School of Computing
FACULTY OF ENGINEERING & PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

COMP5911M
Advanced Software Engineering

8: More Design Patterns

Nick Efford

https://comp5911m.info

Last Time

UNIVERSITY OF LEEDS

We introduced the design pattern as

- A named, well-understood solution to a common object-oriented design problem
- A tried-and-tested approach
- A formalisation of experience-based knowledge
- A way for novices to learn to be experts, by example

Last Time

UNIVERSITY OF LEEDS

- We explored a hypothetical scenario where the **Strategy** pattern provides a good solution
- We saw an example of how Strategy is used in practice, for component layout in Java's Swing UI framework
- We introduced the **Observer** pattern and saw how it forms the basis for the event handling model in Swing

Objectives

UNIVERSITY OF LEEDS

- To explore some more design patterns
- To see examples of where these patterns are used

Object Creation

UNIVERSITY OF LEEDS

Our game might have code like this in lots of places:

```
GameCharacter character;
...
if (characterType.equals("soldier")) {
  character = new Soldier();
}
else if (characterType.equals("bandit")) {
  character = new Bandit();
}
else if (characterType.equals("farmer")) {
  character = new Farmer();
}
```

What is the problem here?

Design Principle

UNIVERSITY OF LEEDS

Where possible, classes should be **open for extension but closed for modification**.

In other words, it should be possible to extend our design to incorporate new behaviour without having to modify existing code…

Simple Factories

UNIVERSITY OF LEEDS

```
public class CharacterFactory {
  public GameCharacter makeCharacter(String type) {
    if (type.equals("soldier")) {
      return new Soldier();
    }
    else if (type.equals("bandit")) {
      return new Bandit();
    }
    ...
  }
}
```

we are still using `new` to instantiate concrete classes, but have localized this to the factory class…

