principles to add flexibility**.
   We have learned about this in **Module 3: Object-Oriented Principles and Maintainability**. Look back at **S.O.L.I.D. Principles** explained in that module. Make sure you are following those principles when programming and understanding the requirement analysis.
3. **Strive for a maintainable, reusable design**.
   Sometimes, referring to the principles is just too... abstract. If you understand the principles though, you need to get up to this level: looking for a clean, maintainable, and reusable design. A **catalogue of design patterns** will help you to achieve that faster, as you do not really need to design from scratch. This is **why we use Design Patterns**.

Why Use Design Patterns?



This is another version of “Towards a Good Software” that we learned previously. Start from design principles (S.O.L.I.D.). We design our code and class structure based on the common patterns, while maintaining the compliance to the design principles. This will increase long term Maintainability, Extensibility, Testability, and in the end Reusability and Scalability.

Why Use Design Patterns?

- Design objectives: good design that supports the “-ilities”/non-functional requirements
  - Readability & maintainability
  - Extensibility
  - Scalability
  - Testability
  - Reusability

The “-ilities” that mentioned before, in other terminology is “non-functional requirements”. In **Module 3: OO Principles and Maintainability** we have learned that to make good software, we need to think beyond functional requirements (i.e. the functions that users want to do), but also non-functional requirements (i.e. things that make users/developers happy :D).

The Definition of Design Pattern

Elements of a Design Pattern

- Name
  - Describes the pattern
  - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
- Problem
  - Describes when to apply the pattern
  - Answers -- What is the pattern trying to solve?
- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences [Advantages/Disadvantages] (Subjective depending on the concrete scenarios)
  - Results of applying the pattern
  - Benefits and costs

There are four elements in Design Pattern:

- **Name**, of course! This element makes a Design Pattern a new common vocabulary among software engineers. Make sure that it has a relevant meaning, not just a “gimmick” or “technical person pleasers”.
- **Problem**. A design pattern needs to be specific on the general case it wants to solve. A good design pattern does not make an engineer fall into anti-patterns easily, or at the very least, gives engineers a sense of whether the pattern is suitable for their needs or not.
- **Solution**: Describe every element with their relationships, responsibilities, and collaborations (data flow) within the elements. If necessary, you can put an implementation example.
- **Consequences**: This **is important!** This element eases engineers to consider whether a design pattern suits them or not. Sometimes, people call it as Advantages or Disadvantages. Explain the result that the design pattern wants to achieve. Explain about the design principles it complies to, and the design principles it might violate when overused or misused. Also explain the benefits and costs in terms of maintainability, performance, and security.

There are four **types of design patterns** (**bold** means it will be explained in this module and slide, underline means it will be explained as bonus material):

1. **Behavioural** patterns: focus on behaviours of classes or designs a certain behaviour. https://refactoring.guru/design-patterns/behavioral-patterns
   - Chain of Responsibility: https://refactoring.guru/design-patterns/chain-of-responsibility
   - **Command**: https://refactoring.guru/design-patterns/command
   - Iterator: https://refactoring.guru/design-patterns/iterator
   - Mediator: https://refactoring.guru/design-patterns/mediator
   - Memento: https://refactoring.guru/design-patterns/memento
   - **Observer**: https://refactoring.guru/design-patterns/observer
   - **State**: https://refactoring.guru/design-patterns/state
   - **Strategy**: https://refactoring.guru/design-patterns/strategy
   - **Template Method**: https://refactoring.guru/design-patterns/template-method
   - Visitor: https://refactoring.guru/design-patterns/visitor
2. **Structural** patterns: focus on structure and hierarchies of classes. https://refactoring.guru/design-patterns/structural-patterns
   - **Adapter**: https://refactoring.guru/design-patterns/adapter
   - Bridge: https://refactoring.guru/design-patterns/bridge
   - **Composite**: https://refactoring.guru/design-patterns/composite
   - **Decorator**: https://refactoring.guru/design-patterns/decorator
   - **Façade**: https://refactoring.guru/design-patterns/facade
   - Flyweight: https://refactoring.guru/design-patterns/flyweight
   - Proxy: https://refactoring.guru/design-patterns/proxy
3. **Creational** patterns: focus on object creation or instantiation. https://refactoring.guru/design-patterns/creational-patterns
   - **Factory Method**: https://refactoring.guru/design-patterns/factory-method
   - **Abstract Factory**: https://refactoring.guru/design-patterns/abstract-factory
   - **Builder**: https://refactoring.guru/design-patterns/builder
   - Prototype: https://refactoring.guru/design-patterns/prototype
   - **Singleton**: https://refactoring.guru/design-patterns/singleton
4. Compound patterns: a design pattern that consists of multiple design patterns.
   - Model-View-Controller (MVC) for classic web programming**.**
   - Model-View-Presenter (MVP) for Microsoft Windows GUI programming.
   - (Data)Model-View-ViewModel (MVVM) for Android GUI programming.

Pros of Design Patterns

- Add **consistency** to designs by solving similar problems the same way, independent of language
- Add **clarity** to design and design communication by enabling a common vocabulary
- Improve **time** to solution by providing template which serve as foundations for good design
- Improve **reuse** through composition

Design patterns offer **consistency** to designs by solving similar problems "the same way" independent of language. Some languages still have their own unique approach or principles (e.g. between strictly OO languages like Java, or non-pure OO like Rust), but the main principles are still the same. Design patterns will also shorten **time** to create a software solution by providing a template of thinking process, which will lead into good design.

Design patterns often involve interfaces, as they really embraced the "**Program to interface, not implementation**" principle most of the time. This increases **reuse** as composition techniques are better at abstracting concepts than inheritance.

Cons of Design Patterns

- Some patterns come with negative consequences
  - Object proliferation, performance hits, additional layers
- Consequences are subjective & depend on concrete scenarios
- Patterns subject to different interpretations, misinterpretations, and philosophies
- Patterns can be overused and abused → **Anti-Patterns**

Design patterns are not without caveats though. Design patterns sometimes come with performance penalties. For example, the Singleton pattern on the lazy type will have performance hit due to the synchronisation process when creating a new object. Design patterns also often introduce additional layers, such as the Proxy or Façade pattern, but the additional layers do not always count as a disadvantage, because they might be useful.

Some design patterns will also create object proliferation. Object proliferation is the condition where the code generates too many little classes/objects. The symptom is when you need too many classes to do something exceedingly small. This often happened because we overused a design pattern or misused it to a less relevant case. The misuse, overuse, or abuse of design patterns are called **Anti-Patterns**.

Design Patterns in Rust

These presentation slides from the teaching team will only give you the design perspective, not how it will be implemented. In some of the slides, code examples are shown using Java.

To delve deeper on Design Pattern examples using Rust, you can visit these:

- The Rust Programming Language Book, Chapter 17 (OOP Features of Rust): https://doc.rust-lang.org/book/ch17-00-oop.html
- Open-Source Book of Design Patterns in Rust: https://rust-unofficial.github.io/patterns/intro.html
- Refactoring Guru's Example of Rust Design Pattern Codes: https://refactoring.guru/design-patterns/rust

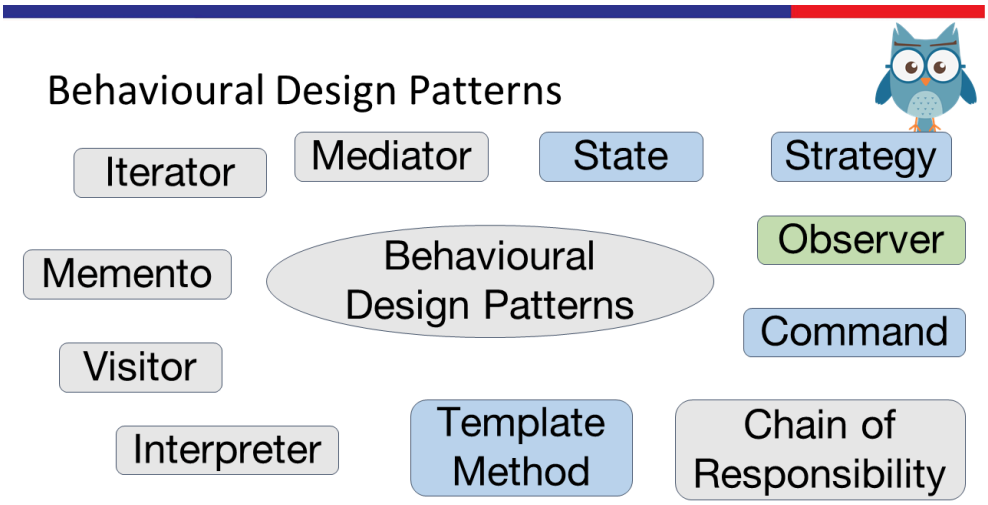
Since we used Rust in the tutorial or some of you may already use Rust for your Group Project, we have compiled some great learning sources about Design Patterns in Rust. Rust has many different OO concepts than the classic OO languages like Java. Some of them are:

- Stricter mutable/immutable, ownership, and lifetime management of variables.
- Rust does not mandate you to use classes all the time, unlike Java.
- There is no static variable in a struct.
- Rust does not support inheritance, as the programming language prefers "has-a" (composition) relationships.
- Rust has a slightly different concept of "interface", called **`trait`**. It is a bit like an abstract class, as we can add concrete methods there.

There are some of great learning materials for Design Patterns in Rust:

1. The Rust Programming Language Book, Chapter 17 (OOP Features of Rust). It explains how Object-Oriented works in Rust. It also includes a State design pattern tutorial. https://doc.rust-lang.org/book/ch17-00-oop.html
2. Open-Source Book of Design Patterns in Rust. It has a great catalogue of design patterns, not limited to OO patterns, and also including Rust-specific ones. https://rust-unofficial.github.io/patterns/intro.html
3. Refactoring.Guru Example of Rust Design Pattern Codes. One of the primary sources of design pattern learning material, now also gives Rust design pattern code examples. https://refactoring.guru/design-patterns/rust

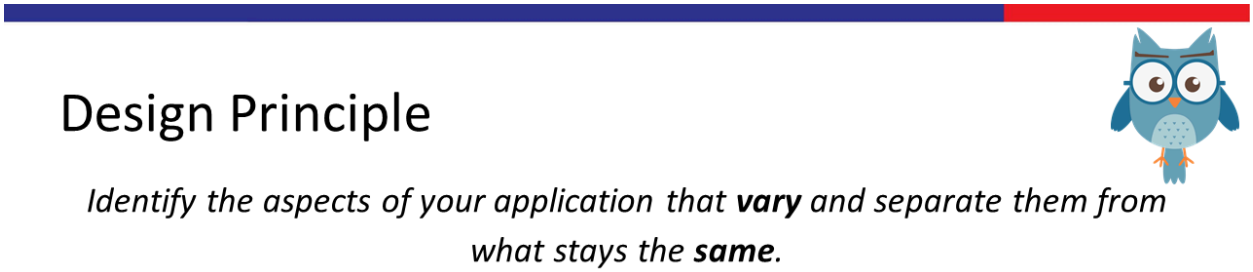
2. Behavioural Design Patterns



In this module, we will learn about Observer (already explained in the Motivation section), Strategy, Command, Template Method, and State pattern.

Strategy Pattern

Related Design Principles



In making design patterns, we need to identify which parts are the same in every case of the problem, and which parts that vary. Separate them so that adding a new variance will not make you change the codes that are not varying. **Open-Closed Principle** (open for extension, closed for modification).

Design Principle

*Program to an **interface**, not an **implementation**.*

- Previously: we were relying on concrete implementation inherited from parent class
- Now: we separate interface of a behaviour from its implementation

Making **interface**s is about separating behaviour to another abstraction. If we use inheritance, we are forced to rely on the parent class, when oftentimes we do not need all the parent class's features. If we use an interface, we can just focus on a certain function that we want, and a group of classes or objects that allow such a function to happen.

Design Principle

*Favor **composition** over **inheritance**.*

- Instead of **inheriting** behaviours, we **compose** the behaviours into an object
- Allows changing behaviour at runtime
- Used in many design patterns

By using interfaces, we compose behaviours into objects. The object then can be swapped to make different behaviours for the same function. We do not need to change the implementation of the class and recompile the program. This is used in many design patterns, such as **Strategy** and **State** pattern.

Strategy Pattern

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Strategy lets the algorithm vary independently from clients that use it

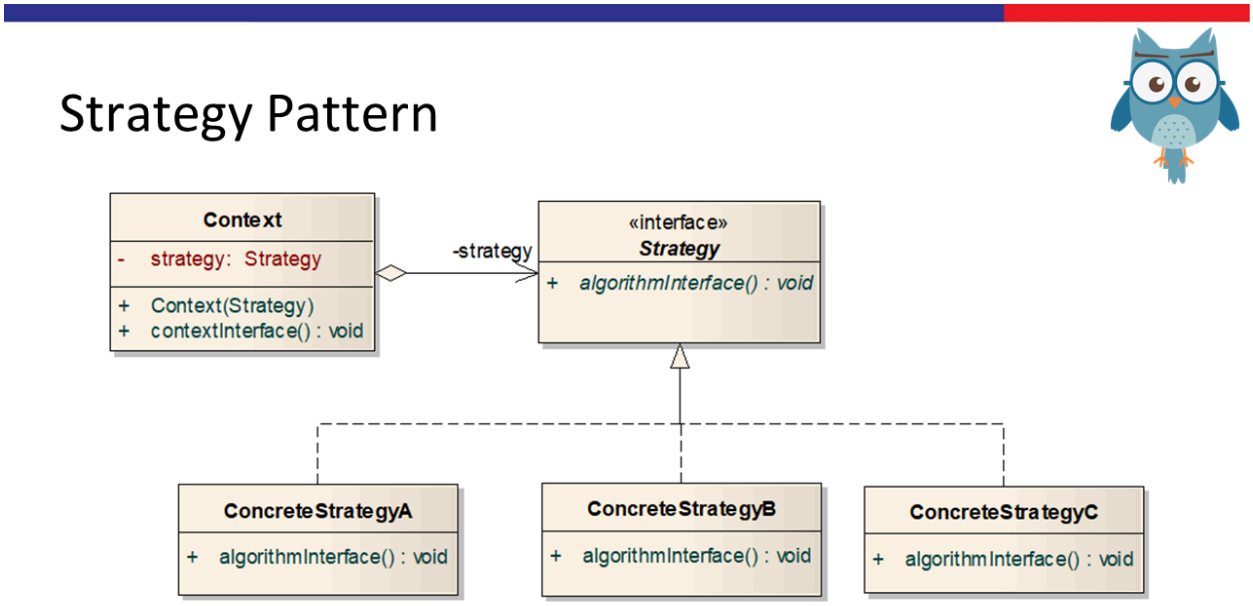
Let us start the explanation with a case study again. Imagine you have a **`Catalogue`** class that contains a list of **`Product`**s. You want to make a sorting feature to sort Products that will be shown to the user. Every Catalogue will have their own preference of what sorting algorithm they want to use. There are a lot of them to choose from, as you have learned in the Data Structures and Algorithms course: Selection sort, Insertion sort, Merge sort, Quick sort, etc.

The admin can change the algorithm via an admin page. So, the sorting algorithm should be able to change at runtime, no need to recompile or restart the app. If we designed the Catalogue class like this (code example is in Rust):

```
01 impl Catalogue {
02     pub fn sortProducts(&self, bool isDescending) {
03         // ... a Quick sort implementation
04     }
05 }
```

We must modify the code (or worse, make another subclass) to change the sorting algorithm. So, we need to separate the sorting algorithm to a separate abstraction, because the algorithm will vary. This is what **Strategy pattern** will do.

The Definition



Based on the case of Catalogue sorting algorithm, here are the roles:

1. **Context** will be the Catalogue.
2. **Strategy** is the abstraction of sorting algorithms.
3. **Concrete Strategy** like Insertion sort, Merge sort, and Quick sort, each will be a concrete implementation class that implements **Strategy**.

Then, where is the flexibility at runtime that we mentioned before? Look at the **`Strategy`** object, it will be saved in the strategy instance variable in the **`Context`**. When we instantiate a new **`Context`**, we are asked to include a **`Strategy`** object. The **`Strategy`** object can also be changed at runtime by making a setter method for strategy instance variables.

To run the sorting algorithm, just run `context.contextInterface()` (in this case will be **`catalogue.sort()`**). Then the **`sort()`** method of **`Catalogue`** will execute **`this.strategy.sort()`**.

Strategy Pattern

- Pros
  - Provides encapsulation
  - Hides implementation
  - Allows behaviour change at runtime
- Cons
  - Results in complex, hard to understand code if overused

Here we discuss the advantages and disadvantages of Strategy pattern. The main advantage is already explained: allowing behaviour change at runtime. Strategy pattern will also encapsulate and hide implementation of the sorting algorithm. Catalogue does not need to know about sorting algorithms, right? By encapsulating the algorithm to a different class, when we make a new sorting algorithm, we do not need to change or make a new Catalogue subclass.

The disadvantage of Strategy pattern is that if overused, the number of classes will be too much. If there are too many things that are separated into strategies, it will also make the Context constructor more complex because more parameters are needed.

Observer Pattern

This pattern has been explained in **Chapter 1** when explaining for **Product Notification System** case study. Observer Pattern is a Behavioural Design Pattern, as it regulates the behaviour of message passing from a Publisher (data owner) to Observers (subscribers) that are interested in the data.

Push and Pull Model

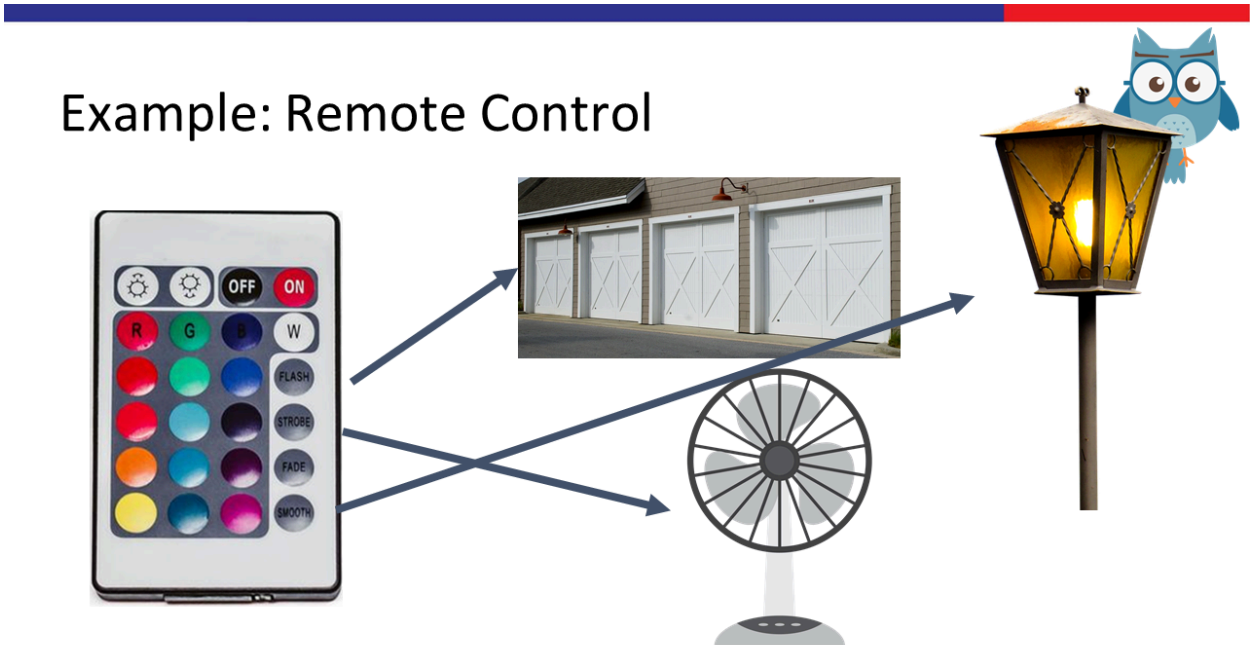
Observer Pattern has two kinds of model:

1. **Push model**
   This type makes the **`Publisher`** responsible to share the data to all **`Observer`**s subscribed to the topic. Publisher will call the **`update()`** method for each **`Observer`** that subscribed to the topic. The **`Observer`** will do nothing other than extracting or processing the data given by the **`Publisher`**.
2. **Pull model**
   This type requires the **`Observer`** to be active. Observer knows which **`Publisher`** it should contact, when asking for data. **`Observer`** then calls **`notify()`** method, then expects data returned from the **`Publisher`**.

Each of the models is useful for different cases. Push model is useful when we need real time notifications. Pull model is useful when we need on-demand notification.

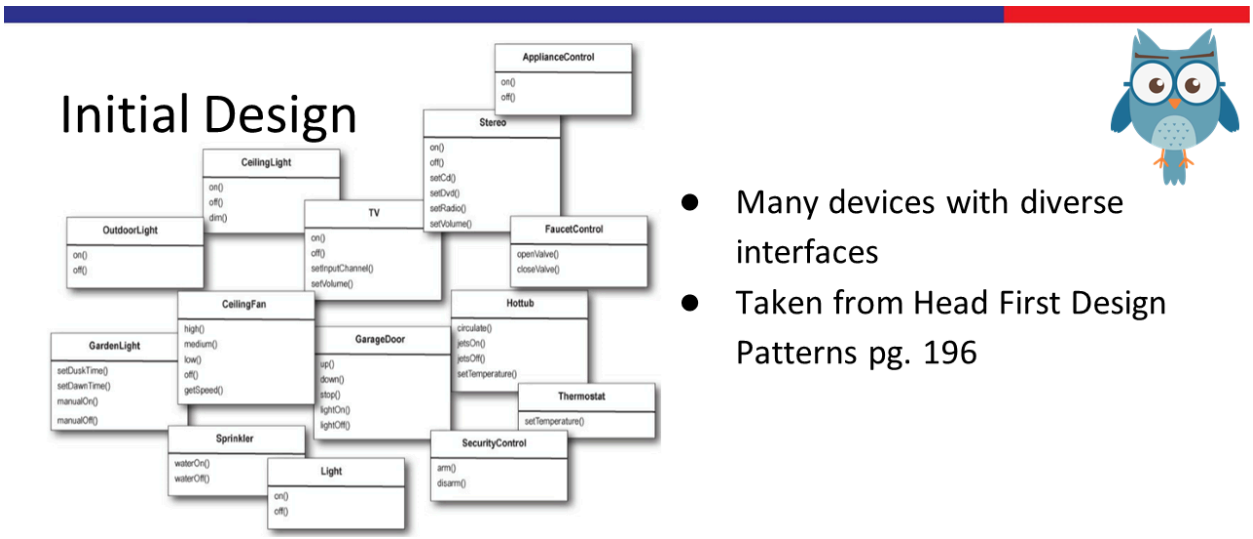
Command Pattern

The Motivation



Imagine you have a generic remote control. A remote control has a set of buttons. Because this is a generic remote control, it is not tied to a single device, but can be used for multiple devices (and even various kinds of electronic devices). Thus, each of the buttons will work differently on various kinds of devices. For example, turning on a lamp is different from turning on a TV.

How can generic remote control exist? How can it handle such a broad variety of interfaces?



- Many devices with diverse interfaces
- Taken from Head First Design Patterns pg. 196

Responsibilities

- What does a remote can/cannot do?
- Does a remote need to know how to perform actions on devices?

We start the discussion about generic remote-control design, by thinking about roles and responsibilities for every party that is involved. What can a remote-control do? Does remote-control only consist of a list of commands? Does a remote-control need to know how to perform actions on devices?

The answer is... remote-control can only **invoke the command**, they do not need to know about how devices perform the actions asked by such a command. If a remote control needs to know about how devices work, then what happens if the device changes its implementation? The remote-control would instantly be useless.

Responsibilities

```
if slot1 is Light:
  light.on()
elif slot1 is HotTub:
  hotTub.jetsOn():
elif ...
# and many more
# devices & slots
```

Bad design

See this example? That is right, it is the bad design we have mentioned previously. It has a lot of if-else conditionals, thus painful to read (see **switch statements** code smell in **Module 4: TDD and Refactoring**). If there is a new device we want to support, it will be even more difficult.

Responsibilities

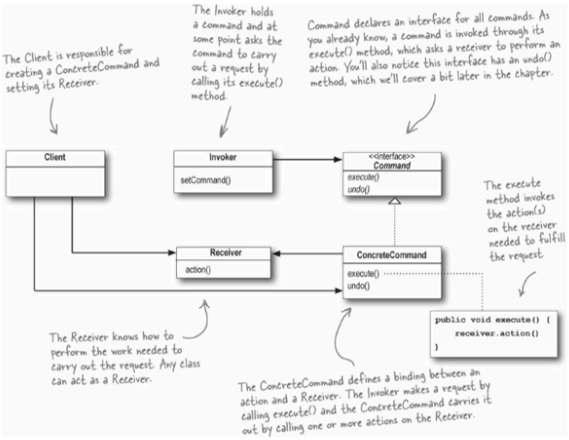
- Client
  - Make Command object & give it to Invoker
- Invoker
  - Delegate execution of Command object to Receiver
- Receiver
  - Execute the instructions
- Command
  - Has instructions
  - Know the Receiver that will perform the instructions

There are 4 (four) roles of this case study:

- **Client**: assume this is us, the developers. We know about the instructions and what devices we want our remote to support.
- **Invoker**: this is the button of a remote control. Invoker saves the reference of a Command. Invoker delegates the Command object to the Receiver (just like a remote-control sending data via infrared).
- **Command**: this is the instruction you want to give to the device. Command knows how to instruct a specific device. Command saves the reference to the device it wants to instruct. A Command object can do this by invoking a method of the specific device that it targets.
- **Receiver**: this is the device. Receiver understands the instructions brought by a Command and will execute it after the instructions being invoked.

Command Pattern

- Encapsulates a request as an object, thereby letting you parameterise other objects with different requests, queue or log requests, and support undoable operations



Taken from Head First Design Patterns pg. 209

Introducing... **Command pattern!** This design pattern translates those roles we have learned before, to a class design. In this case, **`Client`** knows which **`Receiver`** it possesses or supports. **`Client`** also knows which **`Invoker`**s it possesses. **`Client`** will create the **`Command`** object as an instruction to the **`Receiver`**, then associate it to an **`Invoker`**.

There is an interface (abstraction) called **`Command`**. It abstracts the command data, so that command data will be generic regardless of the **`Invoker`** and **`Receiver`** it will be associated with. It has methods to instruct a **`Receiver`** to do things (and/or undo the instruction, and/or batch instructions). Those methods have the information on which method in **`Receiver`** should be executed, for example: **`execute()`** method will invoke **`receiver.turnOff()`**.

Better Option: Functional Programming style (Lambdas)

Simplifying Command Objects: λ

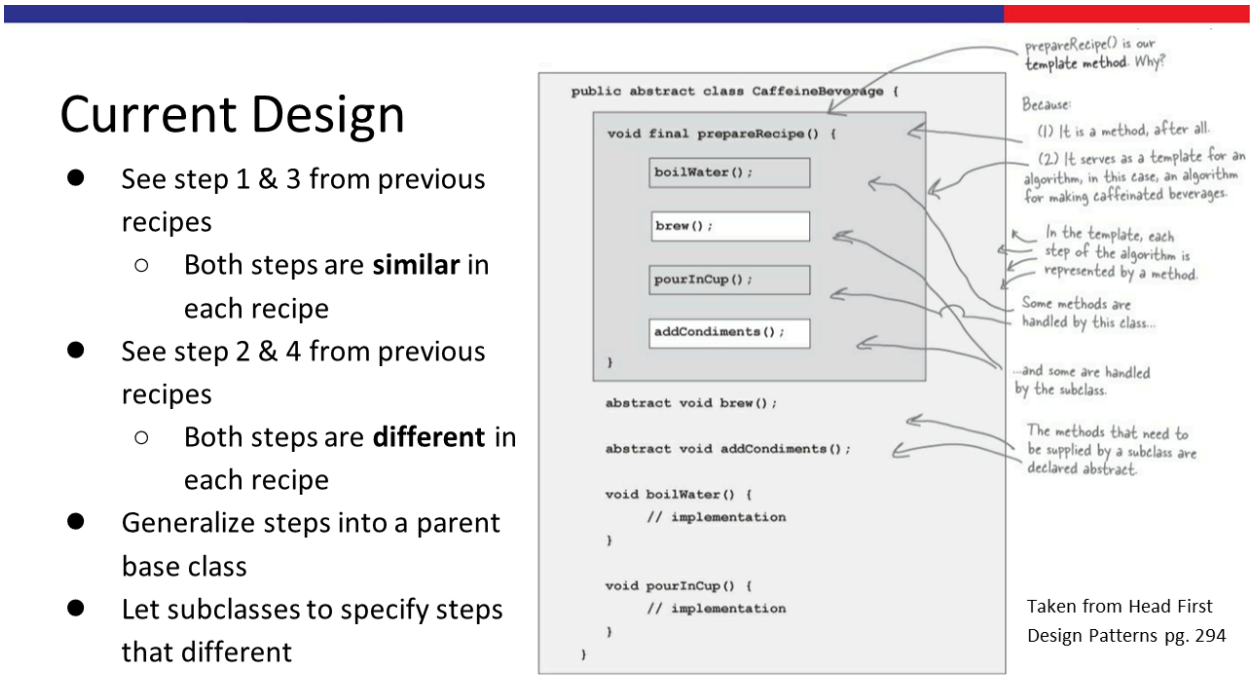
- Suppose that we have to create great number of `Command` classes
- Imagine the boilerplate for every `Command` classes!
- Simple `Command` classes follow SAM (Single Abstract Method) pattern
  - SAM can be refactored into lambda function

Now that you have seen the class diagram, you might wonder how tedious it is to create so many objects to command multiple things. **Beware of object proliferation!** As the **`Command`** only contains a single method (except if it has “undo” or “batch” feature), then we can simplify this by using **`lambda` functions** instead of objects. A lot of programming languages support this (especially Java, Python, and Rust).

