Design Patterns Notes

**Separate the aspects that are varying from the ones that remain the same.**

Let A be a class from which varying aspects are being removed.

**Program to an interface and not the implementation.**

This means “Program to a supertype” 🡪 actual runtime object isn’t locked in the class we want to be flexible.

The declared type of variable should be of supertype, so that the object assigned to them can be any concrete implementation of supertype.

Class declaring them needn’t know about actual object type.

get/set () helps to assign object at runtime (rather than concrete instantiation. new A () better to do getA ()).

Setters help the concrete class to change the value of instance variable dynamically. Thus “*PROGRAM TO INTERFACE”* is helpful.

This increases the reusability.

(Encapsulate Family of Algorithms to a class/interface.)

If two classes are used with **HAS-A** relationship, then it is ***COMPOSITION.***

Instead of inheriting, one class is **composed** of other class (probably methods).

**Favor Composition over Inheritance.**

It helps encapsulating the *family of Algorithms* and makes the code more dynamic.

***STRATEGY PATTERN***

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithms vary independently from clients using them.

A **shared pattern vocabulary** helpful b/w the developers to help easy communication and better for thinking at pattern level.

* When sharing info with pattern name, one intrinsically talks about its characteristics and constraints.
* Precise design just by name.
* Design level abstraction (avoid implementation details).

Using common interface means using a common method (say update ()) to update whatever is varying.

**OBSERVER PATTERN:**

Subject and Observer.

Subject is the object that is going to change.

Observer will be notified whenever the subject is changed.

***“The observer pattern defines one-to-many dependency between the objects so that when one object changes state, all of its dependent objects are notified and updated automatically.”***

Subject and Observer interfaces.

Concrete subject will implement register, remove, notify methods.

Observer will implement update and will interact with concrete subject.

*If two objects are* ***loosely coupled,*** *they can interact with each other but have little knowledge of each other.*

Observer pattern 🡪 observers are loosely coupled.

Because only thing subject needs to know that it implements observer interface.

Add observer anytime.

Never need to modify subject when observer modified.

Subject and observer can be used independently.

Change won’t affect other.

**Strive for loosely coupled designs between objects that interact.**

**Inheritance VS Composition:**

Inheritance leads to static behavior to subclasses and the same behavior for all the subclasses. With Composition, it can do it dynamically and vary the behavior.

With composition, additional functionality can be added which was not thought during the implementation without modifying the super class.

Since there is very minimal or no change in existing code, the chance of buggy code is reduced.  
  
**Classes should be open for extension but closed for modification.**

The open-closed principle.

Applying “open-closed principle” everywhere can be cumbersome and may make the code hard to understand.

**DECORATOR PATTERN:**

🡪 Decorators have the same supertype as the object they decorate.

🡪One or more decorator can be used to wrap the object.

🡪 Since the decorator and the object it wraps have same supertype, the decorators can be passed in place of object.

🡪The decorator adds its behavior either before or/and after the delegation to do its job.

🡪Objects can be decorated anytime. Thus, can be done dynamically.

***“The Decorator Pattern attaches additional responsibilities to the object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”***

Each decorator HAS-A association with component. i.e. it holds an instance variable that references the component.

The decorator can extend the state of component and/or add new behavior to the component.

For *type matching,* decorators go for inheritance not for getting behavior. It is vital to have same type.

New behavior is acquired by composition rather than inheriting it. Thus, get dynamic behavior rather than static as in the case of inheritance.

This way no change in existing code.

If the application relies on concrete component, then the decorator will break the code. If the application is coded against the abstract component type, the decorator will work well.

🡪 Lots of small classes are added to design that makes code hard to understand.

🡪Adding code dependent on specific type causes problem.

🡪 To get the behavior of decorator, the object has to be wrapped with decorator class.

Using Factory (Object) to instantiate the objects.

The factory object can be a static Method too. In that case, it is called Static Factory. The disadvantage is that you cannot subclass and change the behavior of the method.

