Software Design Patterns

Overview

- Basics of design patterns
- Types
- Different design patterns with non software and software examples

Definitions

Christopher Alexander defined a pattern as

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

Components required for patterns:

- Name
- Purpose, the problem it solves
- Way to accomplish this
- Constraints and forces we have to consider in order to accomplish it

Why study design patterns?

Because design patterns-

- Help reuse existing high quality solutions to commonly occurring problems
- Improve individual and team learning
- Shift level of thinking to a high perspective
- Illustrate basic object-oriented principles
- Improves modifiability and maintainability of the code
- Improve team communication and individual learning

Types of patterns:

The Gang of Four are the four authors of the book « Design Patterns: Elements of Reusable Object-Oriented Software »

• Defined 23 design patterns for recurrent design issues, called GoF design patterns

- Creational
- Behavioral
- Structural

THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory
S	Adapter
S	Bridge
C	Builder
B	Chain of Responsibility
B	Command
S	Composite
S	Decorator
S	Facade
C	Factory Method
S	Flyweight
B	Interpreter
B	Iterator
B	Mediator
B	Memento
C	Prototype
S	Proxy
B	Observer
C	Singleton
B	State
B	Strategy
B	Template Method
B	Visitor

Facade pattern

- Provide a unified interface to a set of interfaces in a subsystem.

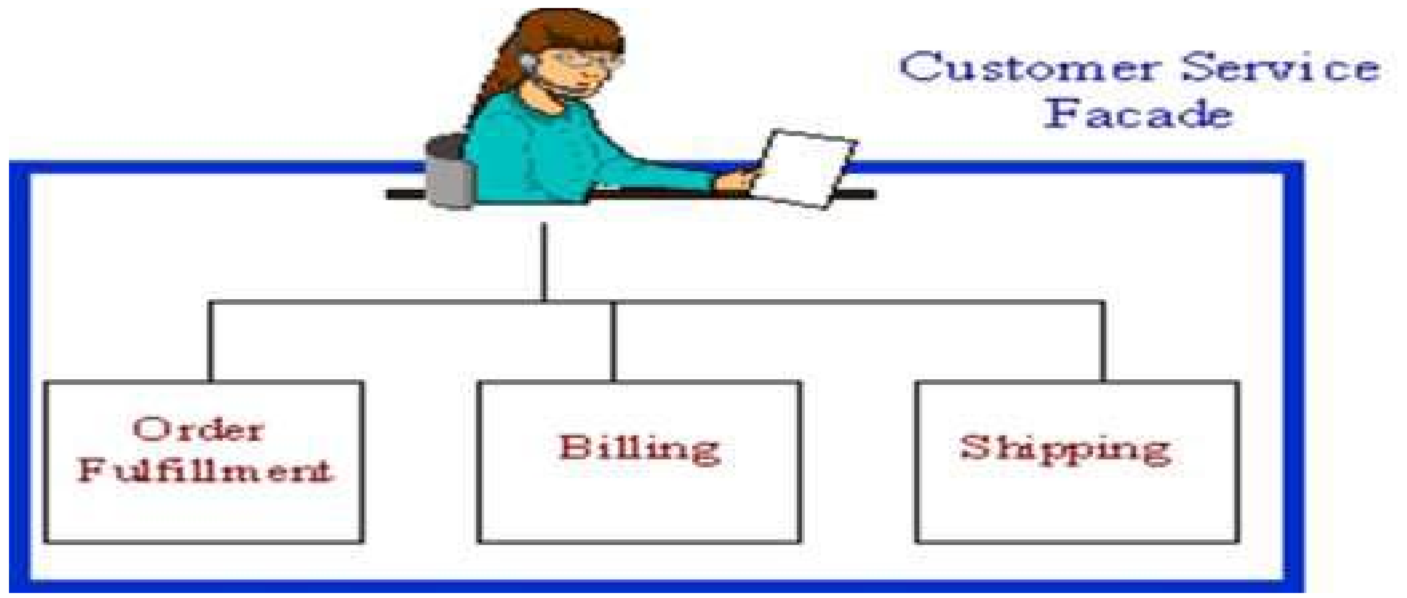
- Defines a higher-level interface that makes the subsystem easier to use.

