Electronics and
Computer Science

University of
Southampton

Design Patterns

COMP6226: Software Modelling Tools and Techniques for Critical Systems

Dr A. Rezazadeh (Reza)
Email: ra3@ecs.soton.ac.uk or ar4k06@soton.ac.uk

November 24

Overview

- Why Design Patterns?
- What is a Pattern?
- Different types of Patterns
- Example patterns
- Key points

use in course work(great)

Why Design Patterns?

- Describes a problem which frequently occurs in our environment, and then describes the core of the solution to the problem
  - Software development is repetitive: Quite often, different programmers have to solve the similar problems.
- Experienced programmers have compared notes and discovered that they arrived at common solutions to the same problems
- Over time, these common solutions, have been systematically documented as the best know approach to solving a given problem

Why Design Patterns?

- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

What is a Design Pattern?

- Design patterns describe a solution to common problems found in the design of systems.
- A good pattern should
  - Be as general as possible
  - Contain a solution that has been proven to effectively solve the problem in the indicated context.
- In order to benefit you must know:
  - The problem you are facing
  - Know design patterns themselves
  - Key is to understand the relationship between classes and how to allocate responsibilities to them

why we need pattern?

Essential Elements of a Design Pattern

There are four essential elements of a design pattern:

- **Pattern name**
- **Problem description**

  This describes when the pattern can be applied: to what kind of problem, and what pre-conditions need to be met
- **Solution description**
  - An abstract description of the solution in terms of a set of classes with particular functionalities and interrelationships
- **Consequences**
  - presenting the results and tradeoffs of using the pattern, helping you decide the best choice from among several promising alternatives.

Different Types of Patterns

Creational patterns

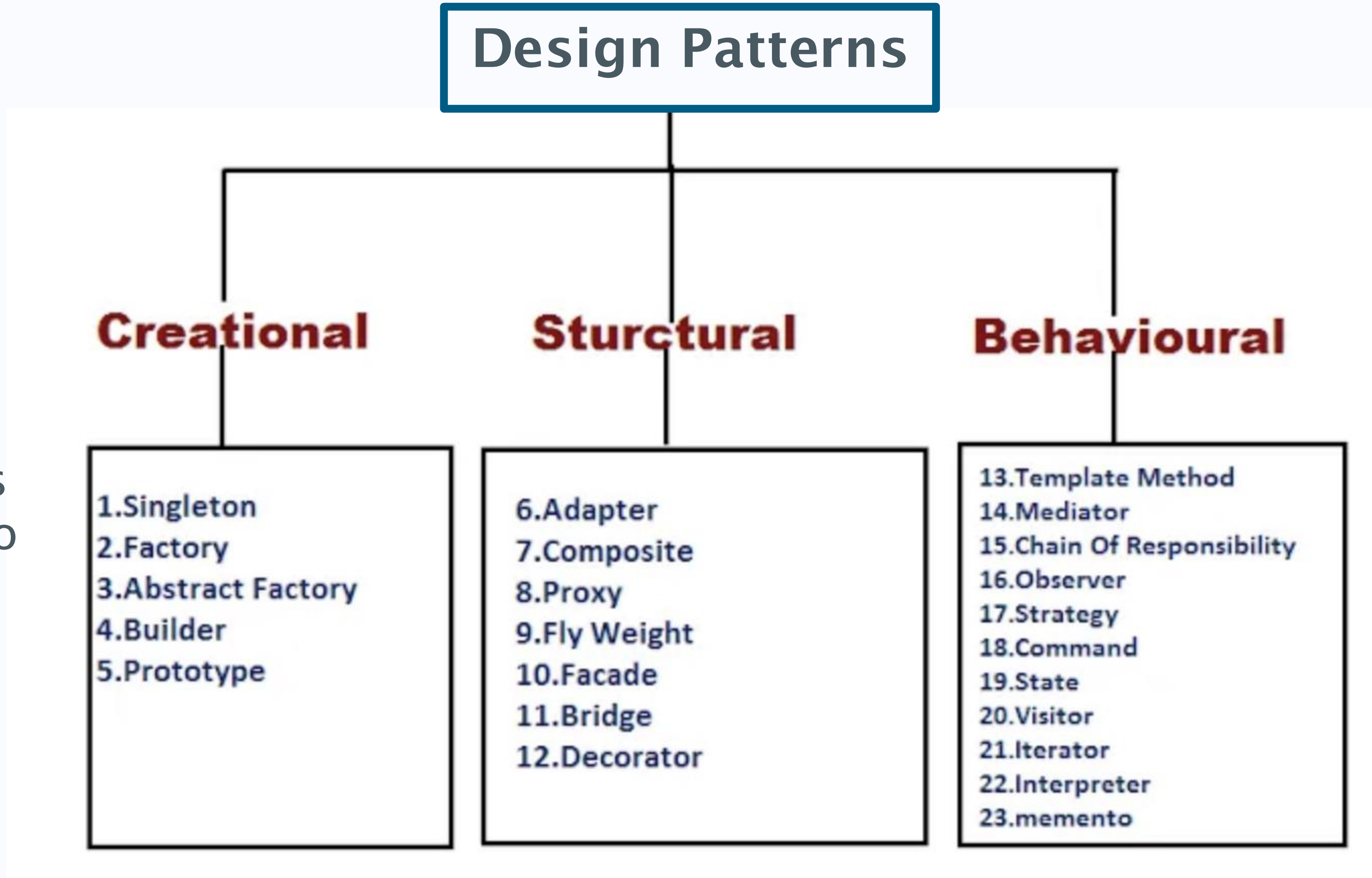
The process of object construction and the instantiation process

Structural Patterns

Composition of classes/objects: how classes and objects are composed to form larger structures