Generic term: *Implement an interface can mean implementation of abstract methods from supertype.*

Abstract method of a class helps in decoupling. If a method (say A) is running a getting an Object of Abstract class, A doesn’t know what concrete class Object it is getting and performing operations on. Thus, A is decoupled.

*A factory method handles the object creation and encapsulates it in subclass. This decouples the client code in abstract superclass from the object creation in subclass.*

Factory method is abstract so that subclass must handle accordingly.

Factory method decouple the client code in superclass to not know about its implementation.

**FACTORY METHOD PATTERN:**

Encapsulates object creation by letting subclass decide what object to create.

Creator class has factory method that creates the product. (Abstract creator and factory methods).

Product class is the object created.

***“The factory method pattern defines an interface for object creation, but lets the subclass decide what class to instantiate. Factory method defers the class instantiation to subclasses.”***

🡪 Apart from factory method, creator class has method that operate on the product.

**Depend upon abstractions. Do not depend upon concrete class. (dependency inversion principle).**

Our high-level component should not depend on our low-level component. They both should depend on abstraction.

The *“inversion”* usage is because the way we think about Object oriented design. Both the components depend upon the abstraction.

**Guidelines to avoid OO design which violate “*Design inversion principle”:***

* No variable should hold reference to concrete class.
* No class should derive from concrete class.
* No method should override an implemented method of any of its base classes.

The above rules should be strived for rather followed all the time.

Abstract Factory gives us interface for creating family of products.

By writing the code which uses abstract factory, we decouple our code from concrete factory.

***“An Abstract Factory pattern provides an interface for creating families of related or dependent object without specifying their concrete class.”***

* Factory method creates object through **inheritance** whereas Abstract factory pattern uses **composition.**

**SINGLETON PATTERN**

***“A Singleton pattern ensures a class has only one instance and provides a global point of access to it.”***

* Ensure synchronization in multi-threaded paradigm. For example, in JAVA, use of *`synchronized` and `volatile*` to ensure this.

**COMMAND PATTERN**

***“A Command pattern encapsulates the request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and supports undo operations”***

Null Object: This helps the user to avoid the null condition check. The Object is not meaningful and avoids unnecessary errors in case of unassigned values.

Macro command pattern where we have an array of commands in a command class that implements the command interface.

Command pattern can be used in Job-queueing, logging requests.

**ADAPTER PATTERN**

***“The Adapter pattern converts the interface of a class into another interface the client expects. Adapter lets the classes work together that couldn’t otherwise because of incompatible interfaces.”***

Good OO principle:

* Adapter is composed with adaptee which allows us to use it with any subclass of the adaptee.
* Adapter binds to adaptee interface, (NOT IMPLEMENTATION!!)

Two kinds of Adapters:

* Object Adapters: Composition to implement the adaptee.
* Class Adapters: Multiple inheritance. Adapter subclasses Target and Adaptee.
* Class adapters can override the behavior of adaptee class. No need to instantiate target and adaptee class.
* Object adapters are flexible. Delegate the code to adaptee.

One can have imperfect adapters as there might be methods in target class that are not supported by adaptee.

**FAÇADE PATTERN**

Simplifies the interface and decouples client code with subsystem components.

Same subsystem can have many façades depending upon client’s need.

The subsystem’s functionality can be still accessed if needed.

***“The Facade pattern provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that make the subsystem easy to use.”***

**Talk only to your immediate friends. (Principle of least knowledge).**

Invoke only those methods that belong to:

* The object itself.
* Objects passed in the parameters.
* Any object method creates/instantiates.
* Any component of object.

**TEMPLATE METHOD PATTERN**

* The algorithm resides in superclass and is declared “final”.
* The template method provides the framework where other subclasses with similar algorithm can be added just by implementing the abstract steps.

The template method defines the steps of an algorithm and allows subclasses to provide the implementations of one or more steps.

***“The Template method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template method lets the subclasses redefine some of the steps of an algorithm without changing the structure of the algorithm.”***

**Hooks** can be used as a step of algorithm with a default behavior which can be overridden in subclasses according to the need.