Template Method Pattern

The Motivation

We have covered this pattern back in the **Module 4: Test-Driven Development and Refactoring** as a refactoring step. You can look back to the Refactoring material if you want to, but we will explain it again deeper here.



For example, here is a coffee maker. Coffee drinks are made using these four steps: (1) boil the water, (2) brew the coffee, (3) pour the coffee into the cup, then (4) add condiments. We know that boiling the water (step 1) and pouring the coffee into the cup (step 3) is the **same for all coffee**. Step 2 and step 4 are different for each type of coffee drink, because a different kind of coffee drink might also use a different type of coffee bean and condiments.

From that illustration, we know that the order of the steps (the algorithm) to make coffee drinks are the **same** for any kind of coffee drink. Some of the steps have varying implementations. Thus, we can make a **template** for the algorithm. We can make a **Template Method** called **`prepareRecipe()`** that contains calls into each step orderly.

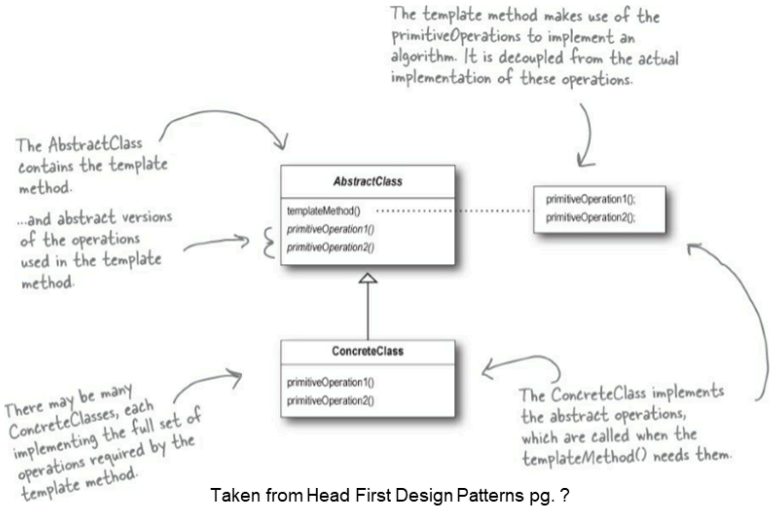
Then, we can implement Step 1 and Step 3 right away in the superclass. For every kind of coffee drink, we need to make a subclass of it. The subclasses will then implement Step 2 and Step 4.

Template Method Pattern

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses
- Lets subclasses redefine certain steps of an algorithm **without changing the algorithm's structure**

From the illustration, we know that the motivation of the **Template Method pattern** is to give the skeleton of an algorithm in a method. This pattern is only suitable if the order of steps done by the algorithm is **the same across all variants**, and the difference only happens in each implementation of some (or all) of algorithm steps. By delegating the implementation duty to subclass, we can easily extend a new variant of the algorithm by making another subclass.

Template Method Pattern



Taken from Head First Design Patterns pg. ?

This is the diagram of the **Template Method pattern**. The superclass is an abstract class. A concrete class that will be the subclass, must implement the abstract methods.

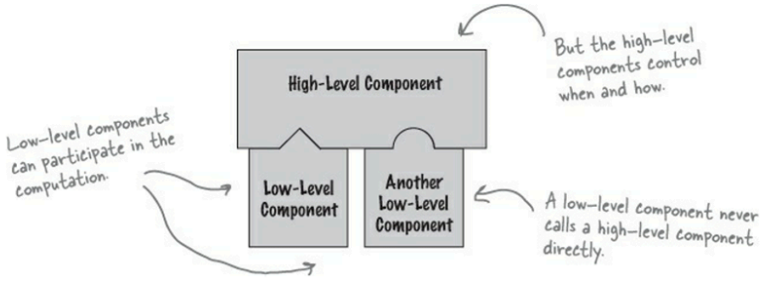
Hook/Extension Point

- Concrete, default instruction in the algorithm that can be overridden
- How is it different with abstract instruction in the algorithm?
  - Review the first point ;)

Although subclasses are only required to implement the abstract methods, we may extend the functionality further; If we want to do a single step in a separate way than the default definition, we can simply override them. The only things that are forbidden in the Template Method pattern are overriding the order of the algorithm or overriding the template method.

Design Principle

Hollywood Principle: "Don't call us, we'll call you."



Taken from Head First Design Patterns pg. ?

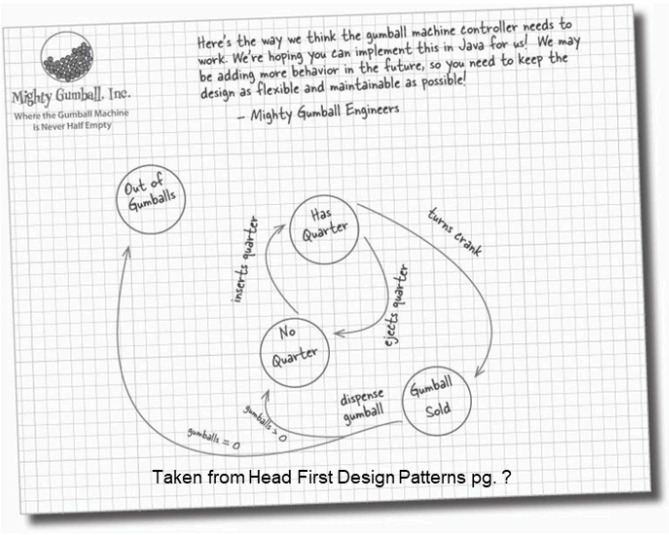
This design principle was inspired by the talent recruitment system of Hollywood movie production houses. They called the targeted talent directly if a role is available. A talent does not need to call Hollywood production houses to seek jobs.

In the context of Template Method pattern, a client will invoke the "template method" of the superclass. The "template method" will then only call implementations in the subclass if there is a difference of implementation in the subclass, or when the implementation in the superclass is absent (abstract method). We consider the superclass as a High-Level Component, and the subclasses as Low-Level Components.

State Pattern

The Motivation

Example: Gumball Machine



In this semester, alongside Advanced Programming, you might also learn the Language Theory and Automata course. If you have taken or currently taking the course, do you feel that diagram is familiar to you? Yep! That is called a **Finite State Machine (FSM)**. In the Language Theory and Automata course, you have learned about FSM to verify strings. The state is the circle nodes. The state contains information about the current approved string. Meanwhile, the edges are the actions, for example: if we found “a” in the next character, move to another state.

In this example, we have a gumball “gacha” machine. The state starts with **No Quarters**, meaning it waits for a quarter coin to be put into. Then if a coin is inserted into a machine, the state moves to: **Has Quarter**. The machine then checks the coin: If the coin is not quarter, go back to **No Quarter** state; else, turn the crank and change the state to **Gumball Sold**.

When the state is **Gumball Sold**, it will eject the gumball. If there are still remaining gumball(s), return to **No Quarter**. If there are no remaining gumballs, then change the state to **Out of Gumballs**, as the final state (the machine will be stopped here).

State Diagram

- Graph that models possible states and its transitions
  - Node: state
  - Directed Edge: transition (action)
- States in Gumball Machine?
  - No Quarter, Has Quarter
  - Out of Gumballs, Gumball Sold

Based on the explanations, there are four states, and four directed edges (actions). The “disperse gumball” action diverges into two conditions though, gumball empty OR gumball(s) still exist. That still counts as one action, we just need to make conditionals inside of the action method.

Initial Design

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

        System.out.println("You can't insert another quarter");

    } else if (state == NO_QUARTER) {

        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");

    } else if (state == SOLD_OUT) {

        System.out.println("You can't insert a quarter, the machine is sold out");

    } else if (state == SOLD) {

        System.out.println("Please wait, we're already giving you a gumball");

    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

Taken from Head First Design Patterns pg. ?

So here is the initial design. We make a single class **`GumballMachine`** that has four methods (methods resembling actions). Then, inside of each method, implement conditional branches for each state. If the machine is in a state, then do something.
This design smells fishy, right? Tell us what is fishy.

New State Means ....

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

First, you'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

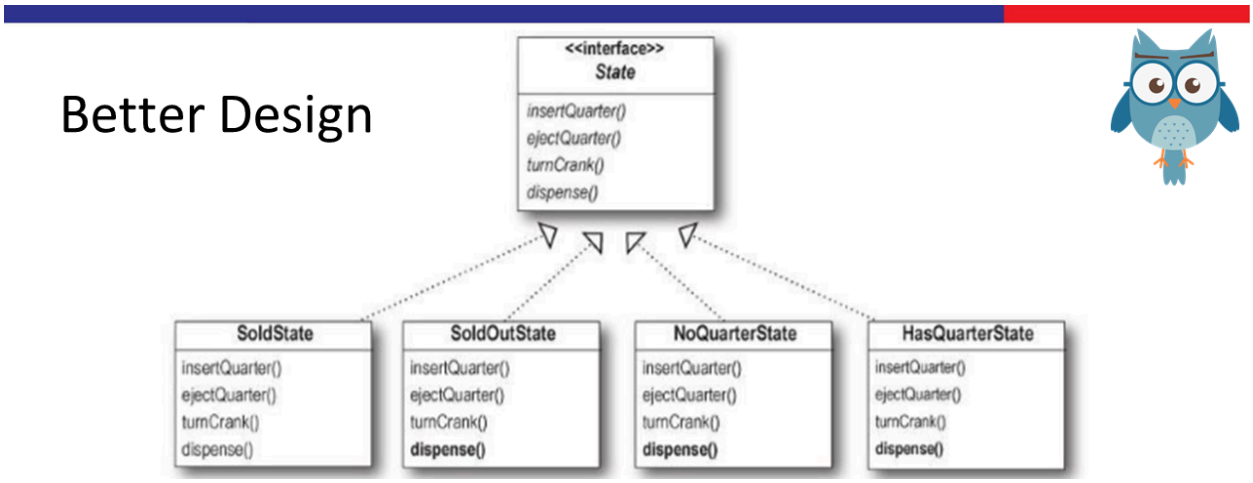
Taken from Head First Design Patterns pg. ?

Okay, now tell us how you will implement another state? What happens if this turns into a “gacha” machine with random probability of getting a single gumball, and we need to define a condition when the **WINNER** got a gumball?

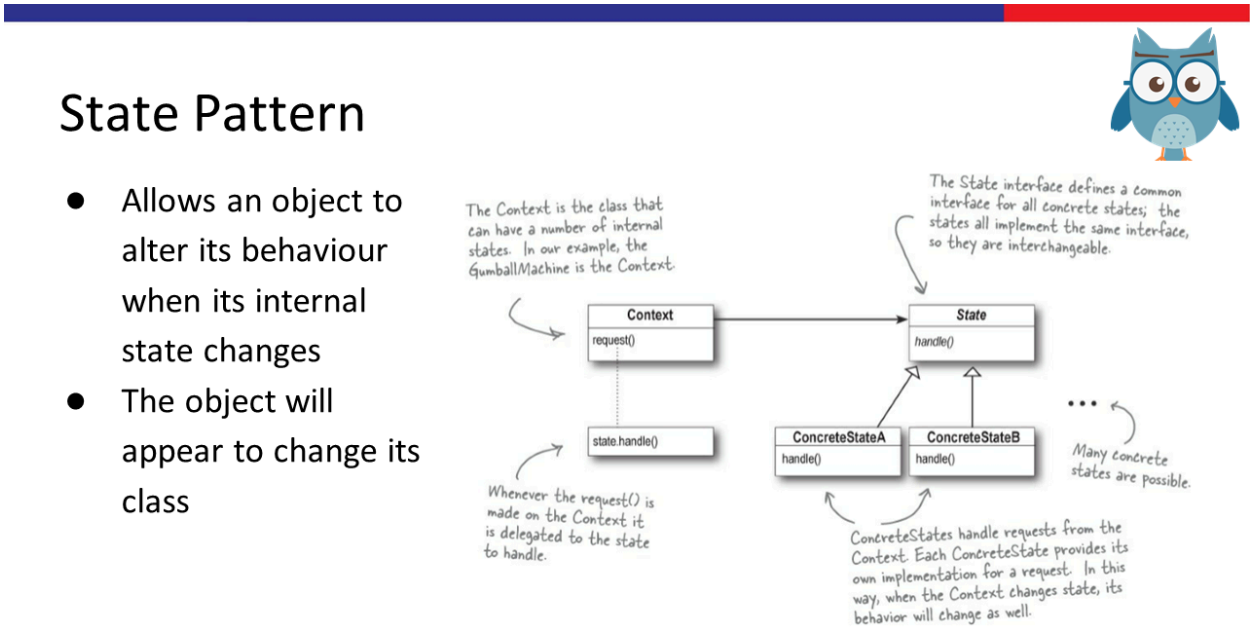
If we want to implement another state, it will be super tedious as we need to modify each action method. This will also make the code harder to read as it is more difficult to understand what happens if a machine is in a certain state. We need to jump here and there across methods just to understand how a state works.

So, let us **abstract the State!**

The Definition



State design pattern introduced an interface to abstract a State. Every kind of state will now have the same action methods. In the case of the gumball machine, which means every state will have 4 (four) methods. Each state will have their own implementation for each action method.

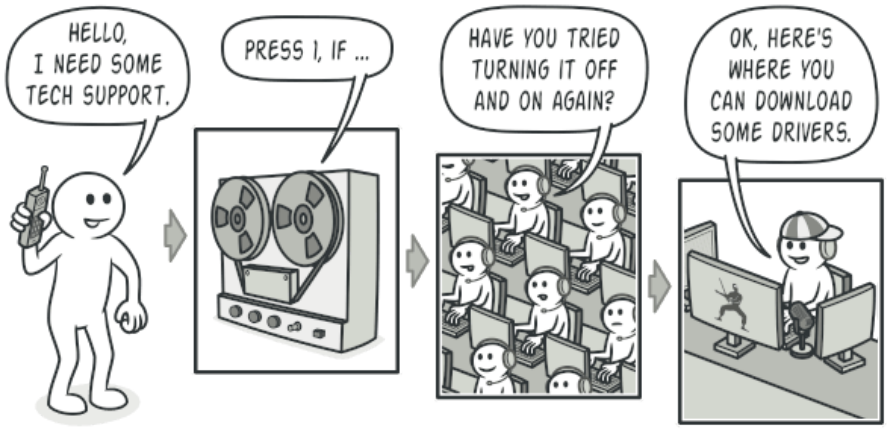


State pattern has similar structure like Strategy pattern, but with different intent. The **State** is a condition of the `Context`, meanwhile **Strategy** is an algorithm variation of the `Context`. To do an action, the action method in **`Context`** will execute the action method of the current state. **`Context`** saves current state into an instance variable. To change the state after executing an action, the action method can invoke **`context.changeState()`** to give a new State object.

Bonus Material: Chain of Responsibility Pattern

This material is not included in the slide. We put this here as a bonus material.

The Motivation



Let us imagine that you have a startup. The peak season is coming, and you are worried you will get a lot of complaints to handle. So, you want to make a customer support system to handle those complaints. In this case, you want the system to handle a complaint using four layers:

- **The automated computer response**, which will give common help or tricks for common problems (or FAQs). It does not have any understanding of the user's problem. If the user is not satisfied with the answer, then the complaint will be passed to...
- **The newcomer customer service staff**: a set of responders that have common knowledge to a problem. The staff are not IT specialists, just experienced users. If the user is still not satisfied with the answer, then the complaint will be passed to...
- **The rookie IT service staff**: a set of responders that have a Computer Science bachelor background and serve as junior developers (or interns). The staff are IT specialists, just that they might not have enough experience to tackle advanced problems caused by intricate bugs. If the user is still not satisfied with the answer, then the complaint will be passed to...
- **The lead of IT staff**: The problem is so advanced that it reaches this point. This case is exceedingly rare, only the leader (or senior engineers) can handle the problem, of course with advanced follow-ups.

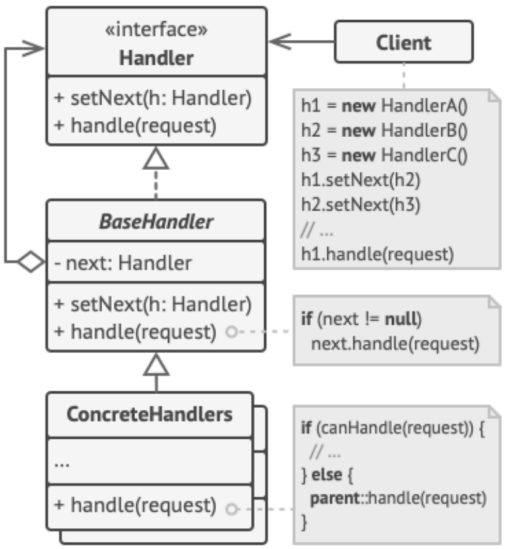
Each of these steps are the **`Handler`**s of the complaint. If the user is satisfied with the answer of a Handler, the user will end the call, thus the call is not passed to another **`Handler`**. If the user is not satisfied, it will reach the final **`Handler`**. The final **`Handler`** will then decide the final verdict of the problem.

The Definition

1 The **Handler** declares the interface, common for all concrete handlers. It usually contains just a single method for handling requests, but sometimes it may also have another method for setting the next handler on the chain.

2 The **Base Handler** is an optional class where you can put the boilerplate code that's common to all handler classes.

Usually, this class defines a field for storing a reference to the next handler. The clients can build a chain by passing a handler to the constructor or setter of the previous handler. The class may also implement the default handling behavior: it can pass execution to the next handler after checking for its existence.



3 **Concrete Handlers** contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and, additionally, whether to pass it along the chain.

Handlers are usually self-contained and immutable, accepting all necessary data just once via the constructor.

4 The **Client** may compose chains just once or compose them dynamically, depending on the application's logic. Note that a request can be sent to any handler in the chain—it doesn't have to be the first one.

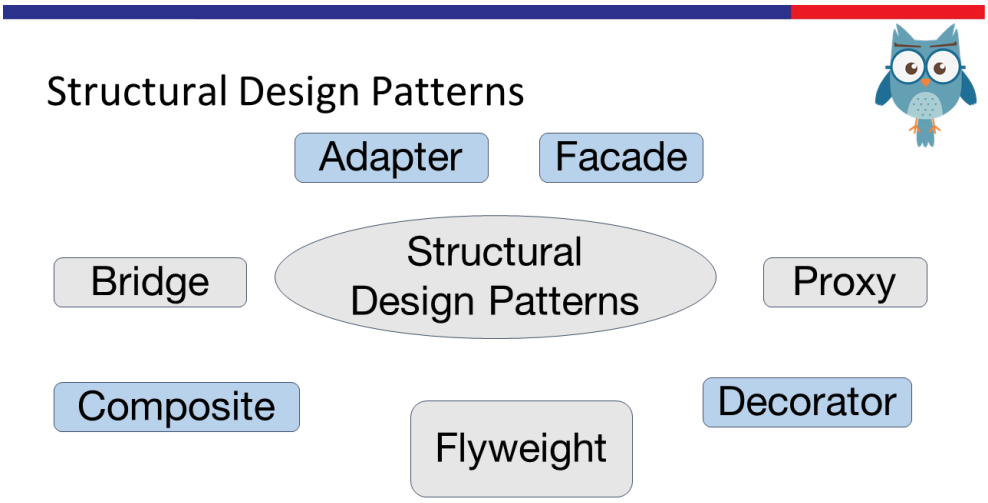
The **`Handler`** is uniform, their job is only handling a request. Thus, we need a **`Handler`** interface to abstract that. The **`Client`** will then create new **`Handler`** chains to form steps on handling requests, by assigning the next **`Handler`** into the previous **`Handler`** using **`setNext(Handler)`** method.

The **Chain of Responsibility** pattern allows for a single common abstract class called **`BaseHandler`** to be created. This is to reduce duplications on common methods or functionalities that are useful for **`Handler`**s to handle incoming requests. A **`ConcreteHandler`** will implement either the **`BaseHandler`** (if it exists) or the **`Handler`** interface.

The advantages of the **Chain of Responsibility** pattern are that we can separate implementations of request handling sequences from the **`Client`** to a class for each step. This pattern complies with **Single Responsibility Principle** due to the separation of concerns between **`Client`** and handling logic, and it also complies with **Open/Closed Principle** due to the ability to make a new **`Handler`** without modifying another **`Handler`** or the **`Client`**.

To make the **`Handler`** chain creation abstracted from the **`Client`**, we can also combine this pattern with **`Factory`** or **`Builder`** pattern. This is the practice that is used by Spring Boot (Java) web framework, when instantiating handlers of Spring Security platform.

3. Structural Design Patterns



In this module, we will learn about Adapter, Façade, Decorator, and Composite patterns.

Adapter Pattern

The Motivation

In real life, an adapter is a tool to convert from one thing to another thing. Our phone/laptop charger is also often called “Adapter”, as it converts the electricity from Alternating Current (AC) to Direct Current (DC). Another example is HDMI to VGA adapter that adapts the new interface (HDMI) as the input to the old interface (VGA) as the output.

Adapter Pattern

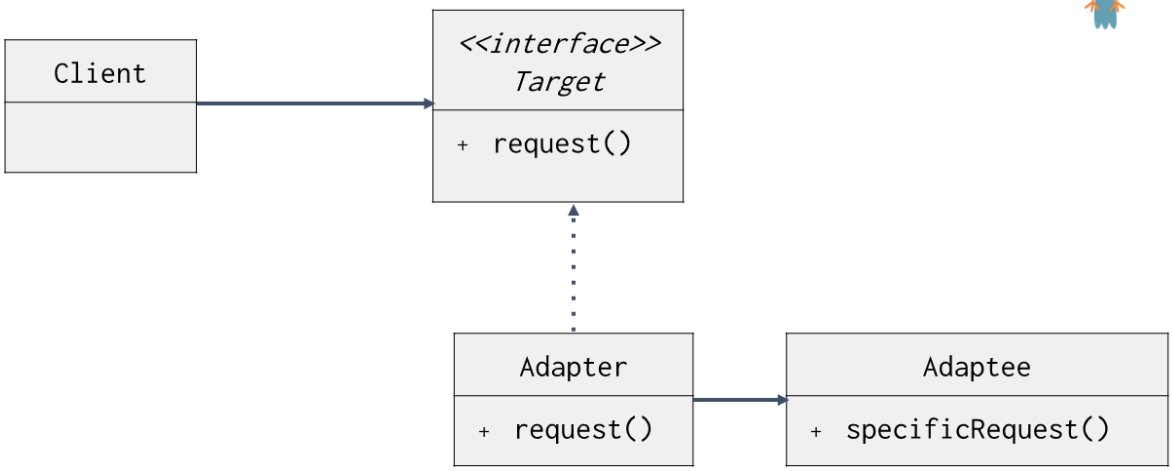
- Converts the interface of a class into another interface the clients expect
- Adapter lets classes work together that could not otherwise because of incompatible interfaces

In software development, we can do exactly that. Let us say we are a new vendor of the government's new software project. The project goal is to integrate all existing systems into one single new system. How will we reuse the old systems, when we cannot change the interface and the old systems are no longer maintained?

The solution will be... to adapt our new code to the old interface. We can make our own interface, then build the adapter class that implements the new interface. The adapter class then accesses the old interface. It also converts the data from the old system to be compatible with our new code.

The adapter is not all about old systems though. For example, we can use the adapter to avoid bottlenecks when doing teamwork. This can be useful when we depend on a code or subsystem made by other teams, but the other teams cannot finish it before we need to implement a feature that depends on the code/subsystem. We can make our own interface (a dummy) first, then we will adapt the other subsystem later when it is done. By doing this, we can also do tests to our code earlier, without waiting for dependencies. We have discussed this case in **Module 4: TDD and Refactoring** in the TDD part.

Structure



This is the basic structure of the **Adapter design pattern**. The **`Client`** (our code) will depend on the **`Target`** interface. This interface then can be implemented by a concrete class that contains our own implementation (for the case of avoiding teamwork bottlenecks), and by an **`Adapter`** class. The **`Adapter`** class will invoke specific methods of the **`Adoptee`** (the other system), then process the return data so the output can be accepted by **`Client`**.

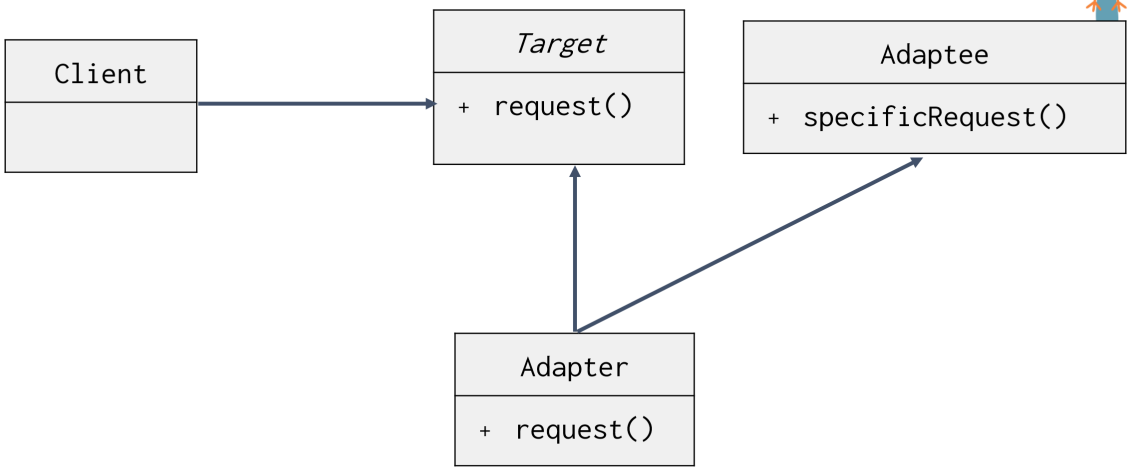
Object & Class Adapters

- Object → **composition** approach
- Class → **inheritance** approach
- Class adapters require **multiple inheritance**

There are two types of adapters:

- **Object Adapter**: The adapter type that will interact with the client via an Adapter object. This uses composition, interface-based approach.
- **Class Adapter**: The adapter class inherits **`Target`** class and **`Adoptee`** class. This is only possible for programming languages that support multiple inheritance, such as Python. The key difference with Object Adapter is that the **`Target`** is a class, not an interface.

Structure - Class-based Approach



In a class-based approach, the **Adapter** inherits two classes: the **Target** class, and the **Adoptee** class. This will clash if some of the Target and Adoptee methods have the same names. You might need a good understanding of Method Resolution Order (MRO) of the programming language you use, to prove which method will be accessed first. This also reduces the predictability of our code, thus making it less readable.

