Exam Review 2

Study Guide for CEN 4020 Software Engineering Exam

Disclaimer: This study guide is a summary of a review conducted on October 30th, 2023, by Christopher Mills of Florida State University. The source material was provided by Te-Yen Wu. It is important to note that this study guide should not be reproduced or published without proper authorization from the original creator. Adhering to honor code policies, it is emphasized that this guide is **not** in violation of any academic integrity standards, provided it is not intended for commercial distribution. The guide is not for sale or purchase, and any infringement on this principle is the sole responsibility of the author, **Aiden Allen**, as there is no intent to engage in such activities.

Topics Covered: This study guide encompasses a wide range of topics within the field of software engineering, spanning from UML modeling to design patterns.

Note: Please respect intellectual property rights and use this guide only for educational purposes.

Topics Covered: This comprehensive study guide addresses various vital aspects of software engineering, designed to assist in your preparation for the CEN 4020 Software Engineering Exam. The topics include:

1. **Software Development Life Cycle (SDLC):** An understanding of different SDLC models, such as Waterfall, Agile, and Iterative, is essential. Comprehend their characteristics, advantages, and disadvantages.
2. **Unified Modeling Language (UML):** Familiarize yourself with UML diagrams, including use case, class, sequence, and activity diagrams.
3. **Requirements Engineering:** Learn how to gather, analyze, document, and manage software requirements. Understand the importance of clear and well-defined requirements in the development process.
4. **Software Design Principles:** Explore design principles like SOLID (Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, Dependency Inversion), DRY (Don't Repeat Yourself), and KISS (Keep It Simple, Stupid).
5. **Design Patterns:** Recognize common design patterns, such as Singleton, Factory, Observer, and Strategy patterns. Be ready to apply them in software design and development.

Software Architecture

- Architectural patters
  - Subsystems
  - Their interaction principles
- The allocation of modules into subsystems
- Data storage paradigms
- Recovery systems in place
- Used frameworks, tools, and languages
- **Object Oriented Design**: detailed view
  - Methods and attributed of each class
    - List of methods (with sterotypes)
    - List of attributes (with types)
    - Pseudocode for each method
  - Interactions between modules
  - Design patterns
  - Programming idioms
  - Algorithms

Core Architectural Principles

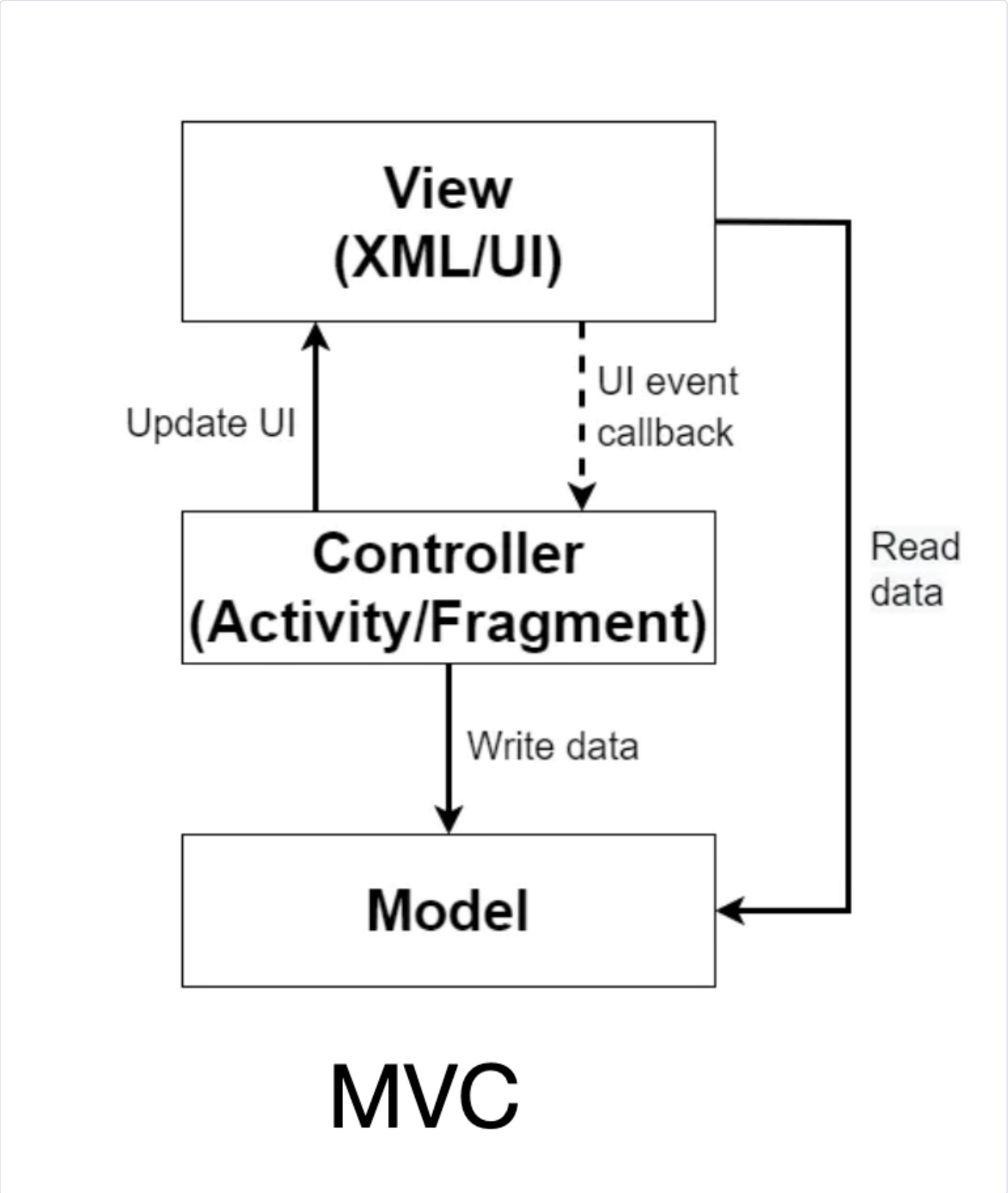
- **Single Responsibility**: Each module should be responsible for only a specific feature or functionality, aggregation of cohesive functionalities
- **Seperation of concerns**: Minimize interaction points to achieve high cohesion and low coupling
- **Principle of least knowledge**: A module should not know about internal details of other modules
- **Don't Repeat Yourself (DRY)**: Do not duplicate functionality
- **KISS (Keep it simple stupid)**: Make it simple only focus on what is needed

