Software Design Pattern

Design:

- process of **transforming the problem into a solution**
- the **solution is also called design**

<u>**Two main aspects**</u>

Product/external design: designing the external behavior of the product to meet the users' requirements. Done by product designers.

Implementation/internal design: designing how the product will be implemented to meet the required external behavior. Done by software architects and software engineers.

Abstraction

- **Guiding principle:** Only consider details that are relevant to the current perspective or the task at hand.
- Large amounts of intricate details is impossible to deal with at the same time→Need abstraction!
- **Data abstraction**: abstracting away the lower level data items and thinking in terms of bigger entities
- **Control abstraction:** abstracting away details of the actual control flow to focus on tasks at a higher level. E.g., print("Hello") is an abstraction of the actual output mechanism within the computer.
- can be applied **repeatedly** to obtain progressively higher levels of abstraction.
- not limited to just data or control abstractions. [**general concept**]

<u>Examples</u>

- An **OOP class** is an abstraction over related **data** and **behaviors**.
- An **architecture** is a higher-level abstraction of the **design of a software**.
- **Models** (e.g., UML models) are abstractions of some aspect of **reality**.

Coupling: measure of the degree of dependence. High coupling (aka tight coupling or strong coupling) is **discouraged**

- **Maintenance is harder** because a change in one module could cause changes in other modules coupled to it (i.e. a ripple effect).
- **Integration is harder** because multiple components coupled with each other have to be integrated at the same time.
- **Testing and reuse of the module is harder** due to its dependence on other modules.

A is coupled to B if a change to B can **potentially (but not necessarily always)** require a change in A.

<u>**Example:**</u>

- A has access to the internal structure of B (this results in a **very high level** of coupling)
- A and B depend on the same global variable
- A calls B
- A receives an object of B as a para. or a return value
- A inherits from B
- <u>A and B are required to follow the same data format or communication protocol</u>

Cohesion: measure of how strongly-related and focused the various responsibilities of a component are. Higher cohesion is better. (keeps related functionalities together while <u>keeping out all other unrelated things</u>)

<u>Disadvantages of **low cohesion** (aka weak cohesion):</u>

- **Lowers the understandability** of modules (difficult to express functionalities at a higher level)
- **Lowers maintainability** because a module can be modified due to unrelated causes or many modules may need to be modified to achieve a small change in behavior.
- **Lowers reusability** of modules because they do not represent logical units of functionality.

Cohesion can be present in many forms.

- Code related to a single concept is kept together
- Code **invoked close together** in time is kept together
- Code manipulates same data structure is kept together

Multi-level design

- **Smaller system**: can be shown in one place.
- **Bigger system:** needs to be done/shown at multi-levels.

Top-down:

- Design the high-level design first
- Flesh out the lower levels later

Especially useful when designing big and novel systems where the **high-level design needs to be stable** before lower levels can be designed.

Bottom-up:

- Design lower level components first
- Put them together to create higher-level systems later.

Usually **Not Scalable for bigger systems**.

When:

- designing a variation of an existing system
- Re-purposing existing components to build new system.

Mix:

- Design the top levels using the top-down approach
- Use bottom-up approach when designing bottom levels

AddressBook

- Level2 has a single-level design.
- Level3 has a multi-level design.

Agile design

- Emergent, **not defined up front.**
- **Design will emerge over time**, evolving to fulfill new requirements and take advantage of new technologies as appropriate.
- Although you will often do some initial architectural modeling at the very beginning of a project, this will be **just enough to get your team going.**
- **Does not produce a fully documented set** of models before you may begin coding

Software Architecture (by software architect)

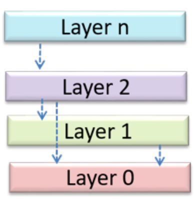
- consists of a set of interacting components
- forms the basis for the implementation

Architecture diagrams are **free-form** diagrams.

- **No** universally adopted **standard** notation
- **Any symbols** reasonable may be used
- **Minimize** the variety of symbols.
- **Avoid** the indiscriminate use of double-headed arrows to show interactions between components. [Some important will be no longer captured.]
- **Follow** various high-level styles (**architectural patterns**)
- Most applications use a **mix** of architectural styles.

