COMP2511

Tutorial 5

Last Week's Tutorial

- Design Principles
- Introduction to Design Patterns

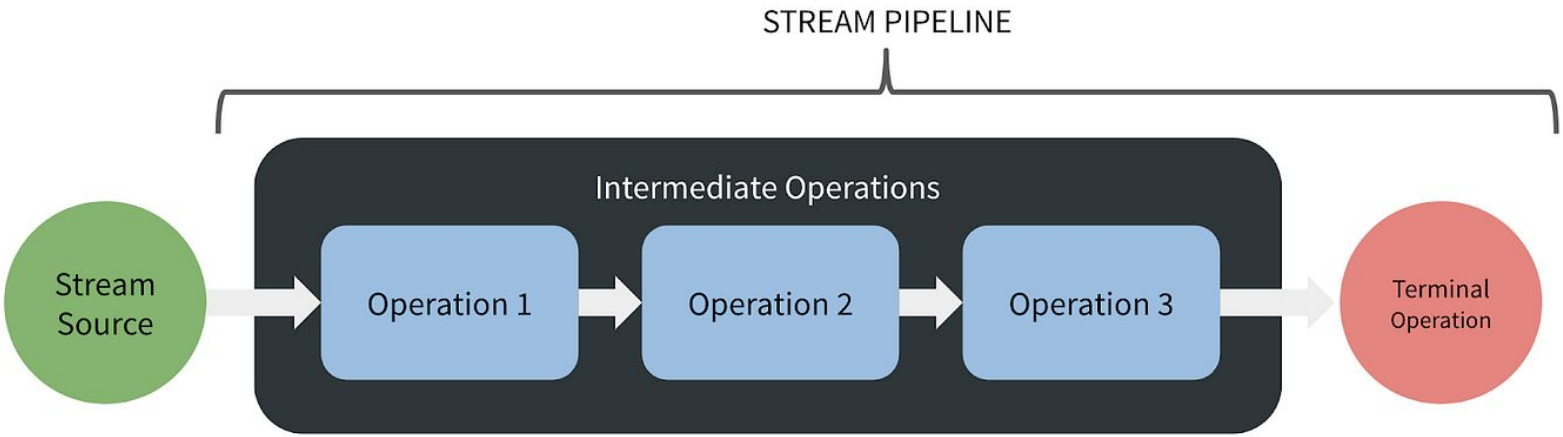
This Week's Tutorial

- Streams
- More Design Patterns
  - Strategy Pattern
  - Observer Pattern

Streams

- Collections in Java (eg. `ArrayList`, `TreeMap`, `TreeSet`, `HashMap`) can be transformed into an *abstracted* sequence of the same elements, referred to as a **stream**.
- Elements in a stream can only be interacted with through methods that provide high-level abstractions to perform operations, such as `map`, `filter`, and `reduce`. These operations can be chained together.
- After all necessary operations have been done, a stream can be converted back into the original type of collection, as required.

Streams


STREAM PIPELINE
Intermediate Operations
Stream Source
Operation 1
Operation 2
Operation 3
Terminal Operation

Streams

- Suppose we have a list of integers and we want to take all of the numbers less than 5, then multiply all remaining numbers by 3. What sequence of stream operations could be used to achieve this?
- `filter`: take only elements that satisfy a specific condition (*predicate*)
  - `.filter(x -> x < 5)`
- `map`: apply a function to all elements in the collection
  - `.map(x -> x * 3)`
- `toList`: convert stream into a list (returned list is **immutable**)