Popular Architectural Patterns

Pattern	Desc
Layered Architecture	a model where a whole network process is divided into various smaller sub-tasks. These divided sub-tasks are then assigned to a specific layer to perform only the dedicated tasks.
Model-View-ViewModel (MVVM)	software design pattern that is structured to separate program logic and user interface controls.
Model-View-Controller (MVC)	is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. These elements are the internal representations of information (the Model), the interface (the View)

Pattern	Desc
	that presents information to and accepts it from the user, and the Controller software linking the two
Client-server	multiple clients request and receive files and services from a centralized server over a local or internet connection. A client uses an application as an interface to connect to the server.
Pipe-and-filter architecture	Data is passed from pipes through a variety of filters, and ultimately ends up in the sink.
Message Bus	a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces.

MVVM vs MVC



Identifying object/class in architecture design

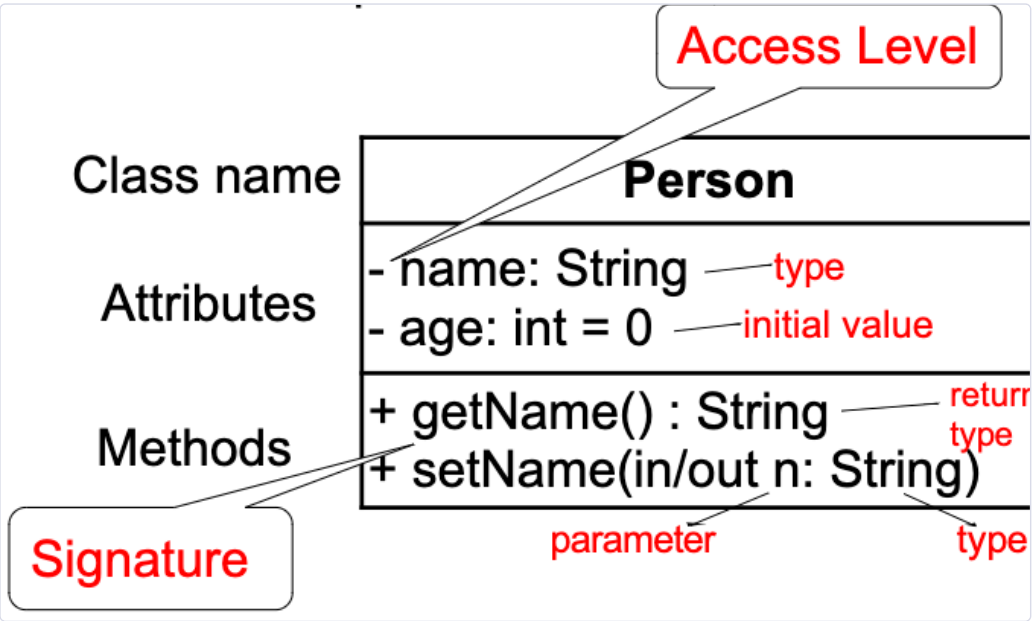
- Use grammatical approach to identify objects and their behaviors.
- Base the identification on tangible things in the application domain
- Look for hierarchies and relationships between components in the architecture. Hierarchies may indicate inheritance relationships between object classes, while relationships may suggest associations, aggregations, or composition