Using The Factory

UNIVERSITY OF LEEDS

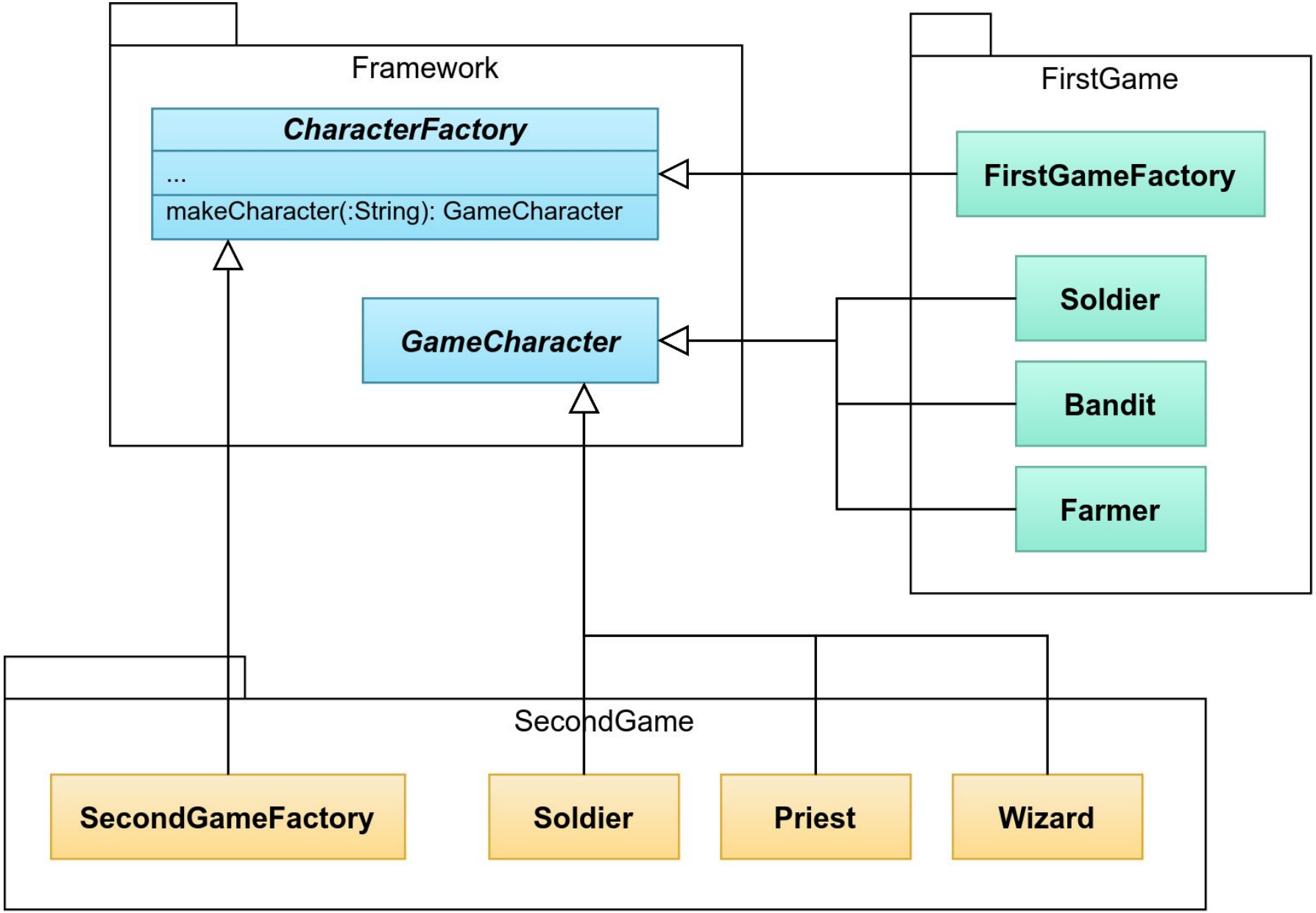
```
public class GameEngine {
  private CharacterFactory factory;

  public GameEngine(CharacterFactory factory) {
    this.factory = factory;
  }

  public void setUpGame() {
    ...
    GameCharacter chr = factory.makeCharacter(chrType);
  }
}
```

no explicit references here to the actual class being instantiated!

We Need To Go Further…

UNIVERSITY OF LEEDS

- If we are using this same approach across multiple games, we might need different factories – each knowing how to create a *different* set of `GameCharacter` objects
- It therefore makes sense to insulate our game code from dependency on a specific factory
- … which means we need another interface (or abstract class) to hide our factory classes


Framework
CharacterFactory
...
makeCharacter(:String): GameCharacter
GameCharacter
FirstGame
FirstGameFactory
Soldier
Bandit
Farmer
SecondGame
SecondGameFactory
Soldier
Priest
Wizard