Non-software example for facade

The customer service representative acts as a *Facade,* providing an interface to the order fulfillment department, the billing department,

and the shipping department.

Object Diagram for facade using Phone Order



Software example

In temple run game the “upgrade” option acts as a facade, giving an interface for selecting a player, for abilities and powerups.

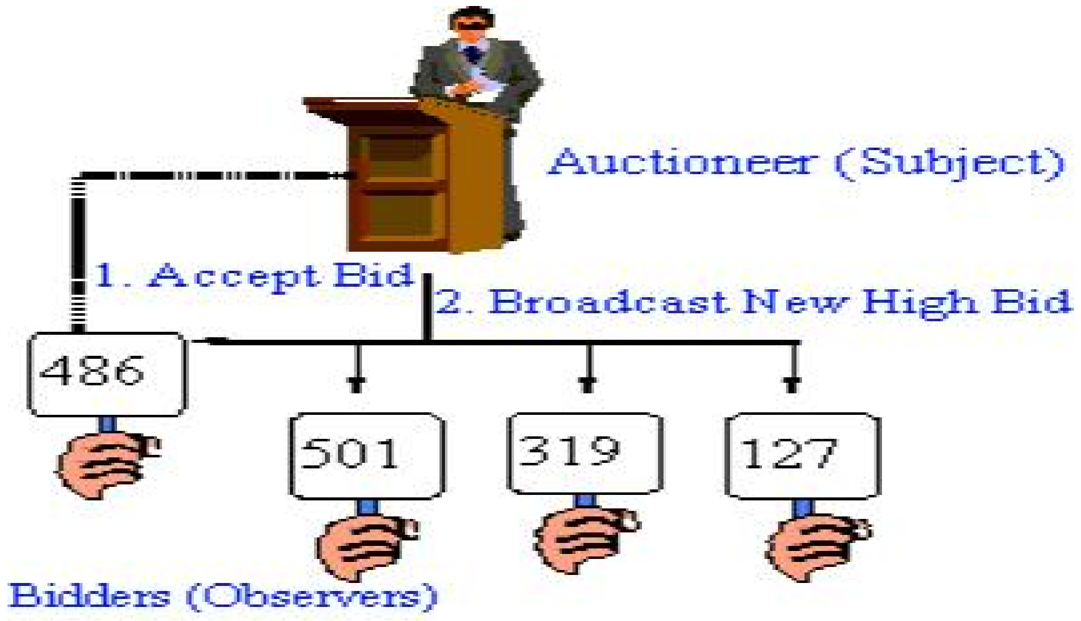
Observer pattern

- Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.

- Observers delegate the responsibility for monitoring for an event to a central object(the subject) so that it can notify a varying list of objects that an event has occurred.

Non-software example

- The following figure shows an Auction example of an observer
- The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid.
- The acceptance of the bid changes the bid price, which is broadcast to all of the bidders in the form of a new bid.



Software example

In temple run game ,when the player completes certain objective(s) following needs to be updated/notified automatically-