Unified Modelling Language

- Standardized specification language
- A language, not a model
- General purpose notation that is used to visualize, specify, construct, and document the design
- Allows developers to visualize software design and architecture

UML - class diagram

- Represents the structure of the system and specifies detailed behavior and relationships between classes



Access Level	Meaning	Explanation
+	Public	Member is visible to all code in application
-	Private	Member is only visible to code in the class
#	Protected	Member is visible only to code inside the class and any derived classes
-	Package	Member is visible only to code inside the same package

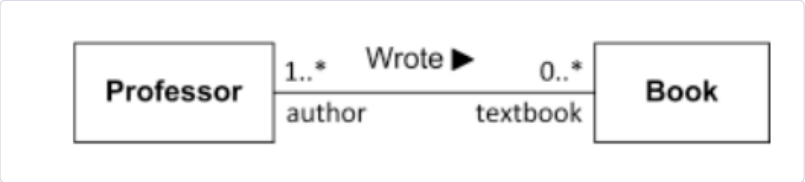
- Informally called "inheritance" or "**Is A** relationship"

- Generalization is adirected relationship between a more general class and a more specific class
- The children classes inherit the attributes and operations of the parent class and can have additional ones

---

Association

- Describes the presence of a relationship between classes (**has a**)
- Name of the association and multiplicity may be placed on the same line



- The association end name is commonly referred to as role
- Professor is an author of a book
- The multiplicity of association denotes how many objects the instance of a class can legitimately have

Aggregation

- An aggregation is a special type of association denoting a **part to whole** relationship
  - Exhaust system "consists of" Muffler and tailpip
  - Muffler "is part of" exhaust system
- Parts can exist without the whole, when the whole is destroyed the parts are not

Composition

- A solid diamond denotes composition
- A strong aggregation
- Parts don not make sense/cannot exist without the whole
- If the aggregate / whole is destroyed, then the parts are destroyed as well

Dependency

- Represents a "using" relationship
- If a change in specification in one class effects another class (but not the other way around) there is dependency

Sequence Diagram

- the interaction that takes place in a collaboration that either realizes a use case or an operation
- High level interactions between user of the system and the system, between the system and other external systems, or between subsystems
- Usually considers small, discrete pieces of the system, individual use cases or operations
- **Horizontal axis**: is for objects, objects that initiates interaction is leftmost
- **Vertical axis**: is for time, Messages sent and received are ordered by time

Message Passing

Word	Definition
Call	Invoke an operation (Synchronous Message), shown with a filled arrow head
Send	Sends a message to an object (Asynchronous), shown with an open arrow head
Return	Return a value to a caller, shown as a dashed line with an open arrow head
Create	Creates an object, shown with a dashed line with an open arrow head

Design Patterns

- Design patterns are template designs that can be used in a variety of subsystems to solve common problems
- Design patterns aid in organizing software in manners that are easier to understand, more maintainable and flexible
- Design patterns describe best practices and capture experience in a way that is possible for others to reuse
- If used incorrectly, can increase complexity

