Design Patterns: Strategy

Mikael Svahnberg*

2023-09-07

Contents

1 Introduction

- (Freeman & Robson, Chapter "intro")
- Freeman & Robson, Chapter 1: Welcome to Design Patterns
- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*
- Design Principle: *Low Coupling*
- Design Pattern: *Strategy*

*Mikael.Svahnberg@bth.se

2 The Challenge

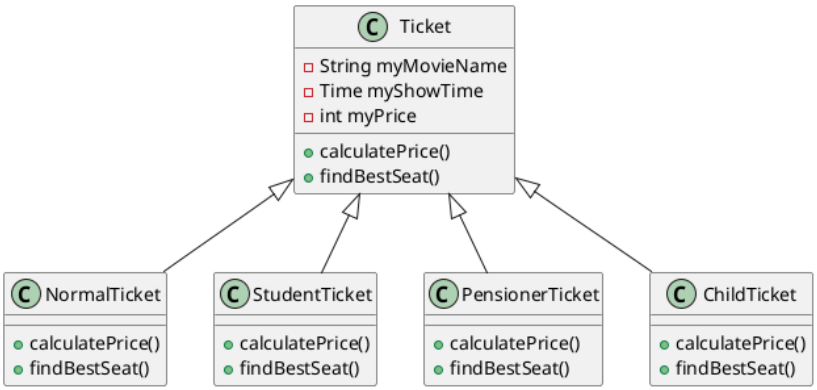
- One type may have many pieces of behaviour that differs.
- If it was only one place, we *could* get away with `switch-case`
- But then we have `findBestSeat()`, and the next behaviour, and the next...

```
enum TicketType {Normal, Student, Pensioner, Child};

int Ticket::calculatePrice(TicketType theType) {
  switch(theType) {
  case Student: return basePrice*0.80; break;
  case Pensioner: return basePrice*0.50; break;
  case Child: return basePrice*0.50; break;
  case Normal:
  default:
    return basePrice;
    break;
  }
}
```

3 The Challenge II

- Next step, an inheritance hierarchy
- Overload all methods that differ, implement separate behaviour.
- What if *some* ticket types, but not all, share the same behaviour?
  - Can we "lift" this behaviour to the base class?
  - Might work if it is only one type of behaviour that is common.
  - Will not work (in Java), if I end up with multiple inheritance.



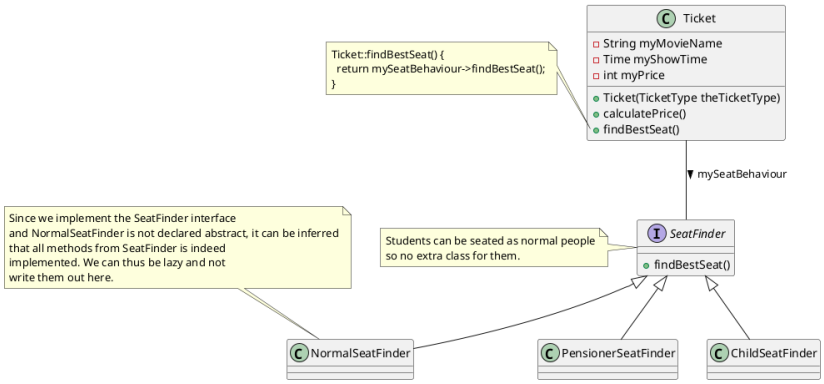
4 High Cohesion and Encapsulation to the rescue!

- Separate similar behaviour to a new class:
  - it will know all there is to know about that particular behaviour.
  - it will *encapsulate* this behaviour
  - it can be modified and extended separately from the rest of the application.

  Identify the aspects of your application that vary and separate them from what stays the same.

- *Encapsulate* the behaviour that varies and hide it from others.
  - Only expose a common API that can be used on all types of behaviour.

