COMP2511

Tutorial 7

Last Tutorial

- Streams
- Strategy Pattern
- Observer Pattern

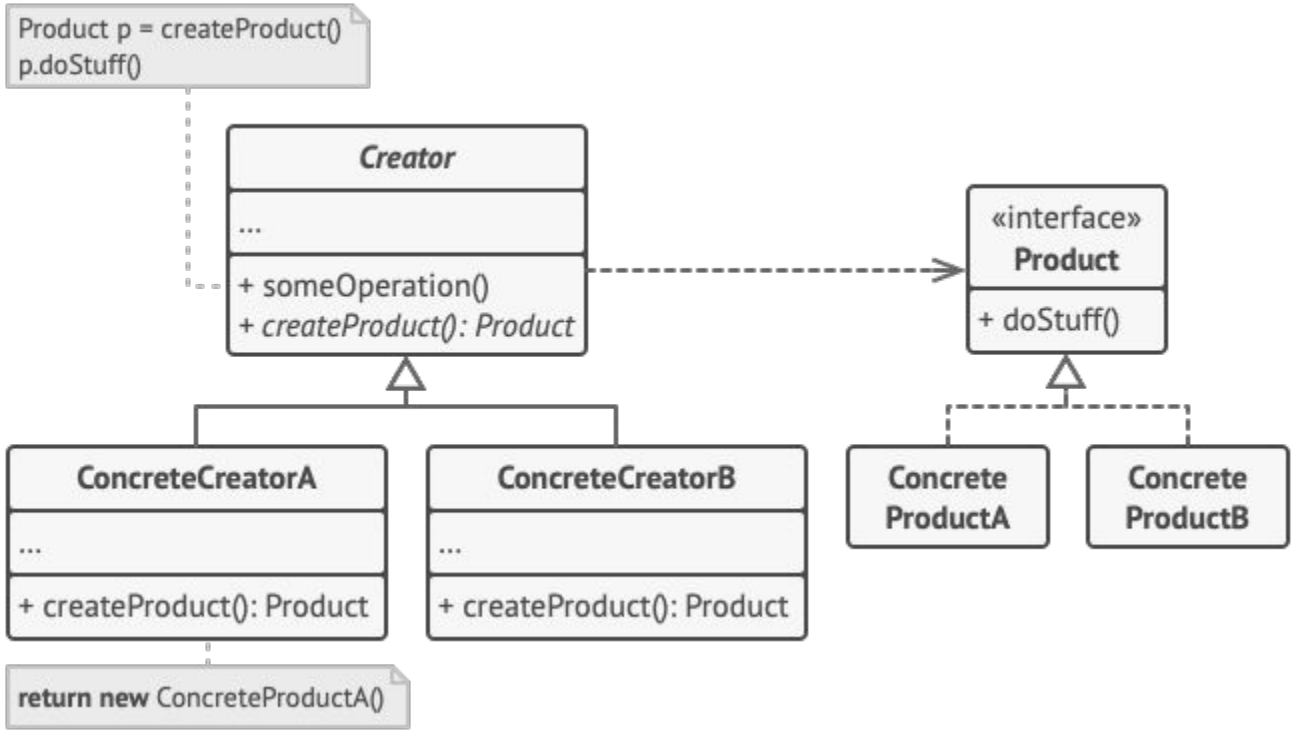
This Tutorial

- Abstract Factory Pattern
- Decorator Pattern
- Singleton Pattern

Factory Pattern (again)

- For the purposes of this course, think of the Factory Pattern as simply moving construction logic to a separate “factory” class.
- This is different to how the Factory Pattern is presented elsewhere.
- In the more typical definition of the Factory Pattern, factories are used to create objects *without specifying their concrete types*.
  - Suppose we had a `Vehicle` interface and `Car`, `Train` and `Plane` classes implementing it.
  - If we had logic that worked for any `Vehicle`, we can make a `VehicleFactory` interface with a prescribed `createVehicle()` method, then concrete `CarFactory`, `TrainFactory` and `PlaneFactory` classes implementing this interface.
  - Then, a vehicle can be created by calling `createVehicle()`, with the *concrete type* (`Car`, `Train`, `Plane`) unknown - but this doesn’t matter as long as we are only calling `Vehicle` interface methods in the program logic (due to polymorphism!).