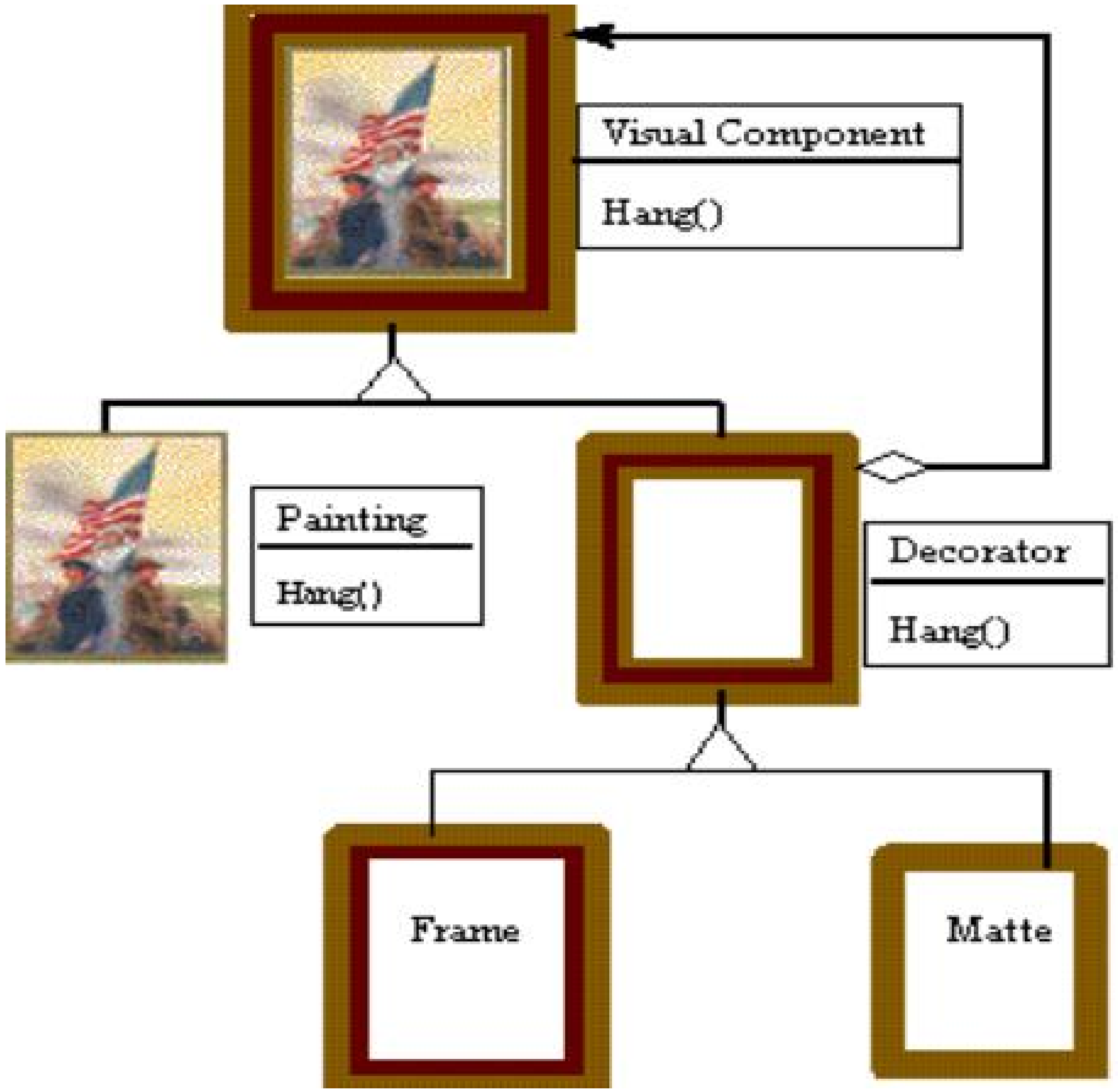
1. Current list(list of objectives to be achieved)
2. Complete list(list of objectives completed)
3. Increase multiplier
4. Increase score

Decorator pattern

- Attach additional responsibilities to an object dynamically keeping the same interface.
- Decorators provide a flexible alternative to subclassing for extending functionality.
- The object that we want to use does the basic function.
- However, we need to add some additional functionality occurring before or after the object’s base function.
- Allows extension of functionalities without resorting subclassing.

Non-software example

- The Decorator attaches additional responsibilities to an object dynamically.
- Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall.
- Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.


Visual Component
Hang()
Painting
Hang()
Decorator
Hang()
Frame
Matte

Software example

- In temple run game, run is a basic object.