5 Separate the changing behaviour



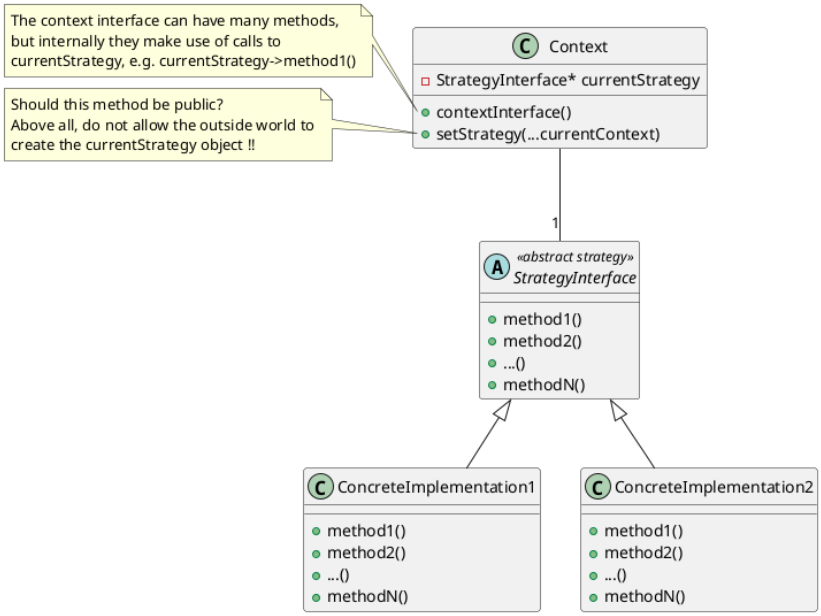
6 Summary: Strategy Pattern

- This was a very simple case, with only one method
- Likely, we have several methods that are all part of the same type of behaviour.
- Only one `findBestSeat` behaviour is active for one specific `ticket:Ticket` at any specific point in time.
  - We call this a *strategy*, in this case a `SeatFinderStrategy`
- Design Principle: *Program to an Interface, not an implementation*
  - The Ticket class and the rest of the world mostly only knows about the interface `SeatFinderStrategy`

  - The interface is stable and define the methods common for all seat finding strategies.
  - The actual implementation of this interface, this API, will vary in each sub-class.
  - The *interface* defines `findBestSeat()` , not `findSeatCloseToParent()` or `findEasyToAccessSeat()`.

- We can do the same for `calculatePrice()`:
  - `interface PriceCalculatorStrategy` , and sub-classes for each type of price calculation
- One instance of *Strategy Pattern* for every type of behaviour that varies.

7 Generic form for Stategy Pattern



8 Inheritance or Composition

- Assume we would only use inheritance:
  - As the number of behaviour types grow, we create new sub-classes
  - As the number of variants of each behaviour type grow, we create new sub-classes
  - $\sum$ The number of combinations grow exponentially
- The *strategy pattern* refactors the problem into separate inheritance hierarchies
  - one inheritance hierarchy for each behaviour type

  - one class for each strategy, or behaviour variant
- The Context class remains simple
- The Context class is now in charge of managing all the strategies
- The rest of the program no longer need to know any of this. We have *encapsulated* the strategies.
- Design Principle: *Favor composition over inheritance*
  - a more maintainable design
  - easier to extend with new behaviour types
  - easier to extend with new strategies

9 Design Patterns and Pattern Libraries

- Design patterns are reusable solutions to known problems
  - With known consequences
  - "encoded experience"
  - Codified in a structured format
  - named
- There is nothing that *requires* you to use design patterns; they are a convenience.
- Design Patterns offer a *shared vocabularly*
  - When you see "strategy" you know what to look for, and how it works

10 Summary

- Design Pattern: *Strategy*
- Design Principles: *High Cohesion* and *Encapsulation*
  - Separate similar behaviour into a new class.
  - Encapsulate what varies under a common API.
- Design Principle: *Program to an Interface, not an Implementation*
  - Program to a supertype, the common API.
- Design Principle: *Composition over Inheritance*
  - Avoid deep and intricate inheritance hierarchies where behaviour can not easily be extended or changed.
- Design Principle: *Low Coupling*

- The rest of the program do not need to know how to create a specific sub-class with the exact right combination of different behaviour strategies.
- Most of the time, the Context class can also forget about which is the current strategy, and only use the common API.
- We may not get *fewer* connections between classes, but they are more *loosely* coupled.

11 Next Lecture: Responsibility Driven Design

- Barnes & Kölling Chapter 8, Designing Classes
- Barnes & Kölling Chapter 9, Well-Behaved Objects
- Book tip: Robert C. Martin, *Clean Code*, Pearson Education, 2009.
- Book tip: Steve McConnell, *Code Complete*, Microsoft Press, 2004.
- Design Principle: *Low Coupling*
- Design Principle: *Encapsulation*
- Design Principle: *Localising Change (High Cohesion)*