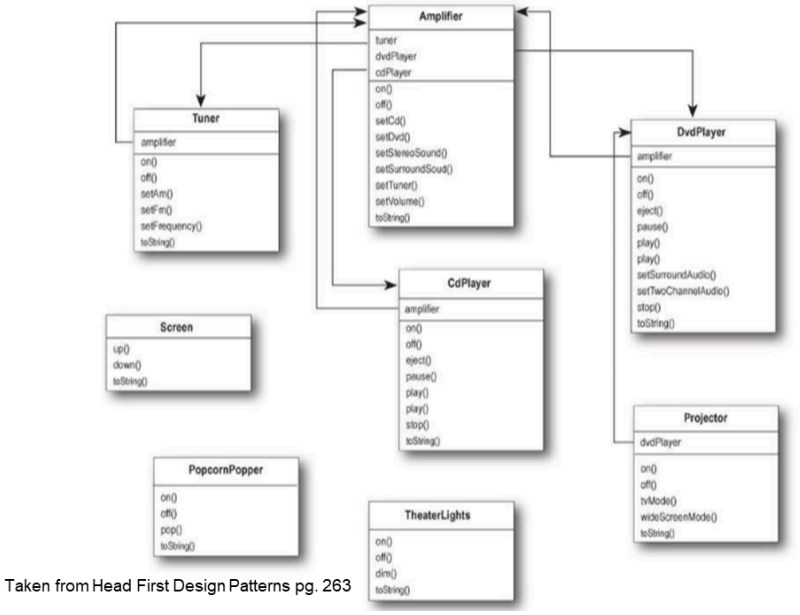
Object-based **Adapter** approach has another advantage like **Strategy** and **State** pattern: the **Adapter** object can be swapped at runtime. Imagine we have different versions of the system, and users sometimes want to change which version of the system they will use, without restarting the app. The app can easily adapt to the requested version of the system.

Both class-based and object-based approaches have the same advantage: We can make new adapters easily, only by making a new **Adapter** class. We do not need to change our existing code; the **Adapter** will adapt it for our system.

Façade Pattern

The Motivation

Initial Design



Taken from Head First Design Patterns pg. 263

Imagine we have such a complex system with a lot of classes inside that depend on each other. Can we just publish it to the public? Then how will a regular user who does not have a good understanding of the program, use this correctly? Of course, the complexity will confuse the user. To solve this problem, we need to do another requirement gathering; look at our target market's most common needs. Then we need to make a new interface that allows those common needs to be done easily.

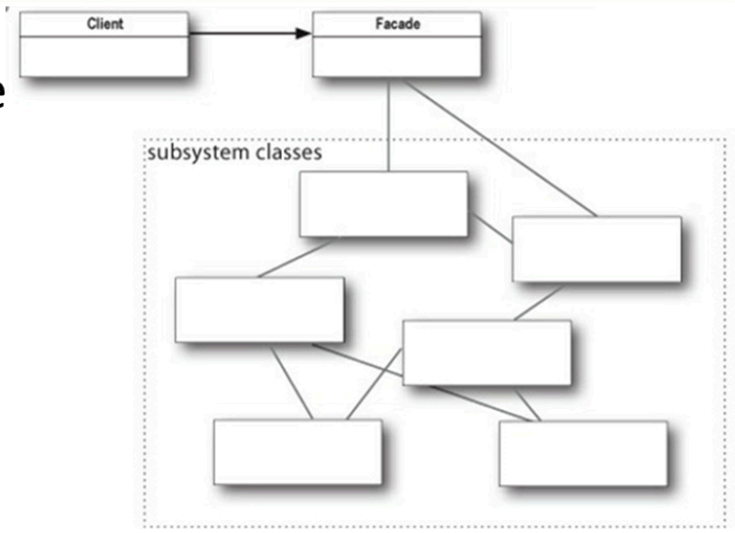
Facade Pattern

- Provides a unified interface to a set of interfaces in a subsystem
- Facade defines a higher-level interface that makes the subsystem easier to use
- It is still possible for client to access classes in the subsystem

That new interface of "common needs" is called **Façade**. It provides a unified interface to a set of interfaces/classes in a subsystem. Façade class serves as the higher-level interface that can be easily used by the user. Users can still access the specific classes in the subsystem if they want to do specific things that are not covered by the Façade.

The Definition

Structure



This is the structure of the **Façade pattern**. There is a class called **Facade** where the **`Client`** will depend on. The **Facade** class contains methods that are commonly used by users. **`Client`**s can still depend on members of the subsystem for its specific (or advanced) needs, although it is not recommended.

Design Principle

Principle of Least Knowledge -- talk only to your immediate friends

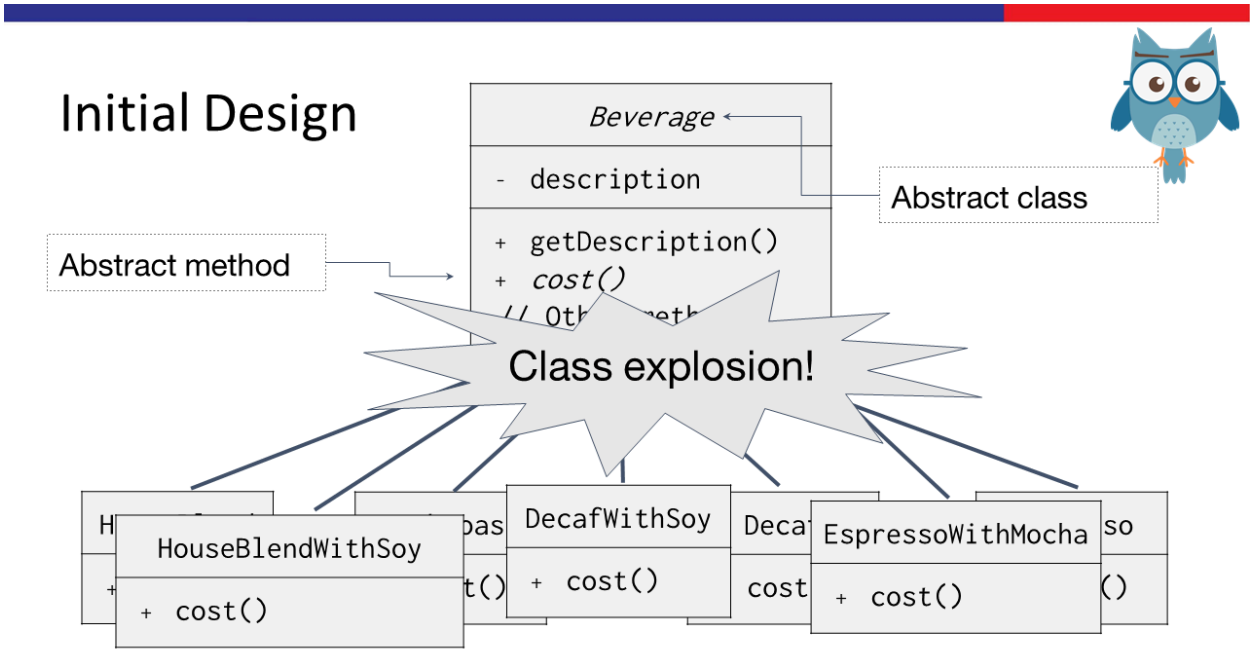
Only Invoke methods that belong to:

- The object itself
- Objects passed in as a parameter to the method
- Any object the method creates or instantiates
- Any components of the object

Façade pattern fulfils the Principle of Least Knowledge. A method should only access to the object itself (including instance variables and other methods inside that class), parameter objects, and objects that the method creates. Of course, accessing other classes is still allowed, but we need to strive for minimum dependency. This is to avoid hassles to modify our code when those dependencies change their interface.

Decorator Pattern

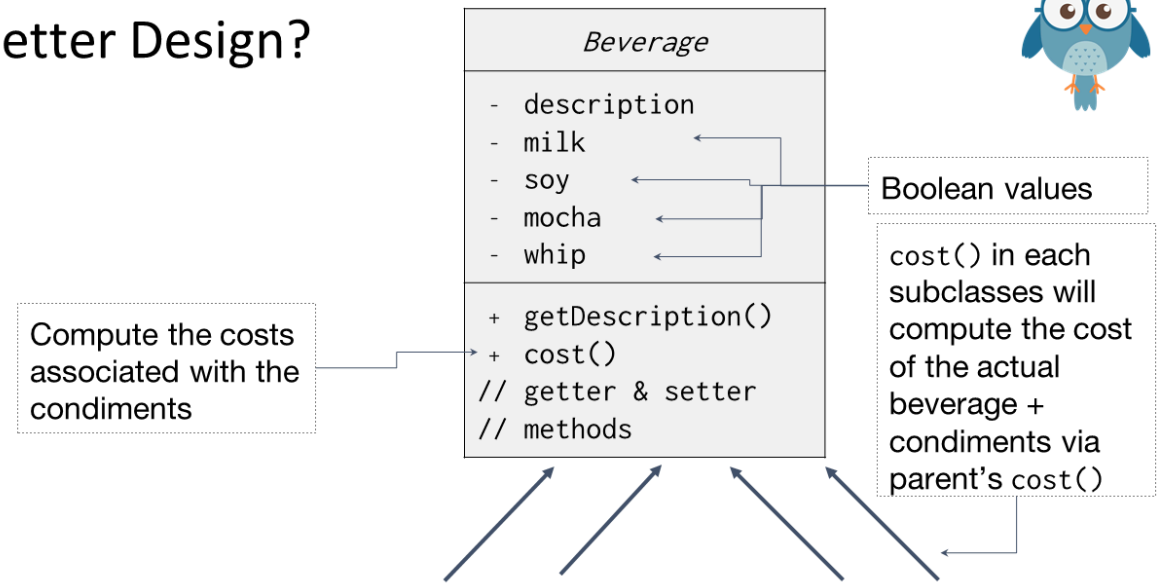
The Motivation



Let us open the discussion using a case study, again. Imagine we have a coffee drink shop. Coffee drinks now have A LOT of variations (thanks to those franchises that show up sporadically :D). Although modern cafes offer lots of coffee drink variations, the base is still the same for years. Let us assume there are four basic categories of coffee: House Blend, Decaf, Espresso, and Dark Roast. We can vary those with condiments, for example: House Blend with Mocha, House Blend with Soy, House Blend with Mocha + Soy.

Do you see the pattern? If there are 4 types of coffee bases and 5 types of condiments, your system will contain approximately $4 * 2^5 = 4 * 32 =$ **128 CLASSES!!!** Wow, that is called “class explosion”! To avoid this problem, we need to make an abstraction.

Better Design?



So here is your first attempt at making an abstraction. You simplified the **Beverage** class so that to make a new variant, we can just build a new object, not implementing a new class. You achieved this by making Boolean values as instance variables of Beverage. Sounds simple enough right?

No! Now tell us, if we asked you to add a few more (not too much, just ten :D) condiments. What will happen to the class and to its methods like **cost()**? It will create another problem: long classes and long methods. You also still need to modify Beverage implementation to add those condiments to the list.

Oh, for reference, here is some of the problem you will face when maintaining this class:

What Could Go Wrong?

- Price changes for condiments?
- New condiments?
- New beverages?
- Double condiments?

Then, what is the best solution?

Related Design Principles

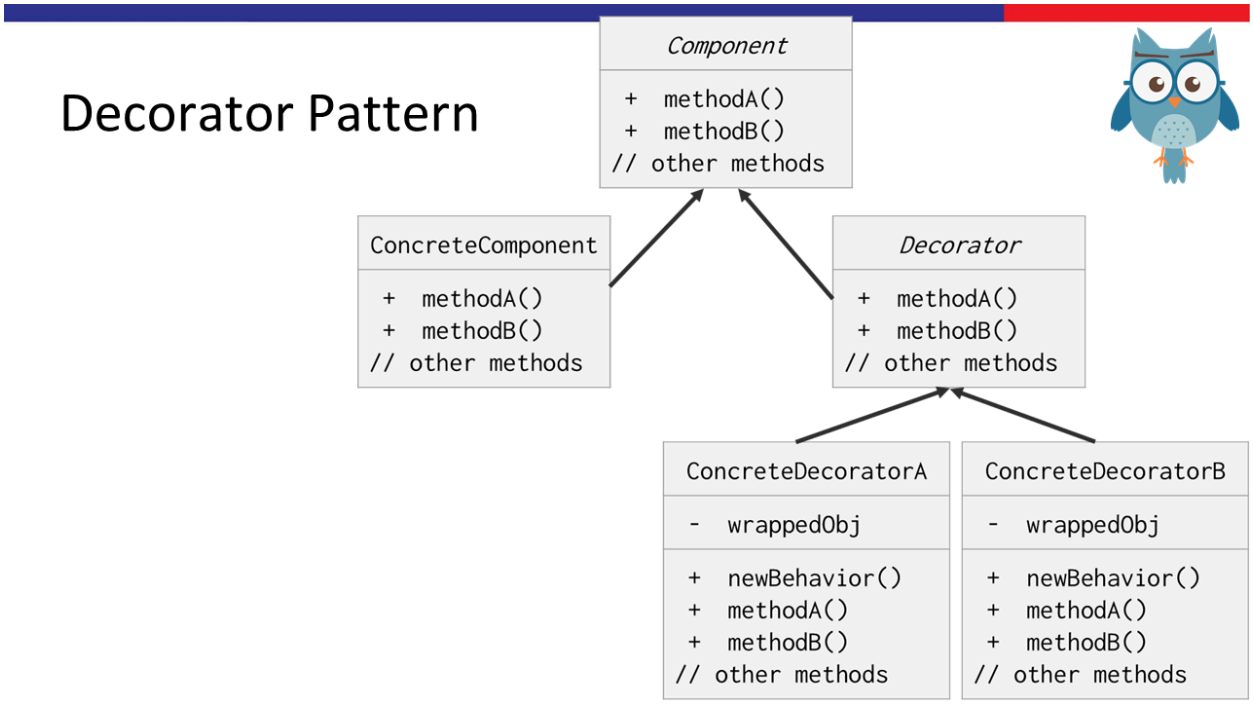
Design Principle

Open-Closed Principle -- Classes should be open for **extension**, but closed for **modification**

- Allows classes to be easily extended to incorporate new behaviour without modifying existing code
- Use with care!

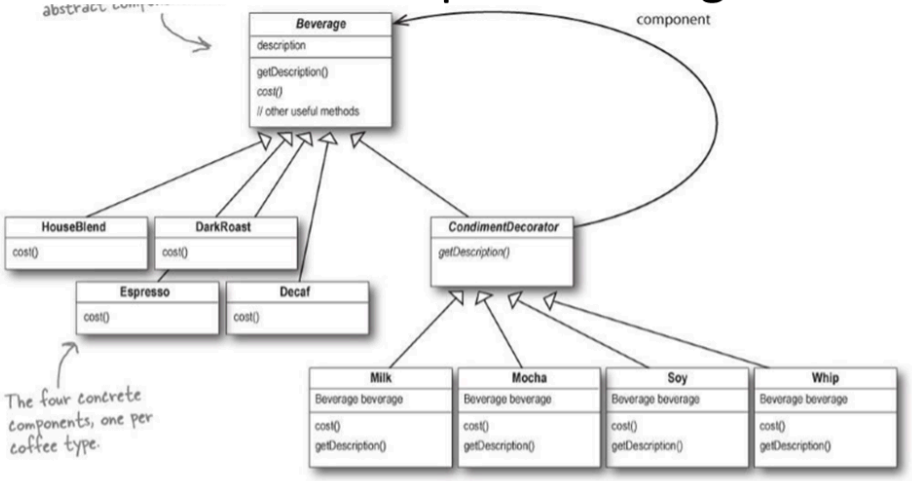
Those problems exist because you only rely on a single concrete class. This will force existing code to be edited when you want to add a new functionality. This violates **Open-Closed Principle** (more on that already discussed at **Module 3: OO Principles & Maintainability**). This principle requires you to make an interface so that extending a new functionality is as simple as adding a new class. Your existing code does not need to be modified. This is **our goal**.

The Definition



So here is the **Decorator design pattern** diagram. There are two interfaces here: **`Component`** and **`Decorator`**. Both are important, do not skip one of them. **`Component`** interface is to make a base object and decorated objects can be treated in the same way. Meanwhile, the **`Decorator`** interface abstracts the decorator classes so it can be used to decorate an object in the same way.

Decorator Pattern Example: Beverages



Back to the coffee drinks case study, here is the new class diagram. **Beverage** will only contain description, getDescription() method, and cost() abstract method. The **Beverage** abstract class is like the **Component** abstraction we mentioned in the previous page. Each of the coffee bases will be represented by a concrete class that extends **Beverage**.

Meanwhile, the **CondimentDecorator** abstract class is the **Decorator** abstraction. Each of the condiments will be represented by a concrete class that extends **CondimentDecorator**. All condiment decorator classes have an instance variable called **beverage**, which saves the reference of wrapped objects. By using this design, we only make concrete classes for the condiment, not the combination of condiments.

So, for example, we want to make a House blend with Soy + Milk, then we will instantiate the object like this: **new Soy(new Milk(new HouseBlend()))**. The order of decorators is not important.

Then, how about calculating the cost? We can do it recursively from the wrapping decorator, down to the deepest wrapped object (the base). We can implement **cost()** in a condiment decorator concrete class like this: (for example, Milk price is 5000 rupiahs not including tax).

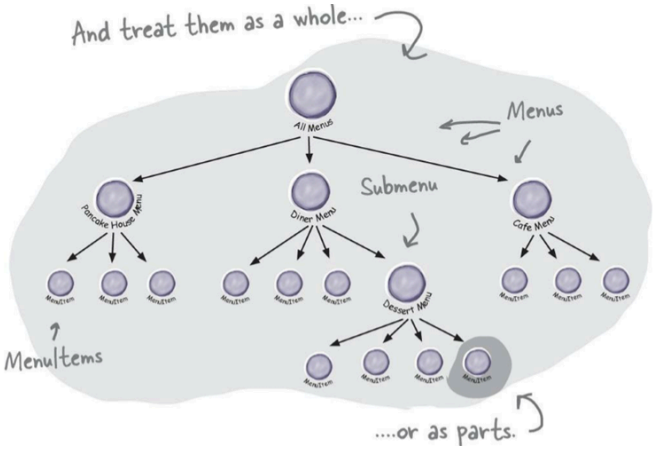
```
01    public double cost() {
02        return this.beverage.cost() + (5000 * TAX_RATE);
03    }
```

By using this design, we can easily add more base classes and condiment classes without modifying existing code **Open-Closed Principle**.

Composite Pattern

The Motivation

Example: Nested Menu



Imagine you want to extend your café's front-end site to have a lot of services. You want to make a menu bar that is easy enough to add submenus. You think about this tree structure. So, how to define this in an object-oriented realm?

Real World Example: DOM?

- What is DOM?
  - Document Object Model
  - An n-tree
- HTML document → DOM
  - Node?
  - Children?
- How do we manipulate HTML tags using JS?

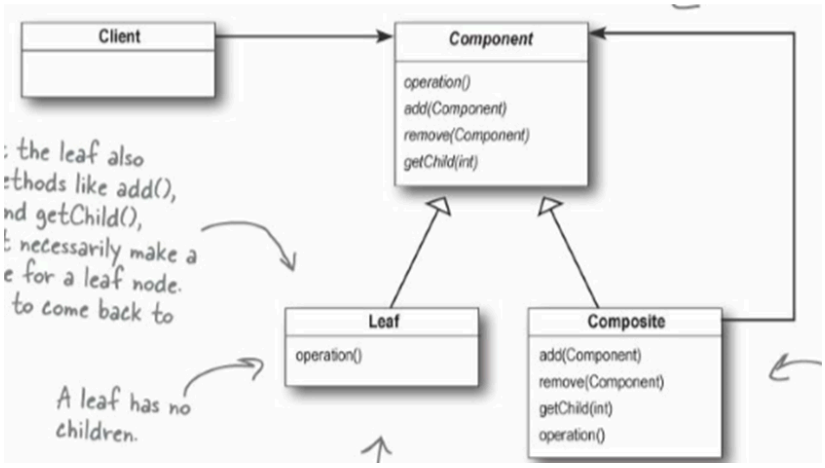
In a tree structure, we want to treat an item, be it a leaf (a node that has no children), or a node with children, without any discrimination. If a node has children, the action will be aggregated throughout the subtree. In real life example, this is what happens when you use JQuery or JavaScript DOM API to add new components or modify existing ones in an HTML page. The CSS styling also works this way, by inheriting the styles through all its children.

Composite Pattern

- Allows you to compose objects into **tree** structures to represent **part-whole** hierarchies
- Composite lets client treat **individual objects and compositions** of objects **uniformly**

This tree structure, in the Object-Oriented design pattern realm, is called **Composite pattern**. **Composite pattern** allows you to compose objects into tree structures and represent part-whole hierarchies. Part-whole hierarchy means that you can extract a subtree from a tree, and still can treat them uniformly.

Composite Pattern



This is the class diagram example of **Composite pattern**. There is an abstraction called **`Component`**, which makes uniform treatment across all elements possible. **`Client`** only needs to interact/depend on that abstraction instead of individual types like **`Leaf`** or **`Composite`**.

Meanwhile, there are two concrete classes: **`Leaf`** and **`Composite`**. The **`Leaf`** is the end of the tree because it has no children. Meanwhile, **`Composite`** contains a list of **`Component`**s as its children. The operations of a **`Composite`** will be done recursively by asking all its children to do the same action. By using this design, we can also easily make new elements, by calling **`add(component)`**. We can also remove an element, using the **`remove(component)`** method.

Bonus Material: Proxy Pattern

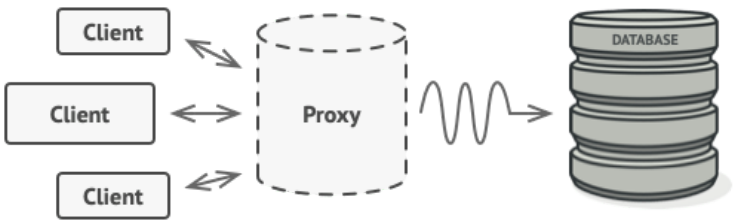
This material is not included in the slide. We put this here as a bonus material.

The Motivation

Let us imagine that you have very secret data that cannot be accessed directly. For example, a bank's gold and money storage. Of course, customers of that bank cannot store or take money directly from the money storage. If we allow that, someone can just take away all that money. So, we need to build a system where users can interact to store or take money, but without letting direct access to the storage. The system checks whether a user has an account on the bank, and stores/takes money using the defined rigorous procedure. This system is the **Proxy**.

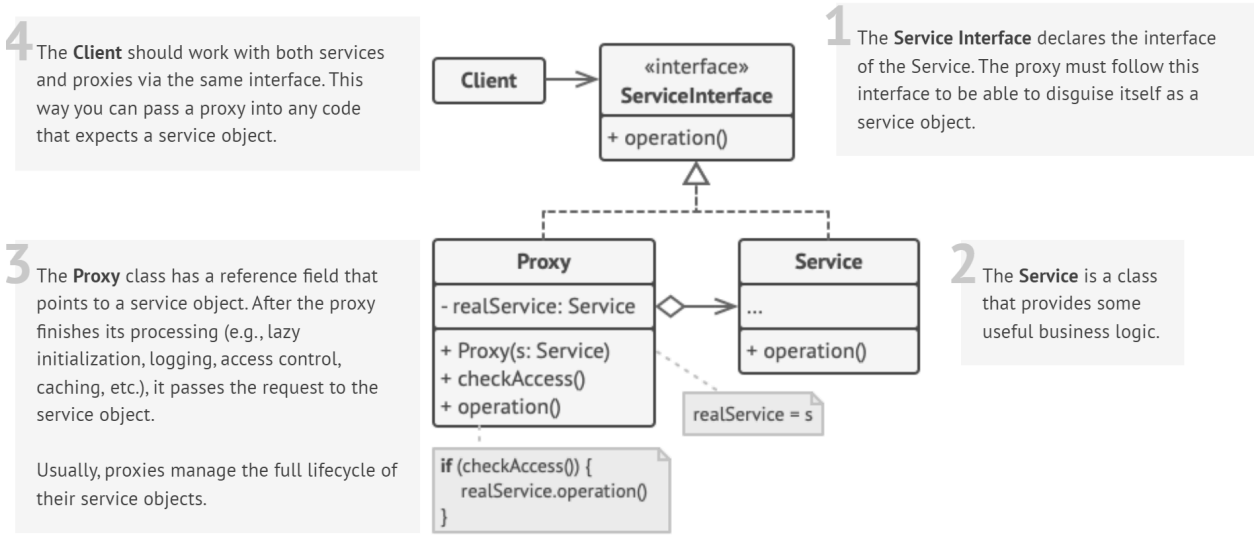
What is the difference between a **Mediator**, a **Façade**, and a **Proxy** then? Both seem to serve as "man in the middle", right? Yes, they are all middle-mans, but they have different purposes:

- A **façade** serves a few main (or commonly used) functionalities, so users can use those functionalities easily, without calling each element in the subsystem one by one.
  Is the client allowed to access elements in the subsystem directly though? Yes, they can. Especially when the case is not common enough to be summarised in a **Façade**.
- A **mediator** serves as an intermediary not only for a single resource, but for multiple resources that previously depend on each other. A mediator serves to break those dependencies, so that if a class wants to call (or interact with) another class, it should call a mediator first. This reduces coupling between classes.
  If there is another class that wants to access only a single element in that system, is it possible to access the element directly? Yes, they are still allowed to do so, but if the dependency becomes complex, change the dependencies to use the mediator.
- A **proxy**, meanwhile, serves as a "gate" for a resource. Clients are **not allowed** to access the resource directly, unlike the **Mediator** or the **Façade**.



A proxy is not always about security. It can also mean an improvement of performance, e.g. the lazy instantiation of a Database connection. We can also have multiple proxy types. We can also combine it with other patterns, such as the Decorator pattern, to make a single proxy object that has multiple capabilities.

The Definition



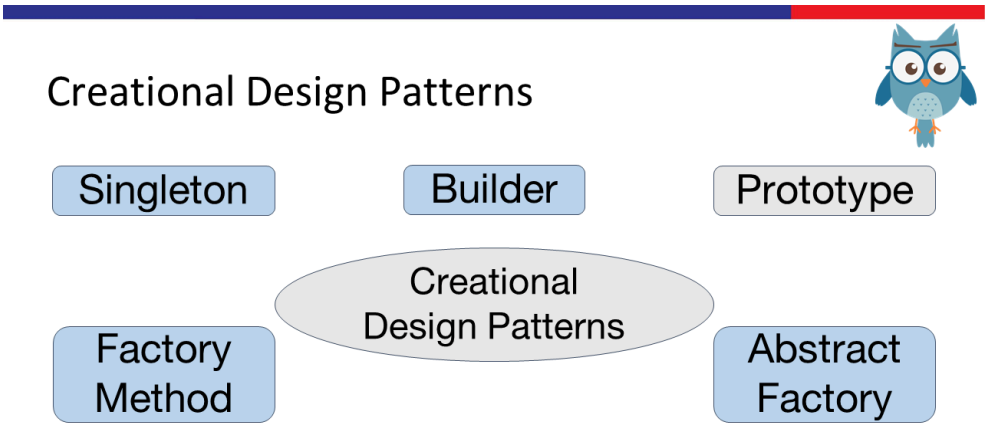
The proxy object and the real service object have the same interface as its abstraction. This is to make sure that a **`Proxy`** covers all its protected **`Service`**'s use cases. In every method that will call a **`Service`**'s use case, a **`Proxy`** can add new logic before or after the call to the original service. In the case explained by the diagram, it will check whether the **`Client`** has access to the service or not, thus the logic is placed before the service's **`operation()`** method execution. The **`Proxy`** object, in most cases, is responsible to instantiate the **`Service`**.

There are six common applicability of **Proxy** pattern: Lazy initialization (virtual proxy, can be combined with Singleton pattern), access control (protection proxy), local execution of a remote service (remote proxy, can be via HTTP, RMI, or RPC), logging proxy, caching proxy, and smart reference (auto destroy object if no more objects used it).

The advantages of Proxy pattern are that we can add checks and validations when accessing a class, without modifying either **`Client`** or the original **`Service`** code, thus complying with Open/Closed Principle. Lifecycle management and controls of the **`Service`** object can also happen within the **`Proxy`** object, without **`Client`**s knowing about it. We can also make a **`Proxy`** object work even when the original **`Service`** is not ready or available.

The disadvantages of Proxy pattern are that if overused, it will introduce a lot of classes and might introduce performance penalties. To avoid class explosion, we can use the **Decorator** pattern to introduce a **`Proxy`** object that consists of multiple types of Proxies. Since proxies and the original service have the same interface (the **`ServiceInterface`**), it serves as a **`Component`** interface. The original **`Service`** is the base class that can be wrapped by **`Proxy`** objects.