Solid

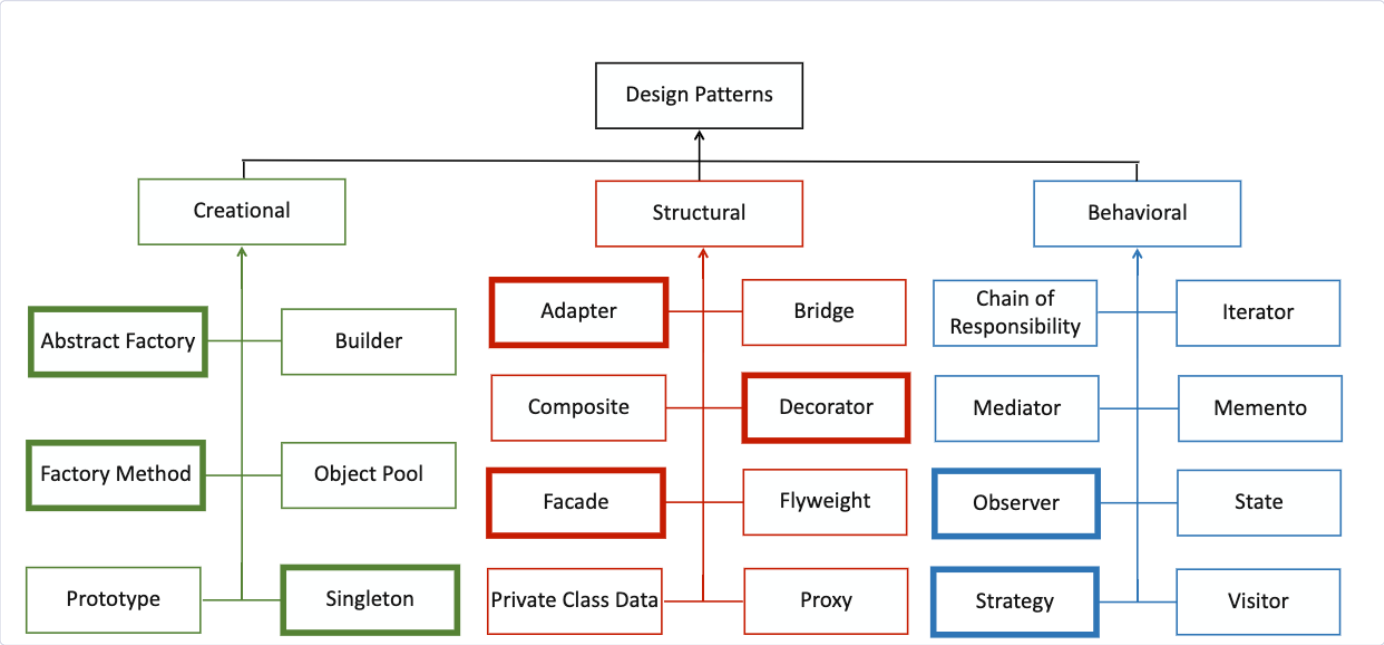
- **Single Responsibility principle**
- **Open closed principle**
- **Liskov substitution principle**
- **Interface segregation principle**
- **Dependency inversion principle**

Types of design patterns

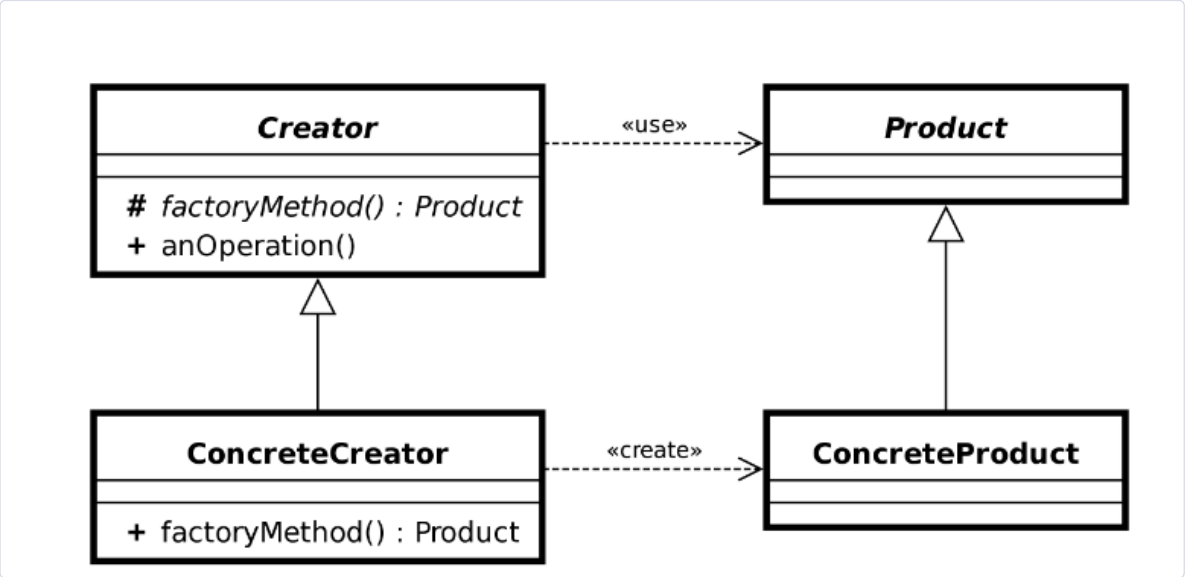
- **Creational**: Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- **Structural**: Ease the design by identifying a simple way to realize relationships between entities

- **Behavioral**: Identify common communication patterns between objects and realize these patterns