**Don’t call us, we’ll call you. (The Hollywood principle).**

The high-level component has the control over the algorithm. This avoids the circular dependency in the applications.

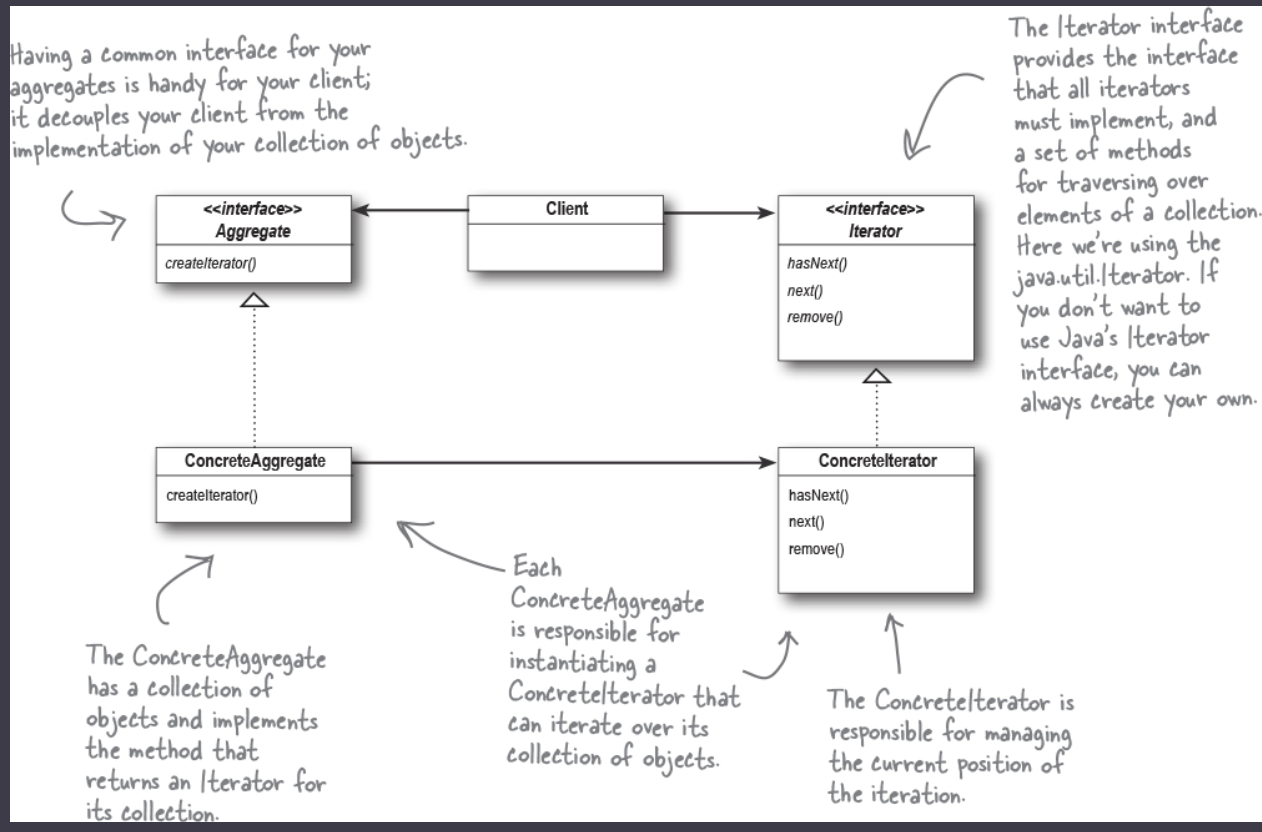
A method can be defined static and the template method pattern can be used by this adjustment.(trade-off)

**ITERATOR AND COMPOSITE PATTERN**

Iterator Interface: { hasNext(), next(), [remove()] }.

***“The iterator pattern provides a way to access elements of an aggregate object without exposing the underlying representation.”***

- places the task of iteration on iterator rather than on aggregate, which simplifies the aggregate interface (see C9/A2/menu).