Behavioural Patterns

The way classes and objects interact/patterns of communication between classes



Design Patterns

Creational

1.Singleton
2.Factory
3.Abstract Factory
4.Builder
5.Prototype

Sturctural

6.Adapter
7.Composite
8.Proxy
9.Fly Weight
10.Facade
11.Bridge
12.Decorator

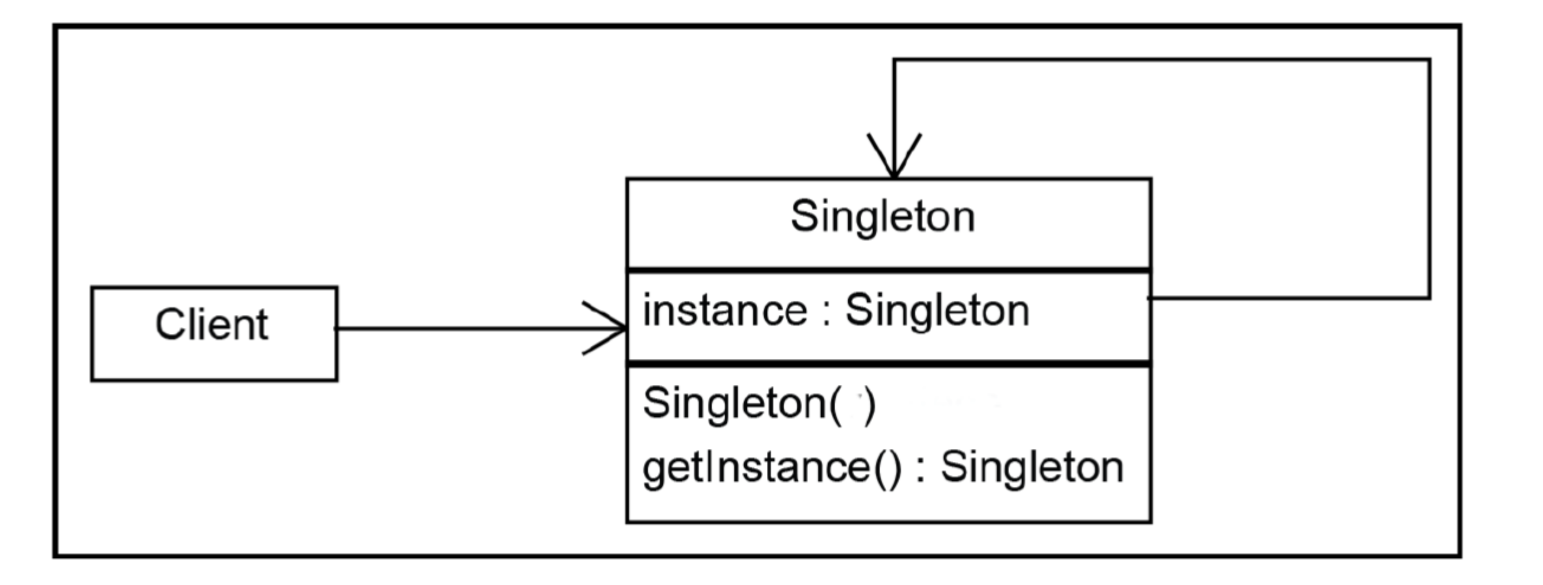
Behavioural

13.Template Method
14.Mediator
15.Chain Of Responsibility
16.Observer
17.Strategy
18.Command
19.State
20.Visitor
21.Iterator
22.Interpreter
23.memento

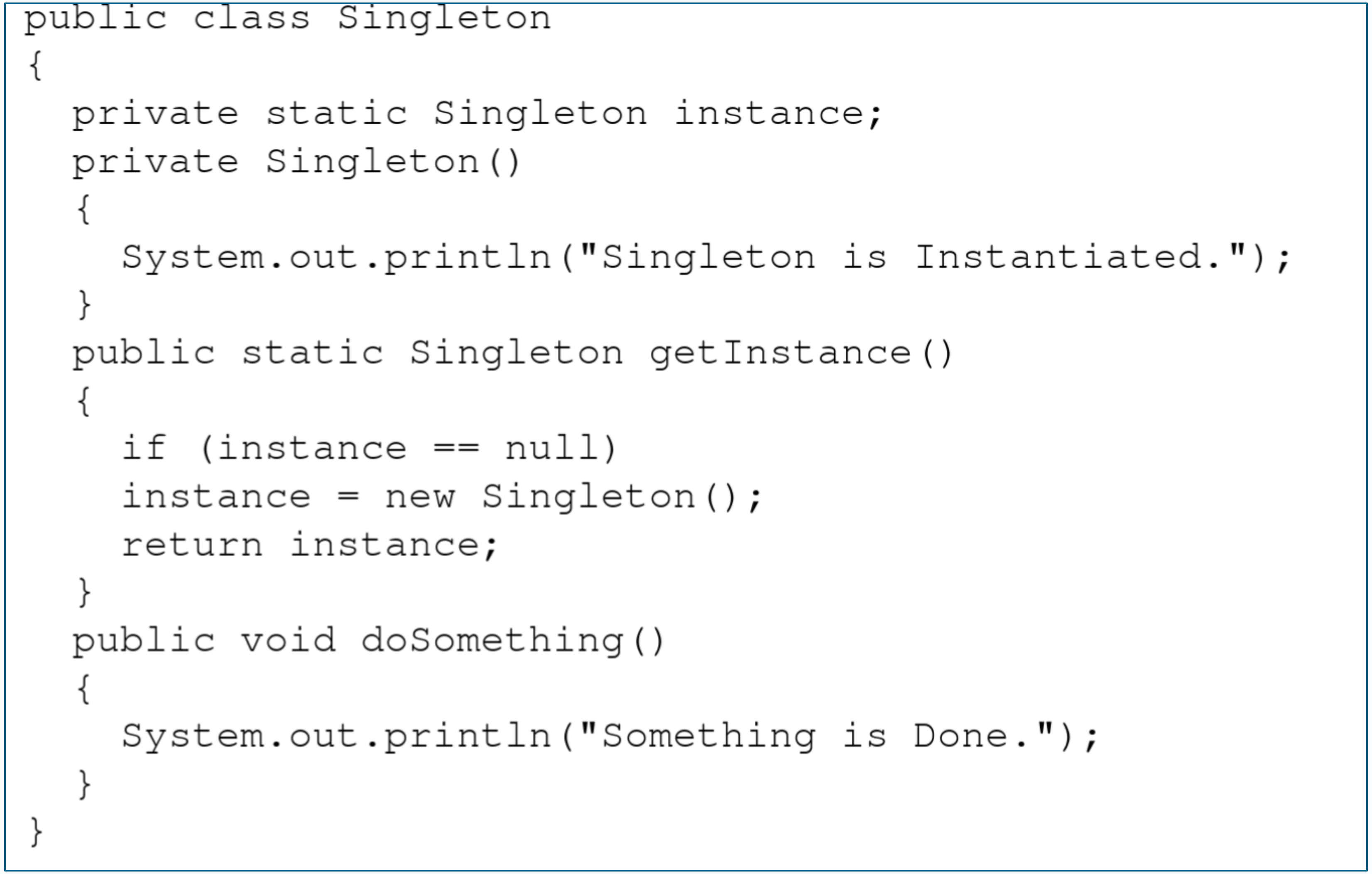
Singleton pattern

why we need it (admin of app)

- The **singleton** pattern is used to ensure that only a single instance of an object can be created.
- It also provide a global access to that instance



Java Code for Singleton pattern



```
public class Singleton
{
  private static Singleton instance;
  private Singleton()
  {
    System.out.println("Singleton is Instantiated.");
  }
  public static Singleton getInstance()
  {
    if (instance == null)
    instance = new Singleton();
    return instance;
  }
  public void doSomething()
  {
    System.out.println("Something is Done.");
  }
}
```