Using `GameEngine`

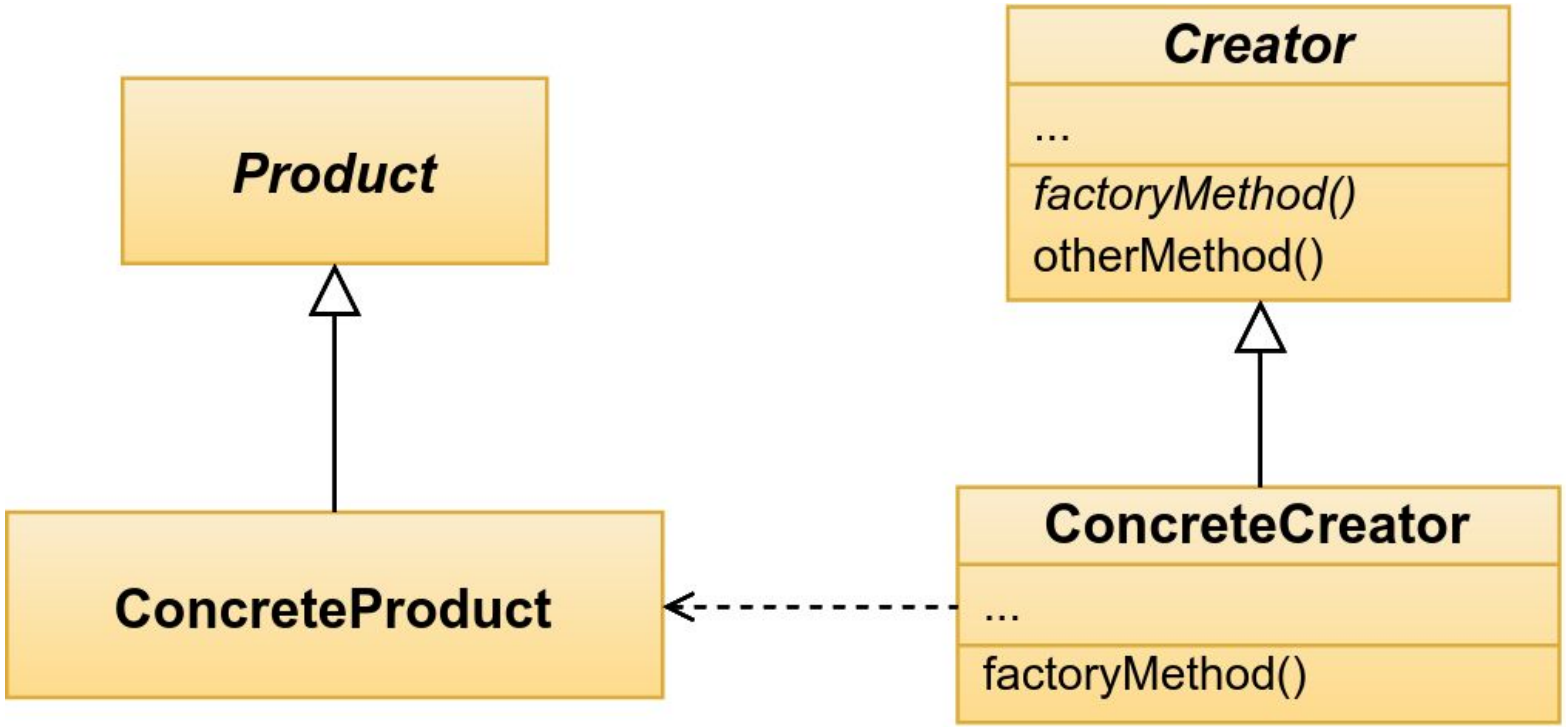
UNIVERSITY OF LEEDS

```
public class FirstGame {
  private GameEngine engine;

  public FirstGame() {
    CharacterFactory factory = new FirstGameFactory();
    engine = new GameEngine(factory);
    ...
  }
  ...
}
```

same `GameEngine` can be used in all games; we just need to plug in the appropriate factory object…