**A class should have only one reason to change. (Single Responsibility principle).**

Every responsibility of class is an area of potential change. More than one responsibility means more than one area of change.

**Cohesion:** how closely a class or module supports SRP.

High cohesion – designed around a set of related functions.

Low cohesion – designed around a set of unrelated functions.

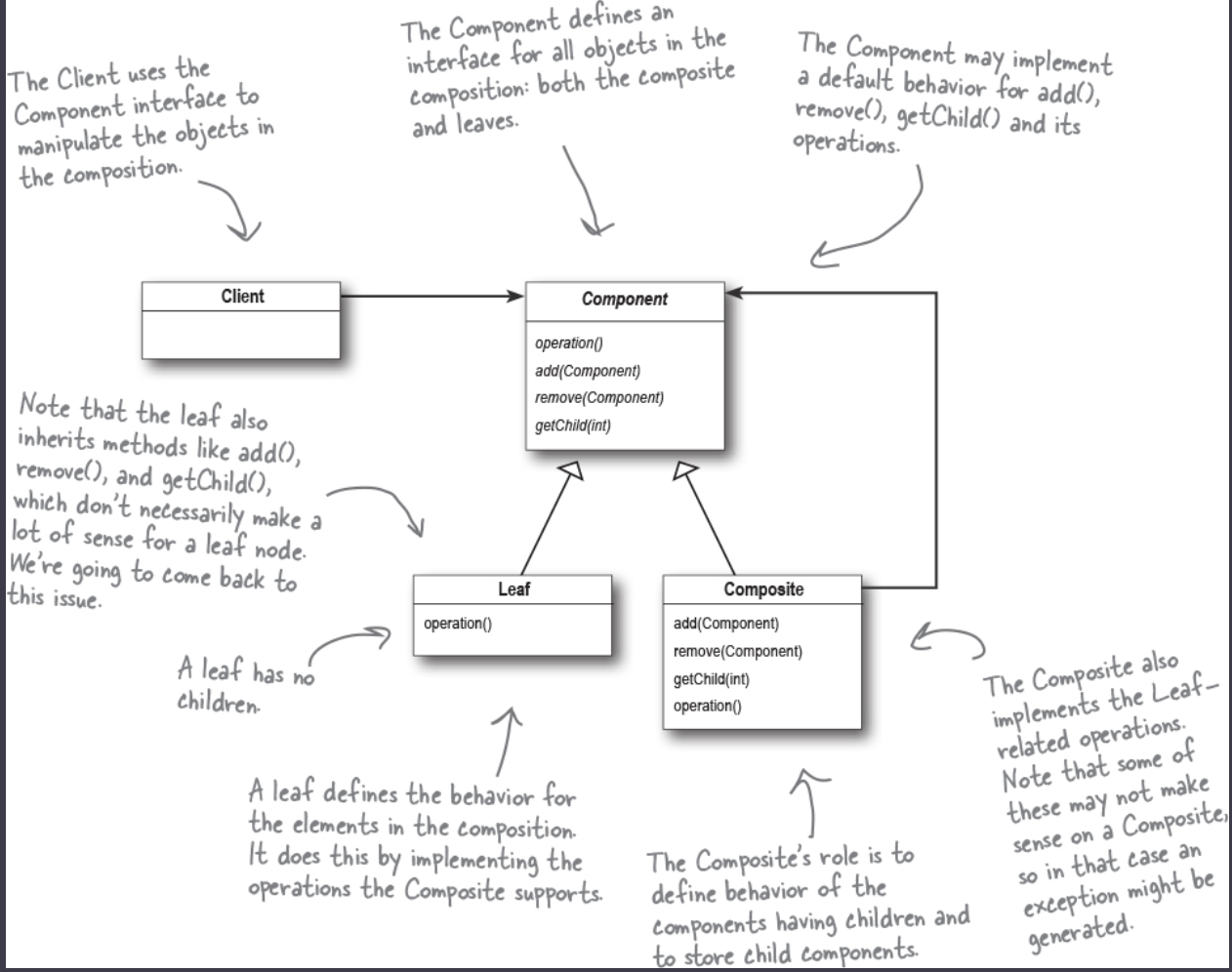
By providing the iterator, we can make the code using it decoupled, thus extensible.