Design patterns hierarchy

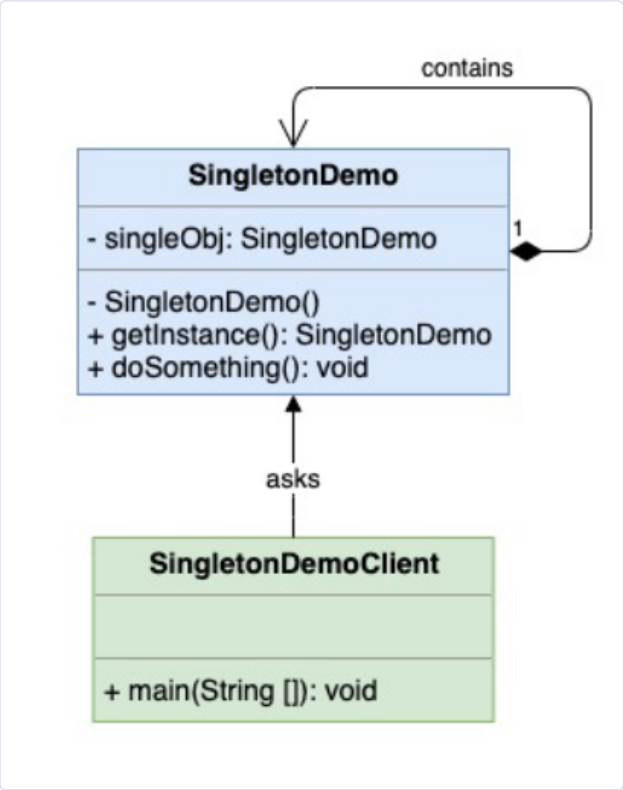


Factory Pattern



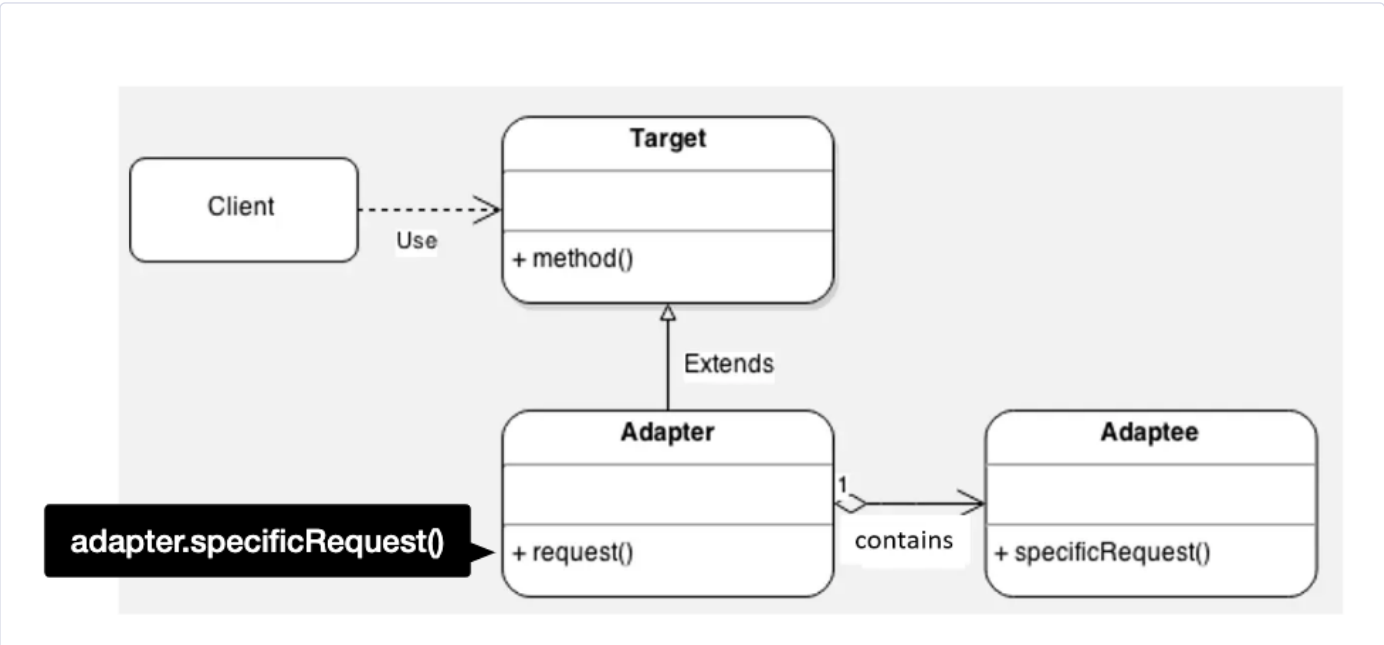
- Use when
  - The creation of objects should be independent of the system using them
  - Systems should be capable of using multiple families of objects / object
  - (Abstract Factory) Families of objects must be used together
  - Libraries must be published without exposing implementation details
  - Concrete classes should be decoupled from clients

Singleton Pattern



- Use when
  - Exactly one instance of a class is needed throughout system
  - Multiple instances would incur loading massive duplicate memory
  - Controlled access to a single object is needed