The Factory Method Pattern

UNIVERSITY OF LEEDS



More Patterns

UNIVERSITY OF LEEDS

- Decorator
- Façade
- Composite

Decorator

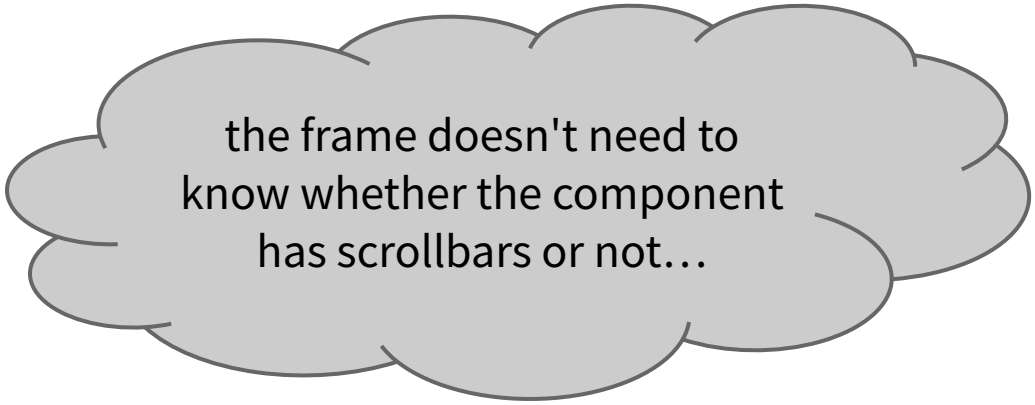
UNIVERSITY OF LEEDS

- **Delegation pattern**: a class has methods that forward calls to identical methods supplied by another class
- Behaviour of an object is modified dynamically by ‘wrapping other objects around it’