What if the object on which we are iterating itself has a sub-collection???? Answer lies in:

**COMPOSITE PATTERN**

***“The composite pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.”***

Part-whole hierarchy: the tree is made of parts but can be treated as whole. i.e. node can be an element(leaf) or a pointer representing another collection of elements. But at top-level, it is treated as whole.



**Tradeoff between Single Responsibility and Transparency**

Transparency: With component being extended by both leaf and composite, client can treat them uniformly. i.e. both leaf and composite are transparent to client.

To have transparency, we tradeoff single responsibility.

**STATE PATTERN**

Use of state diagrams; similarity with “strategy pattern”.

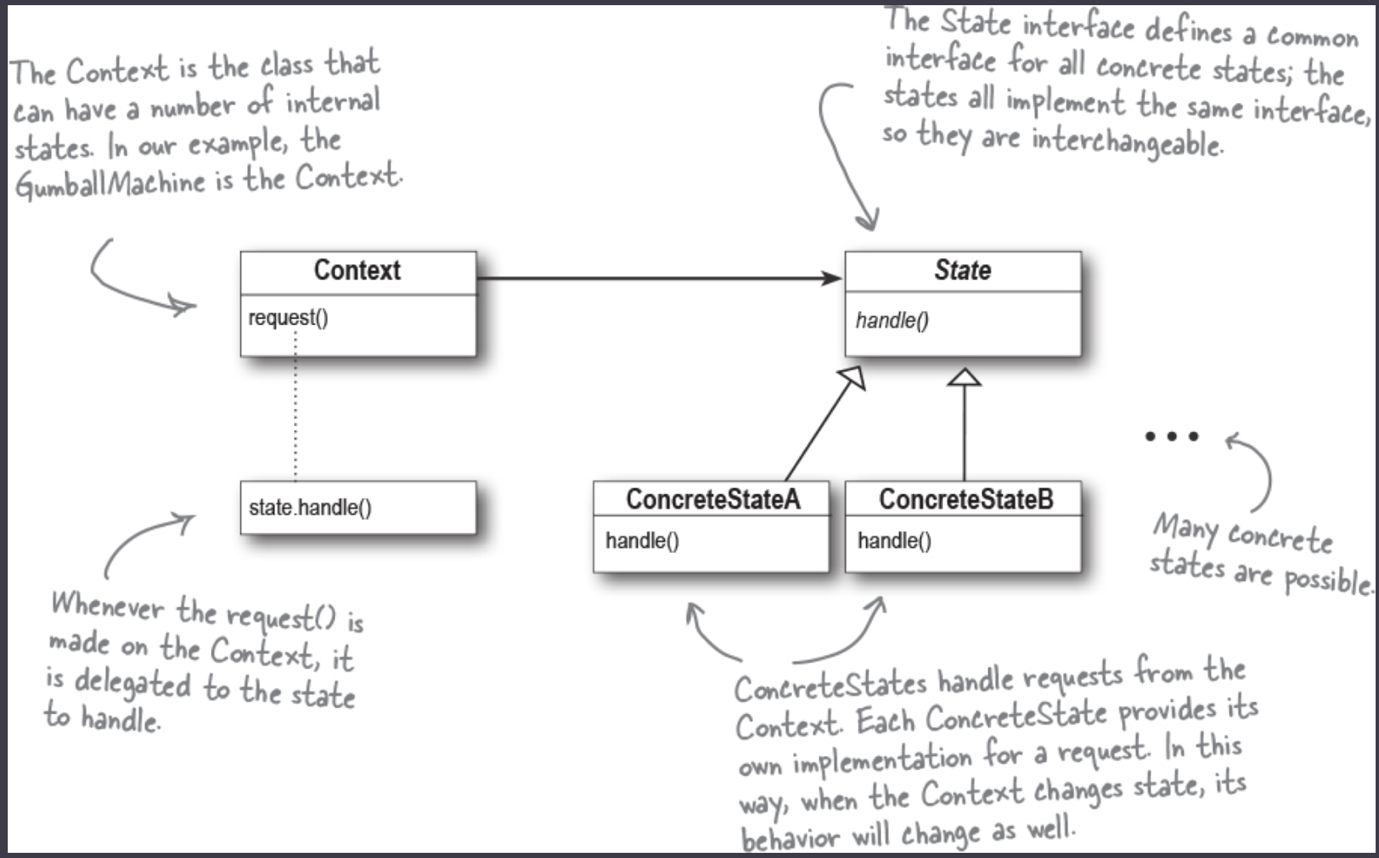
Implementing State M/c:

* Gather up the states
* Instance variable to hold the states(can enumerate them.)
* Gather up the actions.
* Class to represent state machine and for each action, have a method.

Here the varying part is state, which needs encapsulation.

***“The State pattern allows an object to alter its behavior when its internal state changes. The Object will appear to change its class”.***

Internally, we change the behavior of the object by composition by referring different state objects, which “appear to change the class”.



**State vs Strategy:**

* In state pattern, the state of the object changes over time based on request and internal states, while in Strategy, client generally provides the object the context is composed of.
* Strategy is flexible alternative to subclassing, while State is alternative to conditionals in the context. Encapsulating behaviors in a state can help change the behavior of object by simply changing the state of the context.

There can be multiple instances of context within application which can share states, provided the states don’t keep their own internal context. (State as static instance variable).

State patterns increase number of internal classes and avoids inflexible monolithic conditionals.

**PROXY PATTERN**

Proxy object in the local heap on which client will call the remote methods. The proxy pattern will handle the network call and return the required value(s).

***“The proxy pattern provides a surrogate or placeholder for another object to control access to it.”***

**COMPOUND PATTERN**

Model-View-Controller: patterns working together

🡪 User interacts on view.

🡪 View tells controller about the interaction

🡪 Controller asks model to change the state wrt. Interaction.

🡪 Model changes the state and notifies the change to view(observer).

🡪 View may ask for further details from model to change the Interface for user.

Controller interprets the input of view and modifies the model accordingly.

🡪 If controller logic is added to view => view has **TWO** responsibilities.

* View is tightly coupled with the model. Bad for reusability.

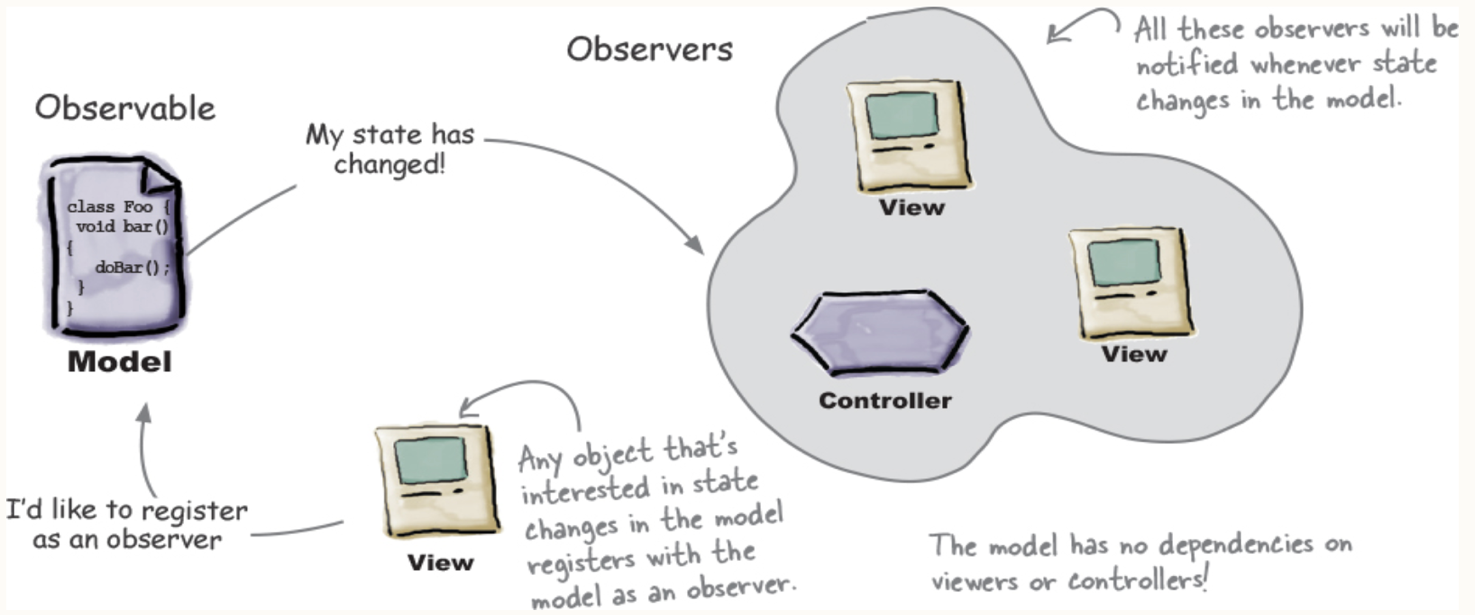
Patterns used in MVC (generally)