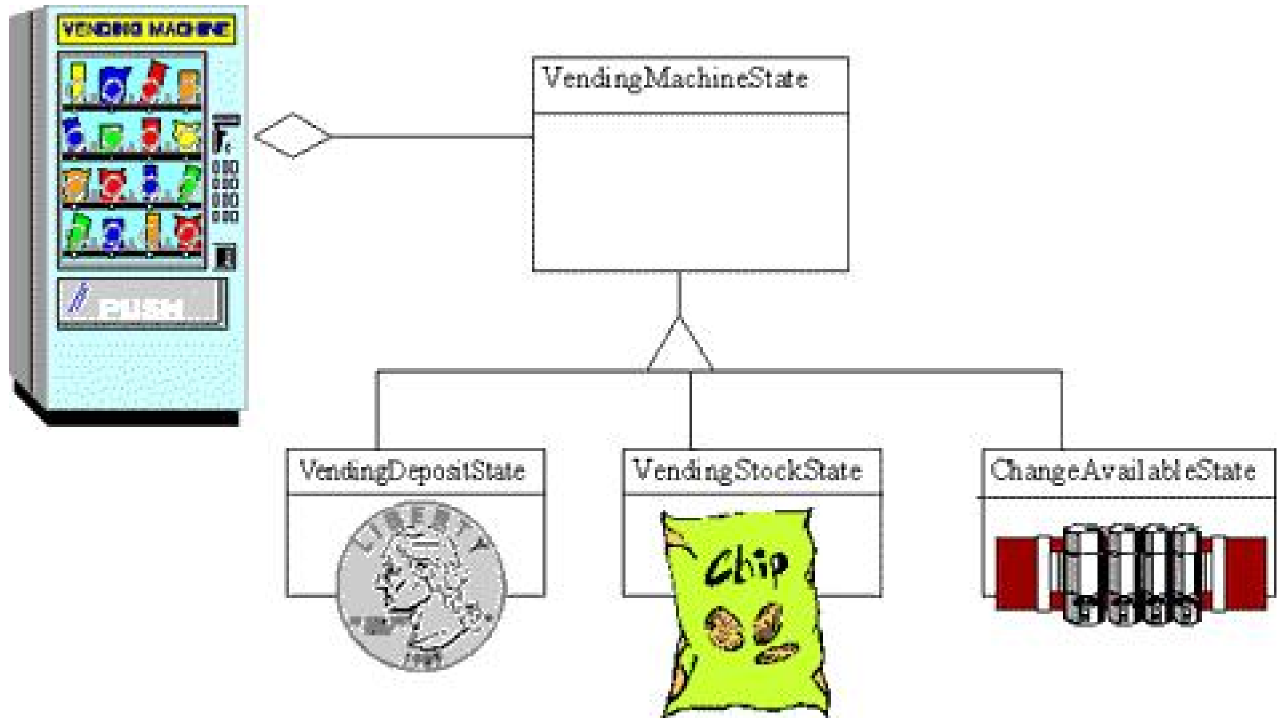
- The player can simply go on running without taking any powerups(boost, shield,coin magnet).