- Decorator and recipient of decoration have same interface, so decoration is transparent

Example: Java Swing

UNIVERSITY OF LEEDS

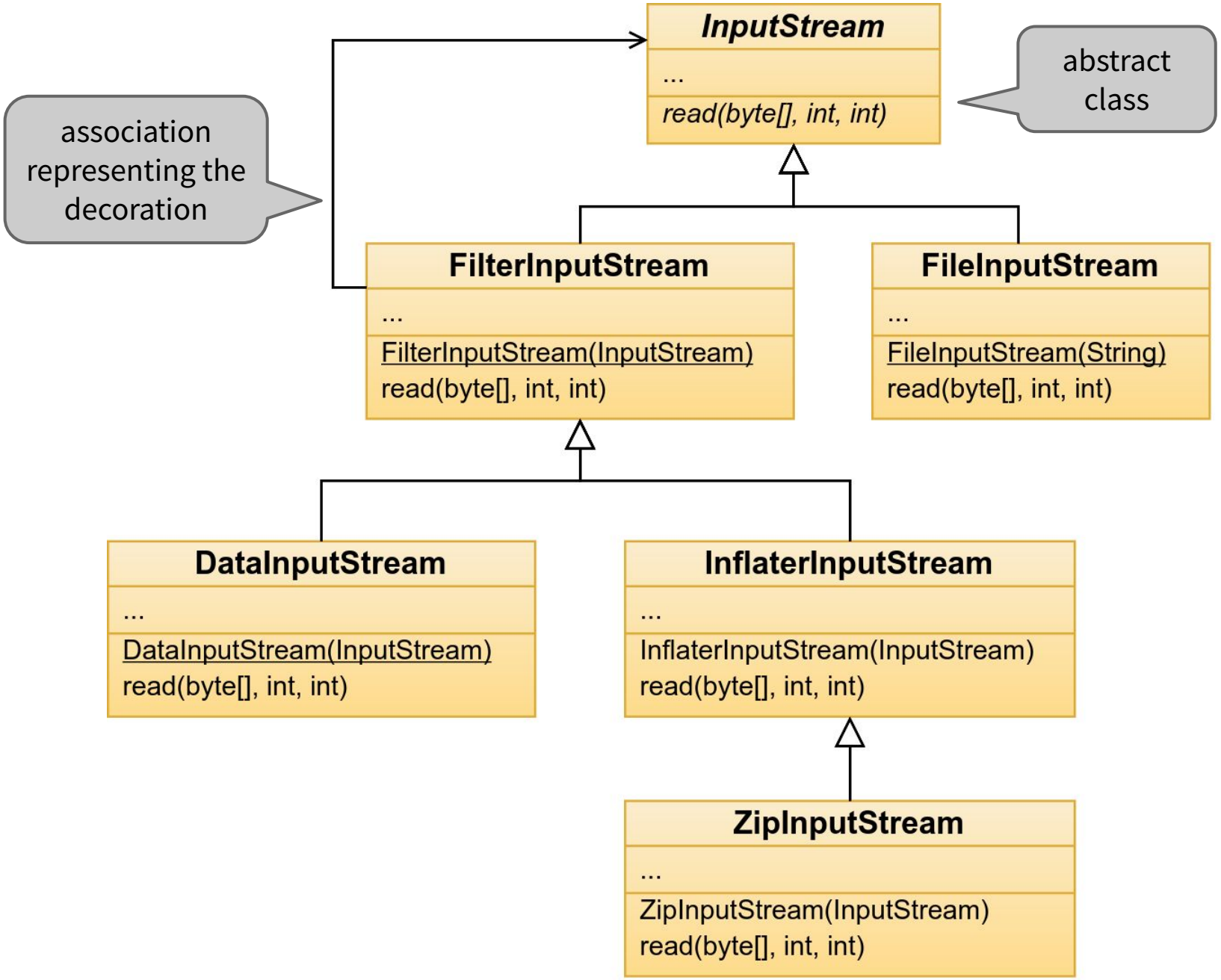
```
JLabel view = new JLabel(new ImageIcon(image));
JFrame frame = new JFrame(filename);

// no decoration
frame.add(view);

// scrollbars as decoration
frame.add(new JScrollPane(view));
```



