**1. Introduction to Design Patterns**

**Goal**:

* Understand what design patterns are and how they provide reusable solutions to common design problems in software development.

**Recap**:

* Software design with interfaces (Software, Hardware).
* Modular design and external services in Software Engineering.

**2. Definition and Importance of Design Patterns**

**What is a Pattern?**

* Inspired by Christopher Alexander in architecture, design patterns in software describe a problem that occurs frequently and its core solution, which can be used repeatedly without doing it the same way twice.( Well-Proven and Reusable)

**Definition by Martin Fowler**:

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

**Characteristics**:

* **Smart**: Elegant solutions not obvious to novices.
* **Generic**: Not dependent on a specific system, programming language, or application domain.
* **Well-Proven**: Identified from real object-oriented (OO) systems.
* **Simple**: Involves only a few classes.
* **Reusable**: Documented in a way that makes them easy to reuse.

**3. Components of Design Patterns**

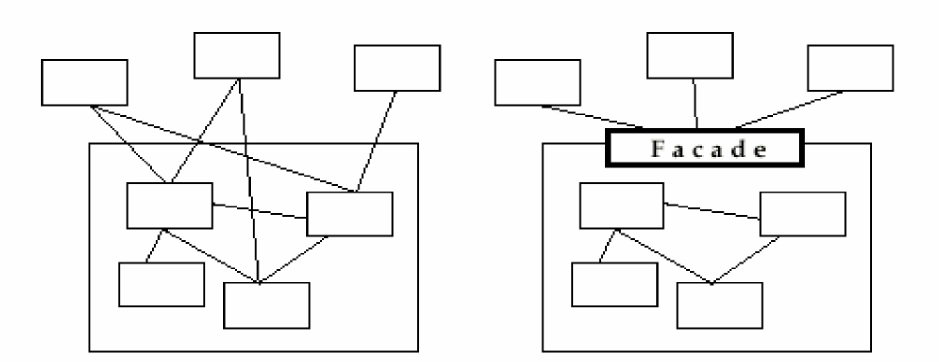
1. **Pattern Name**:
   * A handle used to describe a design problem, its solution, and consequences.
2. **Problem**:
   * Describes when to apply the pattern, explaining the problem and its context.
3. **Solution**:
   * Describes the elements making up the solution and their relationships.
4. **Consequences**:
   * The results and trade-offs of using the design pattern.

**4. Types of Design Patterns**

1. **Structural Patterns**:
   * Deal with the composition of classes and objects (e.g., Adapter, Facade, Proxy).
2. **Behavioural Patterns**:
   * Characterize the way classes or objects interact and distribute responsibility (e.g., Observer).
3. **Creational Patterns**:
   * Make a system independent from how its objects are created, composed, and represented (not covered in this unit).

**5. Structural Patterns**

**5.1 Facade Pattern**

****

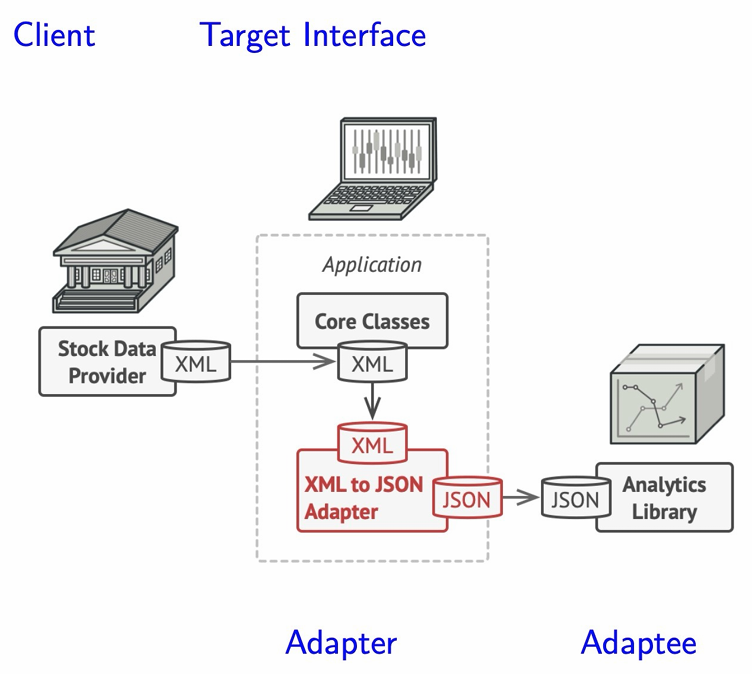
**Purpose**:

* Provides a unified interface to a set of objects in a subsystem, making the subsystem easier to use.
* Facade allows us to provide a closed architecture

**Example**:

* A video conversion system where the Facade pattern simplifies interaction with a complex video conversion framework.

