Design Patterns

CITS4401 Software Requirements and Design

Week 10

Recap

Software design with interfaces

- Software
- Hardware

- In Software Engineering: Modular design and External services

Software Design Patterns

What is a Pattern?

- Inspired by the work of Christopher Alexander, who first described patterns in Architecture:

"Each pattern describes a problem which occurs over and over again in our environment, then describes the core of the solution to that problem, in such a way that you can use this solution a million time over, without ever doing it the same way twice"

- The idea was embraced by computing analysis and design theorists and practitioners. Martin Fowler's definition of a pattern:

"An idea that has been useful in one practical context and will probably be useful in others"

What is a Pattern?

Effective patterns provide solutions that are used again and again..

Design Patterns

- Smart
  - an elegant solution not obvious to a novice
- Generic
  - not dependent upon a system, programming language or application domain
- Well-Proven
  - has been identified from real OO systems
- Simple
  - is usually quite small, involving only a handful of classes
- Reusable
  - is documented in such a fashion that it is easy to reuse

Design Patterns

- a Pattern Name

  a handle we can use to describe a design problem, its solutions and consequences

- the Problem

  describes when to apply the pattern. It explains the problem and its context

- the Solution

  describes the elements which make up the solution and their relationships

- the Consequences

  the results and trade-offs of using the design pattern

Design Patterns

- Class
- the pattern is primarily concerned with classes, they deal with the relationships between classes and their sub-classes. These relationships are established through Inheritance and are static.

- Object
- the pattern is primarily concerned with object relationships, which are more dynamic and can change at run-time

Design Pattern Taxonomy

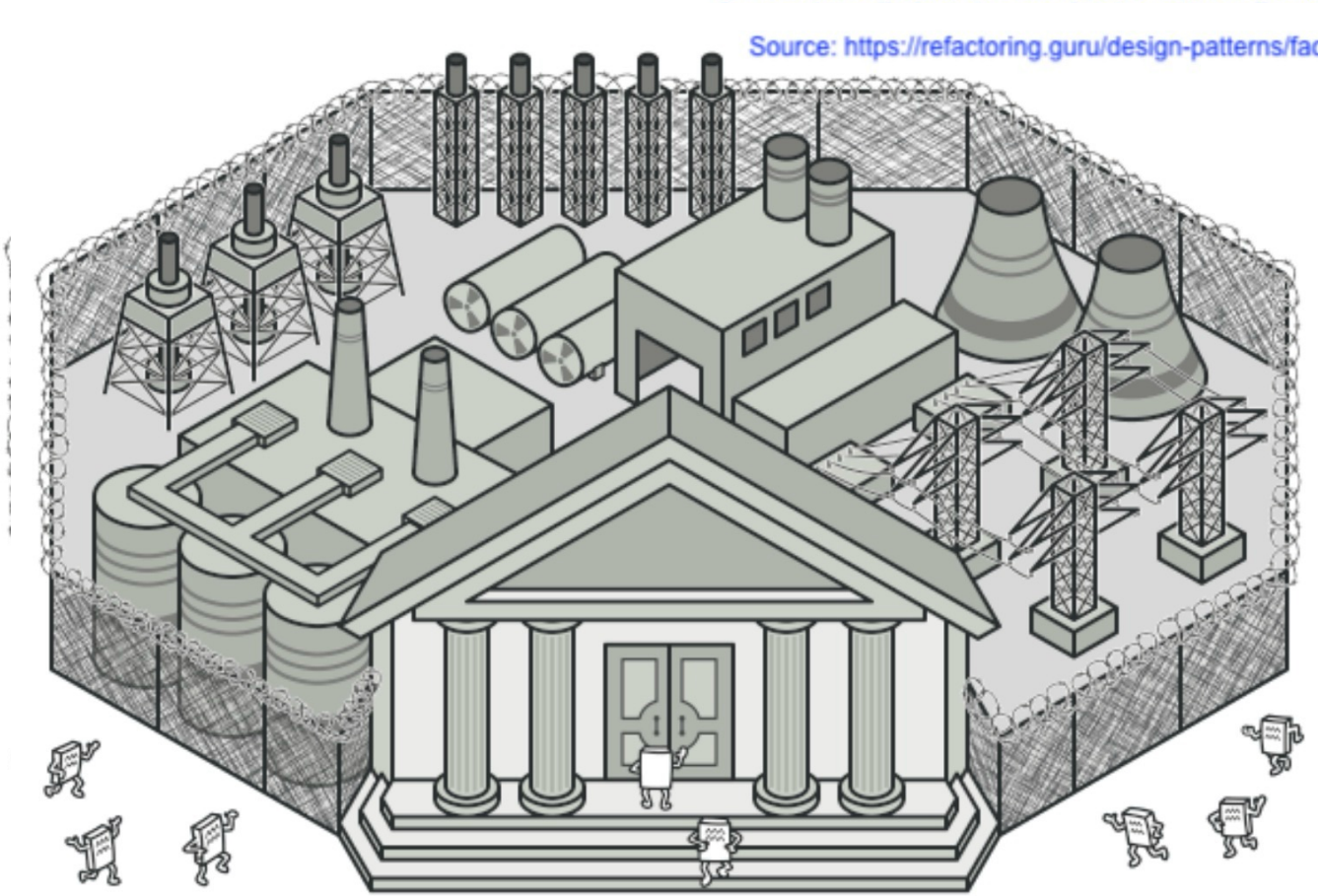
- Structural
  - deal with the composition of classes and objects, e.g.
  - Adapter,
  - Facade
  - Proxy
- Behavioural
  - characterize the way in which classes or objects interact and distribute responsibility, e.g.
  - Observer
- Creational
  - Making a system independent from the way its objects are created, composed and represented (not covered in this unit).