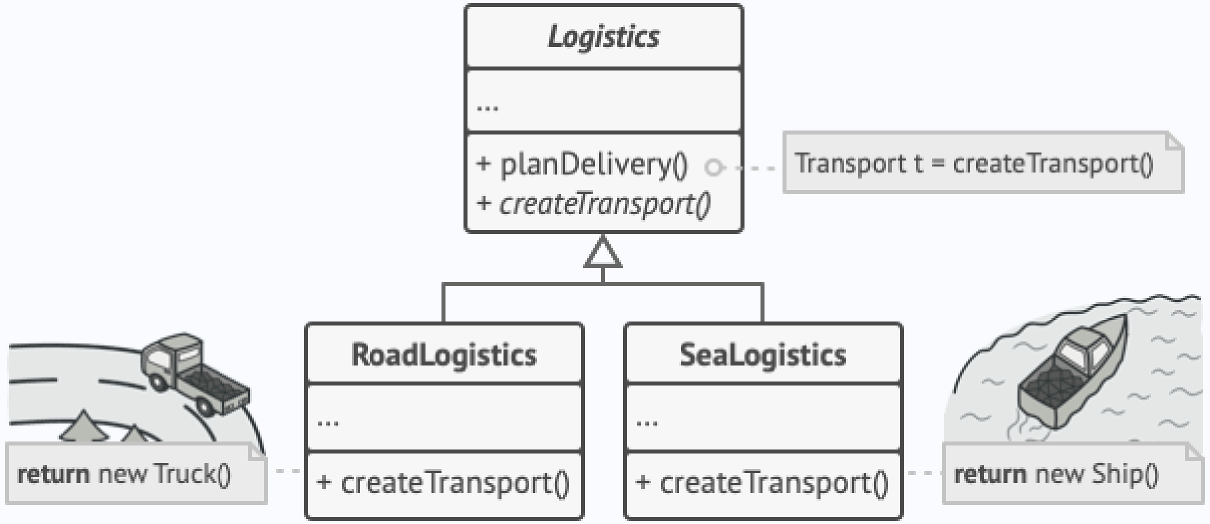
In a multi-threaded application, you should synchronize `getInstance()`

Factory method pattern

- **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
- Problem: Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the `Truck` class.
- After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate **sea** logistics into the app.
  - At present, most of your code is coupled to the `Truck` class. Adding `Ships` into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

Factory method pattern – **Solution**

- The Factory Method pattern suggests that you replace direct object construction calls (using the **`new`** operator) with calls to a special **`factory`** method.
- Don’t worry: the objects are still created via the **`new`** operator, but it’s being called from within the factory method.
- Objects returned by a factory method are often referred to as **products**.



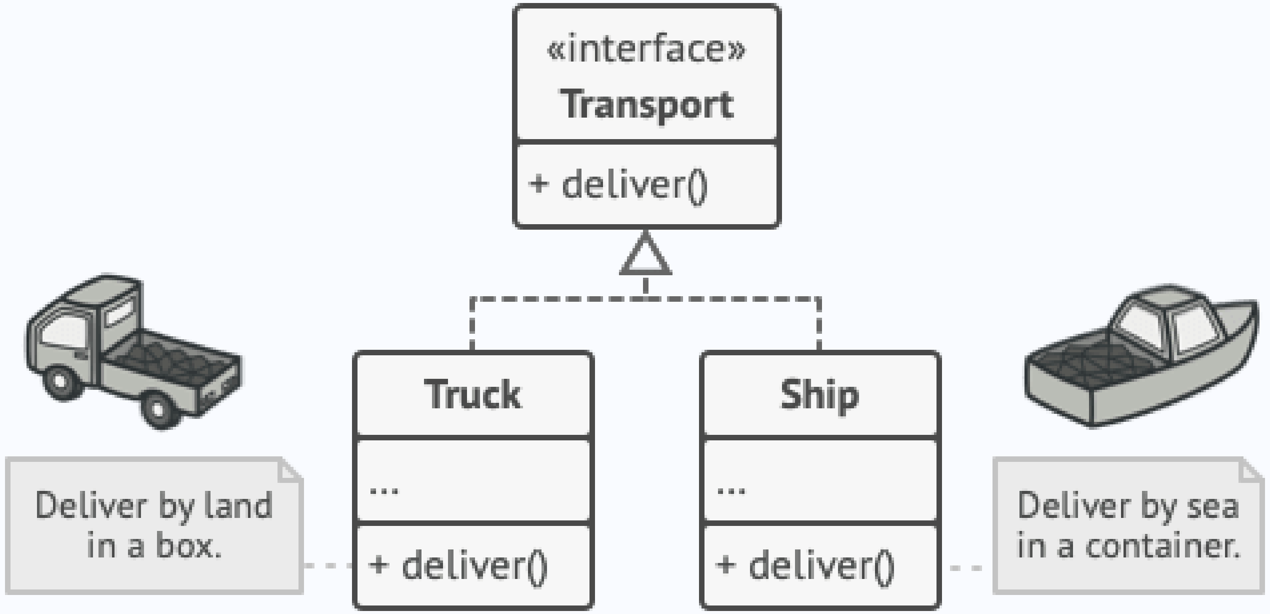
https://refactoring.guru/design-patterns/factory-method

Factory method pattern – Limitations

- At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another.
- However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.
- There’s a slight **limitation** though: subclasses may return different types of products only if these products have a common base class or interface.
- Also, the factory method in the base class should have its return type declared as this interface.

Factory method pattern – Interface impemenation

- Both `Truck` and `Ship` classes should implement the `Transport` interface, which declares a method called `deliver`. Each class implements this method differently: trucks deliver cargo by land, and ships deliver cargo by sea. The factory method in the `RoadLogistics` class returns truck objects, whereas the factory method in the `SeaLogistics` class returns ships.



Adapter Pattern (Structural)

- The adapter design pattern (often referred to as the wrapper pattern or simply a wrapper)
  - translates one interface for a class into a compatible interface.
- An adapter allows classes to work together that normally could not because of *incompatible* interfaces, by providing its interface to clients while using the original interface.
- The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small.

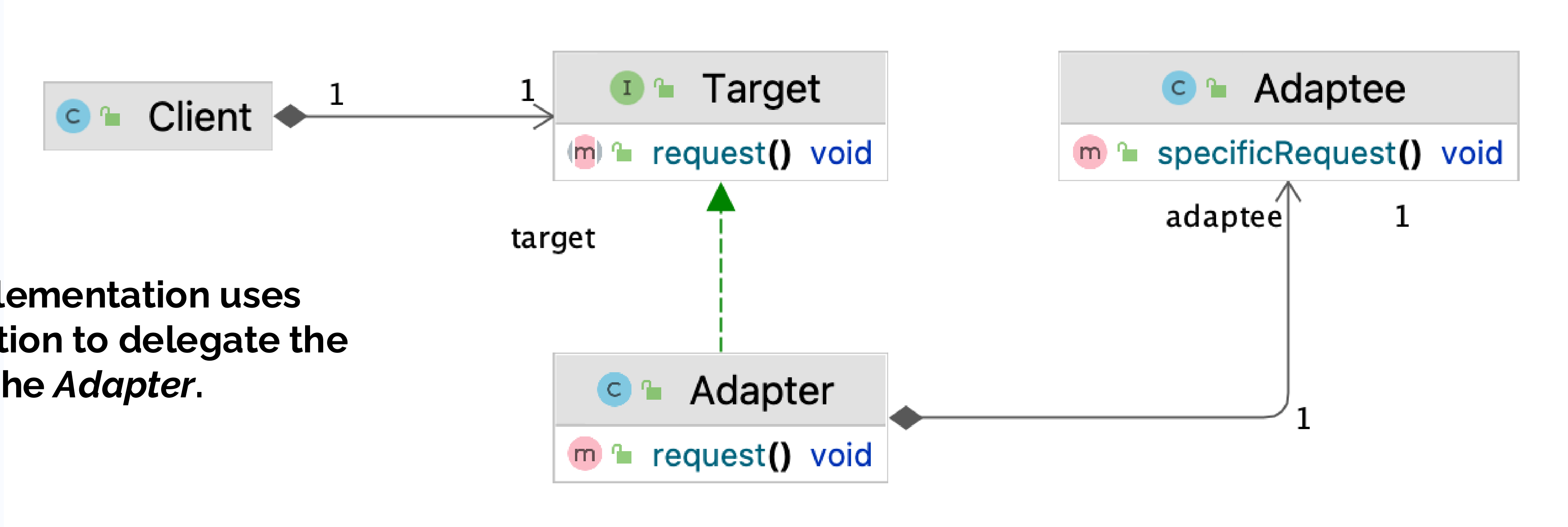
Adapter Pattern – Cont.