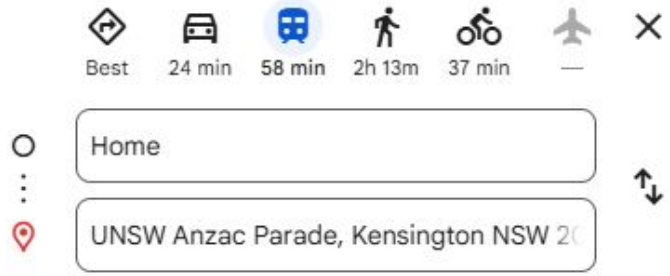
```
List<Integer> l = List.of(1, 2, 3, 4, 5, 6);
l = l.stream().filter(x -> x < 5).map(x -> x * 3).toList();
```

```
[3, 6, 9, 12]
```

Live Example: Streams

- In **src/stream** there is a program that uses standard for-loops to print out a list of strings, and parse integers from a list of strings and print out the new list of parsed integers.
  - Rewrite the first for-loop using the `forEach()` method and a lambda expression.
  - Rewrite the second for-loop using streams and the `map()` method.

Strategy Pattern

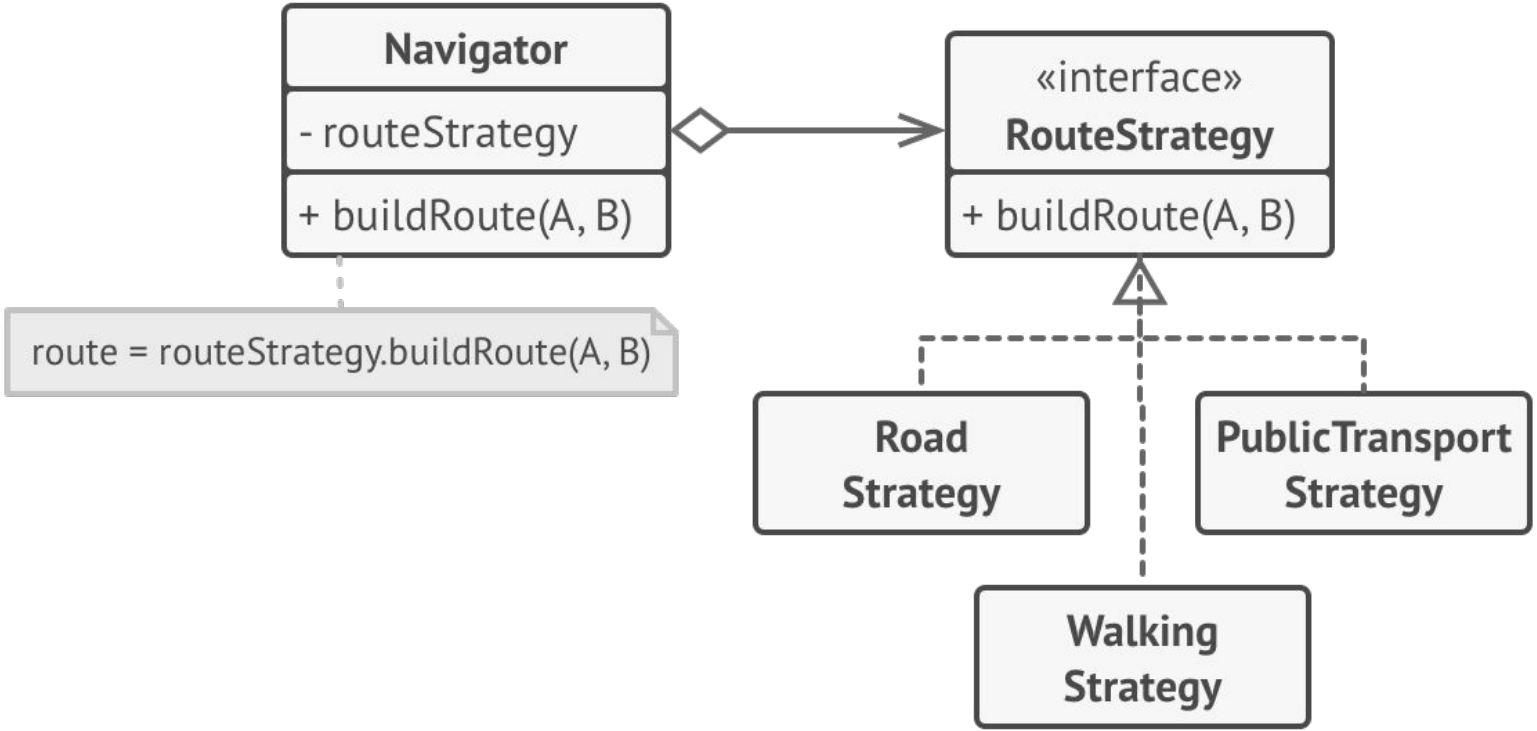


- Consider how you can alternate between different modes of transport while planning a route on Google Maps - you see different routes depending on if you were to walk, drive or take public transport.
    - The specific details of the route, e.g. the distance you have to cover, the turns you have to take, any costs incurred etc. may be **very** different across these modes of transport.
    - Overall though, the common aspect of each of these routes is that they all go the same destination. You can think of them as being “different ways to do the same thing”, and being able to easily switch between them is what the Strategy Pattern is all about.

Strategy Pattern

- The Strategy Pattern is a (behavioural) design pattern where multiple implementations are defined for a specific action, and the class is able to switch between these implementations at **run-time**.
- Rather than putting all of the different implementations in a single class, each implementation is separated into different classes called **strategies**.