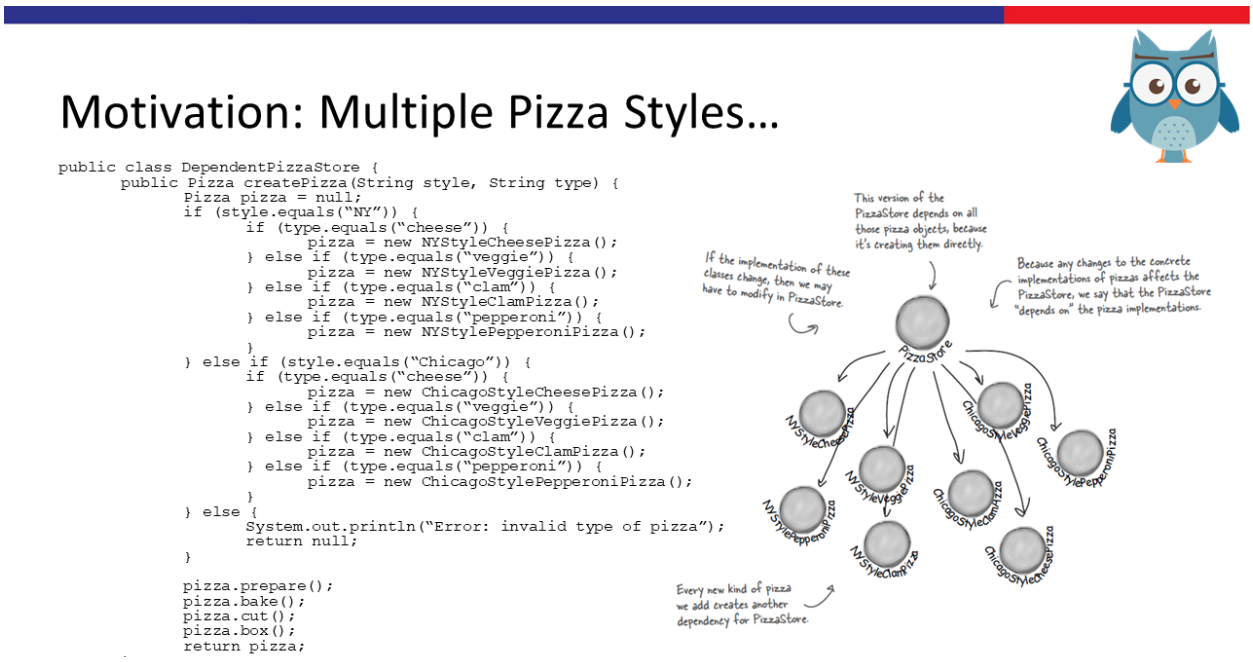
4. Creational Design Patterns



In this module, we will learn about Factory Method, Abstract Factory, Singleton, and Builder patterns.

Factory Method Pattern

The Motivation

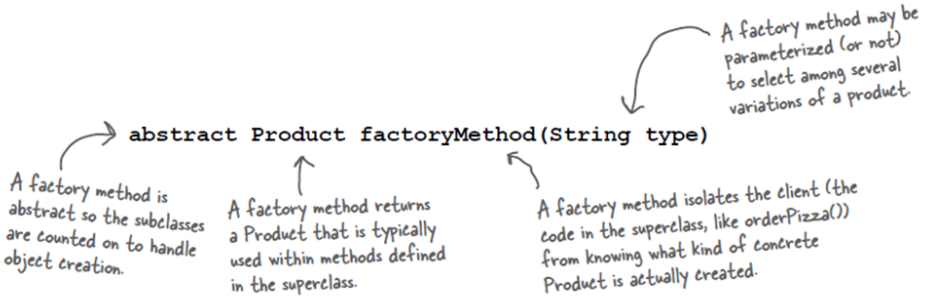


Imagine you have a **Pizza** franchise. Every region has its own taste preference; thus, the franchise must adapt its **Pizza** style for each region. Because our **PizzaStore** is still centralised, we need to modify the **orderPizza()** function each time a new **Pizza** style or type is created. The **orderPizza()** method will be too long. What procedure can we extract to a new method?

The Definition

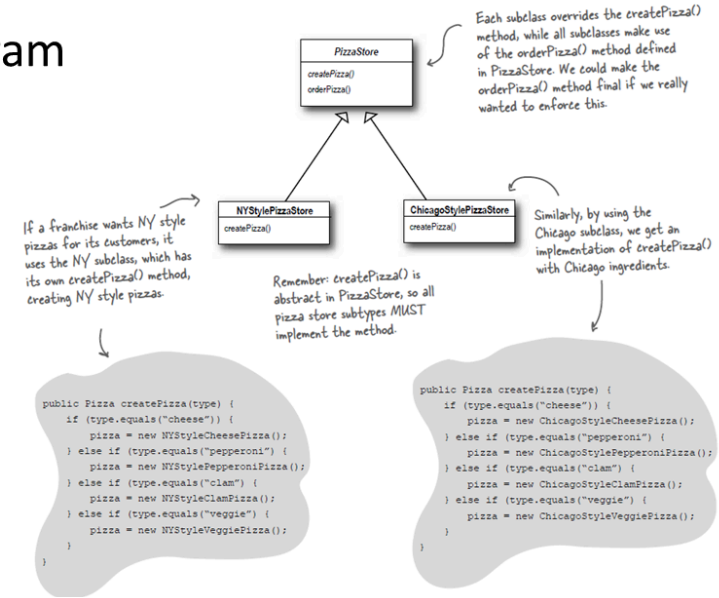
Factory Method Pattern

"Factory Method handles object creation and encapsulates it in a subclass. This decouple the client code in the superclass from the object creation code in the subclass" (Freeman, 2014).



If you guessed that the lengthy if-else branches to make a new **`Pizza`** object can be separated to a new method, you are right! But refactoring those conditionals involves more than a simple "extract method". We also need to abstract the method because each branch represents a different **`Pizza`** type.

The Diagram



Factory Method pattern makes the main class an abstract class. A subclass of that main class must implement the factory method. In the case of **`Pizza`** franchise, each region will have their own **`PizzaStore`** (for example **`NYPizzaStore`**).

Factory Method Pattern

```
public abstract class PizzaStore {
     public Pizza orderPizza(String type) {
          Pizza pizza;

          pizza = createPizza(type);

          pizza.prepare();
          pizza.bake();
          pizza.cut();
          pizza.box();

          return pizza;
     }

     abstract Pizza createPizza(String type);
}
```

Usually, we create a new object using its constructor directly, but this time **we separate the object creation process to a different method**.

The object creation process **must be implemented** by each franchise.

How about languages that isn't based on Object-Oriented like Rust?
Rust **does not** support abstract classes. Use **trait** to achieve same feat.

The factory method then will be used in other methods that need a new object instance. In this case, previously, the **`orderPizza(type)`** method was tasked to create **`Pizza`**s. Now, the method only needs to call **`createPizza(type)` factory method** to get a new **`Pizza`** object. Then, what happens if the programming language does not support abstract classes? Use interfaces or similar features. In Rust programming language, we can use **`trait`**s.

Factory Method Pattern in Rust – use Trait

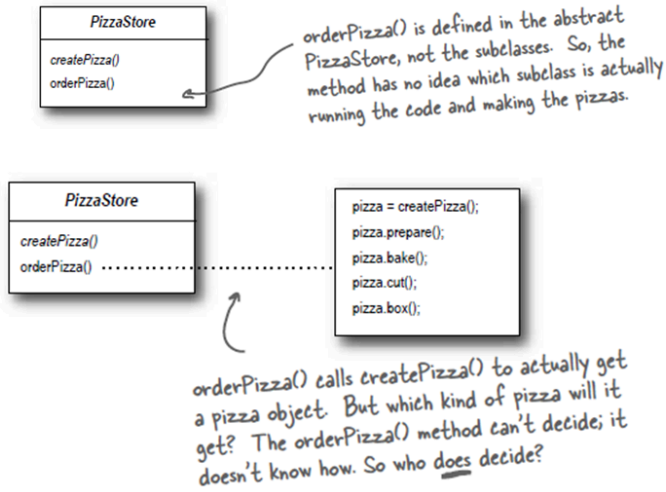
```
pub trait Button {
     fn render(&self);
     fn on_click(&self);
}

pub trait Dialog {
     fn create_button(&self) -> Box<dyn Button>;

     fn render(&self) {
          let button = self.create_button();
          button.render();
     }
}
```

The object creation process **must be implemented** by each franchise.

We separate the object creation process to a different method.

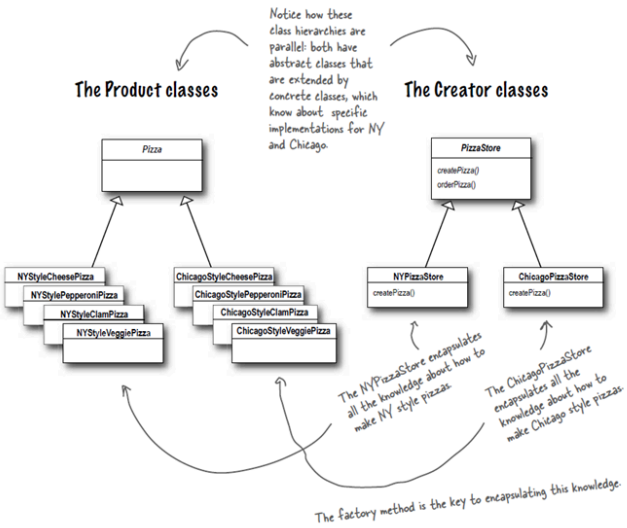
`Trait`s in Rust are a bit like an abstract class. A **`trait`** can contain a concrete method. To make a **factory method**, make a method with no implementation that returns a desired type of object. The return type is **`Box<dyn Button>`**, which means that it will be put to heap memory (because of **`Box<>`** struct) and the object implements Button trait (because of **`dyn Button`**).

Goal: Decoupling of Object Creation Process



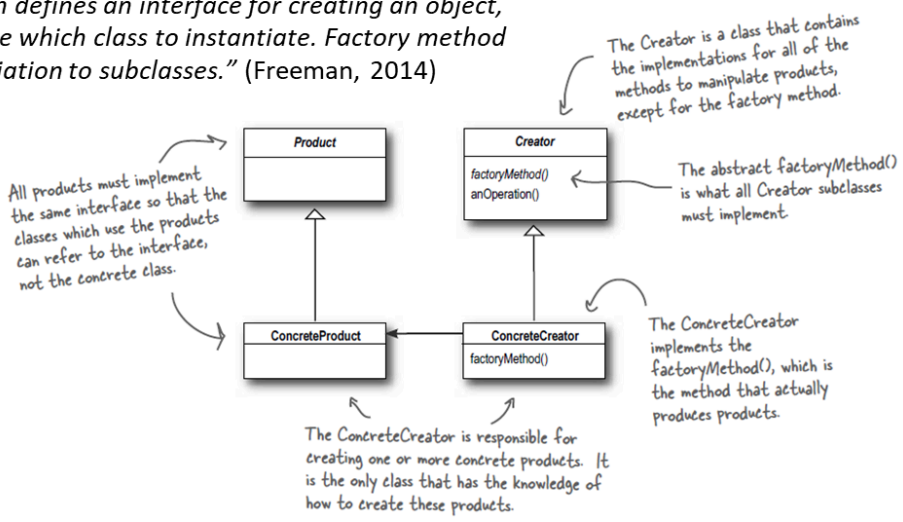
By separating object creation logic to a separate method, we can make other methods decoupled from the object creation logic. Remember **Single Responsibility Principle**, a method or a class should only have one responsibility. **`orderPizza()`** should not make the **`Pizza`** by itself. Factory method should be the only element that knows how to create the object.

Goal: Decoupling of Object Creation Process



Goal: Decoupling of Object Creation Process

"Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses." (Freeman, 2014)



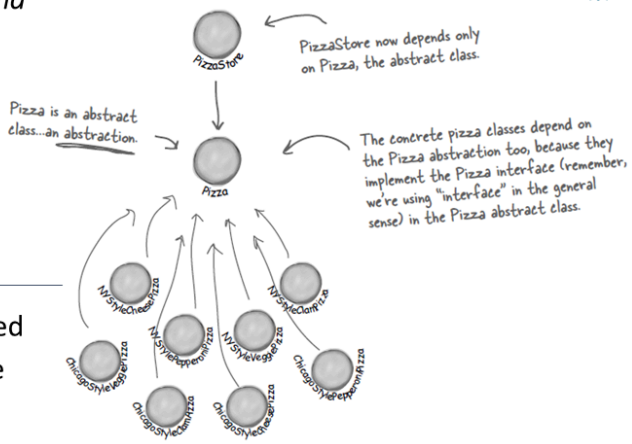
Factory Method, by definition, defines an interface for creating an object, then lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses. This will make adding more Pizza styles easier, because we only need to implement each type of Pizza based on that style and a subclass of **`PizzaStore`** that contains a **factory method** for the **`Pizza`** style. This design complies with the **Open-Closed** design principle.

Dependency Inversion Principle

"Depend upon abstractions. Do not depend upon concrete classes." (Freeman, 2014)

High-level components **do not** have dependencies to low-level components, they should have dependencies to abstractions instead.

After applying Factory Method, we applied that principle so our class diagram will be like this:



This design also complies with the **Dependency Inversion** design principle. This is because the **`PizzaStore`** now only depends on **`Pizza`**, signified by the return type of **`orderPizza(type)`** and the abstract method of **`createPizza(type)`**. The dependency to concrete classes is now handled by the subclasses of **`PizzaStore`** that implements **`createPizza(type)`**.

Guideline to follow Dependency Inversion Principle

1. No variables that has reference to concrete class.
2. No classes that derived from a concrete class.
3. No methods that override methods that are already implemented in the base class.

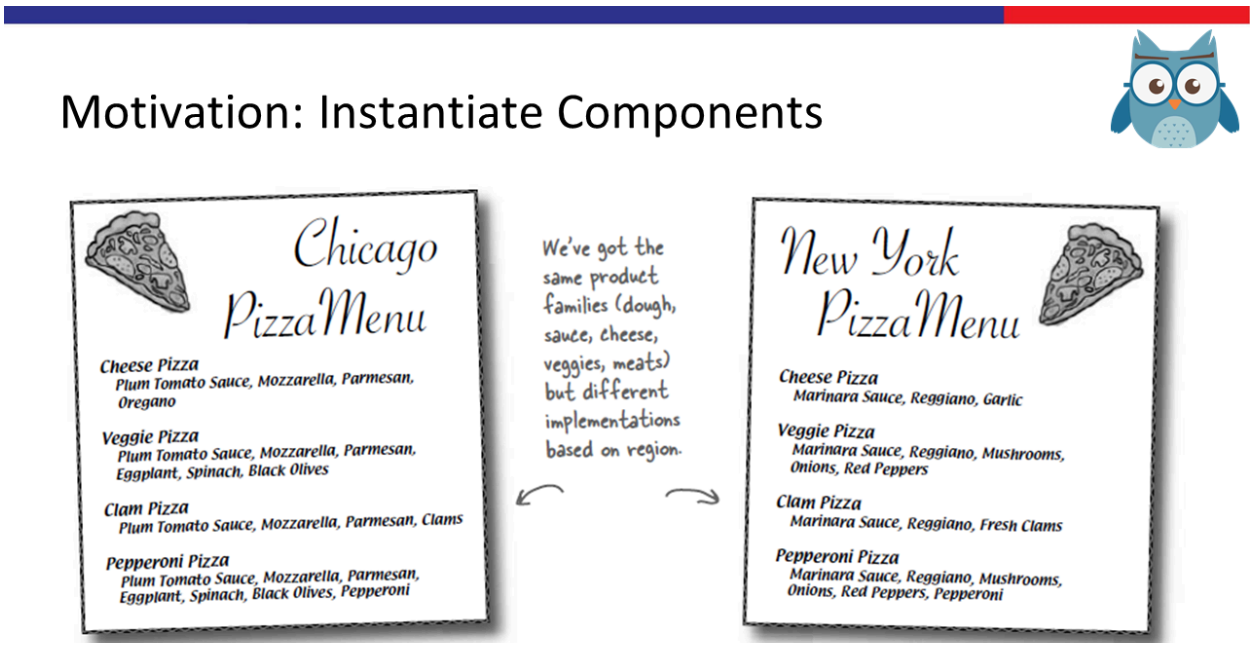
Unfortunately, the guideline is **impossible** to follow.
For example, we always instantiate String. String is a concrete class.
That's fine because String implementation won't change in foreseeable future.

There is a guideline to follow the **Dependency Inversion** principle ideally. You are not allowed to make variables that reference concrete classes. You are also not allowed to derive a class from a concrete class (abstract class is fine though). You are also not allowed to override methods that are already implemented in the parent class.

Unfortunately, the guideline is **impossible** to follow. Sometimes, you need to refer to a concrete class, like **`String`**. That is fine because we can guarantee that **`String`** implementation will not change in the foreseeable future.

Abstract Factory Pattern

The Motivation (and Refactoring Process)

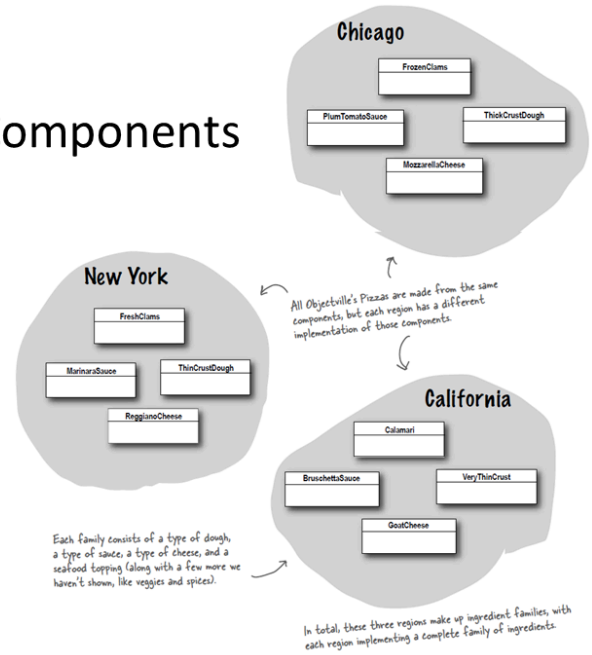


In the **Factory Method** pattern example, we still have an unsolved problem: There is too much class to just define a single **`Pizza`** of a certain type and style, without any details of the ingredients. We have only abstracted the **`PizzaStore`** for each regional style. Here is the example of **`ChicagoPizzaStore`**:

```
01 public class ChicagoPizzaStore {
02     public Pizza createPizza(type) {
03         Pizza pizza;
04         if (type.equals("cheese")) {
05             pizza = new ChicagoCheesePizza();
06         } else if (type.equals("pepperoni")) {
07             pizza = new ChicagoPepperoniPizza();
08         } else if (type.equals("clam")) {
09             pizza = new ChicagoClamPizza();
10         } else if (type.equals("veggie")) {
11             pizza = new ChicagoVeggiePizza();
12         }
13         return pizza;
14     }
15 }
```

Then what happens if we want to make a new **`Pizza`** type, for example **"Fish Pizza"**? We still need to change each **`PizzaStore`** subclass. The refactoring goal is that we can make an **ingredient factory** so the **`PizzaStore`** and its subclasses can use this **ingredient factory** to make components that make a **`Pizza`** of such type.

Motivation: Instantiate Components



In the Factory Method example, we have known that a Pizza franchise should adapt to each region's taste preferences. This will make the ingredients and toppings of **`Pizza`**s different across regions. In the previous example though, we are treating a variant (type, style) of **`Pizza`** as a single object. Now, we need to treat the **`Pizza`** object as a product of composed ingredients.

Abstract Factory Pattern

In Factory Method, the object creation process is in the method.
In **Abstract Factory**, we abstract each component creation process to a **Factory interface** that contains object creation methods.
Each "style" will implement its concrete version of the Factory interface.

```
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
}
```

Abstract Factory pattern introduced a **`Factory`** interface, which is just **a bunch of factory methods**. In this case, **`PizzaIngredientFactory`** requires factory methods of each ingredient (there are six ingredients) that will be implemented differently across regions.

Abstract Factory Pattern

```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza doesn't care which factory is used, as long as it is an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

In the **Factory Method** pattern, we used to call the factory method within the class. When we use **Abstract Factory**, our code will depend on the ingredient factory. To make an ingredient or component, our code will call a method of that factory. Our code does not care which ingredient factories are being used in the code. So, we can change which ingredient factory that will be used at runtime.

Abstract Factory Pattern

Here is the example of main class that will use the components made by the Factory:

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;
    abstract void prepare();

    // implementation methods...
}
```

This is an example of the abstraction of the main product (**`Pizza`**). The main product consists of components that will be composed into. The main product also has a method to compose the component objects, in this case it is the **`prepare()`** abstract method.

How to implement this in languages that do not support abstract classes like Rust? A **`trait`**, even though it may have a concrete method, but it cannot have instance variables. If that is the case, then we can make the variables in the **`struct`**s that implement that **`trait`**. The **`prepare()`** method implemented by the **`struct`** is still able to compose the components anyway.

Abstract Factory Pattern

Here is the example of main class implementation that will use a concrete Factory to create each of its components:

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

This is the implementation example of the main product. In this case, **CheesePizza** is a type of **Pizza**. So, it will implement the constructor method and the **prepare()** method. The **prepare()** method here will compose three ingredients (**dough**, **sauce**, **cheese**) by calling each ingredient's factory method in the **ingredientFactory**. The constructor method of **CheesePizza** asks for a **PizzaIngredientFactory** object. This means, we can instantiate a **CheesePizza** with Chicago style by putting **ChicagoPizzaIngredientFactory** object when creating the **CheesePizza** object.

Combining Factory Method & Abstract Factory

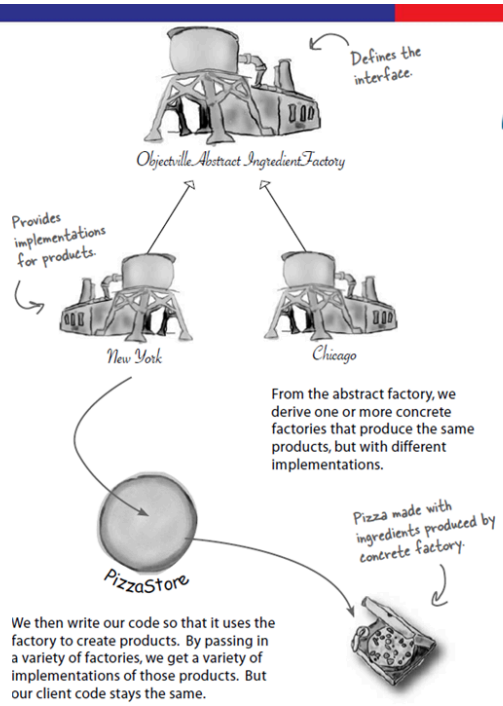
We can combine Abstract Factory Pattern with Factory Method Pattern:
Factory Method will instantiate the main class and the Factory of components, while Abstract Factory will instantiate the components of the main class.

```
public class NYPizzaStore extends PizzaStore {
     protected Pizza createPizza(String item) {
          Pizza pizza = null;
          PizzaIngredientFactory ingredientFactory = new NYPizzaIngredientFactory();
          if (item.equals("cheese")) {
               pizza = new CheesePizza(ingredientFactory);
               pizza.setName("New York Style Cheese Pizza");
          } else if (item.equals("veggie")) {
               pizza = new VeggiePizza(ingredientFactory);
               pizza.setName("New York Style Veggie Pizza");
          } // ...
          return pizza;
     }
}
```

We can also combine the **Abstract Factory** pattern with the **Factory Method** pattern. The Abstract Factory pattern will help you create the ingredients, while the Factory Method pattern will help you create the main product class (**`Pizza`**) that composed the ingredients. In this case, we will modify the createPizza(type) method that we have implemented previously, to incorporate the new PizzaIngredientFactory.

The conditionals are still there, but this time, there is no more **`ChicagoCheesePizza`** and **`NewYorkCheesePizza`** class, only **`CheesePizza`** class. By using this pattern, we finally can define a single **`Pizza`** with great details of the components, just like the menus list that we served to our customers. We may also use the same component for some pizza ingredient, the example is when both **`Chicago`** and **`NewYork`** agree to have **`Lettuce`** as their **`Veggie`**.

Abstract Factory Pattern: The Illustration

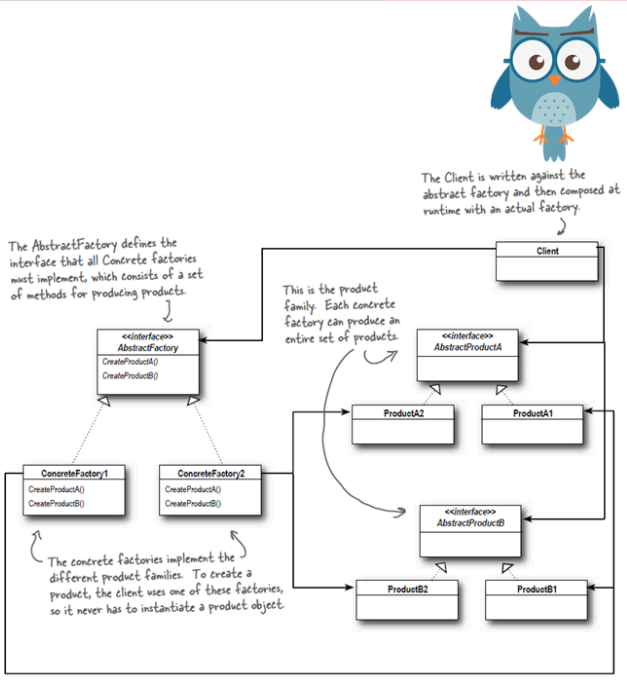


Here we give you the illustration that might help you summarise what we have covered in this subchapter. You can zoom in if the illustration's font size feels too small for you.

To make a new branch (for example, **Padang**), we can implement a new ingredient factory. The ingredient factory provides implementations for products. The implementation can be different for each concrete ingredient factory. We then write our code so that it uses the ingredient factory to create products. Our code will be the same regardless of the variation.

Abstract Factory Pattern: The Definition

"Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes." (Freeman, 2014)

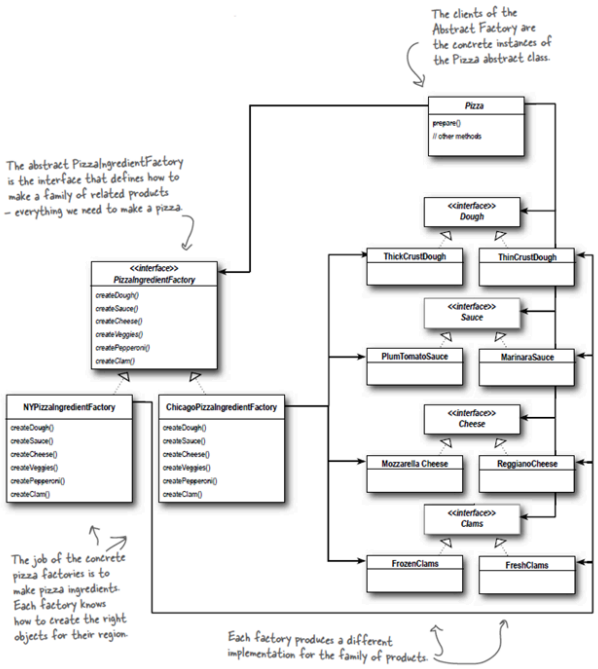


The definition of **Abstract Factory** pattern is to provide an interface for creating families of related or dependent objects (these objects are what we called as "ingredients" previously) without specifying their concrete classes.

Here is the diagram for the **`Pizza`** case:

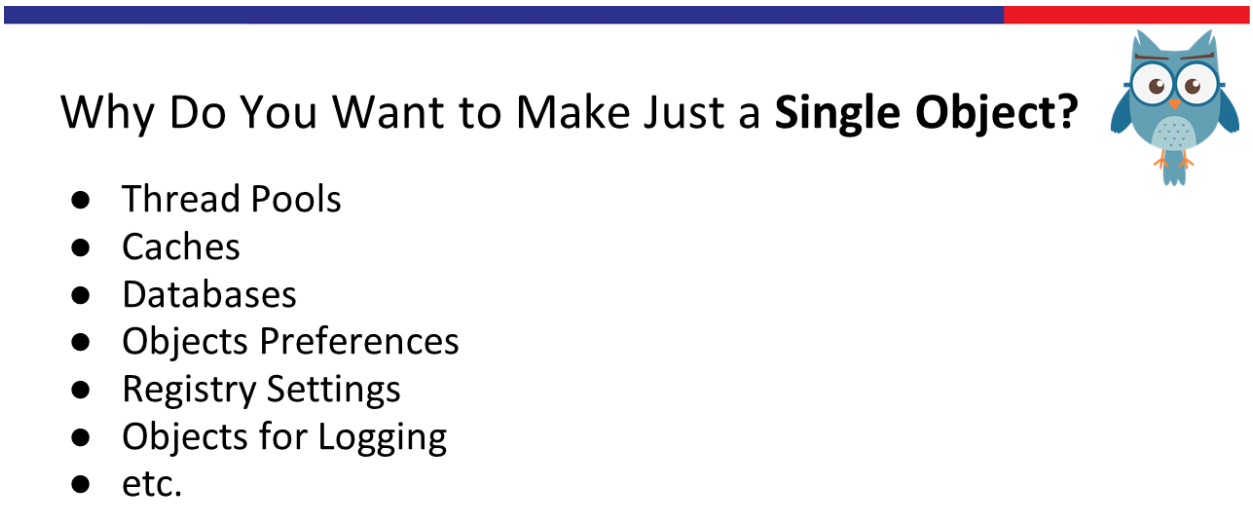
Abstract Factory Pattern: The Definition

"Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes." (Freeman, 2014)



Singleton Pattern

The Motivation



What will you do if you need a place in your code that can save data persistently throughout the lifetime of your app? What will you do if you need a place in your code that can share the same data through all context, all the time? To achieve those goals, we need a class that will only generate a single object. This is because we want to use that single object for all kinds of context. If we use multiple objects, the state of data will be different for each object.