- When an existing class performs the services that a client needs but has different method names. In this situation, you can apply the **Adapter** pattern.
  - One class has methods which do the desired thing, but another class needs to call methods with different names

- The adapter pattern can be implemented in two different ways
  - Using **composition**
  - Using *inheritance*

Object Adapter

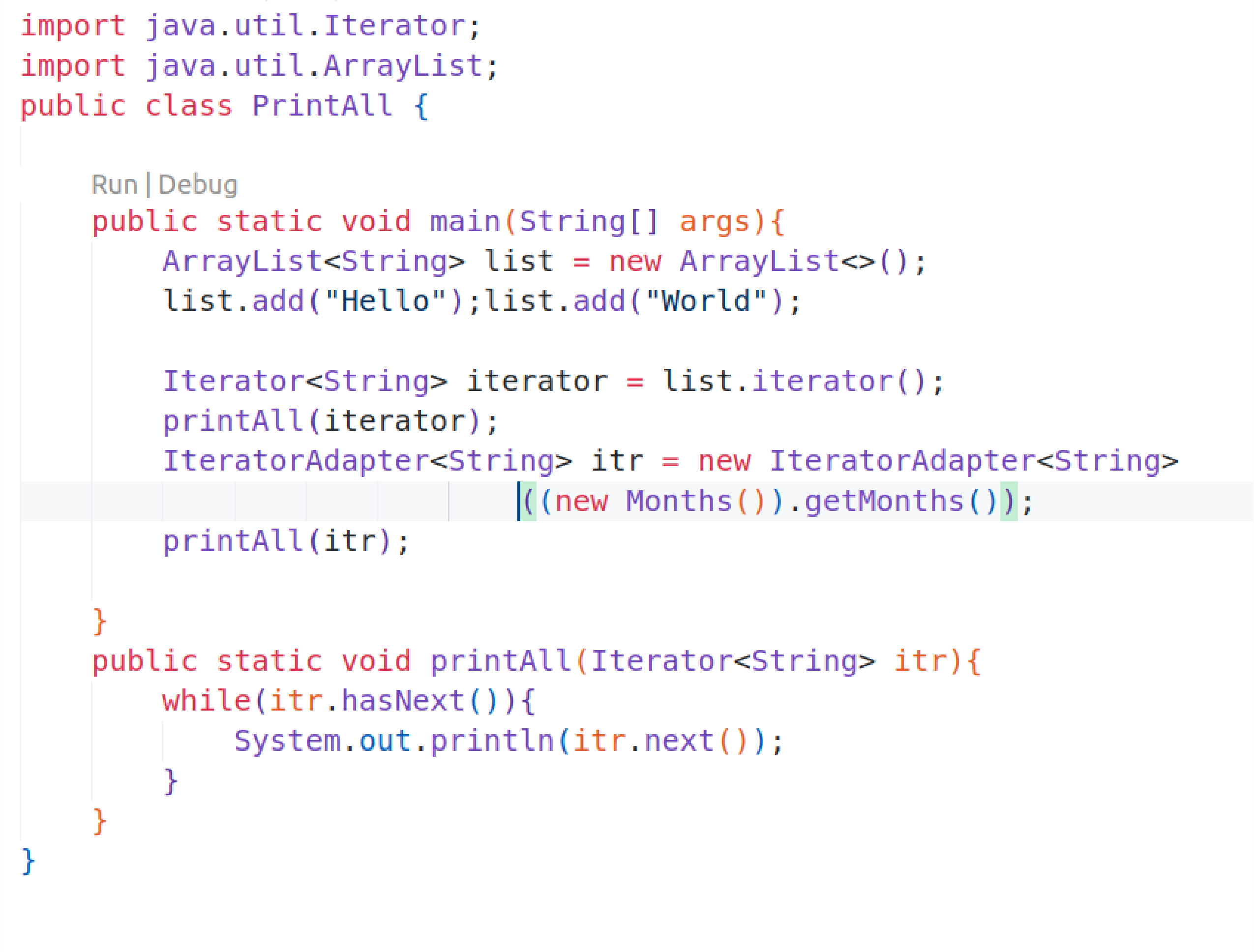
- UML and java code taken from https://www.baeldung.com/