Structural Patterns

Structural Patterns: Facade, Adaptor, Proxy

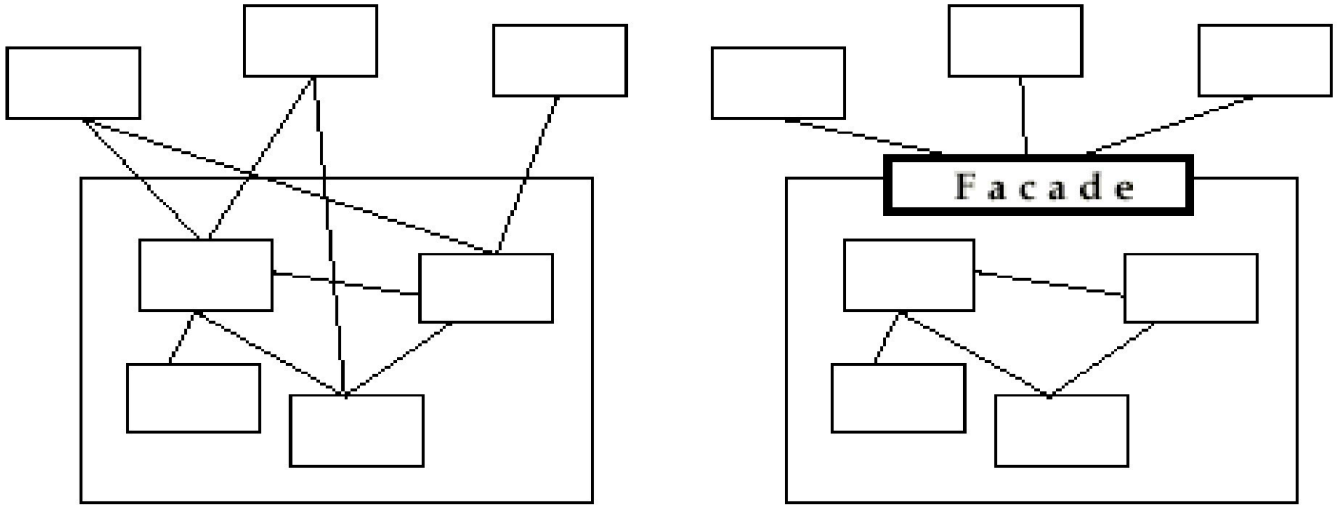
- Structural patterns manage interactions with a subsystem.
- They reduce coupling between subsystems.
  - A subsystem consists of
    - an interface object
    - a set of application domain objects (entity objects) modeling real entities or existing systems
    - one or more control objects
- Realization of Interface Object: Facade
  - Provides the interface to the subsystem
- Interface to Existing systems: Adapter
    - Provides the interface to existing system (legacy system)
    - The existing system is not necessarily object-oriented!
- Defer object creation or initialization: Proxy

Façade Pattern



Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e., it abstracts out the gory details)
- Facade allows us to provide a closed architecture