    - Each strategy should implement a specific **interface** (or inherit an abstract class) that defines the functionality that will differ across each strategy.
    - The original class will store an **instance** of a particular strategy, and executes instructions by calling the appropriate method(s) defined by each strategy.
    - Switching between strategies is done by changing this instance, e.g. through a setter.
    - The important part is that the functionality of the object will differ according to which strategy is being stored, due to polymorphism.

Strategy Pattern: Implementation (for Routing Example)



Live Example: Strategy Pattern

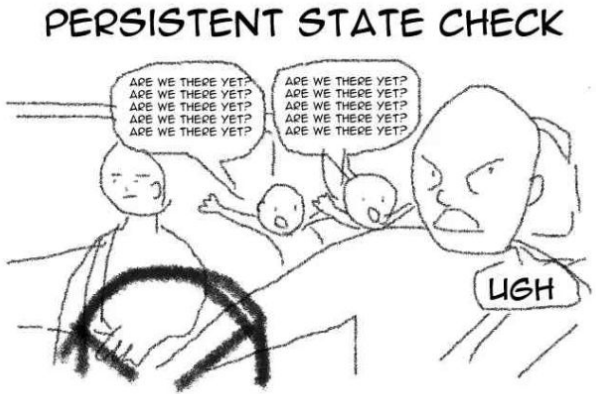
- Inside **src/restaurant** is a restaurant payment system with the following requirements:
  - The restaurant has a menu (a list of meals). Each meal has a name and price.
  - Users can enter their order as a series of meals, and the system returns their cost.
- The prices on meals vary in different circumstances. The restaurant has three different price settings (so far):
  - **Standard** - normal rates
  - **Holiday** - 15% surcharge on all items for all customers
  - **Happy Hour** - registered members get a 40% discount, standard customers get 30%
- Why is the current code bad quality? **Refactor the code using the Strategy Pattern to handle the three settings**, then implement a new **Prize Draw** Strategy - every 100th customer (since the start of the promotion) gets their meal for free!