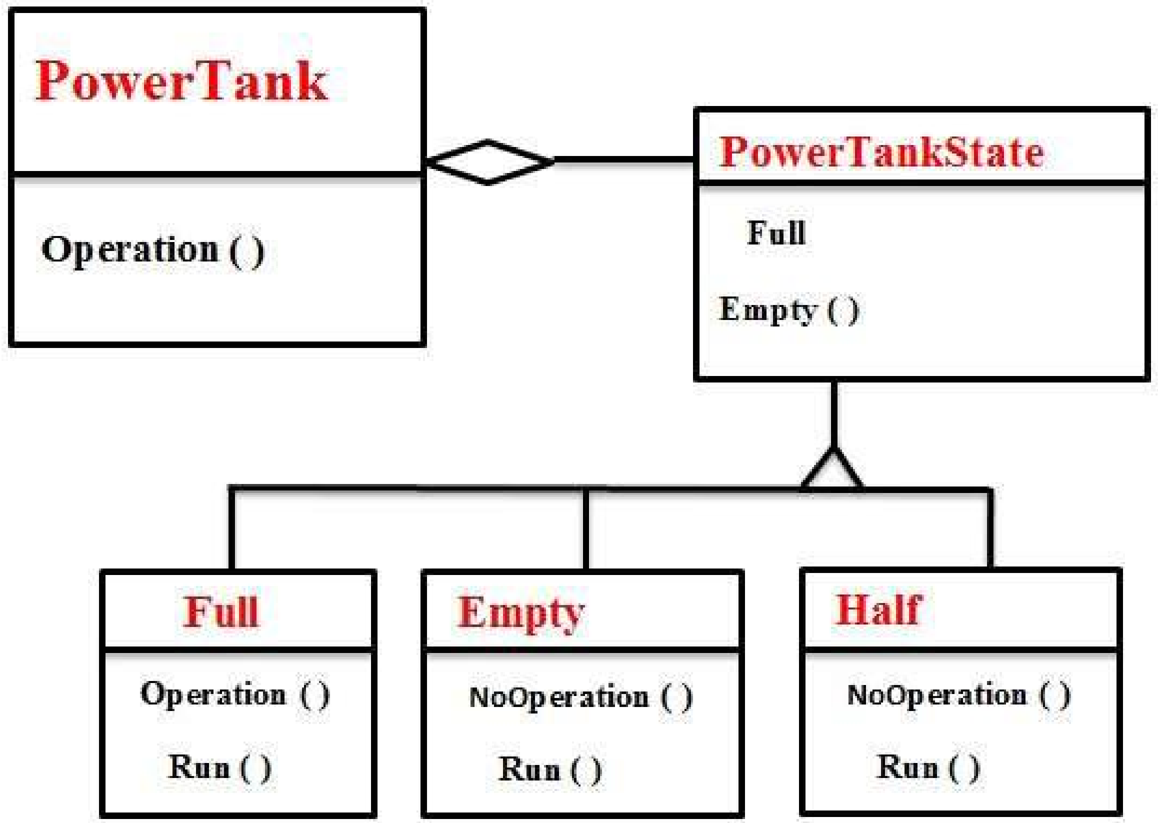
- Additional functionality like Boost(runs faster),shield(protects from fire n other obstacles),coin magnet(attracts the coins automatically while running) will be added if the player takes any of the above mentined powerups which will be appearing on the screen while running.

State pattern

- Allow an object to alter its behavior when its internal state changes.
- The object will appear to change its class.

- This pattern can be observed in a vending machine.
- Vending machines have states based on the inventory, amount of currency deposited, the ability to make change,the item selected, etc.
- When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change,deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion


VENDING MACHINE
PUSH
VendingMachineState
VendingDepositState
VendingStockState
ChangeAvailableState
LIBERTY
Chip


PowerTank
Operation ( )
PowerTankState
Full
Empty ( )
Full
Operation ( )
Run ( )
Empty
NoOperation ( )
Run ( )
Half
NoOperation ( )
Run ( )