Facade Pattern

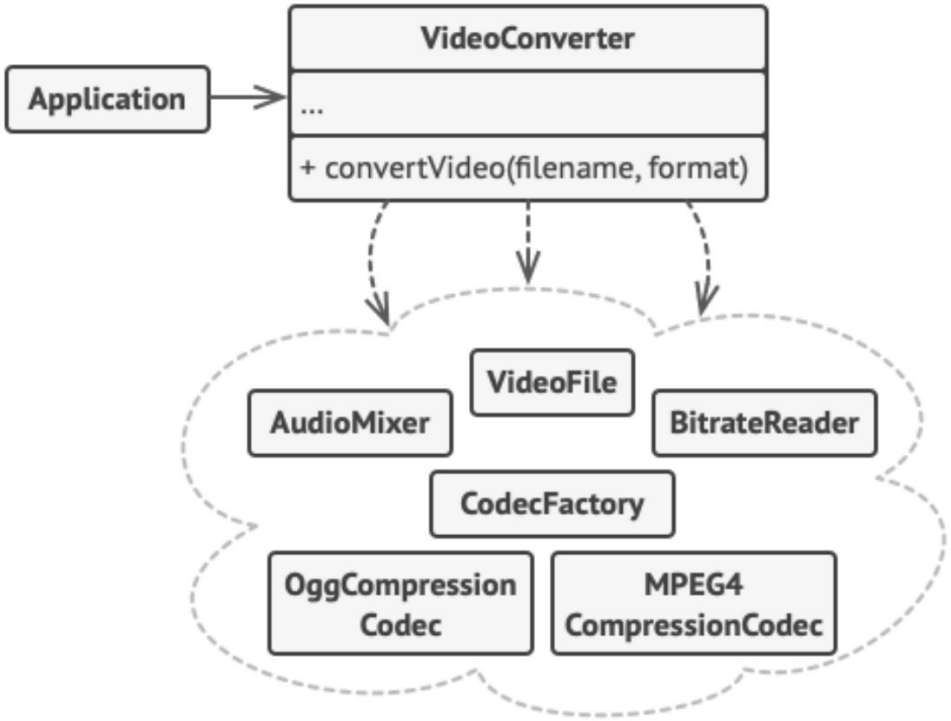
```
class VideoConverterFacade {
    method convertVideo(videoFileName, format) {
        videoFile = VideoFile(videoFileName)
        audioCodec = BitrateReader(videoFile, format)
        audioCodec.convert()
        audio = AudioMixer(audioCodec)
        return audio}
}

class VideoFile {
    constructor(videoFileName) {
        // Initialize video file
    }
}

class BitrateReader {
    constructor(videoFile, format) {
        // Initialize bitrate reader with video file and
format
    }

    method convert() { // Convert bitrate
    }
}
```

```
class AudioMixer {
    constructor(audioCodec) {
        // Initialize audio mixer with audio codec
    }
}
```

```
// Usage
converter = VideoConverterFacade()
convertedAudio =
converter.convertVideo("example_video.mp4",
"mp3")
```

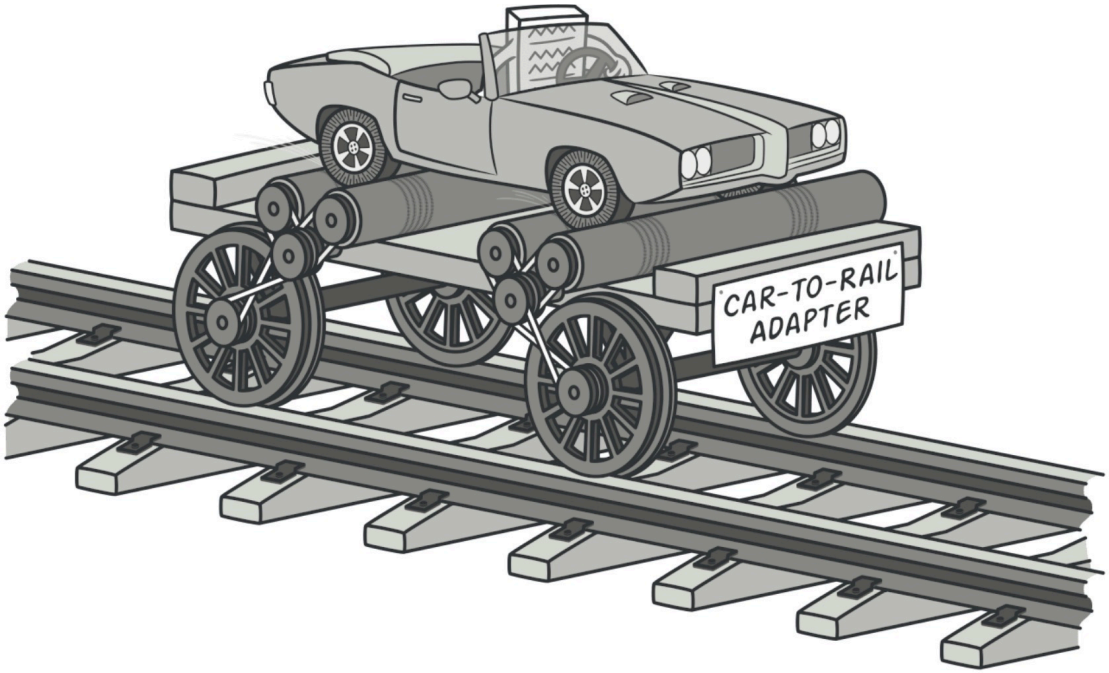
Facade Example

- In this example, the Facade pattern simplifies interaction with a complex video conversion framework.