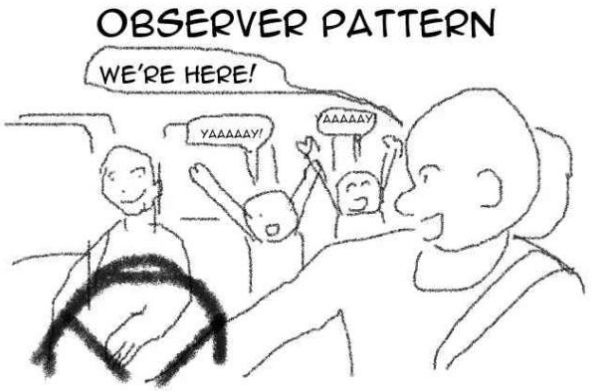
Observer Pattern

- If you were subscribed to a YouTube channel and wanted to watch that channel's videos as soon as they came out, would you rather:
  - Keep refreshing the YouTube channel every 5 seconds?
  - Receive a notification whenever a new video has been uploaded?
- The latter case is more sensible and what happens in reality - when a channel publishes a new video, its subscribers will get notified.
- We can frame this in a particular way: There is an important **event** (a new video being uploaded) that some people are waiting on. As soon as that event happens, everyone waiting will get **notified** and respond to it; this is the idea of the Observer Pattern.

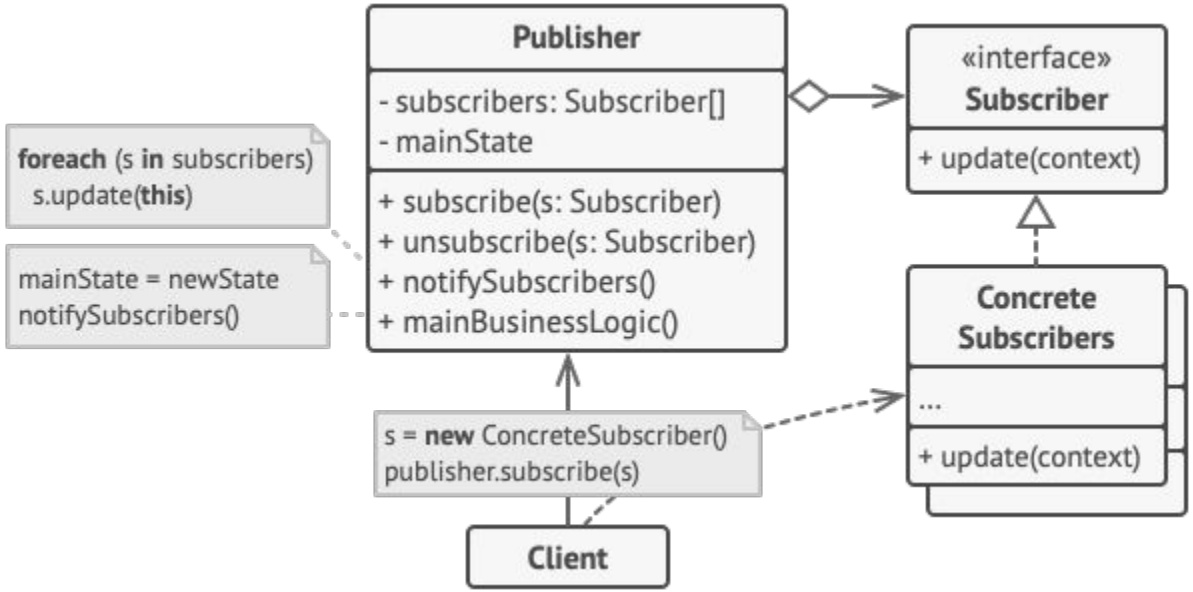
Observer Pattern

- The Observer Pattern is a design pattern where a **subject** maintains a list of **subscribers**, who wait on some event to occur with the subject.
    - Once the event occurs, the subject will **notify** each of its observers to respond to that action.





Observer Pattern: Implementation



Live Example: Observer Pattern

- In **src/youtube**, create a model for the following requirements of a YouTube-like video creating and watching service using the Observer Pattern.
  - A YouTuber has a name, subscribers and videos.
  - A User has a name and can subscribe to any YouTuber.
  - When a YouTuber posts a new video, the video gets added to the ‘Watch Later’ lists of all Users subscribed to that YouTuber.
  - Write a simple test with print statements inside YoutubeTest.java.