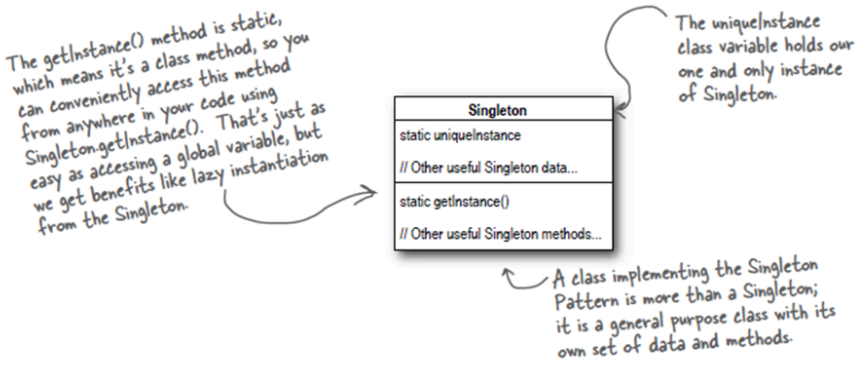
Another case is if you want to use an object with a huge performance penalty, for example initialising a database from other computers in a network. To avoid the performance penalty, you need to instantiate it as rarely as possible. If possible, it is better for you to use a single object that is shared to all functions that need that object.

This is what the **Singleton pattern** comes in handy!

The Definition

Singleton Pattern Diagram

"The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it." (Freeman, 2014).



The Singleton pattern is quite simple. It does not involve dependency or relationship to any other class, unlike other design patterns. This design pattern only involves one **`Singleton`** class that has a **`static`** variable to save the **`Singleton`** object. The difference between this kind of class and all other kinds of classes, is the constructor of **`Singleton`**s are private. This prevents the constructor to be used by objects outside of the class, thus preventing another object from being instantiated.

To get or create an object, users of this class must use the **`getInstance()`** static method, meaning it can be executed like this: **`Singleton.getInstance()`**. The **`getInstance()`** will return the **`Singleton`** object that is already inside the **`static`** variable. If the **`static`** variable is still empty, **`getInstance()`** will create a new object using the private constructor.

Tackling Concurrency Issues: Lazy vs Eager Singleton

Notice: This subchapter will use Java programming language for some of the code examples. The example for Singleton in Rust is completely different, and in Rust, the distinction between Lazy vs Eager is not too relevant.

There are two types of Singleton pattern: **Eager** and **Lazy**. **The eager version of singleton will instantiate the object right at the beginning**. To make an Eager singleton, you just need to define the object directly in the static variable definition, for example: **`private static Singleton instance = new Singleton()`**. The advantage of Eager singleton is the object creation process does not have any concurrency issue, as the object will be created at the start of the program, just like any other static variables. The disadvantage is the program will consume more memory for objects that might not be used immediately.

The lazy version of singleton will only instantiate the object when an object is requested for the first time. The **`getInstance()`** method will involve conditionals to check whether an object has ever been created or not. If there is no object instantiated in the static variable yet, then the object will be instantiated and saved to the static variable. If there is already an object, return that object. Here is the code example:

```
01 public class Singleton {
02     private static instance;
03     private Singleton { // ... implement constructor }
04     public static Singleton getInstance() {
05         if (instance == null) {
06             instance = new Singleton();
07         }
08         return instance;
09     }
10 }
```

The lazy version of singleton will have less memory usage at the beginning of the program because the object will be created on-demand. The major disadvantage is the **`getInstance()`** method is not thread safe. We have learned in **Module 6: Concurrency** that Race Condition may happen in this lazy singleton scenario. This is because two threads (or more) may execute line 05 (the **`if`** statement) simultaneously, and both will return **`true`** because at that time, no one made a Singleton object yet. Then both will return different Singleton objects because both tried to create new Singleton objects because of the Race Condition.

To handle the concurrency issue, we need to synchronise the execution of **`getInstance()`** method so that it can only be executed by one thread at one time. The mechanism is commonly called the **Double-Checked Locking Mechanism**. There are two steps to create that: add the **`volatile`** keyword to the static variable that will contain the Singleton object, then implement a synchronisation mechanism in the **`getInstance()`** method.

Why do we use the **`volatile`** keyword? **`volatile`** keyword guarantees that the change of value in a variable will be instantly reflected to all the other threads that also use the value. Unfortunately, this keyword does not guarantee atomicity, so a variable with **`volatile`** keyword will still be prone to lost updates if the operations done by threads are not atomic. To ensure atomicity, implementing synchronisation to the method or operation is necessary. You can learn more about the **`volatile`** keyword here: https://ioflood.com/blog/java-volatile/ and https://www.baeldung.com/java-volatile.

So, here is the code example of **Double-Checked Locking Mechanism** in Java. Changes from the previous code are printed **bold**.

```
01 public class Singleton {
02     private volatile static Singleton instance;
03     private Singleton() { // ... implement constructor }
04     public static Singleton getInstance() {
05         if (instance == null) {
06             synchronized (Singleton.class) {
07                 if (instance == null) {
08                     instance = new Singleton();
09                 }
10             }
11         }
12         return instance;
13     }
14 }
```

Why do we put **`synchronized` blocks** inside the conditional branch? Why not use **`synchronized`** directly without nesting the same conditional twice? This is intentional because if we directly use **`synchronized`**, we will introduce a significant performance penalty every time **`getInstance()`** is executed. **`synchronized`** block will force us to wait for other threads done using that code block before our current thread continues. By keeping the outer conditional, we can skip the **`synchronized`** block altogether if the object is already created.

Then, why do we redundantly check the same conditional again inside the **`synchronized`** block? This is to avoid previous conditional mistakes due to Race Condition affecting the object creation result. When the outer conditional is executed, it might be **`true`**, but after doing synchronisation, the code knows that the real condition is **`false`**, so no object will be created.

This **double-checked locking mechanism** is mandatory to make a lazy Singleton for most programming languages. In Rust, the implementation is different.

Singleton Pattern in Rust

Rust **does not** support static variables inside a struct. You need to make a static variable **outside of a struct** or instantiate to a variable with static (or the longest) lifetime.

Alternatives to Singleton in Rust:

- Instantiate the object in main() function, or
- Instantiate the object using static variable.
- If the object needs to be mutable (such as HashMap): use **`lazy_static`** macro and **`std::sync::Mutex`** (Rust enforces thread-safe operations so Mutex is a must. Look back at Module 6: Concurrency)

See https://refactoring.guru/design-patterns/singleton/rust/example#example-1

Refactoring.Guru has the example of Singleton code for Rust in https://refactoring.guru/design-patterns/singleton/rust/example.

The “eager version” is not really a Singleton like we know in Java and most other programming languages, because it just instantiates the object to a “regular” variable, or to a static variable. No need for **`getInstance()`** method because we can access the variable directly.

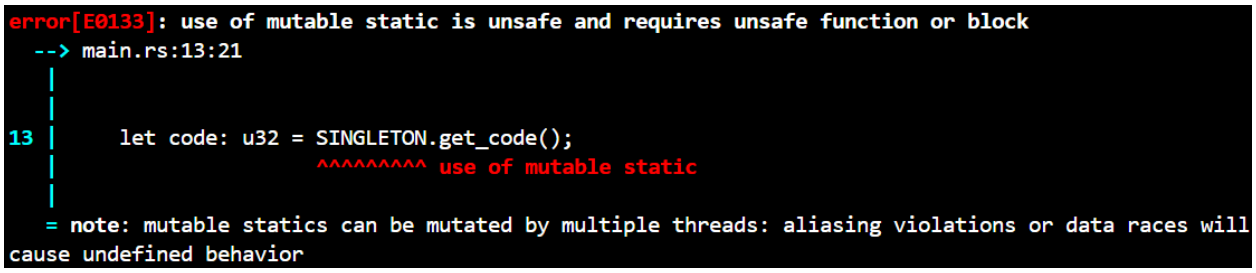
Unfortunately, the “eager version” is not practical for most of the cases, especially the first option (instantiate a variable in **`main()`** function). Although the lifetime is as long as '**`static`** (the variable will be alive until the program ends), the first option restricts the usage only in the **`main()`** function or passed as parameters to any other function executed by **`main`**(). If we used a web framework for example, this is not practical.

The ”lazy version” (the third option) is still different from other programming languages, as it involves an external library (**`lazy_static`**) with a “lazy instantiation” mechanism and **`std::sync::Mutex`** to synchronise the object’s operation. This is because in Rust, **`static`** variables can only be filled by constant functions, tuple structs, and tuple variants. We can still set a new object from an empty struct or a struct with primitive types (str, bool, etc.), but it cannot work if the struct contains non-primitive types. Here is the error shown by the compiler:

```
error[E0015]: cannot call non-const fn `<str as ToString>::to_string` in statics
 --> main.rs:1:68
  |
1 | static SINGLETON: SingletonClass = SingletonClass {name: "Patrick".to_string()};
  |                                                                    ^^^^^^^^^^^^
  |
  = note: calls in statics are limited to constant functions, tuple structs and tuple variants
```

Rust also forbids us to use mutable static variables by default. This code will return error:

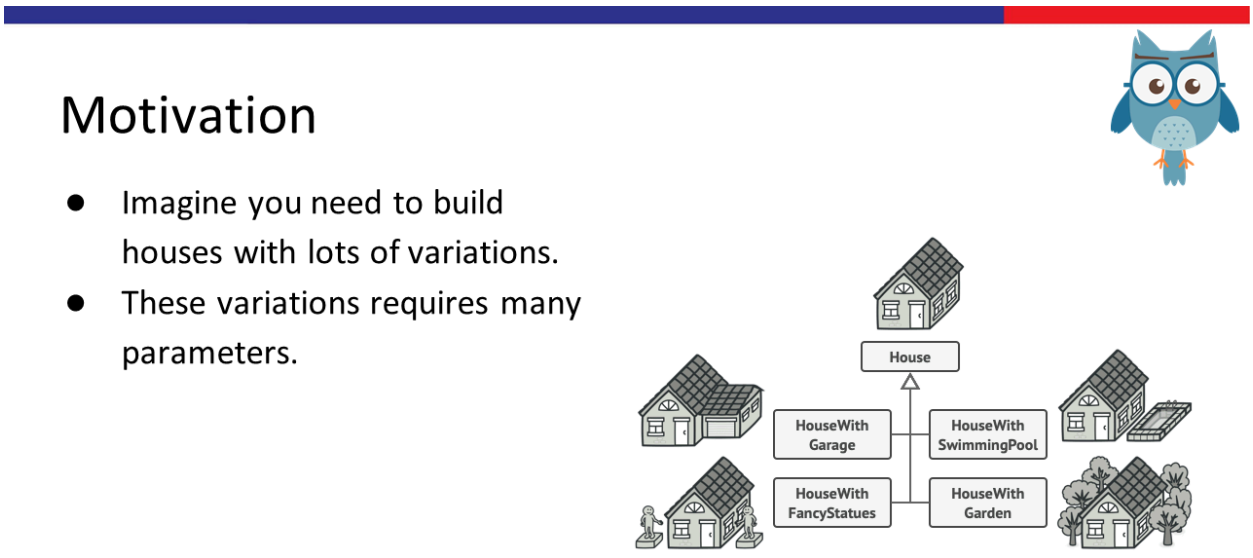
```
01 static mut SINGLETON: SingletonClass = SingletonClass {code: 32};
02 struct SingletonClass {
03     pub code: u32
04 }
05 impl SingletonClass {
06     fn get_code(&self) -> u32 {
07         return self.code;
08     }
09 }
10 fn main() {
11     let code: u32 = SINGLETON.get_code();
12     println!("{}", code)
13 }
```



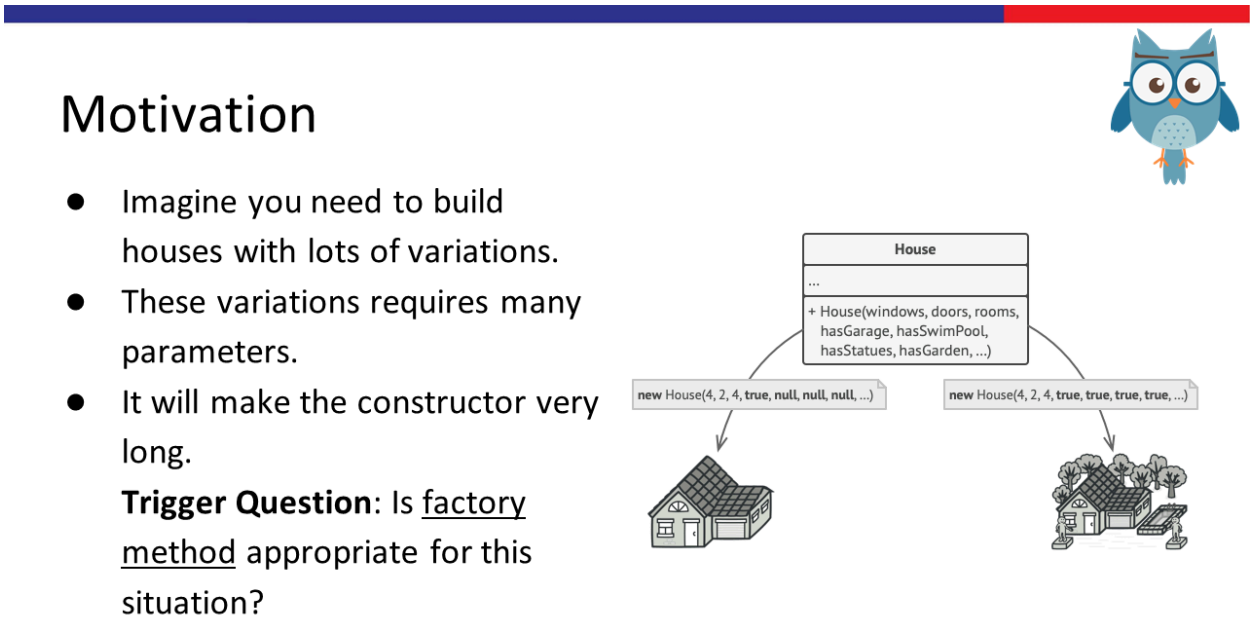
The only option for mutable structs or data structures is to use synchronisation mechanism via **std::sync::Mutex** or **std::sync::RwLock**. Using this synchronisation mechanism directly will automatically raise **E0015** error once again, because they are not constant functions. So, we will make a constant function via **lazy_static** macro. Refactoring.Guru also provides the code example for this case. This module's tutorial will also make use of that "lazy version".

Builder Pattern

The Motivation

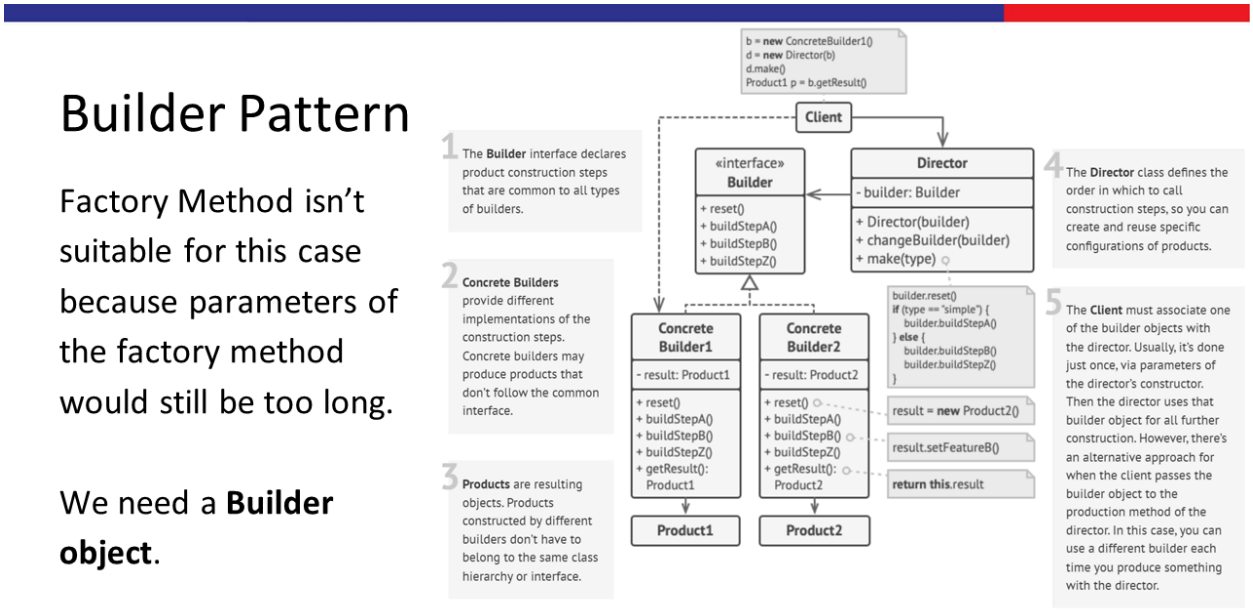


Imagine you want to build houses with lots of different variations. These variations require many parameters. For example, you want to build a house with two statues, a garage, a garden, and twelve doors.



To build complex variations of houses, we currently need to have exceedingly long constructor parameters. This sounds déjà vu with the case of Decorator pattern and Factory Method pattern, right? Can this be solved with Factory Method pattern or with Decorator pattern?

The Definition



Note: Director class is optional. You can use the Builder right away.

Before we continue to the definition of Builder pattern, the answer from the question before is **NO**. This is because if we use Factory Method, we just move the parameter problem from the constructor method to the factory method.

If we use Abstract Factory instead, this case will create a "class explosion" problem as there will be too many variations. Abstract Factory is also less flexible for this case because a factory can only make predetermined objects. In real life factories, engineers set predetermined templates and steps to make objects. The process cannot be easily changed on the go.

If we use the Decorator pattern, this case will create an "object proliferation" problem, and the wrapping mechanism itself will be too tedious. Imagine wrapping **House** with **Door** 12 times, then with **Statue** 2 times, etc. The recursion will be too deep.

Introducing... **Builder pattern**! Builder pattern gives the flexibility aspect that Factory does not have, by letting the **`Client`** or **`Director`** execute object building methods at their will. This can be achieved by making a **`Builder`** interface that contains methods that build parts of the object, and a **`getResult()`** method to get the result. Methods to build the parts (for example: **`buildDoors()`**) will modify the object to instantiate those parts. Methods to build the parts will return the Builder object itself, so they can be chained when building an object. The Builder interface must be implemented by concrete Builder classes because a class might have distinctive styles/subclasses (this reminds us of **`IngredientFactory`** from Abstract Factory pattern).

Builder Pattern Example

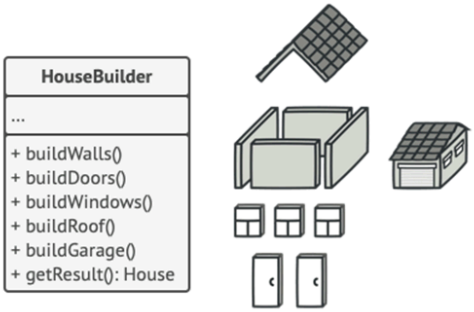
To make a new House with walls, doors, windows, and roofs...

From this:

```
House house = new House(true, true,
                  true, true, null)
```

To this:

```
House house = new HouseBuilder()
                    .buildWalls()
                    .buildDoors()
                    .buildWindows()
                    .buildRoof()
                    .getResult()
```



Advantage: Better code **readability**. No need to look at documentations too often. :)

Because methods inside a **`Builder`** will return the Builder object itself (like **`return this;`**), the building process can be chained together into a single statement. The example is given in the slide, we can build a house by running those chains of method calls. The advantage of Builder pattern is that we know what we build, by calling builder methods with meaningful names. This is a significant improvement in code readability compared to executing a constructor method with parameters full of magic values.

You might want to ask, "We have learned that **Message Chains** is a code smell (look **Chapter 4: TDD and Refactoring** for this), and these chains somehow resemble Message Chains..."
The answer is: **`Builder`** chain is not a Message Chain code smell! Message Chain code smell is when an object gets another object, then gets another object, then gets another object, before it does something. For example: `Employee -> Manager -> Director -> getDirectorId()`. These builder chains are not getting another object as the intermediary. These builder chains are chaining the **`Builder`** object itself, so it does not cause dependency or coupling issues, unlike the Message Chain code smell.

There are real-life cases in programming that uses Builder pattern:

- Database Query Builder in a lot of Object Relational Mapping (ORM) libraries. You can add selections, orders, limits, groups, and other SQL parameters in an Object-Oriented way.
- **`lombok`**'s **`@Builder`** annotation (Java) that converts model classes that have a lot of instance variables to be able to return a Builder object using the **`.build()`** static method, so that it can be built using those "beautiful" chains.

5. Bonus Material: Compound Patterns

We can compose multiple design patterns into a new design pattern. Remember that a **compound pattern is a proper design pattern**. Remember again that a design pattern is a pair of a **common** problem and the solution. It is not as simple as just combining design patterns to an overall code design.

The materials about **compound patterns** are not included in the slide. We put this here as bonus materials. Although this is a bonus material, **integrating design patterns is one of the most useful skills in the industry**. We will learn about compound patterns used in common software developments: Model-View-Controller (used in web programming), Model-View-Presenter (used in desktop app development, Windows), and Model-View-ViewModel (used in mobile app development, Android).

Compound Pattern is More than Just Patterns that Work Together

For this submodule, we will give you an example of the Duck information system from the Head First Design Pattern book, chapter 12. By learning this case study, we hope that you learned how to integrate multiple design patterns into a project.

The Problem

We have **`Duck`** and **`Goose`** classes. Ducks quack, meanwhile, Geese honk. There are four types of Duck: **`DuckCall`** (the quack is “kwek”), **`MallardDuck`** (the quack is “quack”), **`RedheadDuck`** (the quack is “quack”), and **`RubberDuck`** (the quack is “squeak”). The preexisting code already contains those classes and a **`DuckSimulator`** class (in Java, equivalent to main function in Rust) that invokes those quacks and honks.

The problem is that the Duck simulator wants to expand its functionality, not limited to just “honking” and “quacking”. Here is the list of problems we need to tackle:

1. **`Goose`**’s honking is executed differently from **`Duck`**’s quacking, meanwhile we want to execute all of them using the **`quack()`** method to simplify the **`DuckSimulator`**. As Geese is one family in taxonomy as Ducks, then we need to **add an implementation for `Goose` to be compatible with `Duck` variants**.
2. The manager wants **`DuckSimulator`** to handle multiple ducks at once. So, we need to **add implementation to be able to manage `Flock`s of `Duck`s as a whole**.
3. The manager wants **`DuckSimulator`** to be able to count some (or all) Duck quacks. The keyword is: “some”. So, we need to **add implementation to be able to count the total number of quacks made by a `Flock` of `Duck`s**. The limitation is that you are **not allowed to change the `Duck` variant classes**.

4. **`DuckSimulator`** should not build Duck objects on its own. To achieve this, we need to **encapsulate `Duck` variant instantiations**.
5. The manager introduces a new role into the system, called **`Quackologist`**. A **`Quackologist`** observes the behaviour of some (or all) **`Duck`**s. So, we need to **add implementation to be able to observe the behaviour of a `Duck`**.

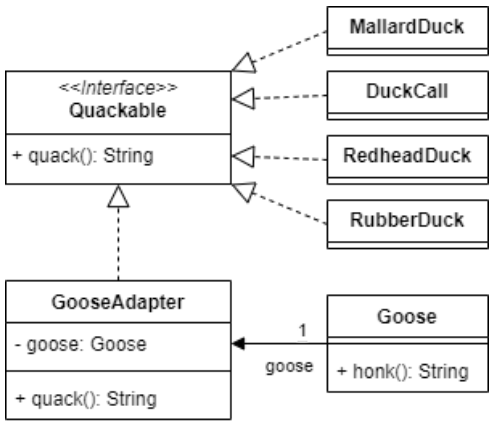
You can look at the repository for the initial Ducks and Goose code here:

- Java: https://gitlab.com/ichlaffterlalu/patterns-work-together-java/
- Rust: https://gitlab.com/ichlaffterlalu/patterns-work-together-rust/

A little heads up from us: Expect different class structures in Rust compared to Java and the original diagram given by the HFDP book. This is because of different approaches that Rust applies for Object-Oriented programming compared to Java, especially about object ownership, traits, and lifetimes. You can review **Module 6: Concurrency** again about object ownership concept.

Adapter Pattern: Adapting **`Goose`** to **`Quackable`** using **`GooseAdapter`**

Look at the diagram below for the solution of the **`GooseAdapter`** problem:

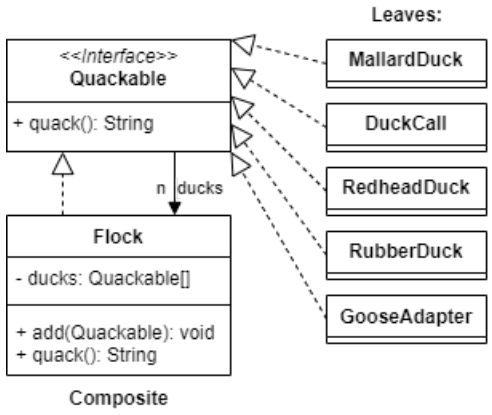


All the duck variants are implementing the **`Quackable`** interface. The **`Quackable`** interface requires the **`quack()`** method to be implemented. Meanwhile, **`Goose`** implements the **`honk()`** method. As we want to adapt **`Goose`**'s honk so it can be invoked like **`Duck`**'s quack, we need to adapt the **`honk()`** method to the **`quack()`** method. So, we will make a **`GooseAdapter`** class that implements **`Quackable`**. The **`quack()`** implementation will call the **`honk()`** method of the wrapped **`Goose`** object. This is the example of **`quack()`** code in **`GooseAdapter`** using Rust:

```
01      fn quack(&self) -> String {
02          return self.goose.quack();
03      }
```

Composite Pattern: Making **`Flock`** of Ducks

Look at the diagram below for the solution of the **`Flock`** problem:

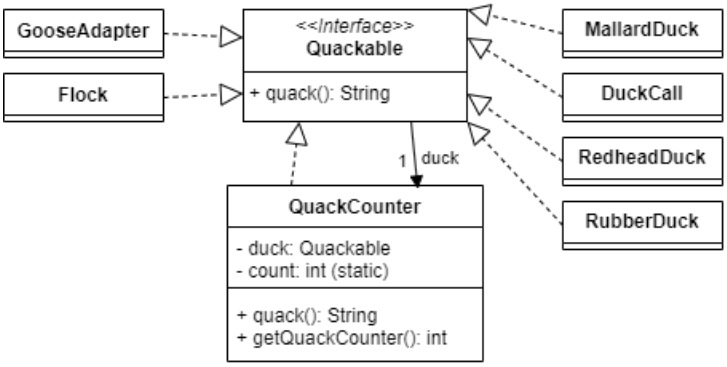


We want to manage a duck, multiple ducks, and a hierarchy of ducks, all in the same way. This can be achieved by implementing a Composite pattern. As all the Duck types and **`GooseAdapter`** class already implement the **`Quackable`** interface, we can assume the **`Quackable`** interface as the Component abstraction. The original classes that are implementing **`Quackable`** are the leaves. The only thing left to be implemented is the Composite.