Example: Java I/O

UNIVERSITY OF LEEDS

- I/O streams can vary in many ways
  - Binary or text-based
  - Buffered or unbuffered
  - Compressed or uncompressed…
- We would like to support this variation
- BUT having a class for every possible combination of stream attributes ⇒ *huge* library of I/O classes!


InputStream
...
read(byte[], int, int)
abstract class
association representing the decoration
FilterInputStream
...
FilterInputStream(InputStream)
read(byte[], int, int)
FileInputStream
...
FileInputStream(String)
read(byte[], int, int)
DataInputStream
...
DataInputStream(InputStream)
read(byte[], int, int)
InflaterInputStream
...
InflaterInputStream(InputStream)
read(byte[], int, int)
ZipInputStream
...
ZipInputStream(InputStream)
read(byte[], int, int)

Example: Java I/O

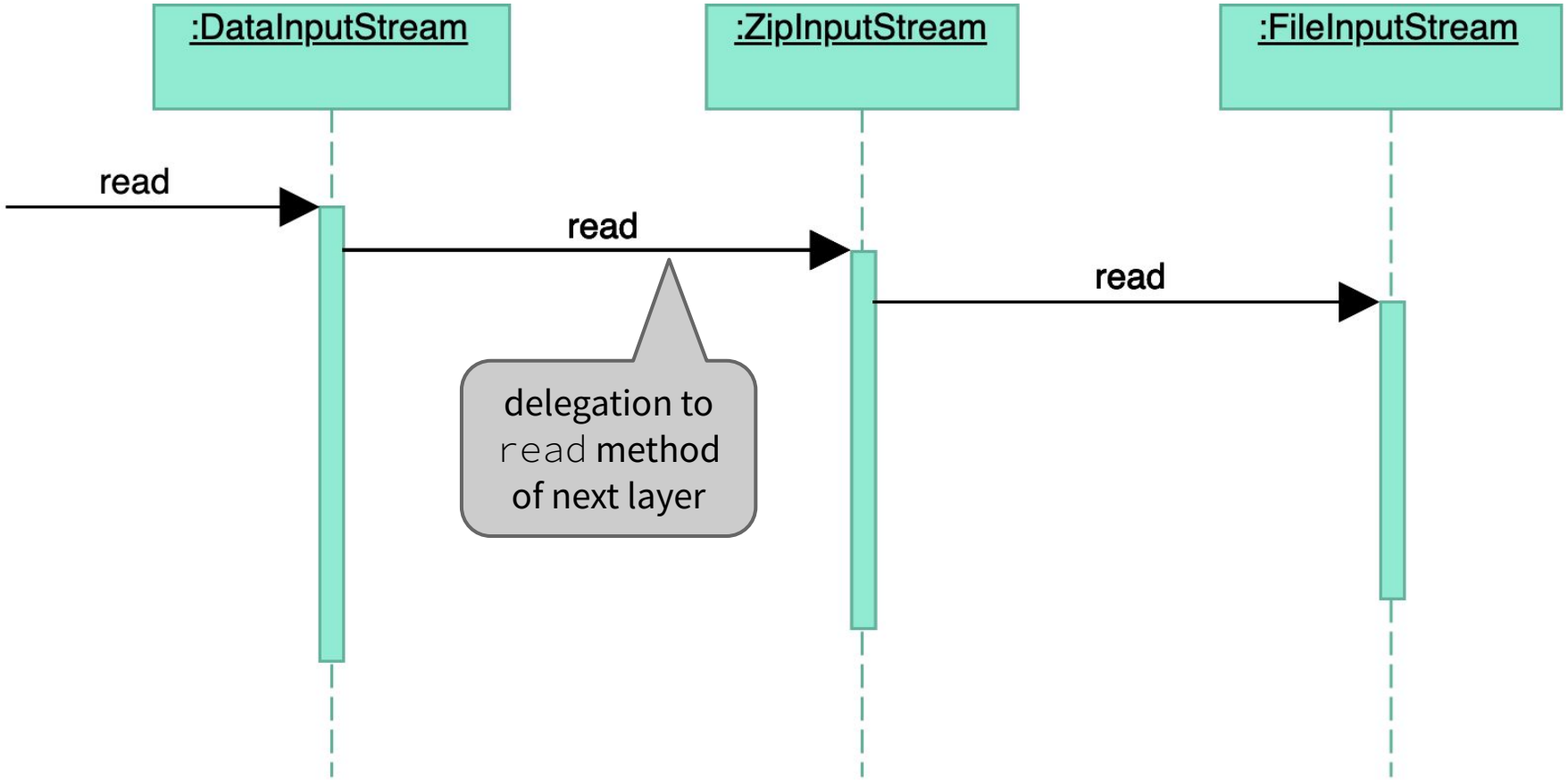
UNIVERSITY OF LEEDS

```
DataInputStream in =
    new DataInputStream(
        new ZipInputStream(
            new FileInputStream("foo.zip")
        )
    );
...
byte[] data = new byte[NUM_BYTES];
in.read(data, 0, NUM_BYTES);
```

what happens when `read` method is called?

Example: Java I/O

UNIVERSITY OF LEEDS



(This is a UML **sequence diagram**)