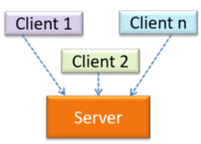
n-tier style:

- Higher layers make use of services provided by lower layers.
- Lower layers are **independent** of higher layers.

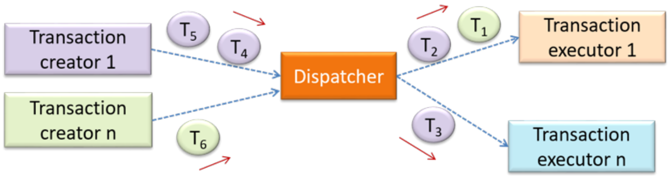


Client-server style has

- at least one component playing the role of a **server**
- at least one **client** component accessing the services of the server.



Transaction processing style: Divides the workload of the system down to a number of transactions → given to a dispatcher that controls the execution of each transaction. Task queuing, ordering, undo etc. are handled by the dispatcher.

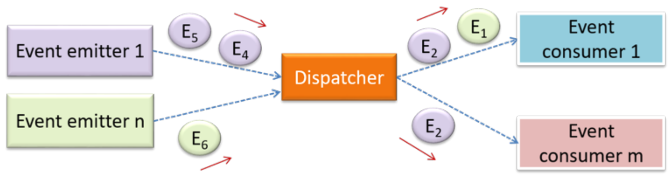


Service-oriented architecture (SOA) style

- Builds applications by combining functionalities packaged as programmatically accessible services.
- Aims to achieve interoperability between **distributed services**.
- May not even be implemented using the same programming language.

Event-driven style: Controls the flow of the application by detecting events from event emitters and communicating those events to interested event consumers.

This architectural style is often used in GUIs. [button clicked]



Design pattern: An **elegant reusable** solution to a commonly recurring problem within a given context in software design. A term popularized by the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software by the so-called "Gang of Four" (GoF)*

Format

- **Context:** The situation or scenario where the design problem is encountered.
- **Problem:** The main difficulty to be resolved.
- **Solution:** The core of the solution. It is important to note that the solution presented **only includes the most general details**, which may **need further refinement** for a specific context.
- **Anti-patterns (optional):** Commonly used solutions, which are **usually incorrect and/or inferior** to the Design Pattern.
- **Consequences (optional):** Identifying the pros and cons of applying the pattern.
- **Other useful information (optional):** Code examples, known uses, other related patterns, etc

Singleton pattern

- **Context:** Certain classes should have no more than just one instance These single instances are commonly known as **singletons.**
- **Problem:** A normal class can be instantiated multiple times by invoking the constructor.

Solution:

- Make the **constructor** of the singleton class **private**
- Provide a public class-level method to access the single instance.

```java
class Logic {
    private static Logic theOne = null;

    private Logic() {
        ...
    }

    public static Logic getInstance() {
        if (theOne == null) {
            theOne = new Logic();
        }
        return theOne;
    }
}
```

Pros:

- easy to apply
- effective in achieving its goal with minimal extra work
- provides an **easy way to access the singleton object** from anywhere in the codebase

Cons:

- The singleton object acts like a global variable that increases coupling across the codebase.
- In testing, it is **difficult to replace Singleton objects with stubs** (static methods cannot be overridden).
- In testing, singleton objects carry data from one test to another even when you want each test to be independent of the others.

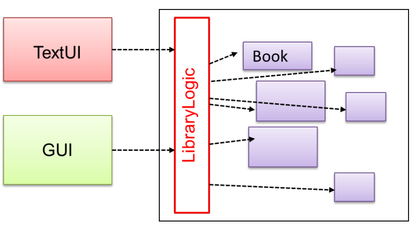
Facade pattern

Context: Components need to **access** functionality deep inside **other components**.

Problem: Access to the component should be allowed without exposing its internal details.