For the Composite, we need to make a new class, called **`Flock`**. **`Flock`** contains a list of ducks (**`Quackable`**s). **`Flock`** itself, just like other classes, also implements the **`Quackable`** interface. For the **`quack()`** implementation, it will loop through every **`Quackable`** object saved in the list and execute each of the **`quack()`** methods.
The next question is, can we make **`Flock`**s of **`Flock`**? The answer is, **yes!**

Decorator Pattern: Making **`QuackCounter`** decorator for **`Quackable`**

Look at the diagram below for the solution of the **`QuackCounter`** problem:



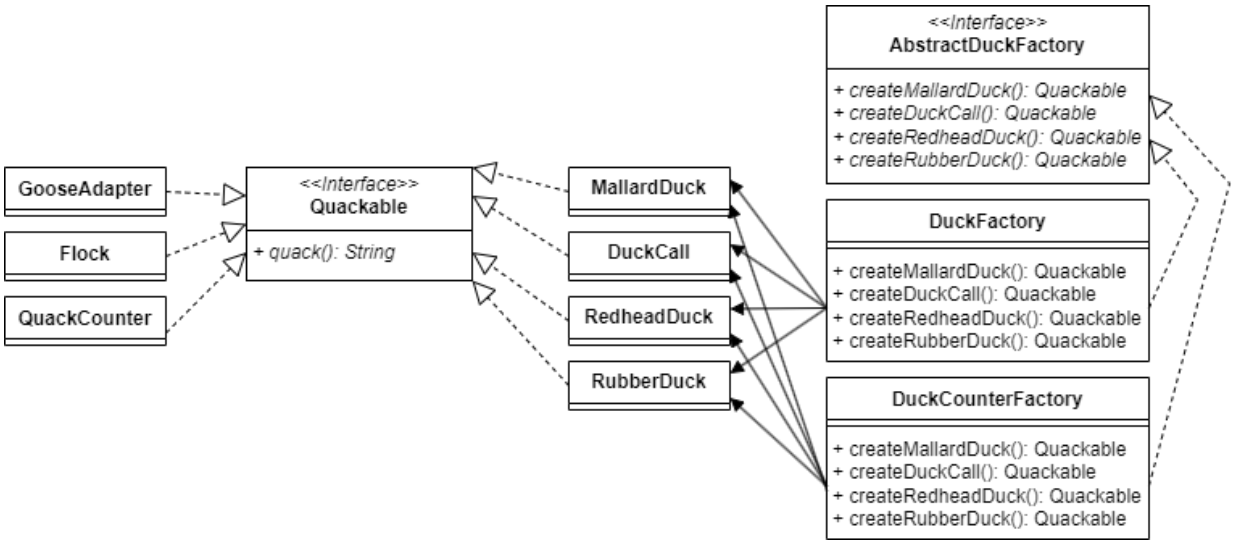
We want to count every quack of the ducks that are in our interest. We may choose which duck we want to count and which not. We can infer that the counting process will be executed **after the duck quacks**, just like when we count people that went into a building. To increase flexibility, we will use the Decorator pattern to add such behaviour into the **`quack()`** method, without modifying any of the Duck implementations. By using Decorator pattern, we can also pick ducks that we want to count by wrapping the duck object with the decorator object.

Decorators and the base classes should be treated the same, that is why we need a common abstraction. We have it already: the **`Quackable`** interface. We just need to create a new Decorator, called **`QuackCounter`**. The **`quack()`** method in **`QuackCounter`** will call its wrapped object's **`quack()`** method first, then increments the **`count`** "static variable" by 1 (one). To get the current count of all quacks, we can use **`getQuackCounter()`** which is a new method that is implemented only in **`QuackCounter`**.

You might want to ask, "Why in this case there is only one interface, not two? In previous material about Decorator pattern, there are two interfaces: **`Component`** and **`Decorator`**."
The answer is that in this case there is no need for a specific **`Decorator`** interface yet, so we can use the **`Component`** interface directly. Also, there are no additional methods that will be commonly implemented across all Decorators (**`getQuackCounter()`** is only specific to **`QuackCounter`**). We will make one if there is another decorator, for example: decorator to append quack messages.

Abstract Factory Pattern: Making factories for plain and decorated **`Quackable`**

Look at the diagram below for the solution of **`Quackable`** instantiation encapsulation problem:



As the Decorator makes the **`Quackable`** object instantiation process becoming more tedious and cluttering the **`DuckSimulator`** main class, we need to encapsulate the object instantiation process

to a separate code. As there are four kinds of duck (**`GooseAdapter`** is not included), a single Factory Method is not enough. We need the Abstract Factory pattern to solve this problem.

We can make an **`AbstractDuckFactory`** as the abstract factory interface. There are four factory methods inside that will create each of the duck types. Those methods have a return type of **`Quackable`** because the mission of this abstract factory is to simplify object creation and decorator wrapping process. If we use concrete classes as the return type (like the **`MallardDuck`** or the **`QuackCounter`**), we cannot make an abstraction out of the factory methods.

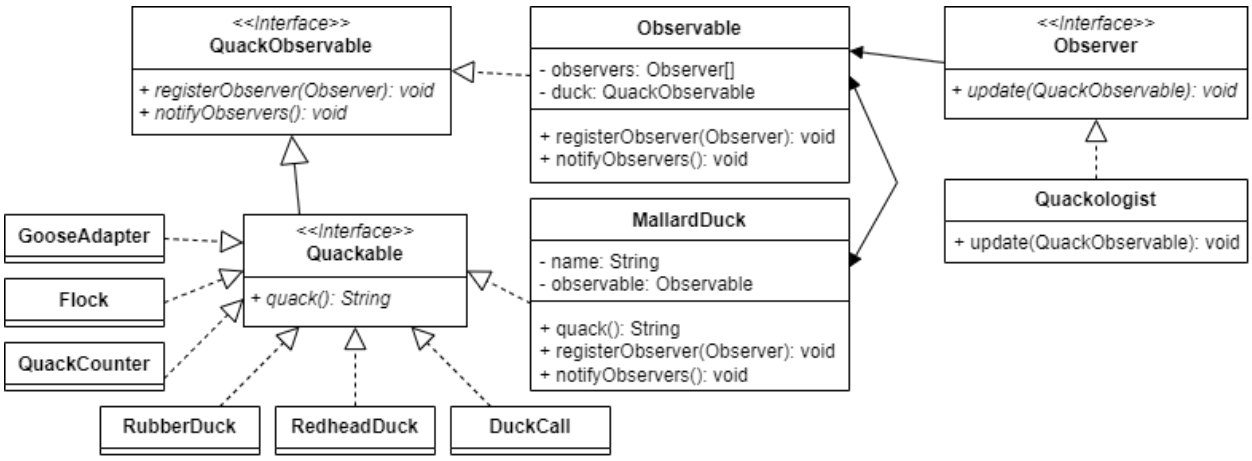
We will implement two different factories using this abstraction, named **`DuckFactory`** and **`DuckCounterFactory`**. **`DuckFactory`** will create plain duck objects of each type. **`DuckCounterFactory`** will create a plain duck object of each type.

Observer Pattern: Making Ducks **`Observable`** to **`Quackologist`**

The solution of the “observing quacks” problem involves many class design changes. This is also the part that differs between Java and Rust, because of the restrictions Rust imposed on object ownership, traits, thread-safe operations, and lifetimes.

Java Approach: Observer pattern with Observable object

This is the class diagram for Java version of this problem’s solution:



In this case, every duck (and duck-alike) will be observable. To achieve that, we need another abstraction for an observable (the thing that is being observed). In this class diagram, that abstraction is represented as a **`QuackObservable`** interface. Then, the original **`Quackable`** interface extends the **`QuackObservable`** interface. All **`Quackable`** objects (duck variants and duck-alike classes) must implement three methods: **`quack()`**, **`registerObserver(Observer)`**, and **`notifyObservers()`**.

If we follow the original Observer design pattern to solve this problem, we need to duplicate **observers** list implementation for all **Quackable** objects. This is tedious. Why not modify it a little bit, by making a different **Observable** class. The **Observable** class will implement **QuackObservable**. The **Observable** contains the main logic on the operating list of **observers**, including logic to add a new **Observer** and notifying data to each member of **observers**.

By making a different **Observable** class, we can delegate the notification logic from an **Quackable** object to the **Observable** object made specifically for the respective **Quackable** object. This is the code snippet of **MallardDuck** to show you the delegation process (statements that are printed **bold** are the delegation logic from **MallardDuck** to **Observable**):

```
01 public class MallardDuck implements Quackable {
02     Observable observable = new Observable(this);
03
04     public void quack() {
05         System.out.println("Quack");
06         this.notifyObservers();
07     }
08
09     @Override
10     public void registerObserver(Observer observer) {
11         this.observable.registerObserver(observer);
12     }
13
14     @Override
15     public void notifyObservers() {
16         this.observable.notifyObservers();
17     }
18 }
```

For the **Observer** interface and **Quackologist** class implementation, it is the same as what we have learned from the previous submodule about the **Decorator pattern**. Their role is the observers. They wait for the **update(QuackObservable)** method to be invoked by the **Observable** object. In this case, the **Observable** object will send the whole duck or duck-alike object to the **Observer**.

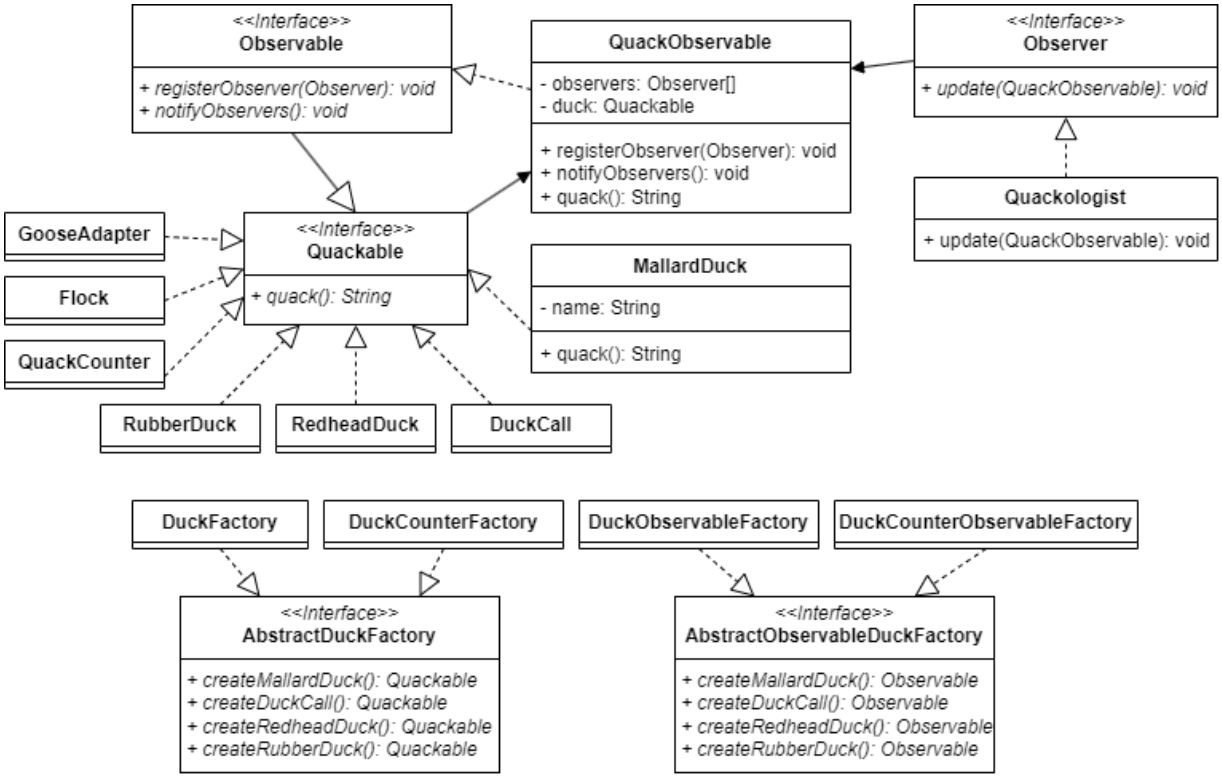
Rust Approach: Decorator + Observer pattern

For Rust, the design will be different, mainly because the ownership restriction in Rust makes it impossible for an object to pass itself to another object. In Java, we can do that easily, for example: **new Observable(this)**. Because of this, the delegation of notification logic from duck or duck-alike objects to **Observable** objects is not possible. We still want to avoid duplication of notification logic, though.

We can avoid duplication of notification logic by using another pattern: **Decorator pattern**. The idea is to make a new decorator called **QuackObservable**, that will wrap a **Quackable** object. The **QuackObservable** decorator struct will contain the notification logic. The **QuackObservable**

decorator struct will also implement **`Quackable`**, by delegating the **`quack()`** logic to the wrapped **`Quackable`** object.

Here is the class diagram for this problem's solution design in Rust:



The difference between the Rust version and the Java version is that **`Observable`** interface now extends the **`Quackable`** interface. In the Java version, it is the opposite. This is because we want to make **`Quackable`** interface still holds the Component abstraction role in Decorator pattern. If we make **`Quackable`** interface extends the **`Observable`** interface, then we should still make notification logic inside of every **`Quackable`**, which violates our goal.

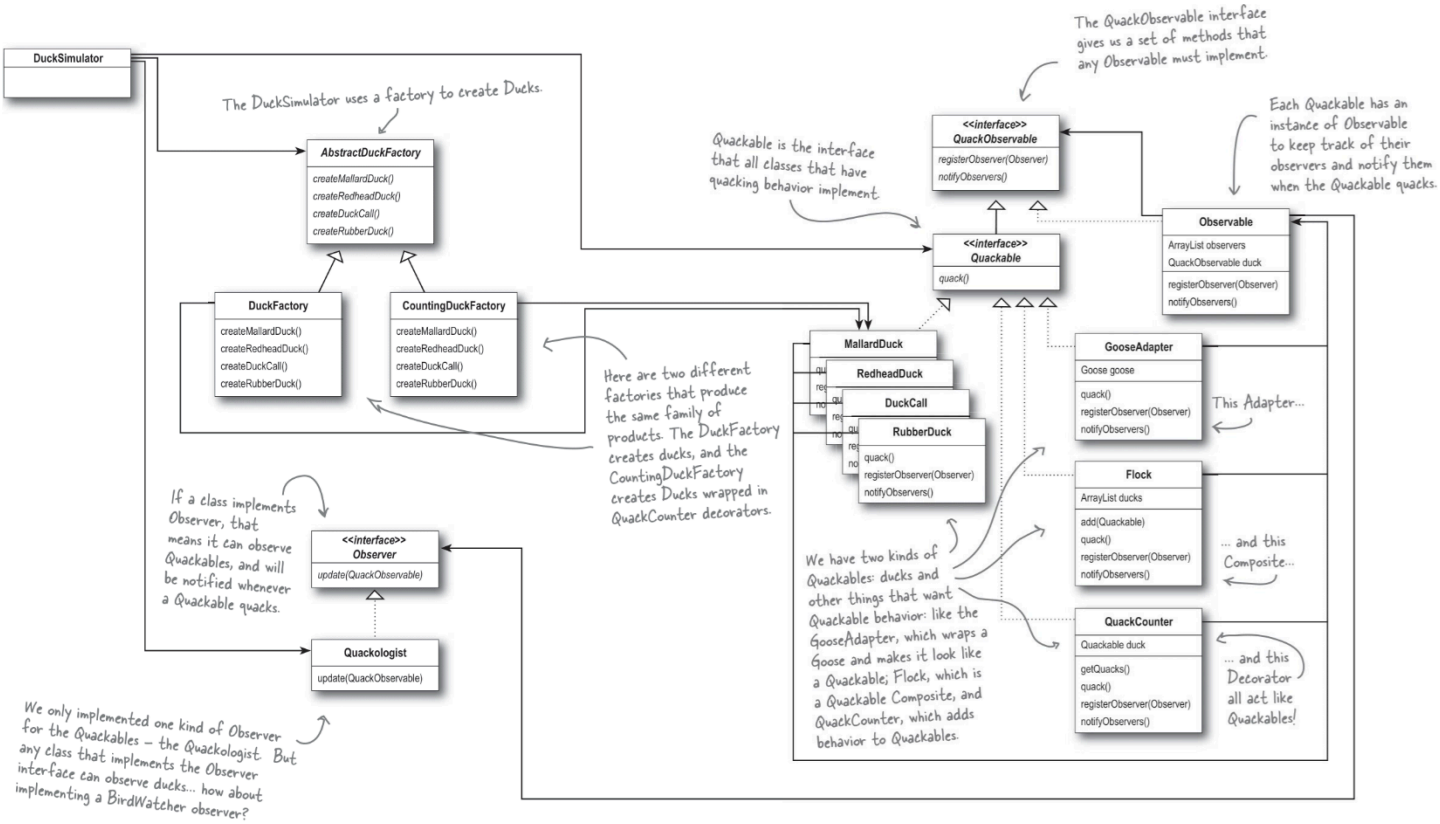
Now that our main problem is solved, the **`DuckSimulator`** is now broken. This is because Rust does not support type casting seamlessly, unlike in Java. We can only upcast (from **`Observable`** to **`Quackable`**), using a library called **`as_dyn_trait`**. Downcast is not supported. This is the error that we will find if we do not use the external library when doing upcasting:

```
error[E0658]: cannot cast `dyn Observable` to `dyn Quackable`, trait upcasting coercion is experimental
  --> src\main.rs:34:15
   |
34 |     flock.add(duck_call);
   |               ^^^^^^^^^
   |
   = note: see issue #65991 <https://github.com/rust-lang/rust/issues/65991> for more information
   = note: required when coercing `Box<dyn Observable>` into `Box<(dyn Quackable + 'static)>`
```

This casting issue also makes our previous abstract factory (**`AbstractDuckFactory`**) unusable because it will return **`dyn Quackable`** type. Meanwhile, we need to access the **`register_observer(Observer)`** method that only exists in **`dyn Observable`** type. Again, down casting is not supported. So, we need to make another abstract factory interface called **`AbstractObservableDuckFactory`**, which still contains a factory method for each duck type, but this time it will return **`dyn Observable`** type. To see the abstract factory usage, you can look at the **`solution`** branch of the Rust example repository for Patterns that Work Together.

Final Class Design

The HFDP book already has a class diagram with sufficient explanations to summarise the process of integrating those design patterns. The diagram that HFDP provides is based on the Java codebase. Please zoom in to look at the explanations given by the diagram.

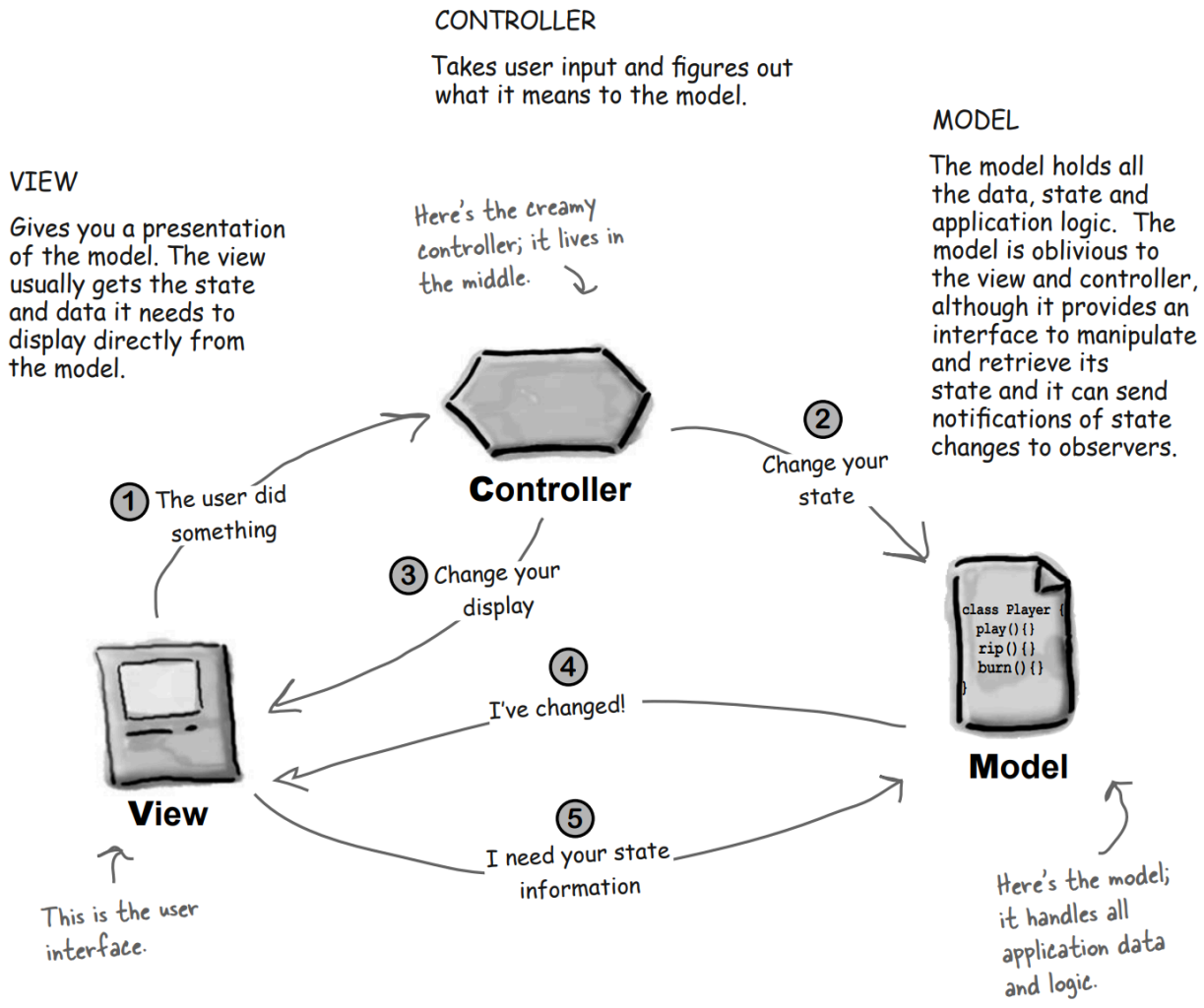


We have successfully integrated the Decorator, Composite, Adapter, and Observer pattern with this case. The solution might have room for improvements, though. Write your suggestions to improve the design (can be either Java version or Rust version, or both) here.

Model-View-Controller (MVC)

Model-View-Controller is a compound design pattern that defines the workflow on how a data is being modelled, processed, and presented. Compound patterns like MVC define roles, responsibilities, and interactions between roles. Individual design patterns that make up MVC define how those interactions work.

Roles, Responsibilities, and Interactions



There are three roles in the definition of Model-View-Controller:

1. **`Model`**: The model holds all the data along with their current state and application logic. The model does not remember any reference of **`View`**s or **`Controller`**s. The model only provides an interface to retrieve or manipulate states. The model acts as the "Publisher" that knows the state of data and can notify it to **`View`**.

2. **`Controller`**: The controller stays in the middle of the program. The **`Controller`** knows how to interpret user inputs from **`View`**, so that it can be used to change the data states in the **`Model`**. **`Controller`** also knows how to change a **`View`**'s display based on the notifications of state changes that a View receives from a **`Model`**.
3. **`View`**: The view is the user interface you (as the user of the program) see, access, and interact with. If the user is doing something with the **`View`**, it will consult the **`Controller`**. The Controller then instructs some changes to the display that a **`View`** shows.

The interaction from a user request to system response in an MVC application is as follows:

1. The user interacts with **`View`**. The user gives some input that will be passed by the **`View`** to the **`Controller`**.
2. The **`Controller`** interprets user input that was given by the **`View`**. Based on the mapping between input possibilities and their respective state changes, a **`Controller`** then instructs the **`Model`** to change the state in the data.
3. After requesting data state changes, the **`Controller`** may ask the **`View`** to change the display, for example, disabling some buttons. **`Controller`** knows which display change is relevant to a set of user inputs or to a state change.
4. When the **`Model`** successfully processed the request of the **`Controller`**, the **`Model`** then changed its state, so it is ready to be called by **`View`**s that are interested in that **`Model`**. The **`View`** then asks the **`Model`** for the detailed data states, periodically, or on demand, using the Pull model Observer pattern.
5. The **`View`** then changes its display using those data states. The **`View`** might redo step no. 3 on consulting the **`Controller`** about how to change the display.

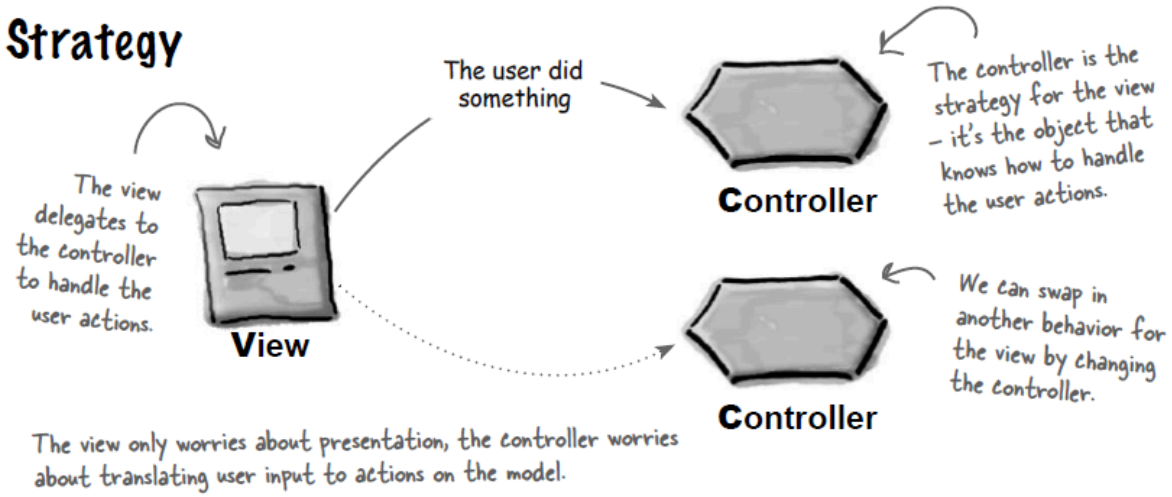
Some frameworks, like Django, have different terminology for this pattern. In Django, this design pattern is called Model-Template-View (MTV) as the Template = View in original MVC, and View = Controller in original MVC.

Design Patterns Used

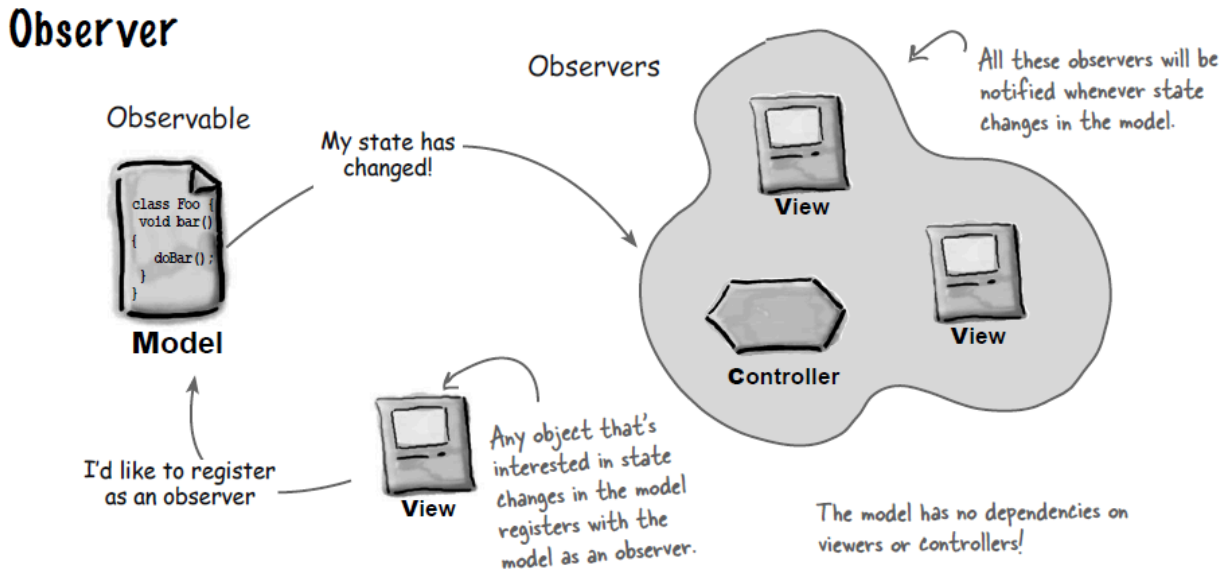
In compound patterns, the key is to not think about individual design patterns strictly by definition. There are some changes necessary to integrate those design patterns. For example, since the **`Model`** does not have the reference to its related **`View`**, it does not have a "list of observers" like the Publisher does in the definition of Observer pattern. Here is the list of individual design patterns that is used to build MVC compound pattern:

1. **Strategy**: The Strategy pattern defines the relationship between a **`View`** and a **`Controller`**. The **`View`** is an object that will be configured by a strategy. The **`Controller`** is the strategy. The **`Controller`** knows how to modify the **`View`** and translate user inputs. To change to a different behaviour, a **`View`** can swap the **`Controller`** object. The **`View`** is only concerned

with visual elements of the application. The Strategy pattern keeps the **`View`** decoupled from the **`Model`** because the logic is on the **`Controller`**.