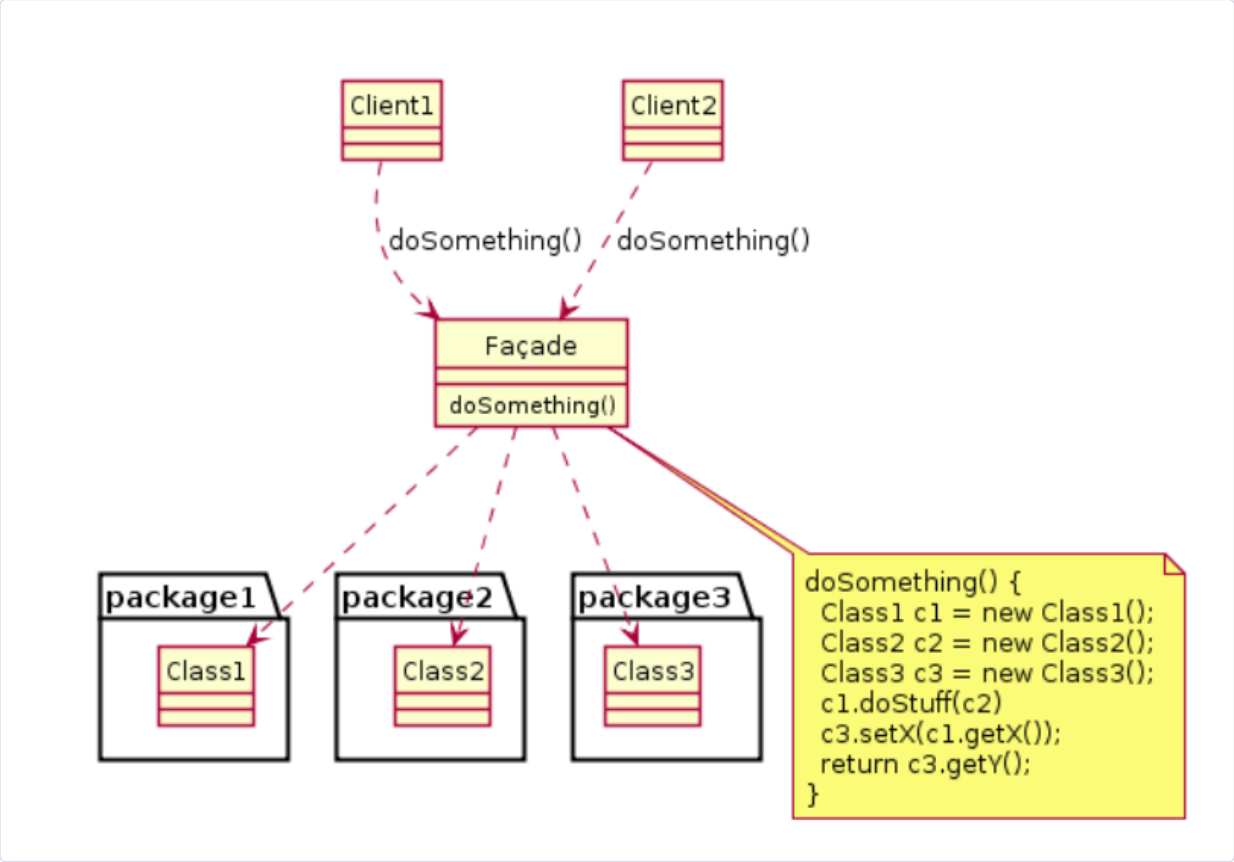
Adapter pattern



- Use when

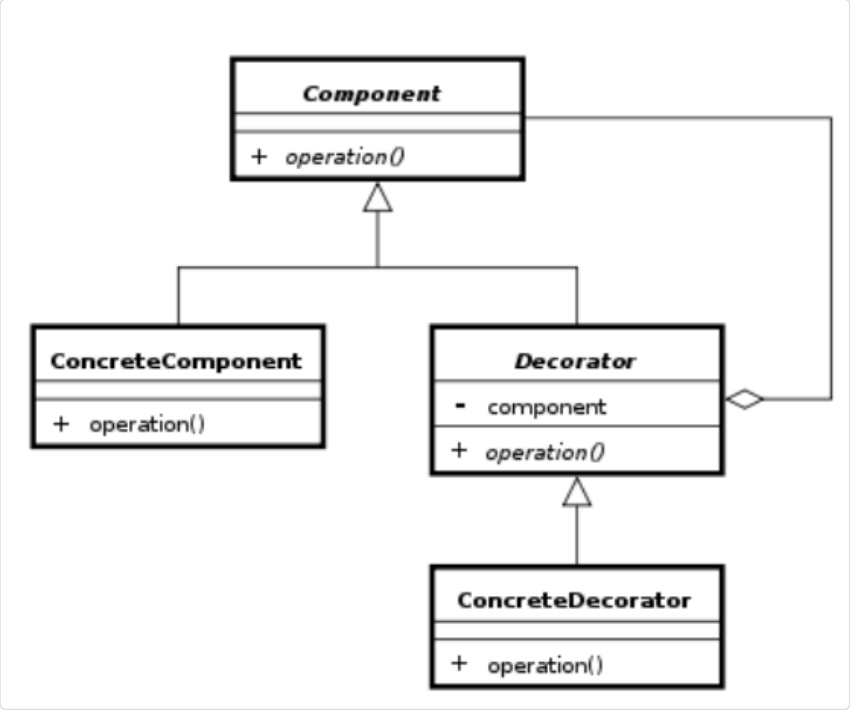
- Two or more classes or components in your system have different interfaces but need to work together
- Unable to use one class or component directly due to differences in method names, parameters, or return types
- If modifying them would lead to significant code changes or break functionality