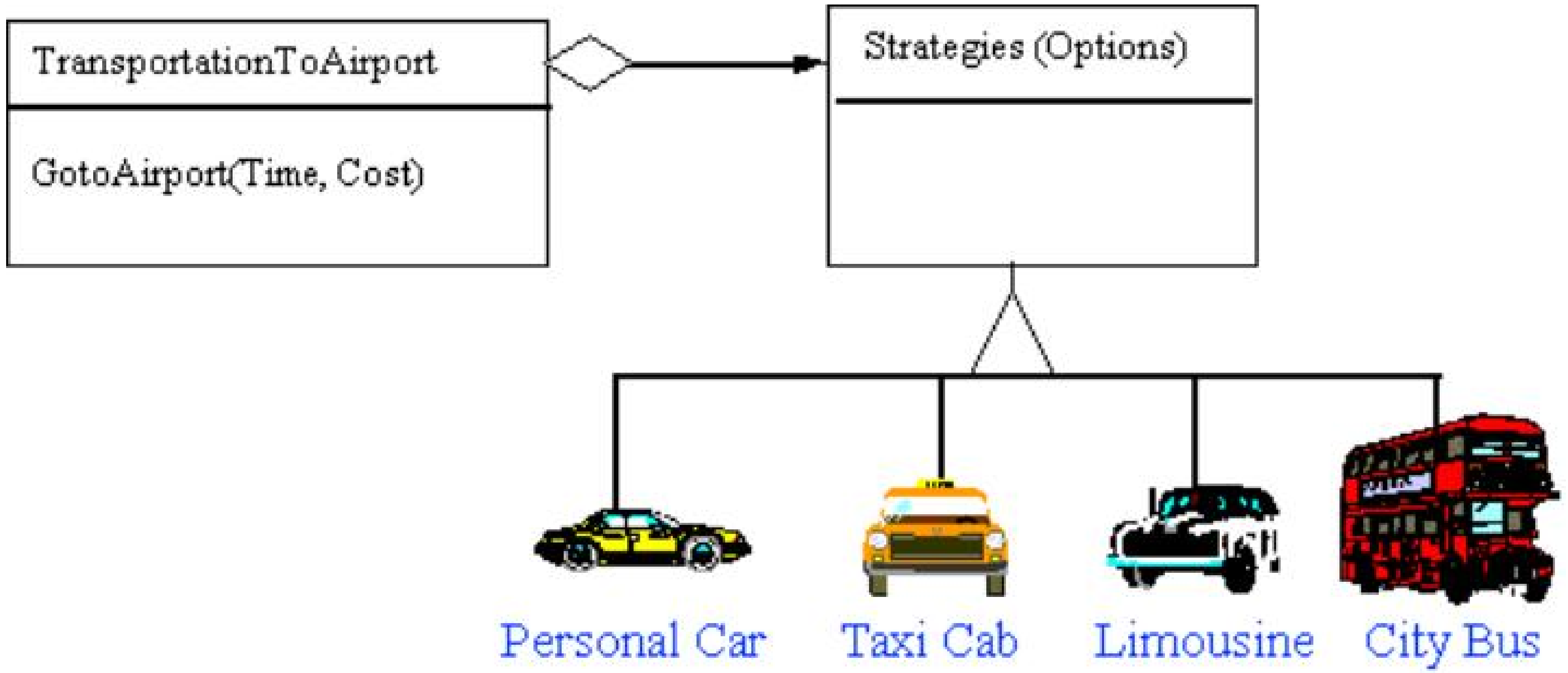
Strategy pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- It enables us to use different rules or algorithms depending on the context in which they occur.
- Separates the selection of algorithms from the implementation of the

  algorithm.

Non-software example

- Modes of transportation to an airport is an example of a Strategy.
- Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably.
- The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.


TransportationToAirport
GotoAirport(Time, Cost)
Strategies (Options)
Personal Car
Taxi Cab
Limousine
City Bus

software example

- In temple run game, when a use double clicks on himself the following changes happen:-
- If the power tank is not filled-no changes
- If the power tank is filled ,one of the following happens until the tank gets emptied

1. If Powerup=boost, engages boost instantly
2. If Powerup=score bonus, then instant 500 point

bonus

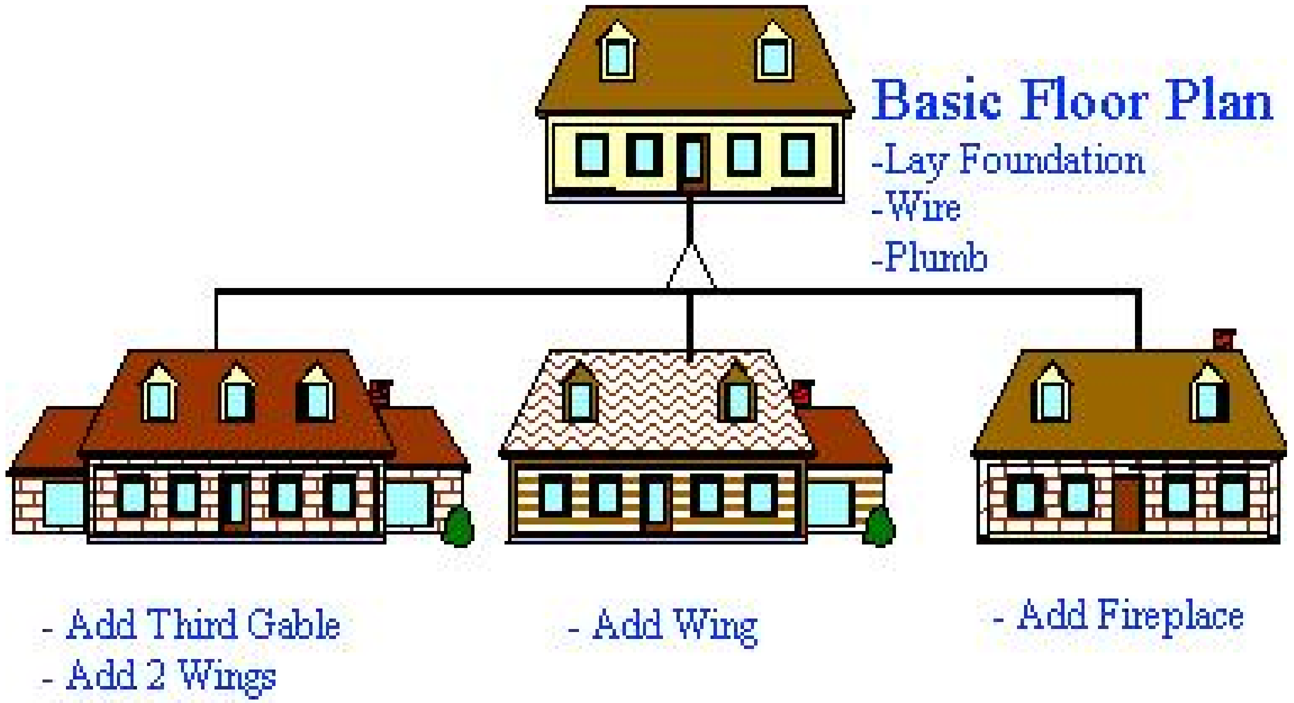
3. If Powerup=coin magnet, then engages coin magnet instantly
4. If Powerup=shield, then engages shield instantly