**Model:** Observer to notify the controller and view.

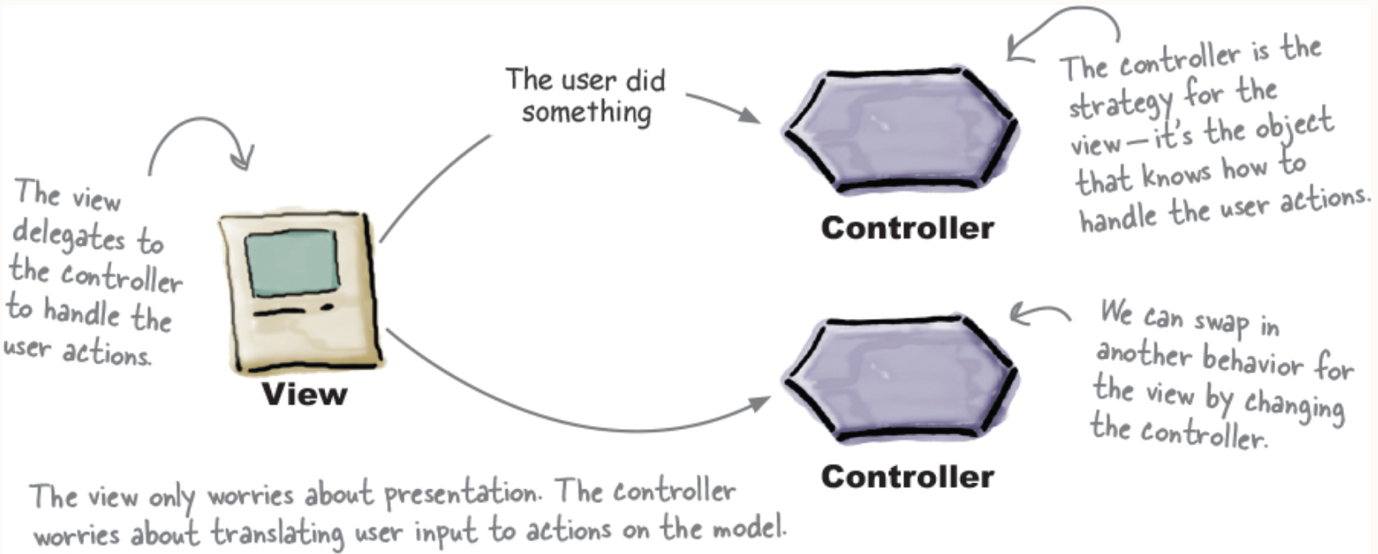
**View and Controller:** Strategy, controller can be replaced to other to change the be behavior of the system. View delegates the actions to controller to handle what should be the output. View just takes care of presentation.