Facade Pattern



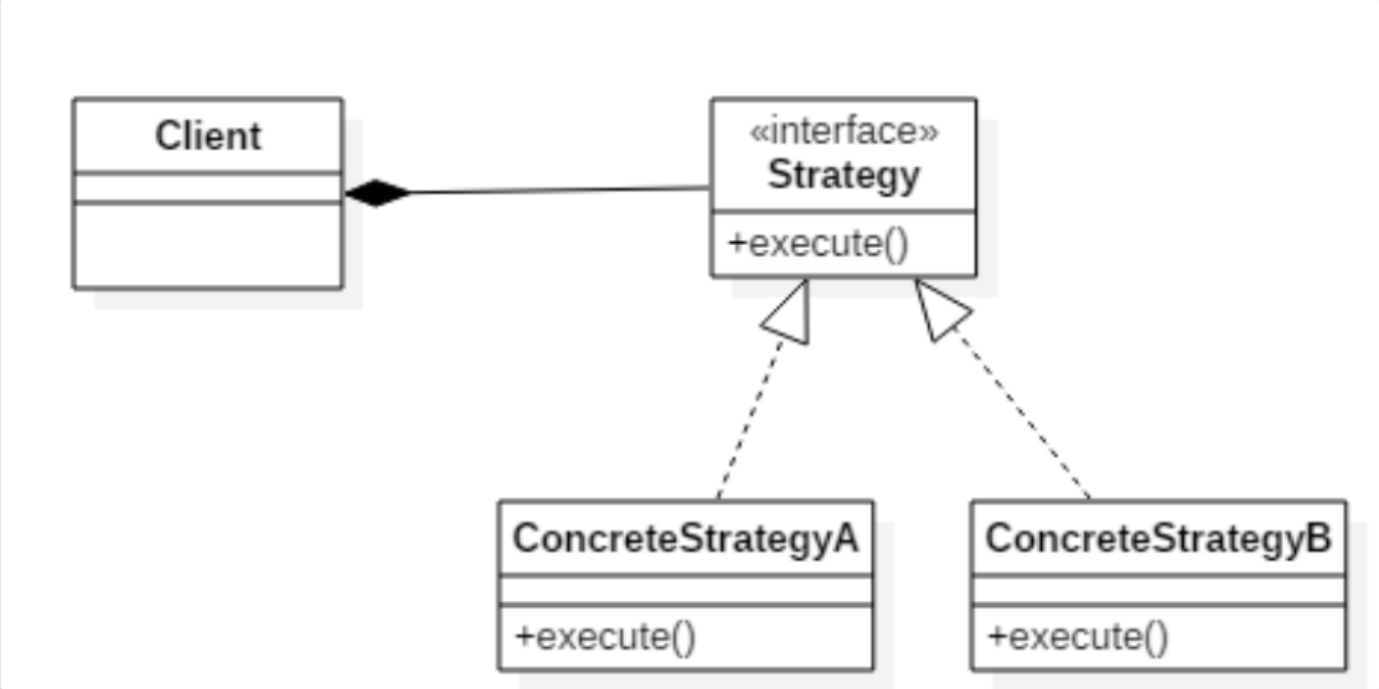
- Use when
  - A simple interface is needed to provide access to complex system
  - There are dependencies between the system implementations and clients
  - System and subsystems should be layered
  - System and subsystems should be layered

Decorator Pattern



- Add or modify of a class or function without modifying its source code
- They are optional features that can be added or removed from objects independently
- Whether there are multiple combinations of behavior that can be applied to the same object