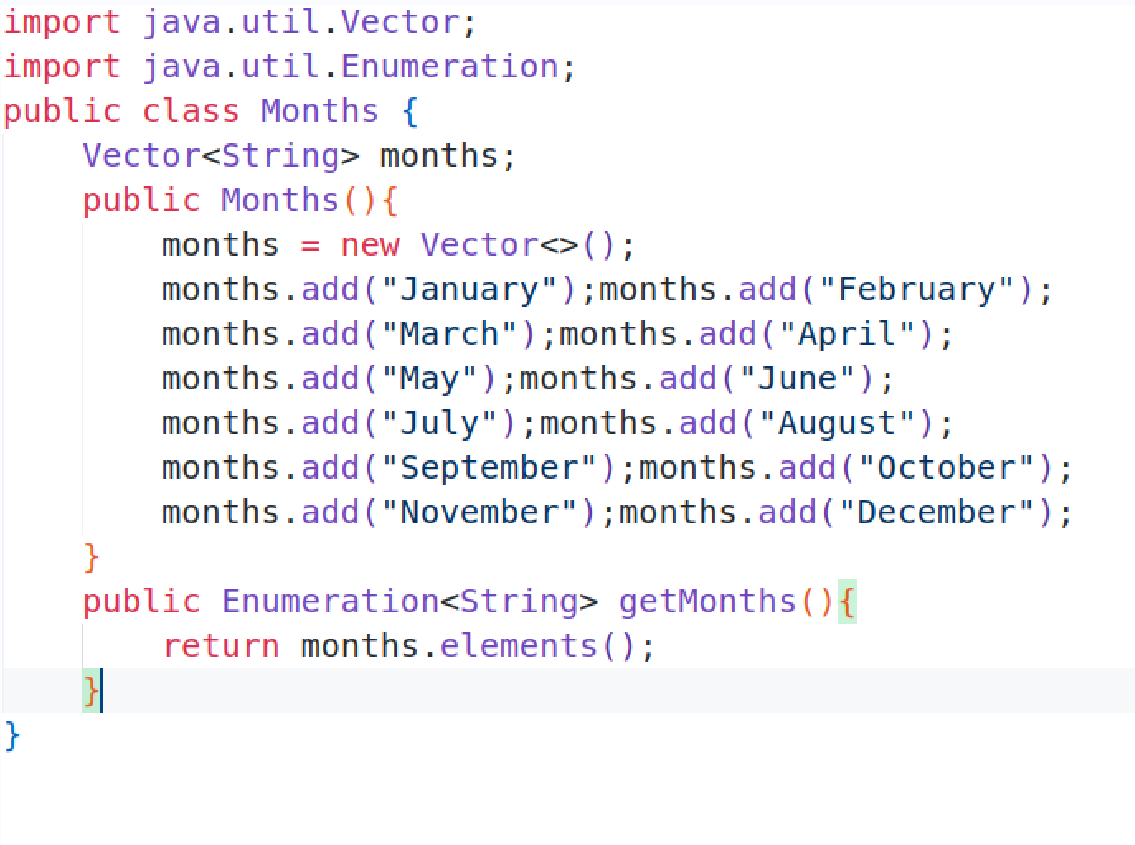
This implementation uses composition to delegate the logic to the *Adapter*.

- In this case, the ***Adapter*** contains the ***Adaptee*** and delegates the `request()` method to the `specificRequest()` method in the ***Adaptee***.

Example using Enumeration as Iterator

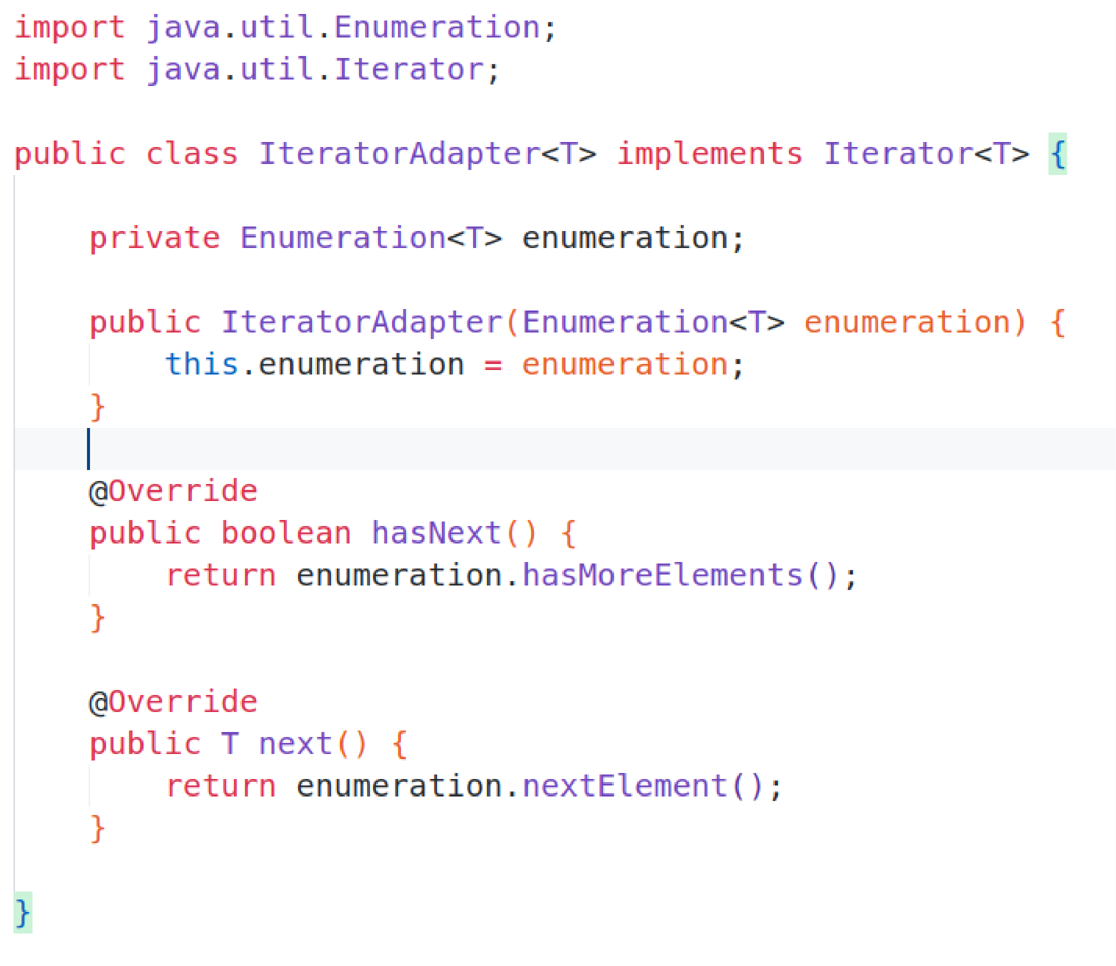


```java
import java.util.Iterator;
import java.util.ArrayList;
public class PrintAll {

    Run | Debug
    public static void main(String[] args){
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");list.add("World");

        Iterator<String> iterator = list.iterator();
        printAll(iterator);
        IteratorAdapter<String> itr = new IteratorAdapter<String>
                           ((new Months()).getMonths());
        printAll(itr);

    }
    public static void printAll(Iterator<String> itr){
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```