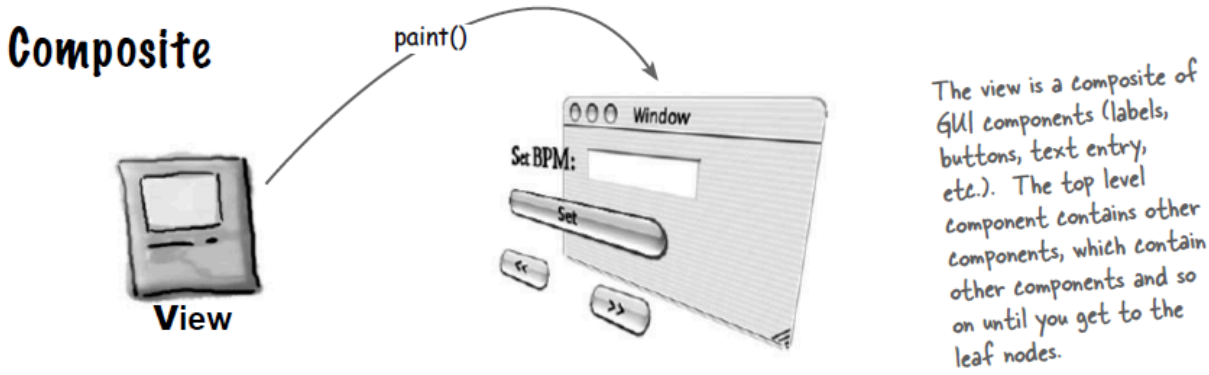


2. **Observer**: The Observer pattern defines the relationship between a **`Model`** and a **`View`** or a **`Controller`**. The **`Model`** does not know which **`View`** or **`Controller`** has interests in its data. The **`View`** or the **`Controller`** meanwhile, knows which **`Model`** it needs to contact for requesting an update of data states, by saving the reference of **`Model`** objects to which they are subscribing. They will retrieve data from the **`Model`** periodically or on demand, using the Pull model Observer pattern.



3. **Composite**: The **`View`** itself is managed using hierarchies. For example: HTML DOM, or XML structure, or structure of widgets. A component can be a member of another component. A component can contain multiple other components. If a **`Controller`** requests a change, the **`Controller`** only needs to contact a relevant component, and let

that component distribute the change to its children through the Composite structure.



The advantages of using MVC pattern is as follows:

MVC in Spring Boot

The MVC in Spring Boot is... slightly different. There are not only three, but 5 (five) roles in Spring Boot's MVC. Here are the additional roles:

1. **Service**: This is separated from the `Model`. A **Service** contains application logic that processes an input from other parties (such as a `Controller`) to update the state of a related `Model`.
2. **Repository**: This is also separated from the `Model`. A **Repository** only concerns data storage and retrieval. It concerns how to save `Model` objects into a database. Usually this is where SQL queries happen when you use a SQL database.

Why is the separation of Service and Repository necessary? This is to fulfil Single Responsibility Principle, where an object of a role should only do what a role needs to do. In the original MVC, the `Model` knows about everything related to the data, from business logics, validations, the data itself, even to the data storage operations. By separating business logic to the **Service** and data storage operations to the **Repository**, a `Model` is now only responsible as a "representative" or structure of a set of data. It can then be easily converted to SQL tables and vice-versa because there is no need to instantiate other unrelated methods or fields.

Advantages and Disadvantages

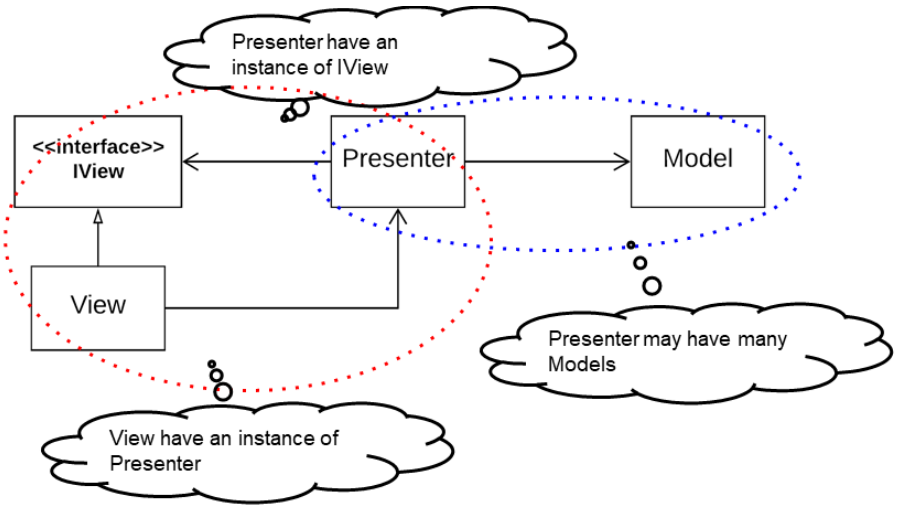
The advantage of MVC is that it increases the code testability and makes it easier to implement new features as it highly supports the separation of concerns. Functionalities of the View can be checked through UI/functional tests if the View respects the Single Responsibility Principle.

The disadvantage of MVC is that it still introduces dependency between code layers, and no parameters to handle UI logic (i.e. how to display the data).

Model-View-Presenter (MVP)

Model-View-Presenter (MVP) is a variant of Model-View-Controller (MVC) compound pattern to model and present data. MVP is often used for GUI programming (especially in Windows).

Roles, Responsibilities, and Interactions



There are three roles defined in Model-View-Presenter (MVP) pattern:

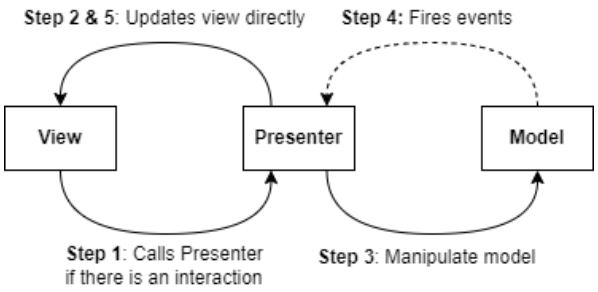
1. **`Model`**: The model holds all the data along with their current state and application logic. The model does not remember any reference of **`View`**s or **`Presenter`**s. The model only provides an interface to retrieve or manipulate states. The model acts as the "Publisher" that knows the state of data and can notify it to the **`Presenter`**.
2. **`Presenter`**: The presenter serves as the event listener of a **`View`**. **`Presenter`** knows how to interpret user inputs from **`View`**, so that it can be used to change the data states in the **`Model`**. The **`Presenter`** also knows how to change a **`View`**'s display based on the notifications of state changes that the **`Presenter`** receives from a **`Model`**.
3. **`View`**: The view is the user interface you (as the user of the program) see, access, and interact with. If the user is doing something with the **`View`**, it will consult the **`Presenter`**. The **`Presenter`** then directly manipulates the display that a **`View`** shows.

The interaction from a user request to system response in an MVP application is as follows:

1. The user interacts with **`View`**. The user gives some input that will be passed by the **`View`** to the **`Presenter`**.
2. The **`Presenter`** may update the **`View`** to change the display, for example, disabling some buttons, as the response of the call from **`View`**.

3. The **`Presenter`** interprets user input that was given by the **`View`**. Based on the mapping between input possibilities and their respective state changes, a **`Presenter`** then manipulates the **`Model`** to change the state in the data.
4. When the **`Model`** successfully processed the request of the **`Presenter`**, the **`Model`** then changed its state, so it is ready to be called by **`Presenter`**s that are interested in that **`Model`**. The **`Presenter`** then asks the **`Model`** for the detailed data states, periodically, or on demand, using the Pull model Observer pattern.
5. When the **`Presenter`** receives the detailed data from the **`Model`**, it will update the **`View`** to change the display based on the data broadcasted by the **`Model`**.

The interaction between a **`Model`**, a **`Presenter`**, and a **`View`** can be drawn as follows:



Design Patterns Used

Here is the list of individual design patterns that is used to build MVP compound pattern:

1. **Strategy**: The Strategy pattern defines the relationship between a **`View`** and a **`Presenter`**. The **`Presenter`** is an object that saves a strategy object to display the data. The **`View`** is the strategy. The **`View`** knows how to display the data given by the **`Presenter`**. To change to a different data display, a **`Presenter`** can swap the **`View`** object. The roles of Strategy pattern in MVP pattern are the opposite of the MVC pattern.
2. **Observer**: The Observer pattern defines the relationship between a **`Model`** and a **`Presenter`**. The **`Model`** does not know which **`Presenter`** has interests in its data. The **`Presenter`**, meanwhile, knows which **`Model`** it needs to contact to manipulate the state, by saving the reference of **`Model`** objects to which they are subscribing. They will retrieve data from the **`Model`** periodically or on demand, using the Pull model Observer pattern.
3. **Composite**: Same as the MVC pattern, **`View`**s are managed hierarchically.

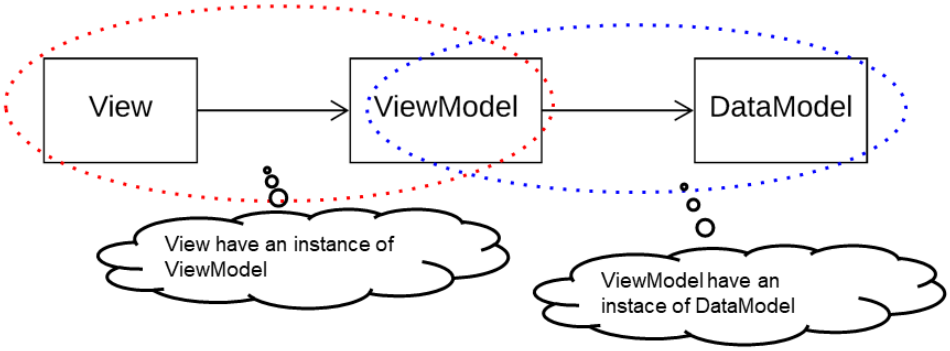
Advantages and Disadvantages

The advantage of MVP is easy code maintenance and testing as the application's model, view, and presenter layer are separated. However, if the code does not follow Single Responsibility Principle, then the Presenter layer tends to expand to a huge all-knowing class.

(Data)Model-View-ViewModel (MVVM)

DataModel-View-Presenter (MVVM) is an “improved” version of Model-View-Controller (MVP) pattern to model and present data. MVVM is often used for mobile GUI programming.

Roles, Responsibilities, and Interactions



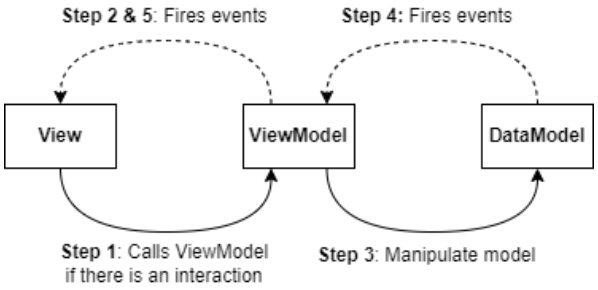
There are three roles defined in (Data)Model-View-Presenter (MVVM) pattern:

1. **`DataModel`**: The DataModel serves as the abstraction of data sources. DataModel will hold the data. The DataModel does not remember any reference of **`ViewModel`**s and **`View`**s.
2. **`View`**: The view is the user interface you (as the user of the program) see, access, and interact with. If the user is doing something with the **`View`**, it will consult the **`ViewModel`**. The **`ViewModel`** then directly manipulates the display that a **`View`** shows.
3. **`ViewModel`**: The **`ViewModel`** exposes data streams that are relevant to the **`View`**. It serves as a link (mediator) between the **`View`** and **`DataModel`**. A **ViewModel** does not remember any reference to **View**s.

The interaction from a user request to system response in an MVP application is as follows:

1. The user interacts with **`View`**. If the user gives some input to the **`View`**, the **`View`** then fires an event to the **`ViewModel`**.
2. Because **`ViewModel`** does not know the reference to a **`View`**, the **`ViewModel`** will fire an event if it wants to change the display of a **`View`**.
3. The **`ViewModel`** interprets the event and user input that was given by the **`View`**. The **`ViewModel`** then manipulates the **`DataModel`** to change the state in the data.
4. When the **`DataModel`** successfully processed the request of a **`ViewModel`**, the **`DataModel`** then changes its state and fires an event to **`ViewModel`**s that are interested with the data. The **`ViewModel`** then asks the **`DataModel`** for the detailed data states, periodically, or on demand, using the Pull model Observer pattern.
5. The **`ViewModel`** then fires an event to change the display of a **`View`** based on the data given by **`DataModel`**.

The interaction between a **`Model`**, a **`Presenter`**, and a **`View`** can be drawn as follows:



Design Pattern Used

Here is the list of individual design patterns that is used to build MVVM compound pattern:

1. **Observer**: The Observer pattern defines the relationship between a **`View`** and a **`ViewModel`**, also between a **`ViewModel`** and a **`DataModel`**.
   When the state of **`ViewModel`** changes, it will notify the **`View`** by firing an event, using Push model observer pattern. The **`View`** will then update the display based on the event.
   The **`ViewModel`**, meanwhile, knows which **`DataModel`** it needs to contact to manipulate the state, by saving the reference of **`DataModel`** objects to which they are subscribing. They will retrieve data from the **`DataModel`** periodically or on demand, using the Pull model Observer pattern.
2. **Mediator**: The **`ViewModel`** can be mentioned as an example of a mediator between a **`View`** and a **`DataModel`**.
3. **Composite**: Same as the MVC pattern, **`View`**s are managed hierarchically.

Advantages and Disadvantages

Advantages of MVVM are:

- It enhances the reusability of code by separating concerns of each layer.
- All modules are independent which improves the testability of each layer.
- Makes project files maintainable and easy to make changes.

Unfortunately, this design pattern is not ideal for small projects, because it will introduce too much boilerplate. For small projects, you can use MVP pattern for less boilerplate. Also, if the data binding logic is too complex, the application will be a little harder to debug.

Tutorial & Group Project Progress

In this tutorial, you will be working on 2 (two) projects. You will implement a notification system that sends notification from a Publisher web app to one or more Receiver/Subscriber web app(s). You will be using Rust in this tutorial, with Rocket as the web framework. For the tutorial, as usual, the checkpoints are marked with **red font colour**.

Starting from this module and onwards, only tutorials are available here. The "exercise" will be replaced by the Group Project progress report. You must implement this week's material into your group project, whenever relevant. **This also makes weekly tutorial and Group Project demonstration to the teaching assistant team mandatory.**

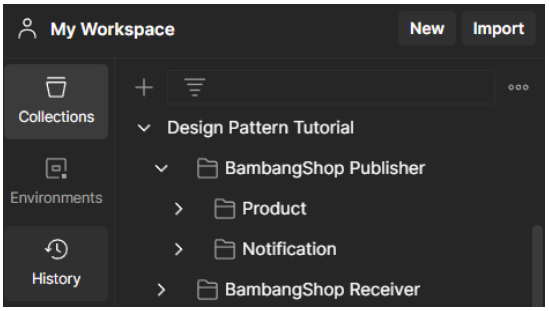
Prerequisites

Before starting the development process, please install the following environments:

1. **Rust**. If you have not installed Rust, visit https://rustup.rs/ to get installer executable for Rust.
2. **Git**
3. **Visual Studio Code** with **rust-analyzer** extension.
4. **Postman** to help you test your works.

Tutorial Preparation

1. Clone https://gitlab.com/ichlaffterlalu/bambangshop **to a new Git repository**. Do not forget to change the **origin** remote URL by executing **`git remote set-url origin [new_repo_link]`** command. This repository contains the **publisher** (or **main**) application.
2. Clone https://gitlab.com/ichlaffterlalu/bambangshop-receiver **to a new Git repository**. Do not forget to change the **origin** remote URL by executing **`git remote set-url origin [new_repo_link]`** command. This repository contains the **subscriber** (or **receiver**) application.
3. Download this Postman collection to help you test your works: https://ristek.link/AdvProgWeek7Postman
4. Import the Postman collection to your Postman workspace, by using the "Import" button on the right side of your workspace's title. If you successfully import the collection, these endpoints will show up in your workspace:



5. In the **main** application, there is a feature already implemented: Products. To check the Product feature, you can run the app using **`cargo run`**.
6. The **main** application by default listens to localhost (127.0.0.1) port 8000.
7. The **receiver** application by default listens to localhost (127.0.0.1) port 8001, but you can change it by editing the **`ROCKET_PORT`** environment variable.
8. For both **main** and **receiver** apps, you can edit/create environment variables using the **`.env`** file, placed in each project's root folder.
9. **Please read `README.md` of each project before you continue doing this tutorial**.

The Problem We Need to Solve

The current condition of BambangShop app can create a new Product, list all Products available in the shop, get a Product by ID, and delete a Product by ID. Pak Bambang wants to expand the exposure of his products sold in BambangShop. This is why Pak Bambang asks you to implement a Notification system.

To use the notification system, BambangShop will provide a **receiver app** that can be used by users to subscribe to one or more product types available. Users can also unsubscribe from a product type. If a user subscribed to a product type, he/she will be notified when:

- There is a new Product with the same product type, or
- A Product with the same product type is deleted from the store, or
- BambangShop promotes a Product with the same product type.

We will solve this problem by using the **Observer pattern**. The **main app** will retain lists of **Subscribers** that follow a product type. Then, when one of the events happens, the **main app** will execute **notify()** to make HTTP requests to every **Subscriber** that subscribed to the relevant product type. The request will contain data for the notification of that current event. These HTTP requests are directed towards the **/receive** URL of the **Receiver app**.

The Main App (Part 1): Models and Repository

To-Do Checklist

1. Create a new file **`src/model/subscriber.rs`**, then make a **`Subscriber`** model struct.

```rust
use rocket::serde::{Deserialize, Serialize};
use rocket::log;
use rocket::serde::json::to_string;
use rocket::tokio;
use bambangshop::REQWEST_CLIENT;
use crate::model::notification::Notification;

#[derive(Debug, Clone, Deserialize, Serialize)]
#[serde(crate = "rocket::serde")]
pub struct Subscriber {
    pub url: String,
    pub name: String,
}
```

2. Edit **`src/model/mod.rs`**, add this line to recognize **`subscriber`** as a submodule.

```rust
pub mod subscriber;
```

3. **Commit your changes to Git with the message: "Create Subscriber model struct."**
4. Create a new file **`src/model/notification.rs`**, then make **`Notification`** model struct.

```rust
use rocket::serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Deserialize, Serialize)]
#[serde(crate = "rocket::serde")]
pub struct Notification {
    pub product_title: String,
    pub product_type: String,
    pub product_url: String,
    pub subscriber_name: String,
    pub status: String
}
```

5. Edit **`src/model/mod.rs`**, add this line to recognize **`notification`** as a submodule.

```rust
pub mod notification;
```

6. **Commit your changes to Git with the message: "Create Notification model struct."**
7. Create a new file **`src/repository/subscriber.rs`**, then make the **`SUBSCRIBERS`** "`lazy static`" variable.

```rust
use dashmap::DashMap;
use lazy_static::lazy_static;
use crate::model::subscriber::Subscriber;

// Singleton of Database
lazy_static! {
    static ref SUBSCRIBERS: DashMap<String, DashMap<String, Subscriber>> = DashMap::new();
}
```

8. Make **SubscriberRepository** struct skeleton in **src/repository/subscriber.rs**.

```
10  pub struct SubscriberRepository;
11
12  impl SubscriberRepository {
13  }
```

9. Edit **src/repository/mod.rs**, add this line to recognize **subscriber** as a submodule.

```
2   pub mod subscriber;
```

10. **Commit your changes to Git with the message: "Create Subscriber database and Subscriber repository struct skeleton."**
11. Implement **add** function in **SubscriberRepository**.

```
13      pub fn add(product_type: &str, subscriber: Subscriber) -> Subscriber {
14          let subscriber_value = subscriber.clone();
15          if SUBSCRIBERS.get(product_type).is_none() {
16              SUBSCRIBERS.insert(String::from(product_type), DashMap::new());
17          };
18
19          SUBSCRIBERS.get(product_type).unwrap()
20              .insert(subscriber_value.url.clone(), subscriber_value);
21          return subscriber;
22      }
```

12. **Commit your changes to Git with the message: "Implement add function in Subscriber repository."**
13. Implement **list_all** function in **SubscriberRepository**.

```
24      pub fn list_all(product_type: &str) -> Vec<Subscriber> {
25          if SUBSCRIBERS.get(product_type).is_none() {
26              SUBSCRIBERS.insert(String::from(product_type), DashMap::new());
27          };
28
29          return SUBSCRIBERS.get(product_type).unwrap().iter()
30              .map(|f| f.value().clone()).collect();
31      }
```

14. **Commit your changes to Git with the message: "Implement list_all function in Subscriber repository."**
15. Implement **delete** function in **SubscriberRepository**.

```
33      pub fn delete(product_type: &str, url: &str) -> Option<Subscriber> {
34          if SUBSCRIBERS.get(product_type).is_none() {
35              SUBSCRIBERS.insert(String::from(product_type), DashMap::new());
36          }
37          let result = SUBSCRIBERS.get(product_type).unwrap()
38              .remove(url);
39          if !result.is_none() {
40              return Some(result.unwrap().1);
41          }
42          return None;
43      }
```

16. **Commit your changes to Git with the message: "Implement delete function in Subscriber repository."**

17. **Write your answer to Reflection Publisher-1 below in README.md of the main project, then commit it.**
18. **Push your commits to your Git repository.**

Reflection Publisher-1

Here are the questions for this reflection:

1. In the Observer pattern diagram explained by the Head First Design Pattern book, Subscriber is defined as an interface. Explain based on your understanding of Observer design patterns, do we still need an interface (or **`trait`** in Rust) in this BambangShop case, or a single Model **`struct`** is enough?

2. **`id`** in **`Program`** and **`url`** in **`Subscriber`** is intended to be unique. Explain based on your understanding, is using **`Vec`** (list) sufficient or using **`DashMap`** (map/dictionary) like we currently use is necessary for this case?

3. When programming using Rust, we are enforced by rigorous compiler constraints to make a thread-safe program. In the case of the List of Subscribers (**`SUBSCRIBERS`**) static variable, we used the **`DashMap`** external library for **thread safe `HashMap`**. Explain based on your understanding of design patterns, do we still need **`DashMap`** or we can implement Singleton pattern instead?

Please write your reflection inside the Main app repository's `README.md` file.

The Main App (Part 2): Service and Controller/Handler

To-Do Checklist

1. Create a new file **src/service/notification.rs**, then make **NotificationService** struct skeleton.

```rust
use std::thread;

use bambangshop::{Result, compose_error_response};
use rocket::http::Status;
use crate::model::notification::Notification;
use crate::model::product::Product;
use crate::model::subscriber::Subscriber;
use crate::repository::subscriber::SubscriberRepository;

pub struct NotificationService;

impl NotificationService {
}
```

2. Edit **src/service/mod.rs**, add this line to recognize **notification** as a submodule.

```rust
pub mod notification;
```

3. Create new file **src/controller/notification.rs**, then insert these dependencies.

```rust
use rocket::response::status::Created;
use rocket::serde::json::Json;

use bambangshop::Result;
use crate::model::subscriber::Subscriber;
use crate::service::notification::NotificationService;
```

4. Edit **src/controller/mod.rs**, add this line to recognize **notification** as a submodule.

```rust
pub mod notification;
```

5. Edit **src/controller/mod.rs**, modify **route_stage** function to add methods in submodule **notification** to be mapped to **/notification** as follows.

```rust
pub fn route_stage() -> AdHoc {
    return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
        rocket
            .mount("/product", routes![product::create, product::list, product::read, produ
            .mount("/notification", routes![])
    });
```

6. **Commit your changes to Git with the message: "Create Notification service struct skeleton."**

7. Add **subscribe** function to **NotificationService** struct in **src/service/notification.rs**.

```rust
pub fn subscribe(product_type: &str, subscriber: Subscriber) -> Result<Subscriber> {
    let product_type_upper: String = product_type.to_uppercase();
    let product_type_str: &str = product_type_upper.as_str();
    let subscriber_result: Subscriber = SubscriberRepository::add(product_type_str, subscriber);
    return Ok(subscriber_result);
}
```

8. **Commit your changes to Git with the message: “Implement subscribe function in Notification service.”**
9. Implement **subscribe** function in **src/controller/notification.rs**.

```rust
#[post("/subscribe/<product_type>", data = "<subscriber>")]
pub fn subscribe(product_type: &str, subscriber: Json<Subscriber>) -> Result<Created<Json<Subscriber>>> {
    return match NotificationService::subscribe(product_type, subscriber.into_inner()) {
        Ok(f) => Ok(Created::new("/").body(Json::from(f))),
        Err(e) => Err(e)
    };
}
```

10. Add **subscribe** handler function in **route_stage** function in **src/controller/mod.rs**. Like step no. 5, but not identical. Edit it as follows:

```rust
pub fn route_stage() -> AdHoc {
    return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
        rocket
            .mount("/product", routes![product::create, product::list, product::read, produ
            .mount("/notification", routes![notification::subscribe])
    });
}
```

11. **Commit your changes to Git with the message: “Implement subscribe function in Notification controller.”**
12. Add the **unsubscribe** function to **NotificationService** struct in **src/service/notification.rs**.

```rust
pub fn unsubscribe(product_type: &str, url: &str) -> Result<Subscriber> {
    let product_type_upper: String = product_type.to_uppercase();
    let product_type_str: &str = product_type_upper.as_str();
    let result: Option<Subscriber> = SubscriberRepository::delete(product_type_str, url);
    if result.is_none() {
        return Err(compose_error_response(
            Status::NotFound,
            String::from("Subscriber not found.")
        ));
    }
    return Ok(result.unwrap());
}
```

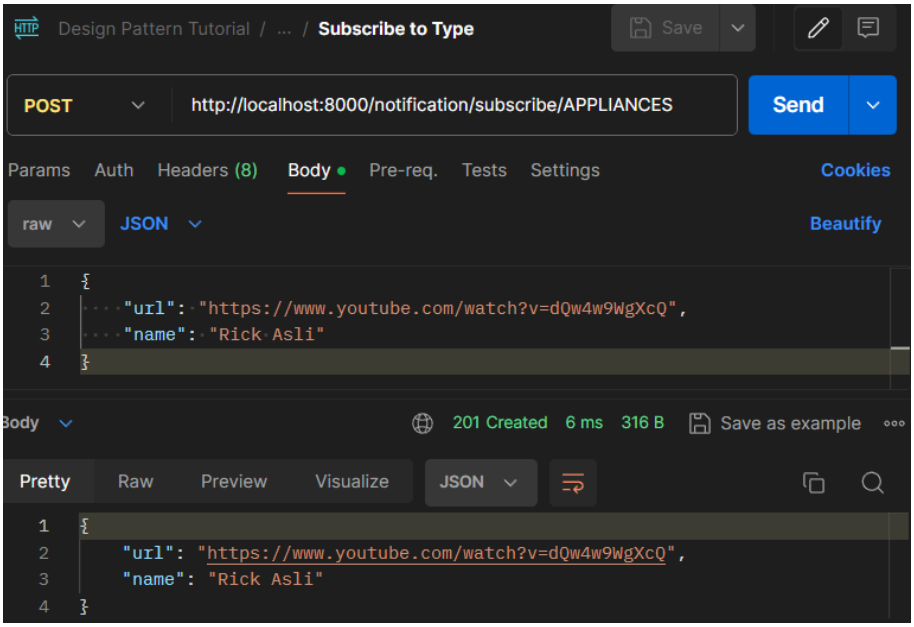
13. **Commit your changes to Git with the message: “Implement unsubscribe function in Notification service.”**
14. Implement **unsubscribe** handler function in **src/controller/notification.rs**.

```rust
#[post("/unsubscribe/<product_type>?<url>")]
pub fn unsubscribe(product_type: &str, url: &str) -> Result<Json<Subscriber>> {
    return match NotificationService::unsubscribe(product_type, url) {
        Ok(f) => Ok(Json::from(f)),
        Err(e) => Err(e)
    };
}
```

15. Add **`unsubscribe`** handler function in **`route_stage`** function in **`src/controller/mod.rs`**. Like step no. 5, but not identical. Edit it as follows:

```rust
pub fn route_stage() -> AdHoc {
    return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
        rocket
            .mount("/product", routes![product::create, product::list, product::read, product::de
            .mount("/notification", routes![notification::subscribe, notification::unsubscribe])
    });
}
```

16. **Commit your changes to Git with the message: "Implement unsubscribe function in Notification controller."**
17. **Write your answer to Reflection Publisher-2 below in `README.md` of the main project, then commit it.**
18. **Push your commits to your Git repository.**
19. **At this point, you can try your new methods by executing `cargo run`.**
20. You can use the provided Postman collection to interact with the app using HTTP requests. Here is the example of the subscription request:



Reflection Publisher-2

Here are the questions for this reflection:

1. In the Model-View Controller (MVC) compound pattern, there is no “Service” and “Repository”. Model in MVC covers both data storage and business logic. Explain based on your understanding of design principles, why we need to separate “Service” and “Repository” from a Model?