Adapter Pattern

- Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Adapter Pattern

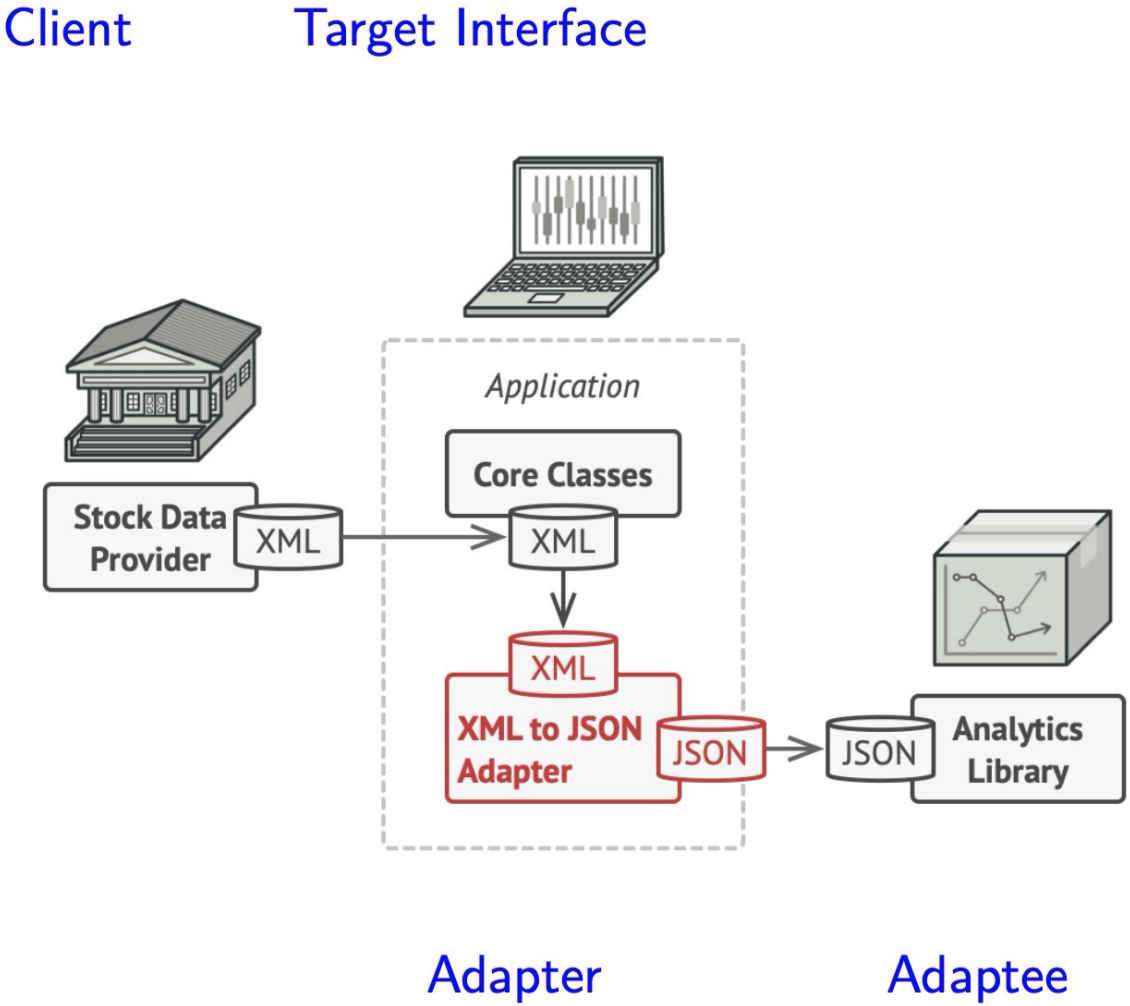
- Problem: You want to use a 3rd party library but it only works with data in a different format to yours (eg JSON instead of XML)
- Solution: Adapter is a special object that converts the interface of one object so that another object can understand it.

- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter.
- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate.

Adapter Terminology

- Client the class that wants to use the third-party library or the external system
- Adaptee a class in the third-party library or the external system that we want to use
- Target interface the desired interface that the client will use
- Adapter this class sits between the client and the adaptee and implements the target interface

Adapter Pattern Example



Adapter Pattern Implementation

- Make sure that you have at least two classes with incompatible interfaces:
  - A useful service class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).

- Create the adapter class and make it follow the client interface. Leave all the methods empty for now.

- Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.

- One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.

- Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

Adapter Pattern Implementation (contd)

```
// Legacy Printer (Third-party or Legacy)
class LegacyPrinter {
    constructor() {
        // Initialize the printer
    }

    printUppercase(message) {
        // Method to print message in uppercase
        console.log(message.toUpperCase());
    }
}

// Client Interface
class PrinterClient {
    print(message) {} // Method to print message
}

// Client Class
class MessageSender {
    constructor(printer) {
        this.printer = printer;
    }

    sendMessage(message) {
        this.printer.printUppercase(message); // Legacy printer expects uppercase
    }
}
```

```
// Adapter Class
class PrinterAdapter {
    constructor(legacyPrinter) {
        this.legacyPrinter = legacyPrinter;
    }

    print(message) {
        const lowercaseMessage = message.toLowerCase();
        this.legacyPrinter.printUppercase(lowercaseMessage); // Convert message to lowercase before printing
    }
}

// Usage
const legacyPrinter = new LegacyPrinter();
const printerAdapter = new PrinterAdapter(legacyPrinter);
const messageSender = new MessageSender(printerAdapter);

messageSender.sendMessage("Hello, World!"); // Output: HELLO, WORLD! (printed in uppercase)
```

Comparison

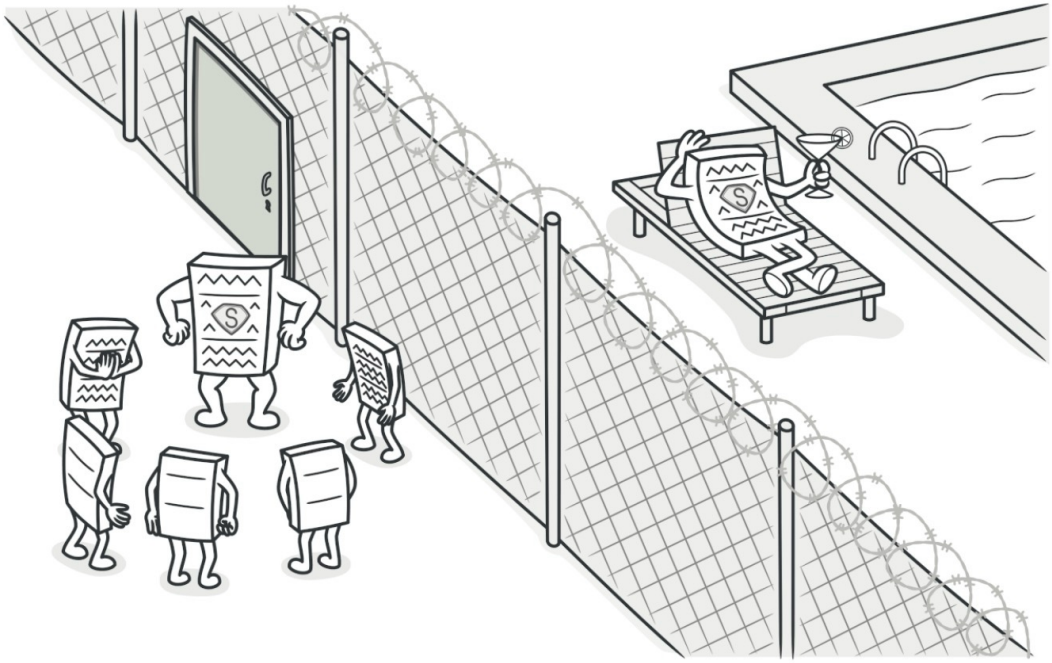
- Facade defines a new interface for existing objects

- Adapter tries to make an existing interface usable
-
- Adapter usually wraps just one object

- Facade works with an entire subsystem of objects

Proxy Pattern

- Proxy is a structural design patterns that lets you to provide a placeholder for another object. A proxy controls access to the original object.

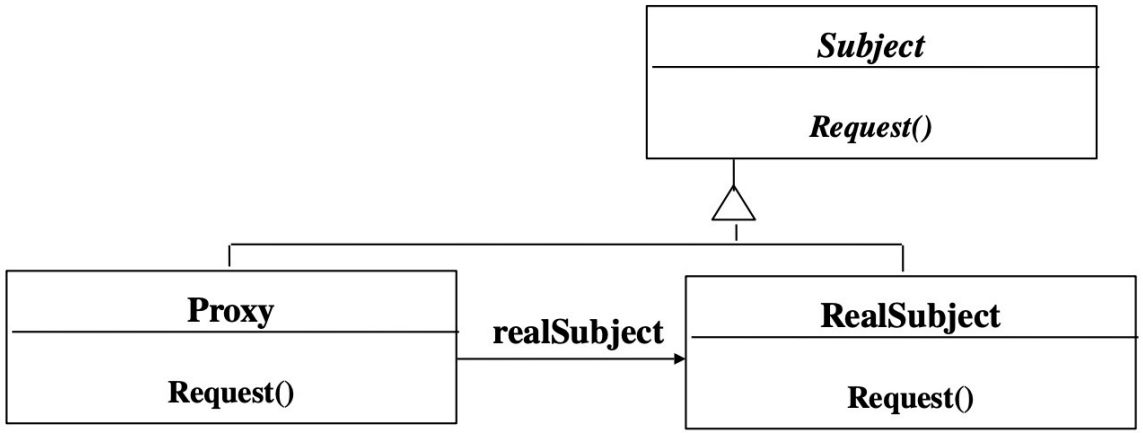


Proxy Pattern

- Proxy pattern:
  - Uses another object (“the proxy”) that acts as a stand-in for the real object
  - Reduces the cost of accessing objects
  - The proxy creates the real object only if the user asks for it