Façade

UNIVERSITY OF LEEDS

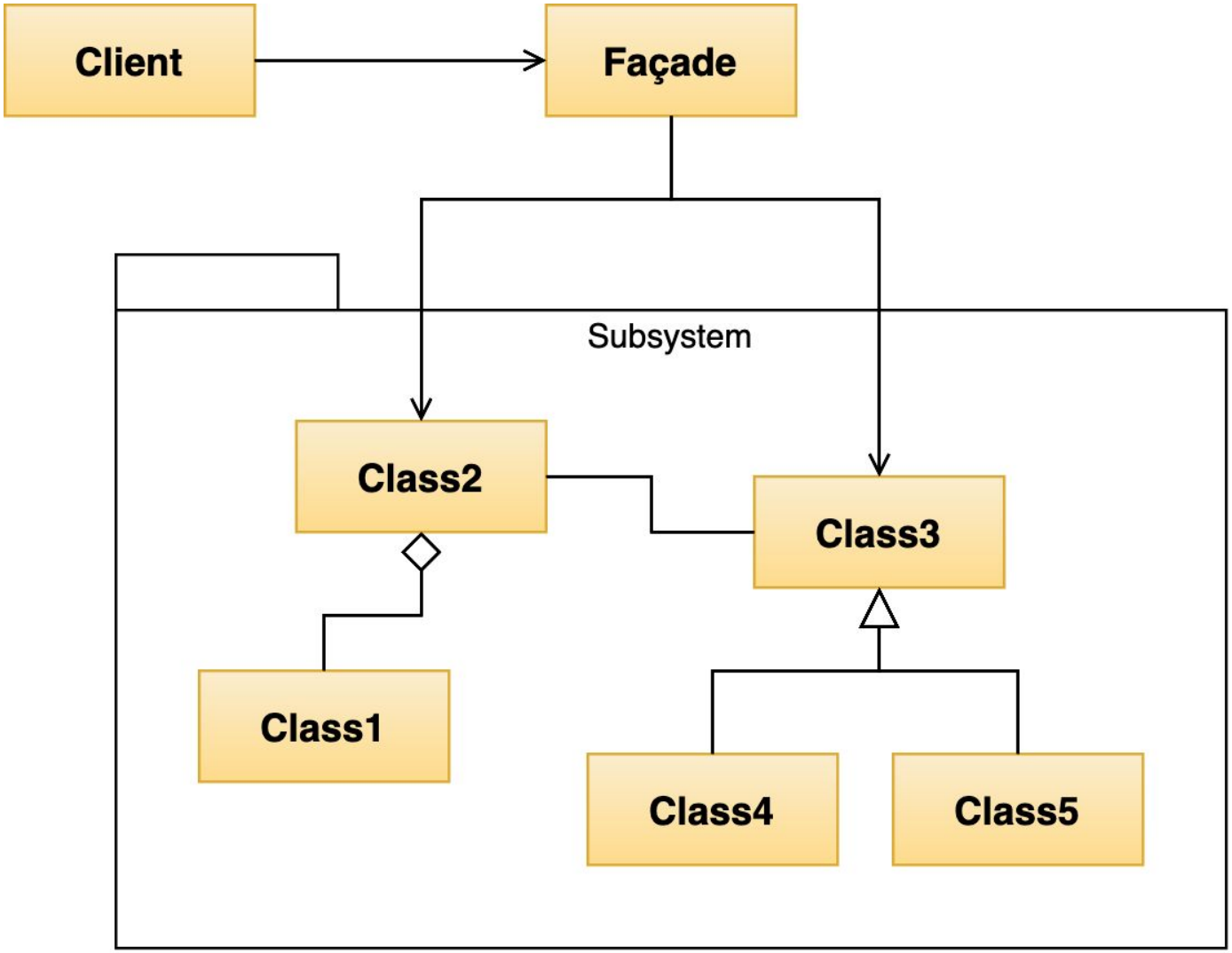
Problem

- Complex subsystem (multiple classes)
- Subsystem implementation could change
- Clients of subsystem need a simplified view of subsystem and a coherent, stable entry point

Solution

- Define a façade class, providing a new & simpler interface
- Façade methods expose desired portion of subsystem functionality to clients, by delegation
- Subsystem classes know nothing about façade, and client need know nothing about subsystem classes