Factory Pattern



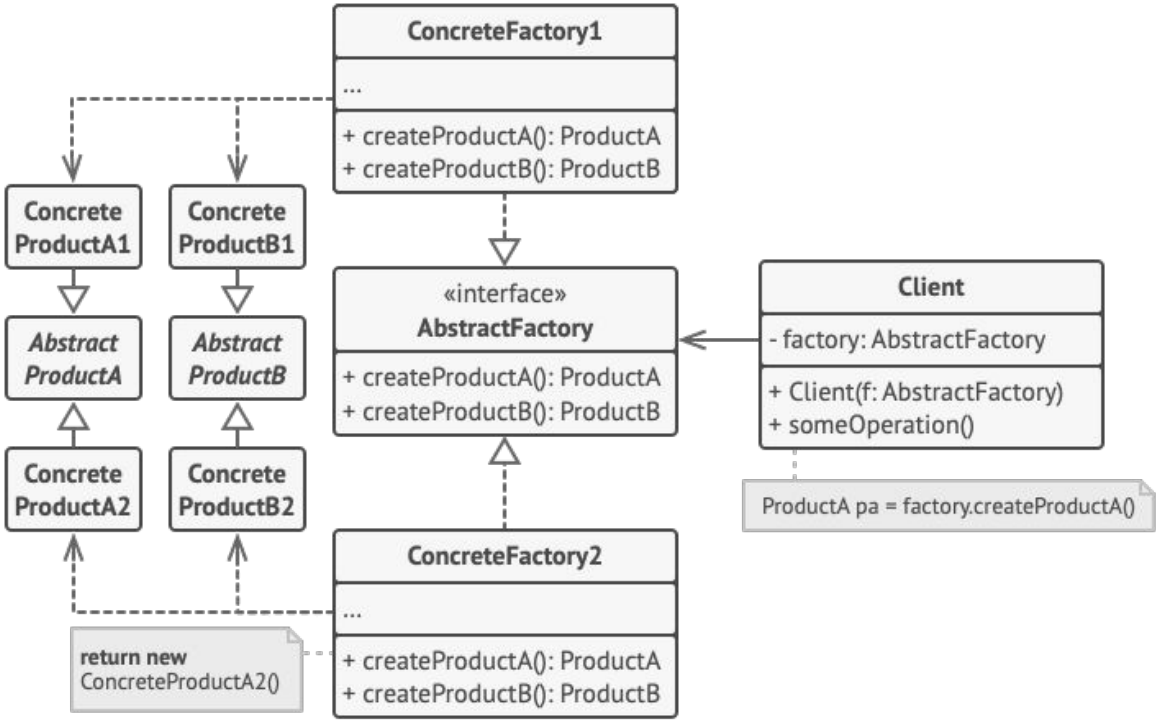
Abstract Factory Pattern

- The Abstract Factory Pattern is a (creational) design pattern that provides interfaces to create **families of related objects** without specifying their concrete classes.
    - As the name suggests, this is very similar to the Factory Pattern. The difference comes into play when the individual items you are creating can be classified into different categories or families.
- The Abstract Factory pattern suggests that you should define a distinct factory for each category, which all implement a common interface that prescribes what objects each of the factories should be able to produce.
    - This comes with the usual benefits of the Factory Pattern, ensures that related objects are kept together and makes changing between different categories very easy.

Abstract Factory Pattern

- For example, consider shirts, pants and jackets. These are group of related objects (clothing) which can be further classified into different categories like colour, size, brand etc.
- If you were categorising clothing by colour, you could have a `ClothingFactory` interface with `createShirt()`, `createPants()`, `createJacket()` methods, and concrete `RedClothingFactory`, `GreenClothingFactory` etc. classes which each provide their own implementation of those methods.