2. What happens if we only use the Model? Explain your imagination on how the interactions between each model (**`Program`**, **`Subscriber`**, **`Notification`**) affect the code complexity for each model?

3. Have you explored more about **Postman**? Tell us how this tool helps you to test your current work. You might want to also list which features in Postman you are interested in or feel like it is helpful to help your Group Project or any of your future software engineering projects.

Please write your reflection inside the Main app repository's `README.md` file.

The Main App (Part 3): Subscriber's Update Logic

To-Do Checklist

1. Implement **`update`** method in **`Subscriber`** model in **`src/model/subscriber.rs`**. This method will send a notification to a **`Subscriber`** based on the provided URL, using HTTP POST request. We use **`tokio`** library to make async functions usable as sync functions, and the **`reqwest`** external library to make HTTP POST requests.

```rust
impl Subscriber {
    #[tokio::main]
    pub async fn update(&self, payload: Notification) {
        REQWEST_CLIENT
            .post(&self.url)
            .header("Content-Type", "JSON")
            .body(to_string(&payload).unwrap())
            .send().await.ok();
        log::warn_!("Sent {} notification of: [{}] {}, to: {}",
            payload.status, payload.product_type, payload.product_title, self.url);
    }
}
```

2. **Commit your changes to Git with the message: "Implement update method in Subscriber model to send notification HTTP requests."**
3. Implement notify function in **`NotificationService`** in **`src/service/notification.rs`**. This method will **prepare notification payload** and then make threads to call **`update()`** method for each **`subscriber`**.

```rust
    pub fn notify(&self, product_type: &str, status: &str, product: Product) {
        let mut payload: Notification = Notification {
            product_title: product.clone().title,
            product_type: String::from(product_type),
            product_url: product.clone().get_url(),
            subscriber_name: String::from(""),
            status: String::from(status)
        };

        let subscribers: Vec<Subscriber> = SubscriberRepository::list_all(product_type);
        for subscriber in subscribers {
            payload.subscriber_name = subscriber.clone().name;
            let subscriber_clone = subscriber.clone();
            let payload_clone = payload.clone();
            thread::spawn(move || subscriber_clone.update(payload_clone));
        }
    }
```

4. **Commit your changes to Git with the message: "Implement notify function in Notification service to notify each Subscriber."**

5. Implement **publish** function in **ProductService** in **src/service/product**. This function will enable the shop owner to promote a product to its subscribers.

```
49      pub fn publish(id: usize) -> Result<Product> {
50          let product_opt: Option<Product> = ProductRepository::get_by_id(id);
51          if product_opt.is_none() {
52              return Err(compose_error_response(
53                  Status::NotFound,
54                  String::from("Product not found.")
55              ));
56          }
57          let product: Product = product_opt.unwrap();
58
59          NotificationService.notify(&product.product_type, "PROMOTION", product.clone());
60          return Ok(product);
61      }
```

6. Implement **publish** handler function in **src/controller/product**.

```
41  #[post("/<id>/publish")]
42  pub fn publish(id: usize) -> Result<Json<Product>> {
43      return match ProductService::publish(id) {
44          Ok(f) => Ok(Json::from(f)),
45          Err(e) => Err(e)
46      };
47  }
```

7. Add **publish** handler function in **route_stage** function in **src/controller/mod.rs**. This is to map the new handler to **/product** endpoint. Edit it as follows:

```
6   pub fn route_stage() -> AdHoc {
7       return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
8           rocket
9               .mount("/product", routes![product::create, product::list,
10                  product::read, product::delete, product::publish])
11              .mount("/notification", routes![notification::subscribe, notification::unsubscribe])
12      });
13  }
```

8. **Commit changes to Git with this message: "Implement publish function in Program service and Program controller."**
9. Import **NotificationController** to **src/service/product.rs**.

```
7   use crate::service::notification::NotificationService;
```

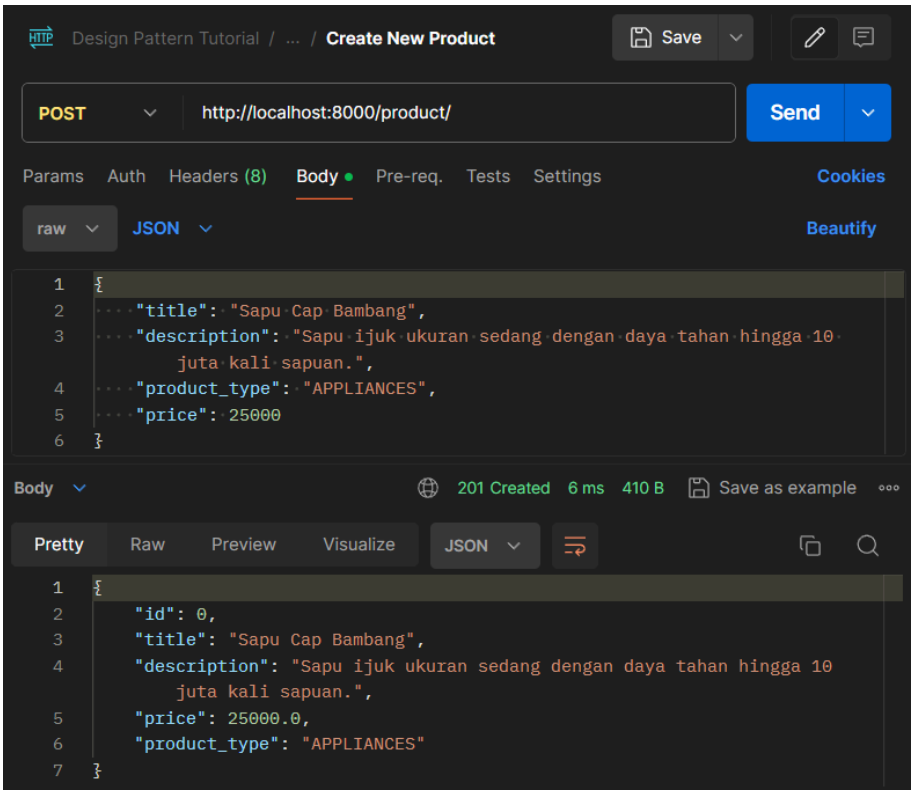
10. Add **NotificationController::notify** function call to **create** function in **ProductService** in **src/service/product.rs** as follows. This will notify the **subscribers** that a new product is created.

```
12      pub fn create(mut product: Product) -> Result<Product> {
13          product.product_type = product.product_type.to_uppercase();
14          let product_result: Product = ProductRepository::add(product);
15
16          NotificationService.notify(&product_result.product_type, "CREATED",
17              product_result.clone());
18          return Ok(product_result);
19      }
```

11. Add **`NotificationController::notify`** function call to **`delete`** method in **`ProductService`** in **`src/service/product.rs`** as follows. This will notify the **`subscribers`** that a new product is deleted.

```rust
pub fn delete(id: usize) -> Result<Json<Product>> {
    let product_opt: Option<Product> = ProductRepository::delete(id);
    if product_opt.is_none() {
        return Err(compose_error_response(
            Status::NotFound,
            String::from("Product not found.")
        ));
    }
    let product: Product = product_opt.unwrap();

    NotificationService.notify(&product.product_type, "DELETED", product.clone());
    return Ok(Json::from(product));
}
```

12. **Commit the changes to Git with this message: "Edit Product service methods to call notify after create/delete."**
13. **At this point, you can try your new methods by executing `cargo run`.**
14. You can use the provided Postman collection to interact with the app using HTTP requests. If you subscribe to a product type, the program then will try to notify you on every product creation, promotion, and deletion in a product type.
    Here is the example of product creation request:



This will invoke the **`update()`** method on the **"Rick Asli" `subscriber` object** if you successfully

implement The Main App tutorial Part 1 until Part 3. The log shown in terminal is as follows:

```
POST /product/ application/json:
   >> Matched: (create) POST /product/
   >> Outcome: Success(201 Created)
   >> Response succeeded.
   >> Sent CREATED notification of: [APPLIANCES] Sapu Cap Bambang, to: https://www.youtube.com/watch?v=dQw4w9WgXcQ
```

15. **Write your answer to Reflection Publisher-3 below in `README.md` of the main project, then commit it.**
16. **Push your commits to your Git repository.**

Reflection Publisher-3

Here are the questions for this reflection:

1. Observer Pattern has two variations: **Push model** (publisher pushes data to subscribers) and **Pull model** (subscribers pull data from publisher). In this tutorial case, which variation of Observer Pattern that we use?

2. What are the advantages and disadvantages of using the other variation of Observer Pattern for this tutorial case? (example: if you answer Q1 with Push, then imagine if we used Pull)

3. Explain what will happen to the program if we decide to not use multi-threading in the notification process.

Please write your reflection inside the Main app repository's `README.md` file.

The Receiver App (Part 1): Models and Repository

To-Do Checklist

1. Create a new file **`src/model/subscriber.rs`**, then make **`SubscriberRequest`** model struct.

```rust
use rocket::serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Deserialize, Serialize)]
#[serde(crate = "rocket::serde")]
pub struct SubscriberRequest {
    pub url: String,
    pub name: String,
}
```

2. Edit **`src/model/mod.rs`**, add this line to recognize **`subscriber`** as a submodule.

```rust
pub mod subscriber;
```

3. **Commit your changes to Git with the message: "Create SubscriberRequest model struct."**
4. Create a new file **`src/model/notification.rs`**, then make **`Notification`** model struct. This struct implements **`Display`** trait (a trait is like an interface in Java). **`Display`** trait can be applied to structs that need to have **`String`** representation.

```rust
use std::fmt::{Display, Formatter, Result};

use rocket::serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Deserialize, Serialize)]
#[serde(crate = "rocket::serde")]
pub struct Notification {
    pub product_title: String,
    pub product_url: String,
    pub product_type: String,
    pub subscriber_name: String,
    pub status: String
}

impl Display for Notification {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        if self.status.to_uppercase().eq("CREATED") {
            return write!(f,
                "Hello {}, let's try our new {} product: {}, only in BambangShop! Check it out: {}",
                self.subscriber_name, self.product_type.to_lowercase(), self.product_title, self.product_url);
        } else if self.status.to_uppercase().eq("DELETED") {
            return write!(f,
                "Hello {}, we informed that our {} product called {} already sold out...",
                self.subscriber_name, self.product_type.to_lowercase(), self.product_title);
        } else {
            return write!(f,
                "Hello {}, let's try our {} product: {}, grab it out before the stock ran out! Check it out: {}",
                self.subscriber_name, self.product_type.to_lowercase(), self.product_title, self.product_url);
        }
    }
}
```

5. Edit **src/model/mod.rs**, add this line to recognize **notification** as a submodule.

```
2   pub mod notification;
```

6. **Commit your changes to Git with the message: "Create Notification model struct."**
7. Create a new file src/repository/notification.rs, then make the **NOTIFICATIONS** "lazy static" variable.

```
1   use std::sync::RwLock;
2
3   use lazy_static::lazy_static;
4
5   use crate::model::notification::Notification;
6
7   // Singleton of Database
8   lazy_static! {
9       static ref NOTIFICATIONS: RwLock<Vec<Notification>> = RwLock::new(vec![]);
10  }
```

8. Make **NotificationRepository** struct skeleton in **src/repository/notification.rs**.

```
12  pub struct NotificationRepository;
13
14  impl NotificationRepository {
15  }
```

9. Edit **src/repository/mod.rs**, add this line to recognize **notification** as a submodule.

```
1   pub mod notification;
```

10. **Commit your changes to Git with the message: "Create Notification database and Notification repository struct skeleton."**
11. Implement **add** function in **NotificationRepository**.

```
15      pub fn add(notification: Notification) -> Notification {
16          NOTIFICATIONS.write().unwrap()
17              .push(notification.clone());
18          return notification;
19      }
```

12. **Commit your changes to Git with the message: "Implement add function in Subscriber repository."**
13. Implement **list_all_as_string** function in **SubscriberRepository**.

```
21      pub fn list_all_as_string() -> Vec<String> {
22          return NOTIFICATIONS.read().unwrap()
23              .iter().map(|f| format!("{}", f.clone())).collect();
24      }
25  }
```

14. **Commit your changes to Git with the message: "Implement list_all_as_string function in Subscriber repository."**
15. **Write your answer to Reflection Subscriber-1 below in README.md of the main project, then commit it.**
16. **Push your commits to your Git repository.**

Reflection Subscriber-1

Here are the questions for this reflection:

1. In this tutorial, we used **`RwLock<>`** to synchronise the use of **`Vec`** of **`Notification`**s. Explain why it is necessary for this case, and explain why we do not use **`Mutex<>`** instead?

2. In this tutorial, we used **`lazy_static`** external library to define **`Vec`** and **`DashMap`** as a "**`static`**" variable. Compared to Java where we can mutate the content of a **`static`** variable via a **`static`** function, why did not Rust allow us to do so?

Please write your reflection inside the Receiver app repository's `README.md` file.

The Receiver App (Part 2): Service and Controller/Handler

To-Do Checklist

1. Create a new file **`src/service/notification.rs`**, then make **`NotificationService`** struct skeleton.

```rust
use std::thread;

use rocket::http::Status;
use rocket::log;
use rocket::serde::json::to_string;
use rocket::tokio;

use bambangshop_receiver::{APP_CONFIG, REQWEST_CLIENT, Result, compose_error_response};
use crate::model::notification::Notification;
use crate::model::subscriber::SubscriberRequest;
use crate::repository::notification::NotificationRepository;

pub struct NotificationService;

impl NotificationService {
}
```

2. Edit **`src/service/mod.rs`**, add this line to recognize **`notification`** as a submodule.

```rust
pub mod notification;
```

3. Create new file **`src/controller/notification.rs`**, then insert these dependencies.

```rust
use rocket::serde::json::Json;

use bambangshop_receiver::Result;
use crate::model::notification::Notification;
use crate::model::subscriber::SubscriberRequest;
use crate::service::notification::NotificationService;
```

4. Edit **`src/controller/mod.rs`,** add this line to recognize **`notification`** as a submodule.

```rust
pub mod notification;
```

5. **Commit your changes to Git with the message: “Create Notification service struct skeleton.”**
6. Add **`subscribe_request`** private **`async`** function to **`NotificationService`** struct in **`src/service/notification.rs`**. As subscribe requests involve HTTP request creation, the approach will be using asynchronous programming, just like what you did before when implementing **`notify()`** function in **The Main App part 3**.

```rust
#[tokio::main]
async fn subscribe_request(product_type: String) -> Result<SubscriberRequest> {
    let product_type_upper: String = product_type.to_uppercase();
    let product_type_str: &str = product_type_upper.as_str();
    let notification_receiver_url: String = format!("{}/receive",
        APP_CONFIG.get_instance_root_url());
    let payload: SubscriberRequest = SubscriberRequest {
        name: APP_CONFIG.get_instance_name().to_string(),
        url: notification_receiver_url
    };
```

```rust
    let request_url: String = format!("{}/notification/subscribe/{}",
        APP_CONFIG.get_publisher_root_url(), product_type_str);
    let request = REQWEST_CLIENT
        .post(request_url.clone())
        .header("Content-Type", "application/json")
        .header("Accept", "application/json")
        .body(to_string(&payload).unwrap())
        .send().await;
    log::warn_!("Sent subscribe request to: {}", request_url);

    return match request {
        Ok(f) => match f.json::<SubscriberRequest>().await {
            Ok(x) => Ok(x),
            Err(y) => Err(compose_error_response(
                Status::NotAcceptable,
                y.to_string()
            ))
        },
        Err(e) => Err(compose_error_response(
            Status::NotFound,
            e.to_string()
        ))
    }
}
```

7. Add **subscribe** function to **NotificationService** struct in **src/service/notification.rs**. This function will make a thread that executes **subscribe_request** function.

```rust
pub fn subscribe(product_type: &str) -> Result<SubscriberRequest> {
    let product_type_clone = String::from(product_type);
    return thread::spawn(move || Self::subscribe_request(product_type_clone))
        .join().unwrap();
}
```

8. **Commit your changes to Git with the message: “Implement subscribe function in Notification service.”**
9. Implement **subscribe** function in **src/controller/notification.rs**.

```rust
#[get("/subscribe/<product_type>")]
pub fn subscribe(product_type: &str) -> Result<Json<SubscriberRequest>> {
    return match NotificationService::subscribe(product_type) {
        Ok(f) => Ok(Json::from(f)),
        Err(e) => Err(e)
    };
}
```

10. Add **subscribe** handler function in **route_stage** function in **src/controller/mod.rs**. This is to map the **subscribe** handler function to the root URL route.

```
pub fn route_stage() -> AdHoc {
    return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
        rocket
            .mount("/", routes![notification::subscribe])
    });
}
```

11. **Commit your changes to Git with the message: "Implement subscribe function in Notification controller."**
12. Add **unsubscribe_request** private **async** function to **NotificationService** struct in **src/service/notification.rs**. Just like **subscribe_request**, this involves HTTP request so the function must be made **async**.

```
#[tokio::main]
async fn unsubscribe_request(product_type: String) -> Result<SubscriberRequest> {
    let product_type_upper: String = product_type.to_uppercase();
    let product_type_str: &str = product_type_upper.as_str();
    let notification_receiver_url: String = format!("{}/receive",
        APP_CONFIG.get_instance_root_url());
```

```
    let request_url: String = format!("{}/notification/unsubscribe/{}?url={}",
        APP_CONFIG.get_publisher_root_url(), product_type_str, notification_receiver_url);
    let request = REQWEST_CLIENT
        .post(request_url.clone())
        .header("Content-Type", "application/json")
        .header("Accept", "application/json")
        .send().await;
    log::warn_!("Sent unsubscribe request to: {}", request_url);

    return match request {
        Ok(f) => match f.json::<SubscriberRequest>().await {
            Ok(x) => Ok(x),
            Err(_y) => Err(compose_error_response(
                Status::NotFound,
                String::from("Already unsubscribed to the topic.")
            ))
        },
        Err(e) => Err(compose_error_response(
            Status::NotFound,
            e.to_string()
        ))
    }
}
```

13. Add the **unsubscribe** function to NotificationService struct in **src/service/notification.rs**. This function will make a thread that executes the **unsubscribe_request** function.

```
pub fn unsubscribe(product_type: &str) -> Result<SubscriberRequest> {
    let product_type_clone = String::from(product_type);
    return thread::spawn(move || Self::unsubscribe_request(product_type_clone))
        .join().unwrap();
}
```

14. **Commit your changes to Git with the message: “Implement unsubscribe function in Notification service.”**
15. Implement **unsubscribe** handler function in **src/controller/notification.rs**.

```
#[get("/unsubscribe/<product_type>")]
pub fn unsubscribe(product_type: &str) -> Result<Json<SubscriberRequest>> {
    return match NotificationService::unsubscribe(product_type) {
        Ok(f) => Ok(Json::from(f)),
        Err(e) => Err(e)
    };
}
```

16. Add **unsubscribe** handler function in **route_stage** function in **src/controller/mod.rs**. This is to map the **unsubscribe** handler function to the root URL route.

```
pub fn route_stage() -> AdHoc {
    return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
        rocket
            .mount("/", routes![notification::subscribe, notification::unsubscribe])
    });
}
```

17. **Commit your changes to Git with the message: “Implement unsubscribe function in Notification controller.”**
18. Add **receive_notification** function to **NotificationService** struct in **src/service/notification.rs**.

```
pub fn receive_notification(payload: Notification) -> Result<Notification> {
    let subscriber_result: Notification = NotificationRepository::add(payload);
    return Ok(subscriber_result);
}
```

19. **Commit your changes to Git with the message: “Implement receive_notification function in Notification service.”**
20. Implement **receive** handler function in **src/controller/notification.rs**.

```
#[post("/receive", data = "<notification>")]
pub fn receive(notification: Json<Notification>) -> Result<Json<Notification>> {
    return match NotificationService::receive_notification(notification.into_inner()) {
        Ok(f) => Ok(Json::from(f)),
        Err(e) => Err(e)
    };
}
```

21. Add **receive** handler function in **route_stage** function in **src/controller/mod.rs**. This is to map the **receive** handler function to the root URL route.

```
pub fn route_stage() -> AdHoc {
    return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
        rocket
            .mount("/", routes![notification::subscribe, notification::unsubscribe,
                notification::receive])
    });
}
```

22. **Commit your changes to Git with the message: “Implement the receive function in Notification controller.”**
23. Add **`list_messages`** function to **`NotificationService`** struct in **`src/service/notification.rs`**.

```
 99   pub fn list_messages() -> Result<Vec<String>> {
100       return Ok(NotificationRepository::list_all_as_string());
101   }
```

24. **Commit your changes to Git with the message: “Implement list_messages function in Notification service.”**
25. Implement **`list`** handler function in **`src/controller/notification.rs`**.

```
33   #[get("/")]
34   pub fn list() -> Result<Json<Vec<String>>> {
35       return match NotificationService::list_messages() {
36           Ok(f) => Ok(Json::from(f)),
37           Err(e) => Err(e)
38       };
39   }
```

26. Add **`list`** handler function in **`route_stage`** function in **`src/controller/mod.rs`**. This is to map the **`list`** handler function to the root URL route.

```
 5   pub fn route_stage() -> AdHoc {
 6       return AdHoc::on_ignite("Initializing controller routes...", |rocket| async {
 7           rocket
 8               .mount("/", routes![notification::subscribe, notification::unsubscribe,
 9                   notification::receive, notification::list])
10       });
11   }
```

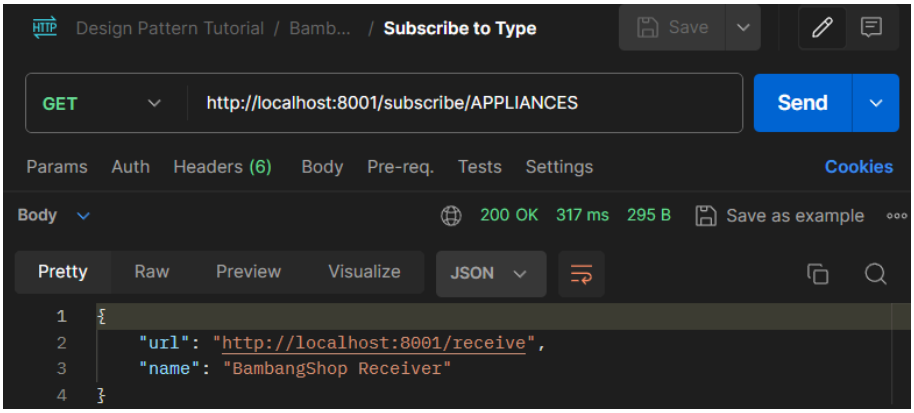
27. **Commit your changes to Git with the message: “Implement list function in Notification controller.”**
28. **Push your commits to your Git repository.**
29. **Create three different instances of Receiver app that listens to port 8001, 8002, and 8003, respectively.** You can change the port and instance name by creating .env file, as follows:

```
1   ROCKET_PORT=8001
2   APP_INSTANCE_ROOT_URL=http://localhost:${ROCKET_PORT}
3   APP_PUBLISHER_ROOT_URL=http://localhost:8000
4   APP_INSTANCE_NAME="Huki Kamal"
```

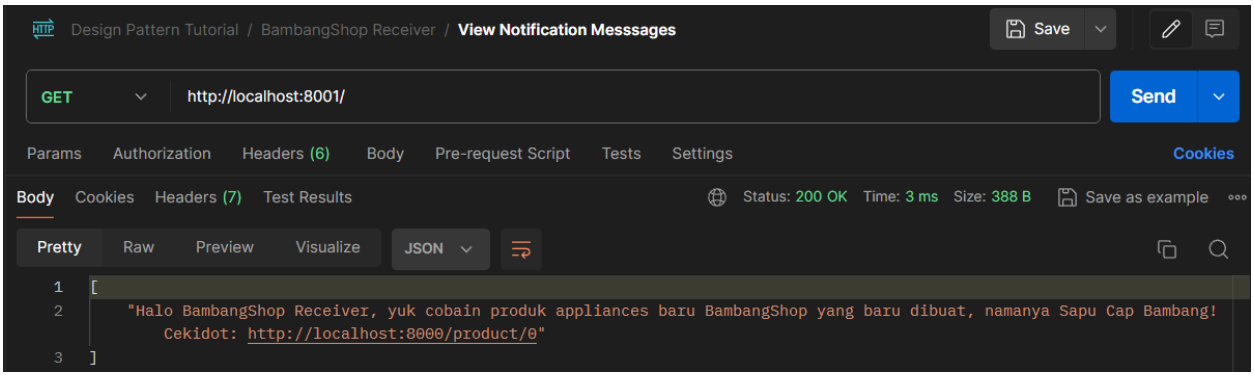
To change the port, change the **`ROCKET_PORT`** variable. To change the name, change the **`APP_INSTANCE_NAME`** variable.

After changing those environment variables, open a new terminal then execute **`cargo run`**. **Repeat until you have three different instances of Receiver app and one instance of Main app running simultaneously.**

30. You can use the provided Postman collection to interact with the app using HTTP requests. Here is the example of the subscription request:



31. If you have successfully subscribed to a product type, you can execute again the create Product, publish Product, or delete Product endpoints of the Main app. See whether the notification comes to the Receiver app by accessing the root of Receiver app URL.



32. **Retry again the step 31 - 32 by subscribing to different product types.** See if all the notifications appear in the correct instances of the Receiver app or not.
33. **Write your answer to Reflection Subscriber-2 below in `README.md` of the main project, then commit it.**
34. **Push your commits to your Git repository.**

Reflection Subscriber-2

Here are the questions for this reflection:

1. Have you explored things outside of the steps in the tutorial, for example: **`src/lib.rs`**? If not, explain why you did not do so. If yes, explain things that you have learned from those other parts of code.

2. Since you have completed the tutorial by now and have tried to test your notification system by spawning multiple instances of Receiver, explain how Observer pattern eases you to plug in more subscribers. How about spawning more than one instance of Main app, will it still be easy enough to add to the system?

3. Have you tried to make your own Tests, or enhance documentation on your **Postman collection**? If you have tried those features, tell us whether it is useful for your work (it can be your tutorial work or your Group Project).

Please write your reflection inside the Receiver app repository's README.md file.

Grading Scheme

Scale

All components will be scored on scale 0-4. Grade 4 is the highest grade, equivalent to A. Score 0 is for not submitting.

Components

- 40% - Commits (Tutorial)
- 20% - Reflections (4% each, 3 from main app, 2 from receiver app)
- 40% - Group Project progress (grading scheme made by teaching assistant)
  For Design Pattern material, the implementation in Group Project can be any design patterns, not just Observer. Framework-bound patterns (like MVC) do not count.

For **Group Project progress**, it is claimable by attending weekly demonstrations to the teaching assistant team. To claim a score for a topic, students need to prove they have implemented a certain week's material as individual work in the Group Project. **Claiming process can be done outside the material's week** (for example: Design Pattern can still be claimed at Week 12), however **there is a limit of maximum 2 (two) topics claimed per week**.

Rubrics

	Score 4	Score 3	Score 2	Score 1
Commits (Tutorial)	100% of the commits **in the PR/MR** are correct according to the tutorial.	>= 75% of the commits **in the PR/MR** are correct.	>=50% of the commits **in the PR/MR** are correct.	<50% of the commits are correct (**or >50% but no PR/MR**).
Reflections	The description is sound, and the analysis is comprehensive.	The description is sound.	The description is not sound although still related.	The description is not sound. It is not related to the topics.