Façade


Client
Façade
Subsystem
Class2
Class3
Class1
Class4
Class5

Composite

UNIVERSITY OF LEEDS

Deals with problems in which

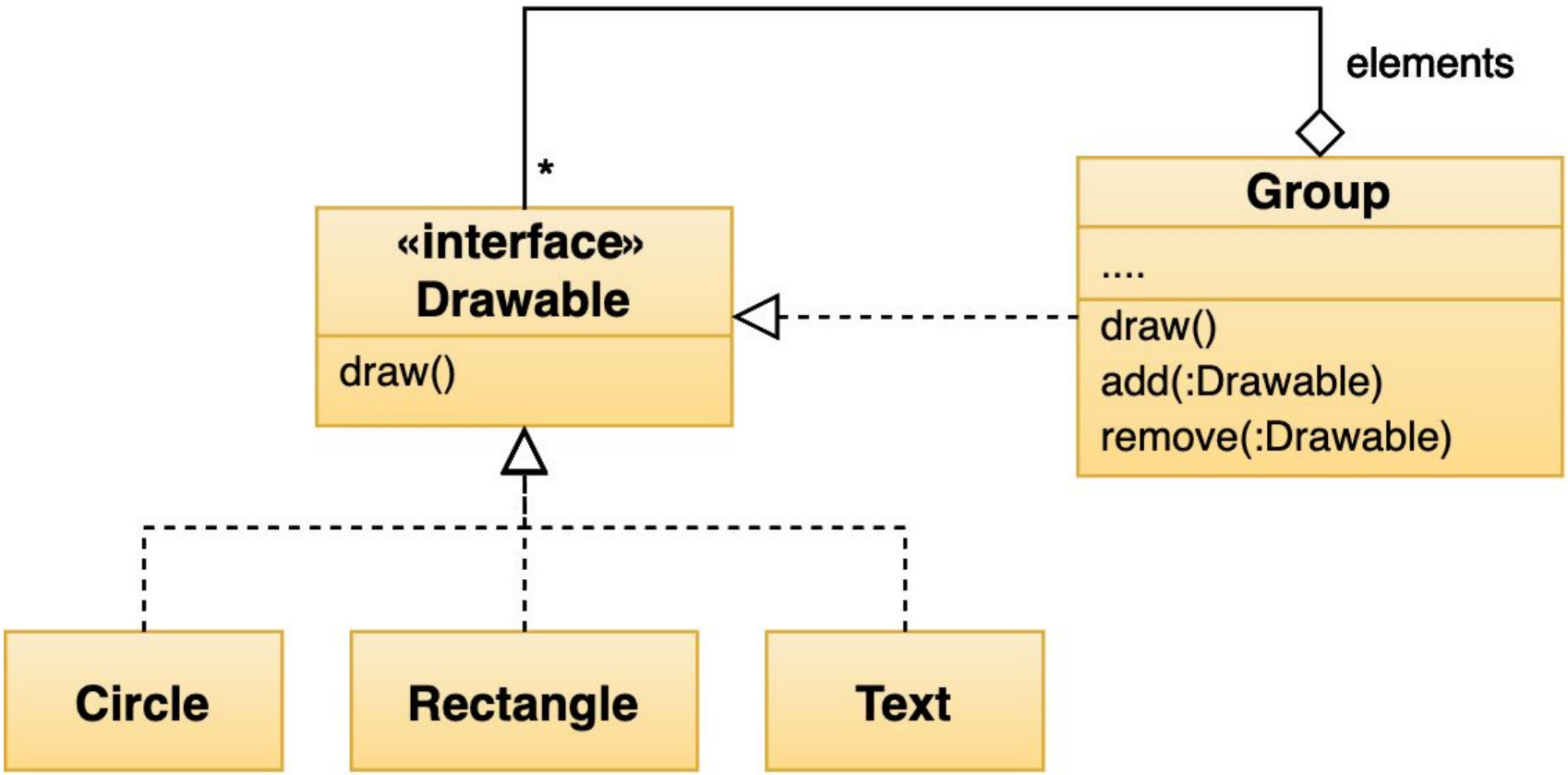
- Objects can be grouped together
- Groups can contain other groups (tree-like hierarchy)

Requires

- One or more **leaf** classes
- A **composite** class, representing a collection of leaf and other composite objects
- An interface implemented by, or abstract superclass of, leaf and composite classes

Composite Example

UNIVERSITY OF LEEDS



Summary

UNIVERSITY OF LEEDS

We have

- Discussed how we can decouple code from specific knowledge about the objects it is using via the **Factory Method** pattern
- Seen how **Decorator** allows us to add functionality to objects in a transparent way
- Noted that **Façade** gives us a stable, simplified view of a complex and changeable subsystem
- Seen that **Composite** gives us a way of managing hierarchical groups of objects

Follow-Up / Further Reading

UNIVERSITY OF LEEDS

- Gamma et al, *Design Patterns: Elements of Reusable Software*
- Refactoring Guru's Catalog of Design Patterns
- Example code from this lecture