**5.2 Adapter Pattern**

****

• 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

**Purpose**:

* Allows objects with incompatible interfaces to collaborate by wrapping one object to hide the complexity of conversion happening behind the scenes.

**Example**:

* Using a 3rd party library that works with JSON while your system uses XML.

**5.3 Proxy Pattern**

**Purpose**:

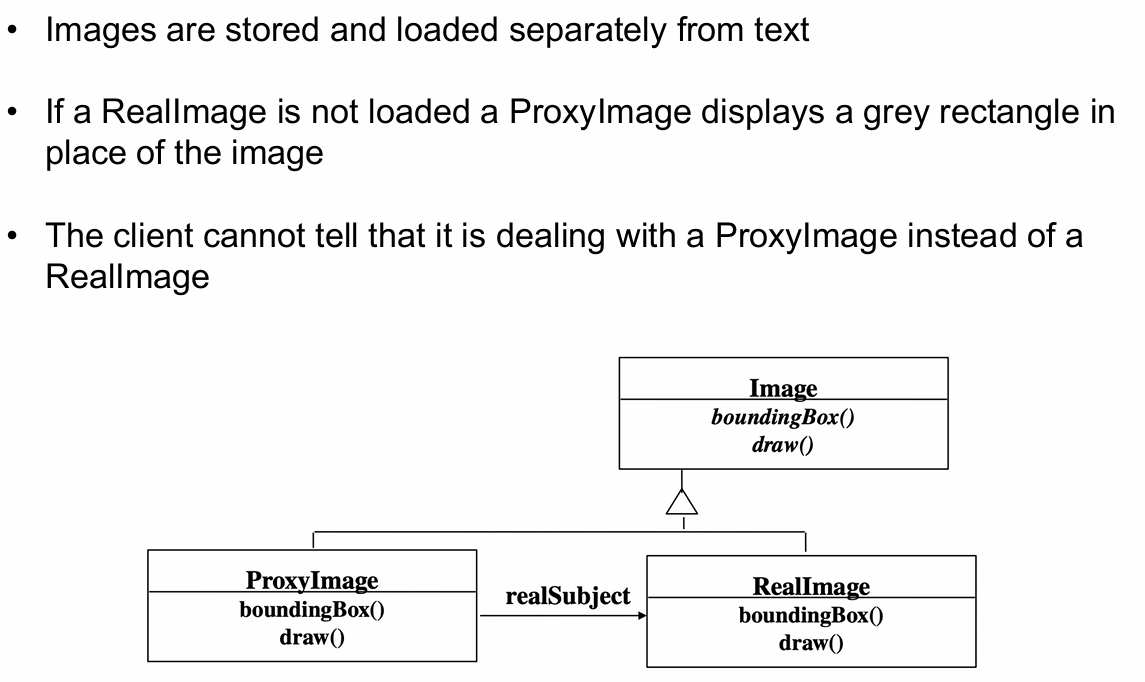
* Provides a placeholder for another object to control access to it, reducing the cost of accessing objects.

**Types**:

* **Remote Proxy**: Local representative for an object in a different address space.
* **Virtual Proxy**: Stand-in for objects that are expensive to create or download.
* **Protection Proxy**: Provides access control to the real object.

**Example**:

* Loading images where a Proxy Image displays a placeholder until the Real Image is loaded.



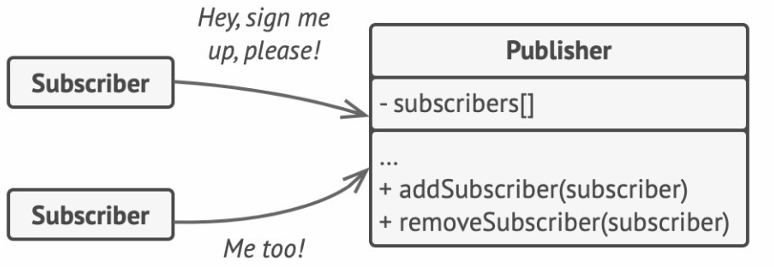
**6. Behavioural Patterns**

**Observer Pattern**

**Purpose**:

* Defines a subscription mechanism to notify multiple objects about events happening to the object they are observing.
* Observer defines a one-to-many dependency between objects, when one object changes state, all its dependents are notified and updated automatically.

**Problem**:

* For example, a customer wants to be notified when a specific product becomes available in a store.
* 

**Solution**:

* **Publisher** notifies subscribers about state changes.
* **Subscribers** track changes to the publisher’s state.

**Example**:

* 