Proxy Pattern

- Interface inheritance is used to specify the interface shared by Proxy and RealSubject.
- Delegation is used to catch and forward any accesses to the RealSubject (if desired)

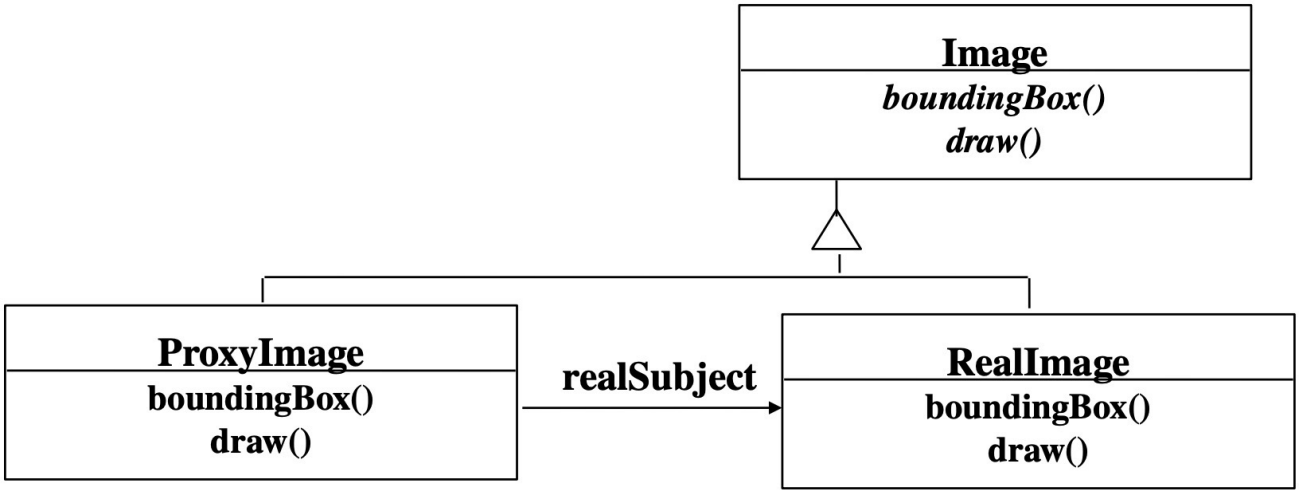


Proxy Types

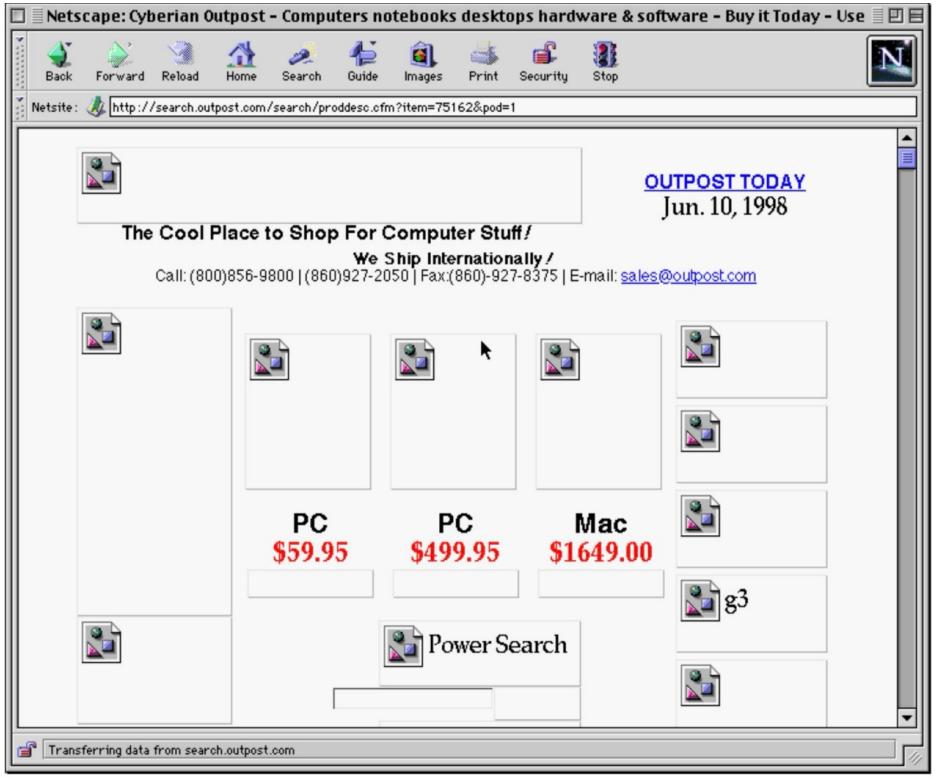
- Remote Proxy
  - Local representative for an object in a different address space
  - Caching of information: Good if information does not change too often
- Virtual Proxy
  - Object is too expensive to create or too expensive to download
  - Proxy is a stand-in
- Protection Proxy
  - Proxy provides access control to the real object
  - Useful when different objects should have different access and viewing rights for the same document.
  - Example: Grade information for a student shared by administrators, teachers and students.