Abstract Factory Pattern Implementation



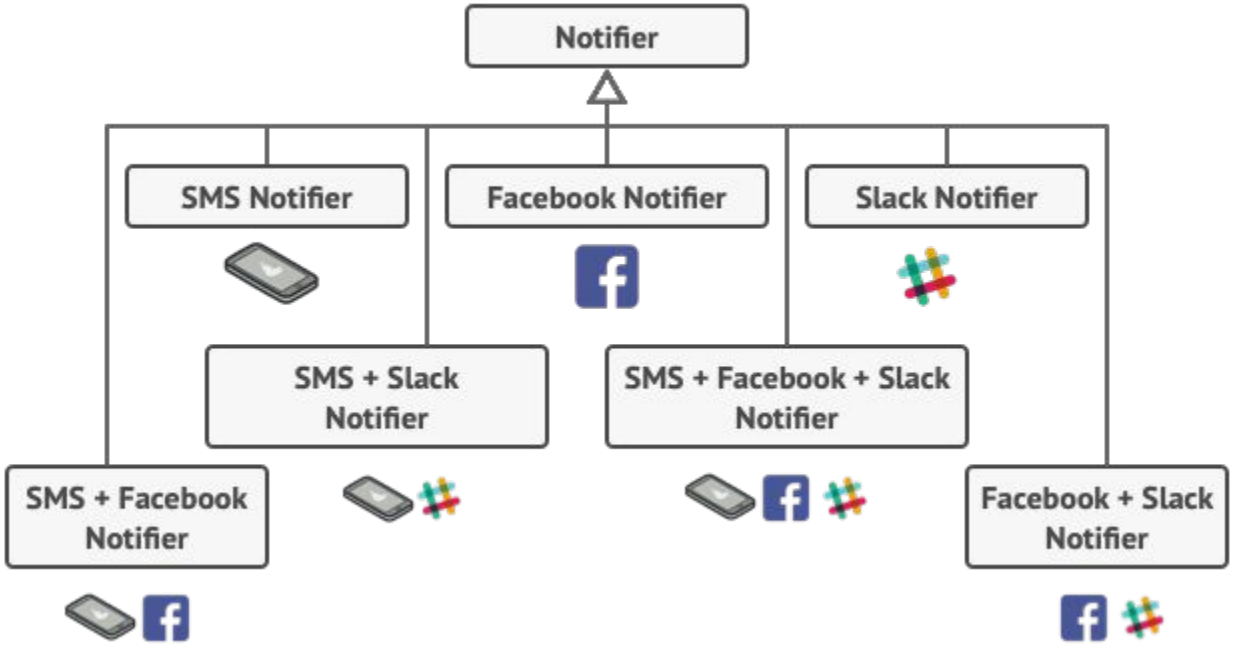
Live Example: Abstract Factory Pattern

- Inside **src/universities**, there is a command-line system for registering students and lecturers from either USYD or UNSW.
  - People involved with UNSW have 7-digit IDs in the form 5XXXXXX.
  - People involved with USYD have 9-digit IDs.
- Apply the Abstract Factory Pattern to improve the quality of the code.

Decorator Pattern

- Suppose (for some reason) we were modelling different combinations of ice cream flavour combinations as classes. For example,
  - `VanillaIceCream`
  - `VanillaAndStrawberryIceCream`
  - `ChocolateAndStrawberryIceCream`
  - `VanillaChocolateAndStrawberryIceCream`
  - and so on...
- What is the immediate issue with this approach?
  - We would end up with *way* too many classes.
- How can we effectively model classes in these scenarios, where we have many different functionalities that may or may not be present and can be stacked on top of each other?