Solution:

Include a Façade class that sits **between** the component internals and users of the component such that **all access to the component happens through the Facade class.**



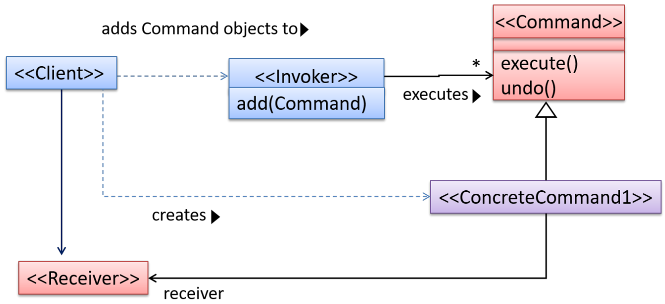
Command pattern

Context: Execute a number of commands, each doing a different task.

Problem: It is preferable that some part of the code executes these commands **without having to know each command type**.

Solution: Have a **general <<Command>> object** that can be passed around, stored, executed, etc without knowing the type of command (i.e. via **polymorphism**).

General Form



- The <<Client>> creates a <<ConcreteCommand>> object passes it to the <<Invoker>>
- <<Invoker>> object treats all commands as a general <<Command>> type.
- <<Invoker>> issues a request by calling execute() on the command
- If a command is undoable, <<ConcreteCommand>> will store the state for undoing the command prior to invoking execute().
- <<ConcreteCommand>> object may have to be linked to any <<Receiver>> of the command before it is passed to the <<Invoker>>.

Model view controller (MVC) pattern

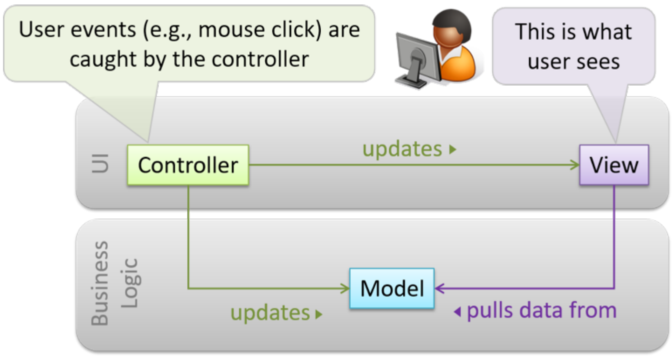
Context: Most applications support storage/retrieval of information, displaying of information to the user (often via multiple UIs having different formats), and changing stored information based on external inputs.

Problem: The high coupling that can result from the interlinked nature of the features described above.

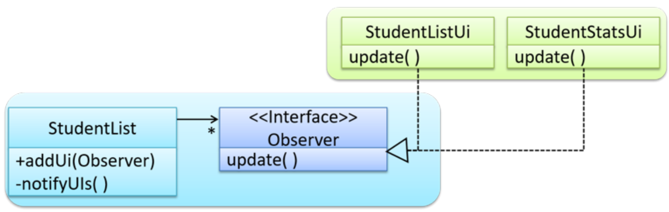
Solution: **Decouple** data, presentation, and control logic of an application by separating them into three different components: **Model, View** and **Controller**.

- **View: Displays** data, **interacts** with the user, and **pulls** data from the model if necessary.
- **Controller**: **Detects** UI events such as mouse clicks and button **pushes**, and takes follow up **action**. **Updates/changes** the model/view when necessary.
- **Model: Stores** and **maintains** data. **Updates** the view if necessary.

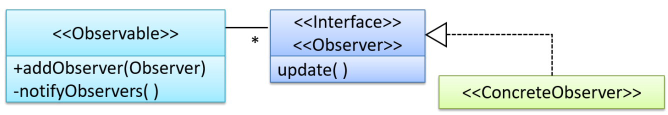
Typically, the **UI** is the combination of **View** and **Controller.**
Note that in a simple UI where there's only one view, Controller and View can be combined as one class.



Observer pattern [**e.g., JavaFX**]
Context: An object (possibly more than one) is interested in being **notified** when a change happens to another object.
Problem: The 'observed' object does not want to be coupled to objects that are 'observing' it.
Solution: Force the communication through an **interface** known to both parties.



General Form



- <<Observer>> is an interface: any class that implements it can observe an <<Observable>>. Any number of <<Observer>> objects can observe (i.e. listen to changes of) the <<Observable>> object.
- The <<Observable>> maintains a list of <<Observer>>
- Whenever there is a change in the <<Observable>>, the notifyObservers() operation is called that will call the update() operation of all <<Observer>>s in the list.