Virtual Proxy Example

- Images are stored and loaded separately from text
- If a RealImage is not loaded a ProxyImage displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a ProxyImage instead of a RealImage



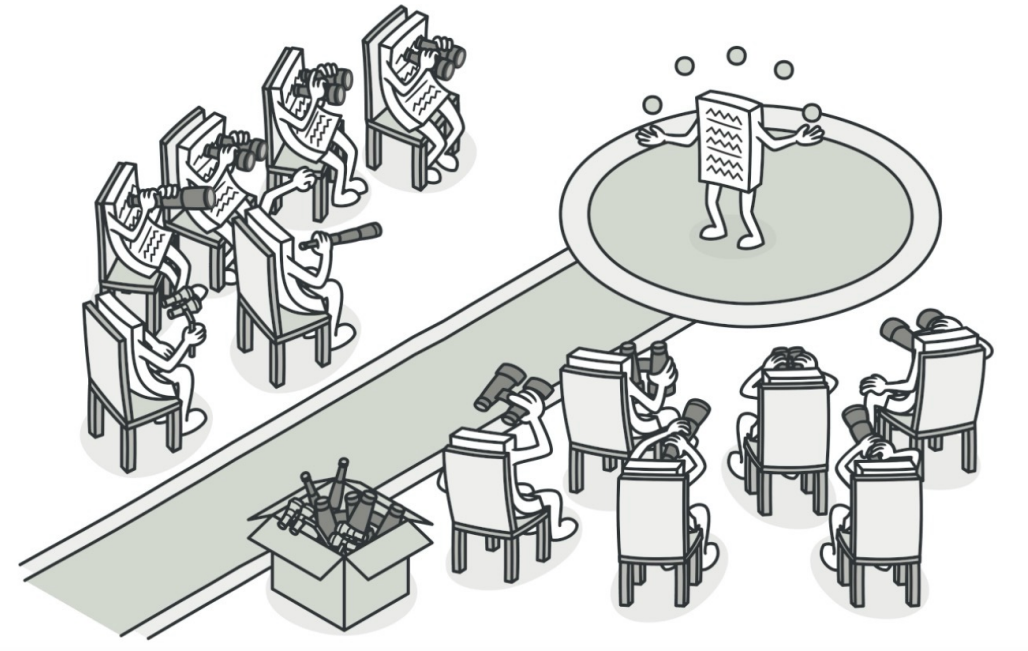
Virtual Proxy Example



Behavioural Patterns

Observer Pattern

- Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object that're observing.

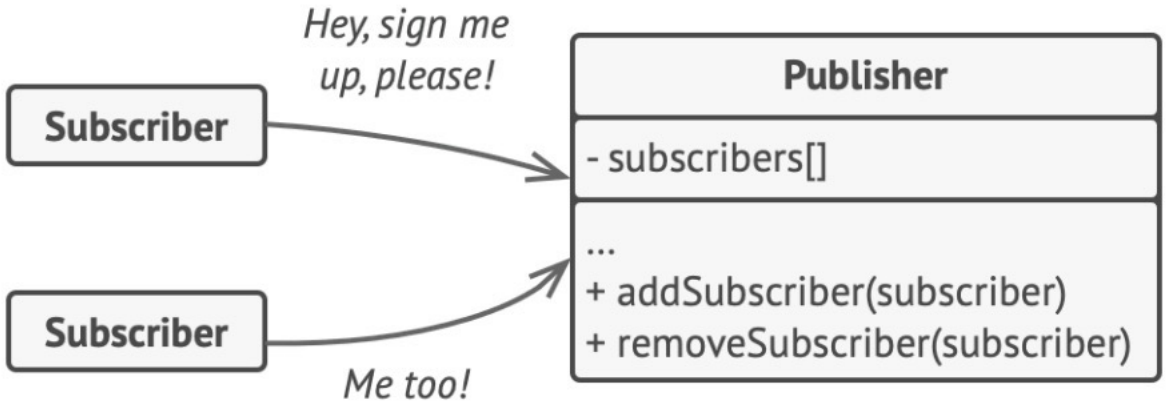


The Problem

- Imagine you have a Customer and a Store
- Customer is very interested in a particular brand of product which should become available in the store very soon
- Customer could visit the store every day and check product availability. But most of these trips would be pointless.
- OR the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available
- Conflict: either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