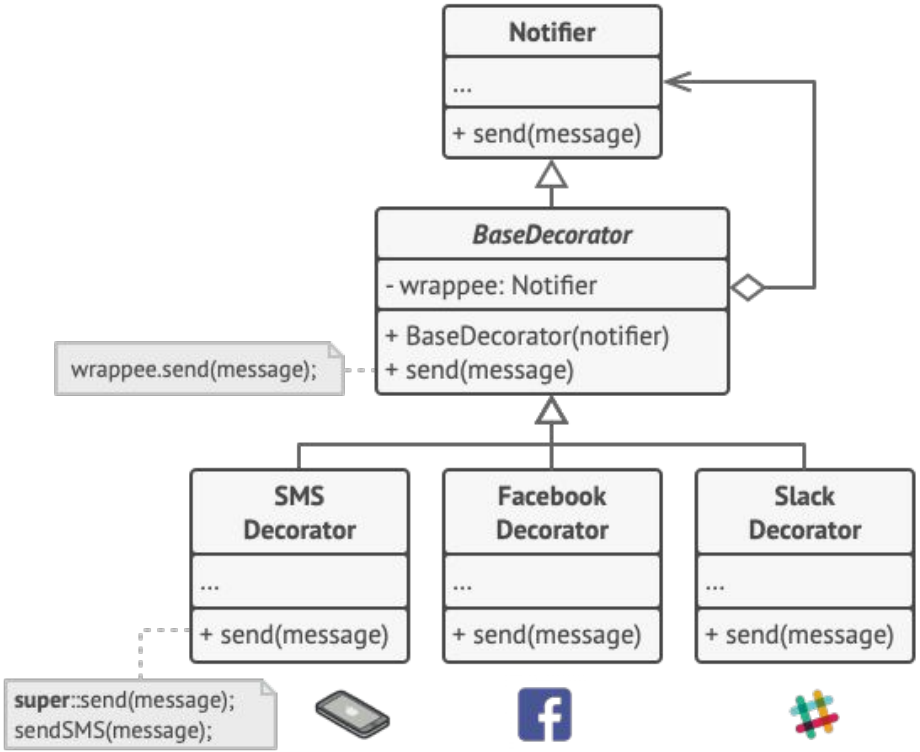
Decorator Pattern



Decorator Pattern

- The Decorator Pattern is a (structural) design pattern that allows for new behaviours to be attached on top of an existing object.
- The existing object gets stored (**wrapped**) in another object, referred to as a **decorator**. The decorator calls on the original functionality of the wrapped object, then modifies it or adds additional functionality.
    - Importantly, both unwrapped and wrapped objects need to implement a common interface, as this will allow multiple decorators to be chained.
    - In the ice cream example, the unwrapped object could be an ice cream cone, and the decorators would be the different scoops of ice cream - we are adding more things on top of the ice cream cone, and they can be added in any order.

Decorator Pattern Implementation



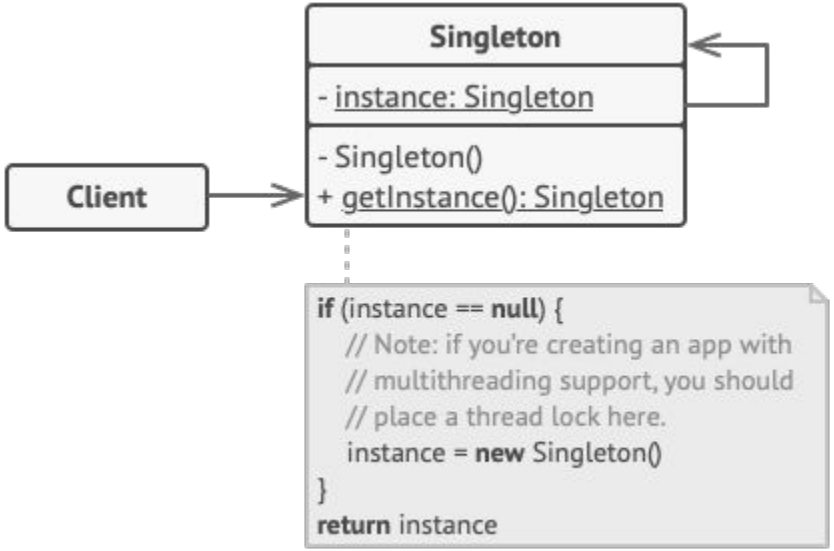
Live Example: Decorator Pattern

- Develop a system in **src/milktea** to print out a milk tea order, where customers are able to order milk tea and optionally add the following toppings on (for an extra cost):
  - Pearls, costing $0.80.
  - Lychee Jelly, costing $0.90.
  - Milk Foam, costing $1.00.

Singleton Pattern

- The Singleton Pattern is a (creational) design pattern that ensures that only one instance of a class exists at any point in time.
  - This may be desired when a class involves some sort of shared resource, and there are issues with multiple people concurrently accessing it.
- The implementation is very simple:
  - Make the class constructor **private**.
  - Make the class store a static instance of itself - this is the **singleton**.
  - Only allow clients to 'instantiate' the object through a static method that returns the instance if it exists (if it doesn't exist, instantiate it first).
  - In Java, the `synchronized` keyword can be added to methods to prevent concurrent access from multiple threads.

Singleton Pattern Implementation



Live Example: Singleton Pattern

- Consider the bank account from Lab 4. What if multiple people try to access the bank account at the same time? Inside **src/heist** are three classes:
  - `BankAccount`: from Lab 4.
  - `BankAccountAccessor`: instances of an access to a bank account to withdraw money a given number of times by given amounts.
  - `BankAccountThreadedAccessor`: threads that will try to concurrently access the bank account.
- Investigate the odd behaviour caused by the race condition, and ensure that only one person can access the bank at a time using the Singleton Pattern.