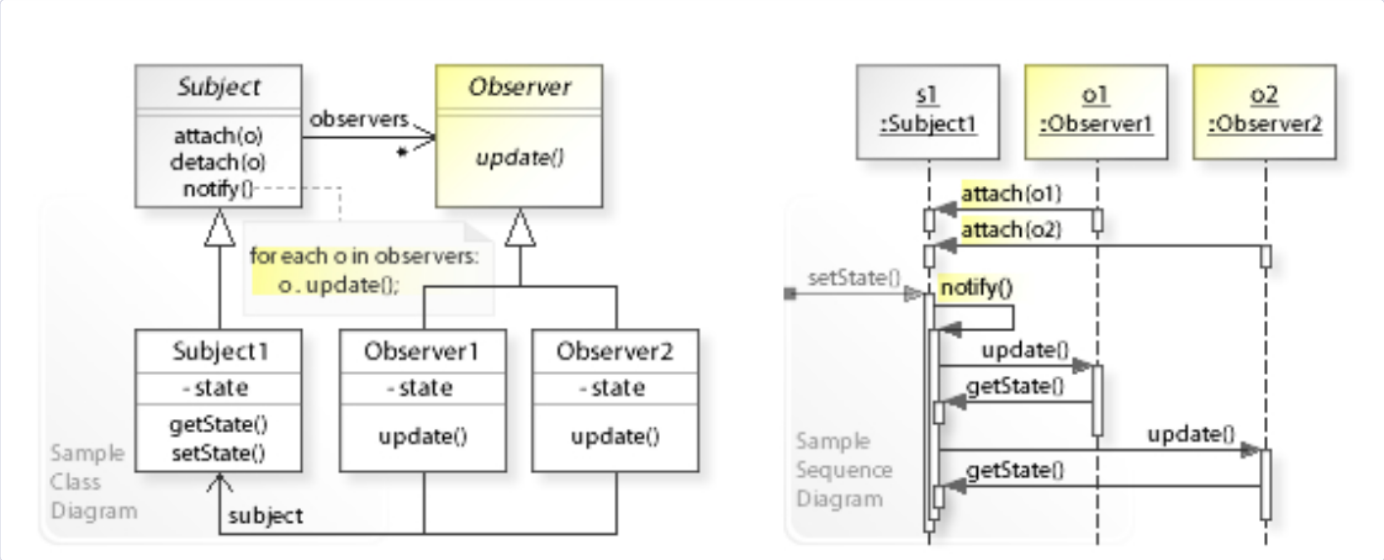
Strategy Pattern



- The only difference between many related classes is their behavior
- Multiple versions or variations of an algorithm are required
- Algorithm access or utilize data that should not be accessed outside the class

- The behavior of a class should be defined at runtime
- Conditional statements are complex and hard to maintain

Observer Pattern



- State changes in one or more objects should trigger behavior in others
- Broadcasting capabilities are required
- Objects should be able to determine the source of the notification

---

Software Implementation

- Process of converting a software design into a functioning software system
- Software implementation is not solely about coding a functional system, although this is obviously important, but other implementation issues are also crucial
  - Configuring management
  - Coding practices
  - Documentation
- They ensure the software implementation is high quality, facilitating collaborations among developers.

Code Smells

- Indicates a deeper problem within code
- fix it before commit when possible
- May apply to variable names
  - must be meaningful and consistent
  - no ambiguity

Long Method

- **Long method**
  - code is easier to write than it is to understand
  - More difficult it is to understand
- Most common techniques - Long Method
  - Extract method
  - Readability and communication improvement

Large Class

- **Large class**
  - Violates single responsibility
  - Lack of insight
- Most common techniques - Large Class
  - Extract class, extract subclass, extract interface
  - They avoid duplication

Long Parameter List

- **Long Parameter list**
  - might happen after several types of algorithms are merged into a single method
  - byproduct of making classes more independent
- Common refactoring techniques - Long Parameter List
  - if one parameter is the output of another method, call that method instead
  - If you are passing in many fields coming from the same object, pass the object instead
  - Replace related parameters with an object
  - Might introduce data class

Duplicate Code

- **Duplicated Code**
  - nearly the same or identical
  - copy / paste is cheap and easy
    - leads to duplication
    - laziness to remove duplication
    - technical debt

Switch Statements

- **Switch Statement**
  - Multiple conditions
  - Likely that there will be more conditions in the future -> long method

- Common Refactoring Techniques - Switch Statement
  - Extract method, replace type code with subclasses, replace conditionals with polymorphism
  - Code organization is improved

Comments

- **Comments**
  - If they are intended to explain something that cannot be easily understood by reading the code, the code might be smelly
  - Self documenting code
- Common Refactoring Techniques - Comments
  - Extract method
  - Split expressions
  - Rename variables

Code Styles

- Common format that has been shown to significantly reduce time it takes for dev's to understand a piece of code
- Not affect the functionality, but may assist the readability of the code
- Checking in unformatted code is a form of technical debt that can lead to increase in software cost

---