```java
import java.util.Vector;
import java.util.Enumeration;
public class Months {
    Vector<String> months;
    public Months(){
        months = new Vector<>();
        months.add("January");months.add("February");
        months.add("March");months.add("April");
        months.add("May");months.add("June");
        months.add("July");months.add("August");
        months.add("September");months.add("October");
        months.add("November");months.add("December");
    }
    public Enumeration<String> getMonths(){
        return months.elements();
    }
}
```

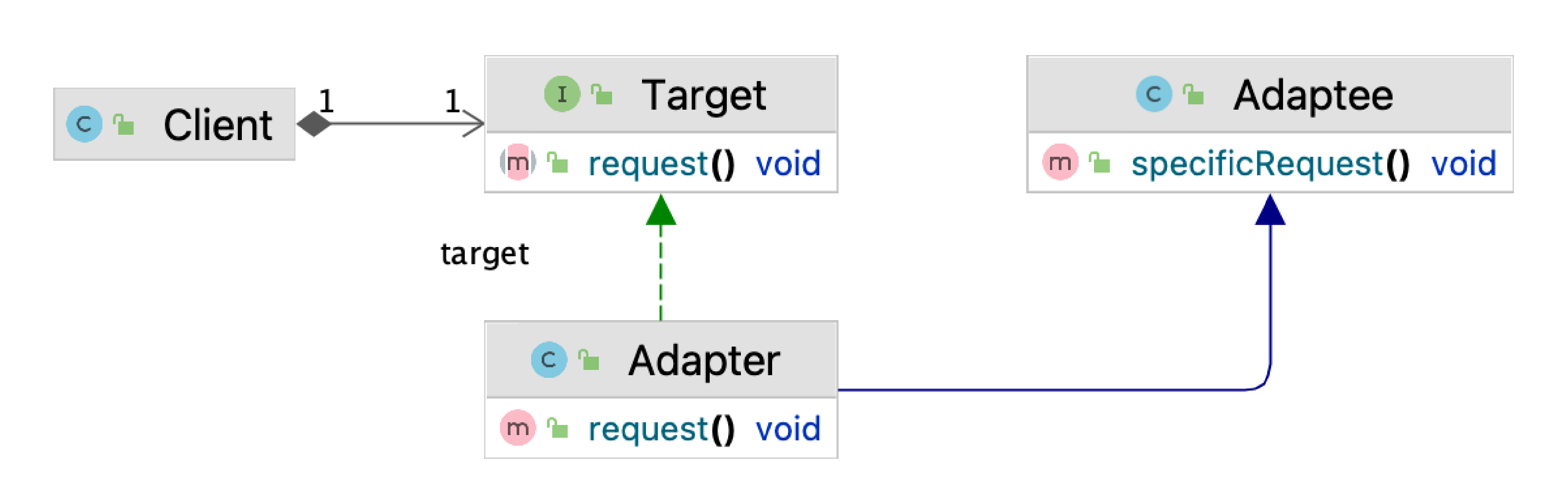
Enumeration as Iterator – Cont.



```java
import java.util.Enumeration;
import java.util.Iterator;

public class IteratorAdapter<T> implements Iterator<T> {

    private Enumeration<T> enumeration;

    public IteratorAdapter(Enumeration<T> enumeration) {
        this.enumeration = enumeration;
    }

    @Override
    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    @Override
    public T next() {
        return enumeration.nextElement();
    }