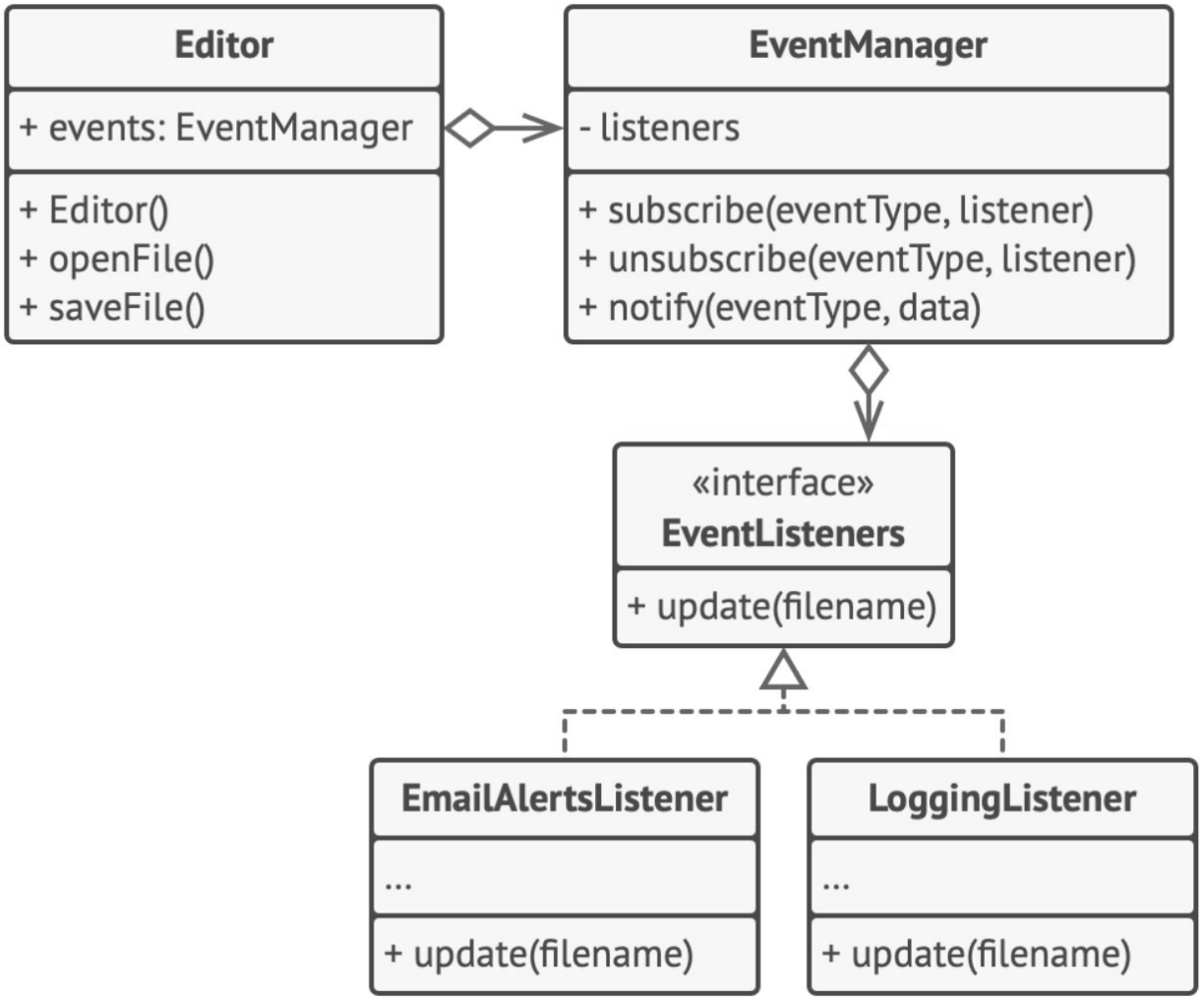
Observer Solution

- Publisher will notify others about changes to its state
- Subscribers are objects that want to track changes to the publisher's state
- Observer pattern adds a subscription mechanism to the publisher class comprising:
- + An array to store a list of references to subscriber objects
- + Public methods to add and remove subscribers from the list



Whenever an important event happens to the publisher, it loops through its subscribers and calls the specific notification method on their objects.

Observer UML Example



Consequences

- Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Observer solutions is also called Publish and Subscribe

- The Observer pattern:
  - Maintains consistency across redundant state
  - Optimizes batch changes to maintain consistency

Summary

- Software design patterns = a reusable solution to a design problem
- Facade: Structural pattern providing a consistent interface to a subsystem
- Adaptor: Structural pattern for interacting with a legacy system
- Proxy: Defer expensive object creation or initialization
- Observer: Behavioural pattern for publish and subscribe

Recommended Reading

- Tutorials on design patterns https://refactoring.guru/design-patterns
- Bruegge & Dutoit, 2010: Chapter 8 and Appendix A
- Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994