*Internally, View also uses* ***Composite Pattern*** *for buttons, window, and other properties of display.*

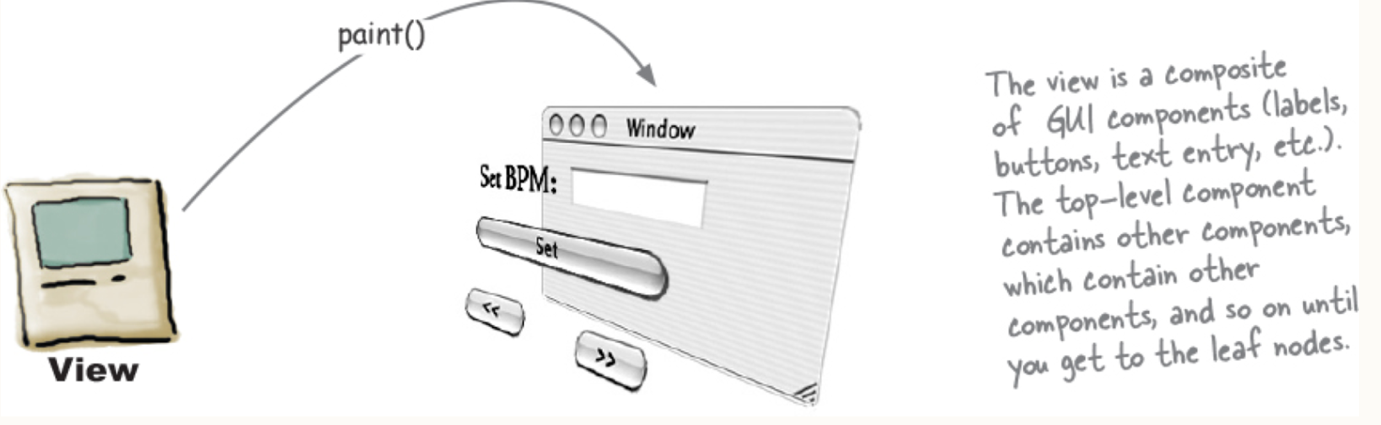
Observer



Strategy



Composite



***“A Pattern is a solution to a problem in context.”***

🡪 Context: situation in which a pattern applies. The situation should be **recurring.**

🡪 Problem: goal that needs to be achieved along with constraints. (also called forces)

🡪 Solution: general design to be applied.

**Pattern Catalog:**

🡪 Name Classification

🡪 Intent

🡪 Motivation

🡪Applicability

🡪 Structure (Class Diagram)

🡪 Participants (Classes and objects and their responsibility)

🡪 Collaboration (of participants)

🡪 Consequences

🡪Implementation

🡪 Uses

🡪 Related Pattern

CLASSIFICATIONS:

🡪 Creational: involve Object creation and all provide a way to decouple client from the object it needs to instantiate.

* Factory, Abstract Factory, Singleton, “BUILDER, PROTOTYPE”

🡪 Behavioral: is concerned with how classes and objects interact and distribute responsibility.

* Strategy, Template, Command, Iterator, State, Observer, “VISITOR, MEDIATOR, MEMENTO, INTERPRETER, CHAIN OF RESPONSIBILITY”

🡪 Structural: compose classes or objects into larger structures.

* Composite, Adapter, Façade, Proxy, Decorator, “BRIDGE, FLYWEIGHT”

🡪 Class pattern: relationship b/w class inheritance. Patterns are established during compile time.

* Template, Factory, Adapter, Interpreter.

🡪 Object Pattern: relationship b/w objects and primarily via composition. Dynamic and flexible.

***“An Anti-pattern tells how to go from a problem to a bad solution”***

🡪 Anti-patterns look good in short-term but are bad in long term

🡪Documenting them help us avoid them and help us find related design pattern for good solution.