}
```

Using inheritance

- This version of the Adapter pattern requires multiple inheritance, which is technically impossible in Java if we're not considering interfaces with ***default*** methods. The main idea is to create the ***Adapter*** by extending both the ***Target*** and the ***Adapter*** classes.
- However, we can implement this in Java when we have the ***Target*** as an ***interface*****,** which is easier to achieve because the ***Target*** is the part we have control of:



Observer Pattern (Behavioural)

- **Intent**
  - Defines a *one-to-many* dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - Also known as publish-subscribe
- **Motivation**
  - The need to maintain consistency between related objects without making classes tightly coupled

Observer Pattern – Applicability

- Use the Observer pattern in any of the following situations:
  - When an abstraction has two aspects, one dependent on the other.
  - Encapsulating these aspects in separate objects lets you to have flexibility and reuse them independently.
  - When a change to one object requires changing others
  - When an object should be able to notify other objects without making assumptions about those objects

Observer Pattern - Key Players

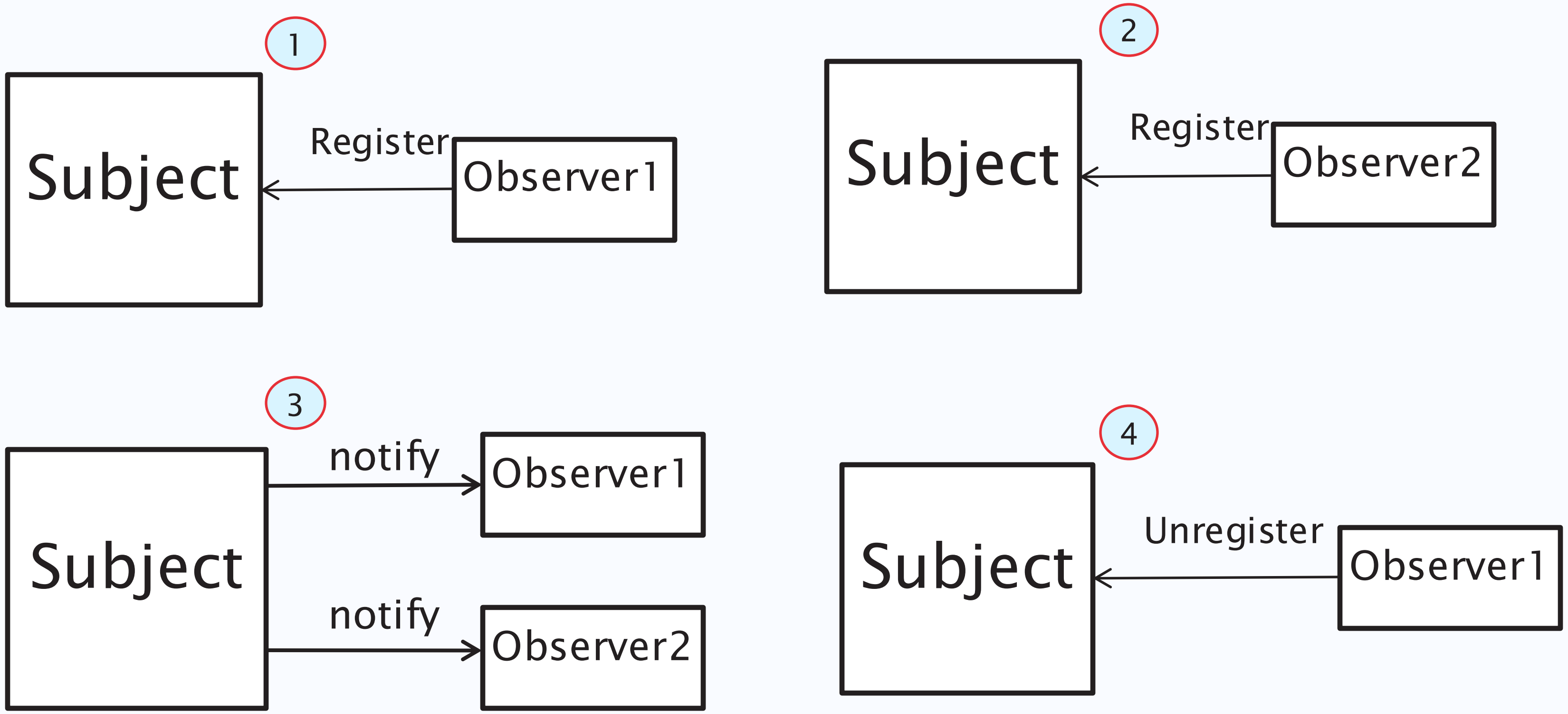
- **Subject**
  - Keeps track of its observers. Any number of Observer objects may observe a subject
  - Provides an interface for attaching/detaching an observer
- **Observer**
  - An updating interface for objects that gets notified of changes in a subject
- **ConcreteSubject**
  - Stores “state of interest” to observers
  - Sends notification when state changes
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's
  - Implements updating interface

Observer Pattern - Advantages

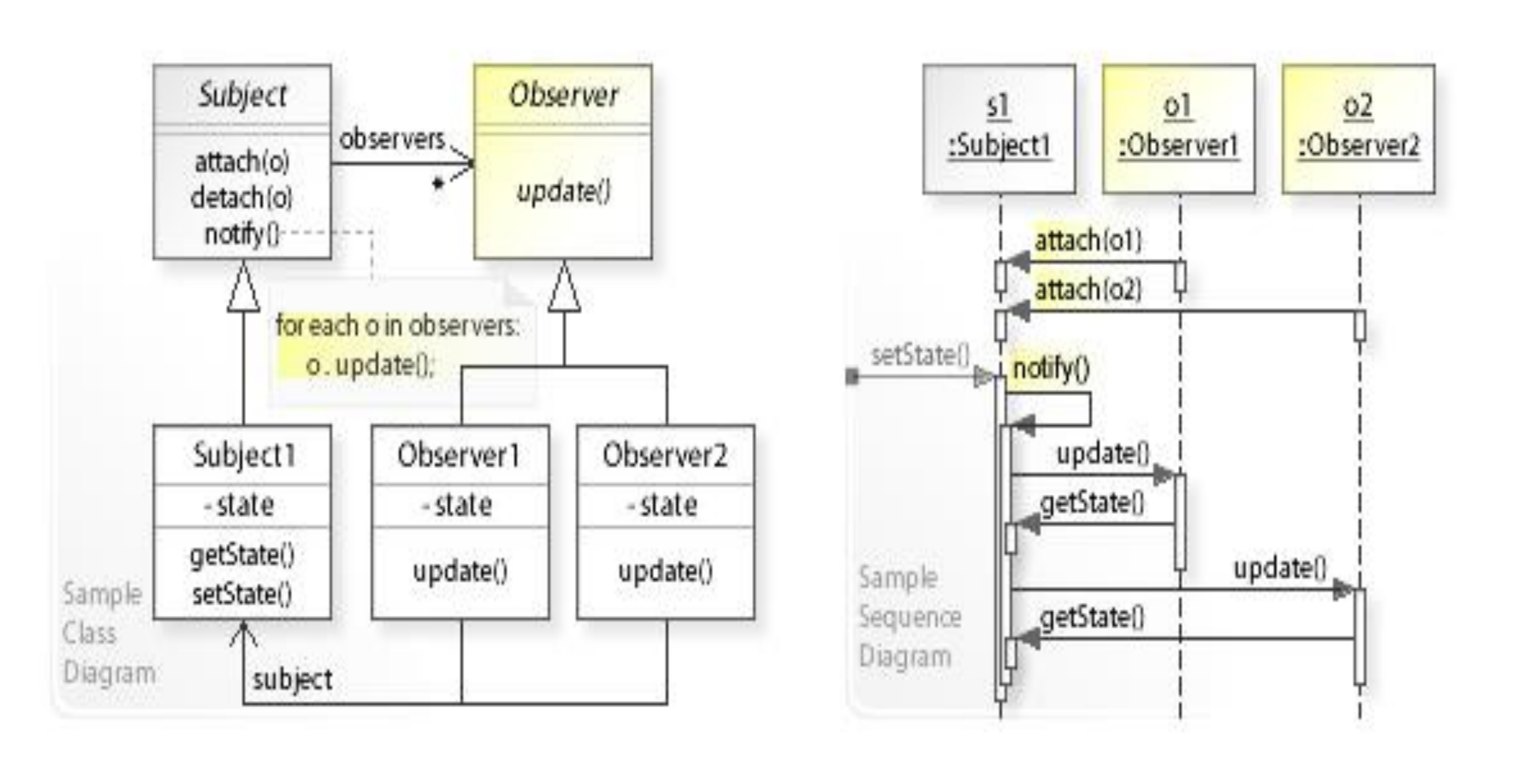
- Minimal coupling between the Subject and the Observer
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - All subject knows its list of observers
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
- Support for event broadcasting
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time

Observer Pattern – How does it Work?

- A number of Observers “register” to receive notifications of changes to the Subject. Observers are not aware of the presence of each other.



Observer Pattern – UML

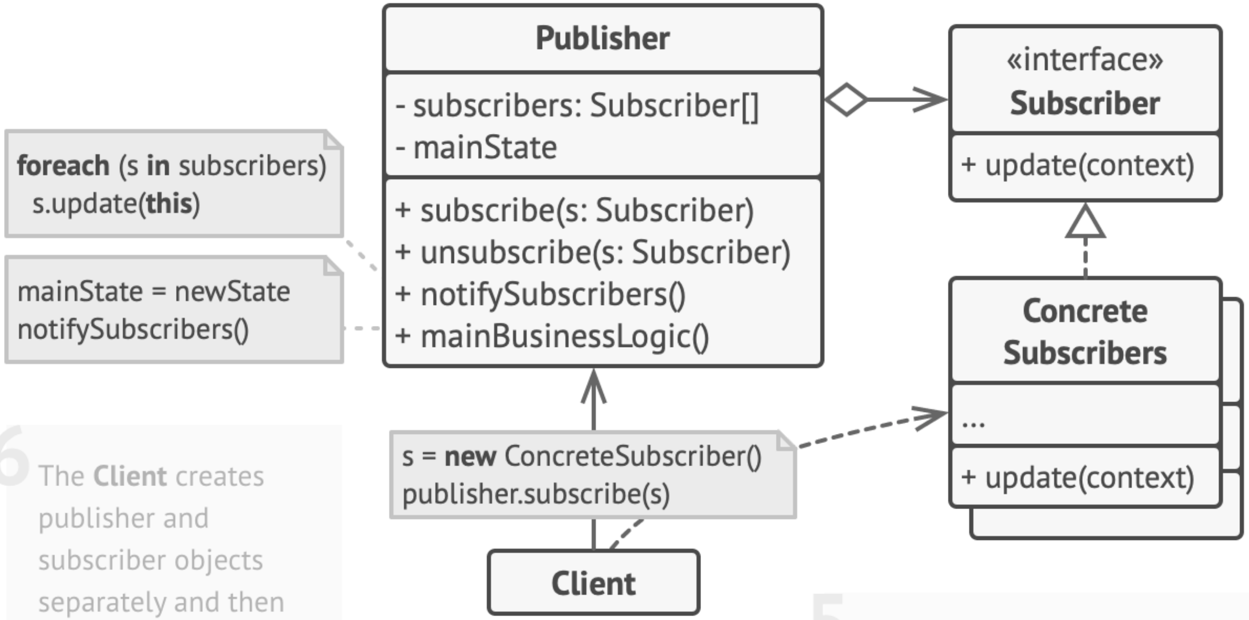


Observer Pattern – Structure

1 The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

2 When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.

3 The **Subscriber** interface declares the notification interface. In most cases, it consists of a single `update` method. The method may have several parameters that let the publisher pass some event details along with the update.



4 **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

6 The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

Composite Pattern - Elements

- Component interface
  - Specifies the interface common to all objects in the composition
- Leaf classes (there could be several, in general)
  - Each implements Component and and defines behaviours for primitive objects in the composition. These objects have no children.
- Composite class
  - Implements Component and defines behaviours for components with children. Normally this will consist of iterating through the children, performing the behaviour with each child.

Composite - Continued

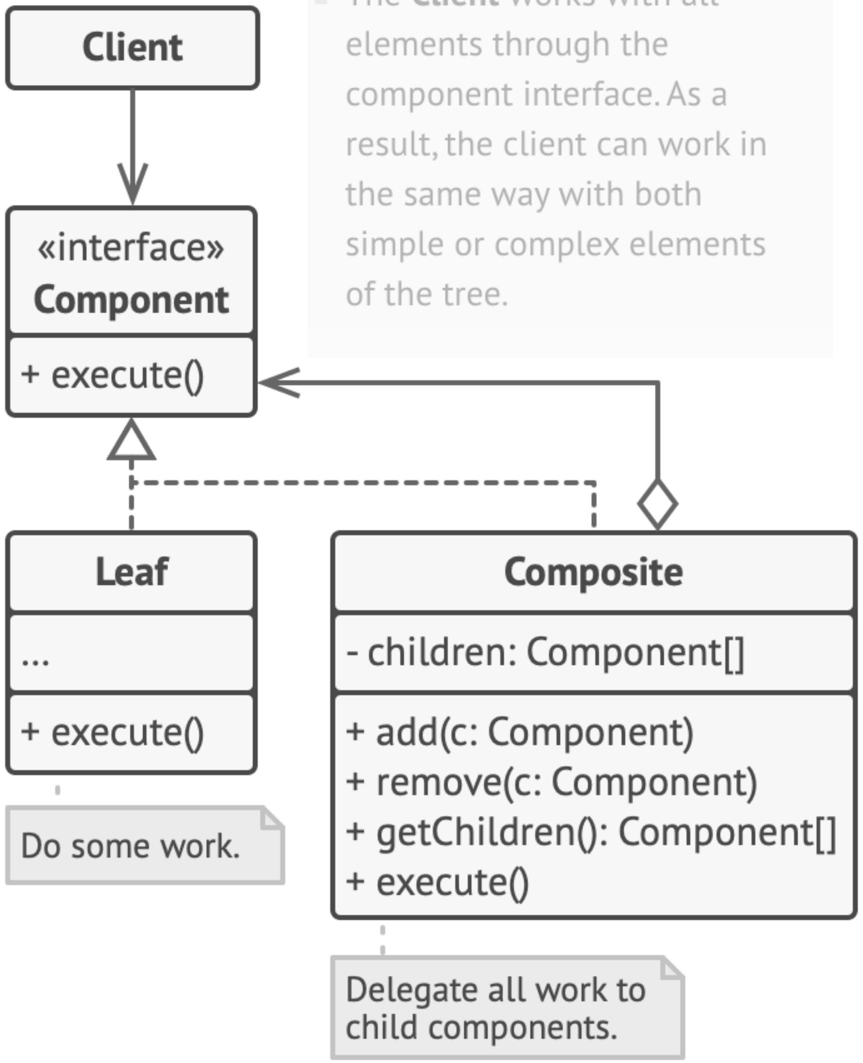
- Applicability – Use composite when:
  - You want to represent part-whole hierarchies of objects
  - You want clients to ignore differences between simple objects and compositions of objects
- Consequences
  - Recursive structural definitions are possible
  - Makes the client simple (uniform treatment)
  - Easy to add new types of component (clients don’t need to be changed)

Composite – Structure

1 The **Component** interface describes operations that are common to both simple and complex elements of the tree.

2 The **Leaf** is a basic element of a tree that doesn't have sub-elements.

Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.



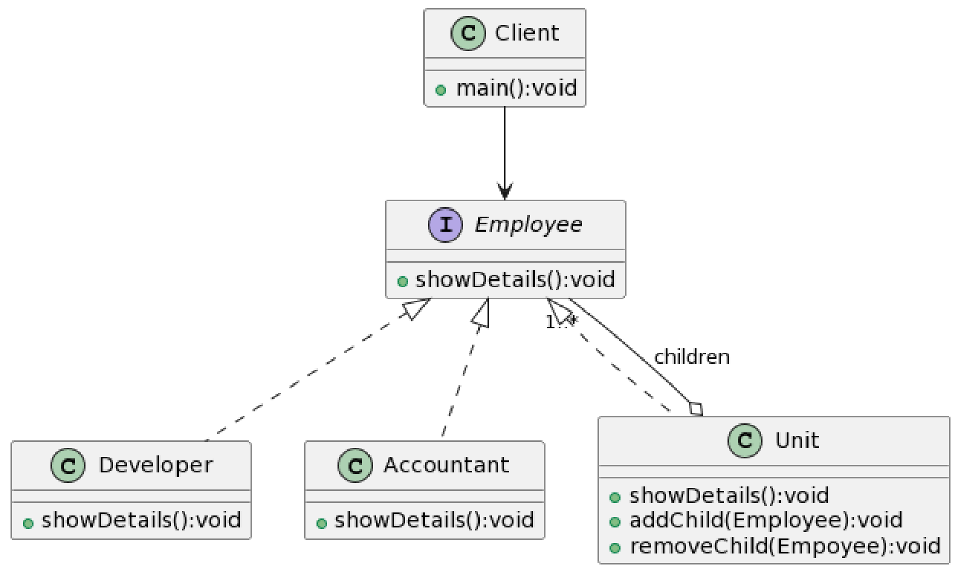
4 The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

3 The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

Composite Pattern - Example

- Consider a company that has employees, units and subunits
- Define a common interface Employee
- Units are composed of Employees and implement Employee interface
- Units can be composed of other units



Design problems

- To use patterns in your design, you need to recognise that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Good design principles

- Program to interfaces not to an implementation
- Separate what changes from what does not
- Loosely couple objects that interact
- Classes should be open for extension, but closed for modification
- Each class should have one responsibility
- Depend on abstractions, not concrete classes

Good use of Design Pattern Principles

- Let design patterns emerge from your design, don’t use them just because you should
- Always choose the simplest solution that meets your need
- Always use the pattern if it simplifies the solution
- Know a good list of the design patterns out there

Key points

- Patterns are distillations of accumulated wisdom that provides a standard jargon, naming and concepts that experienced practitioners apply.

- If all this design stuff are so important, isn’t programming robotic?
  - Short answer: No!
  - Long answer: how to apply a pattern to a problem in hand; Language considerations; Platform considerations; etc.

Resources

- Example code can be accessed from this Repository
- **Design Patterns**

  https://sourcemaking.com/design_patterns

  https://www.oodesign.com/
- **Design Pattern Quick Guide**

  https://www.tutorialspoint.com/design_pattern/design_pattern_quick_guide.htm

YOUR QUESTIONS