Template method pattern

- The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses.
- Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- Allow definition of subsets that vary while maintaining a consistent basic process

Non-software example

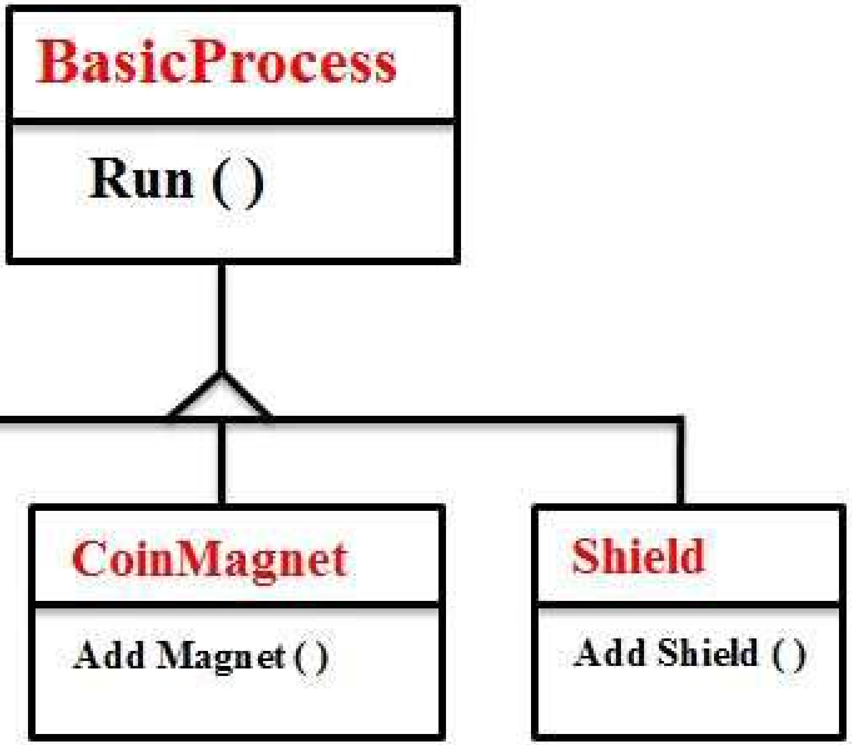
- Home builders use the Template Method when developing a new subdivision.
- A typical subdivision consists of a limited number of floor plans, with different variations available for each floor plan.
- Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house.
- Variation is introduced in the latter stages of construction to produce a wider variety of models.



Variations added to Template Floor Plan

Software example

I



Iterator pattern

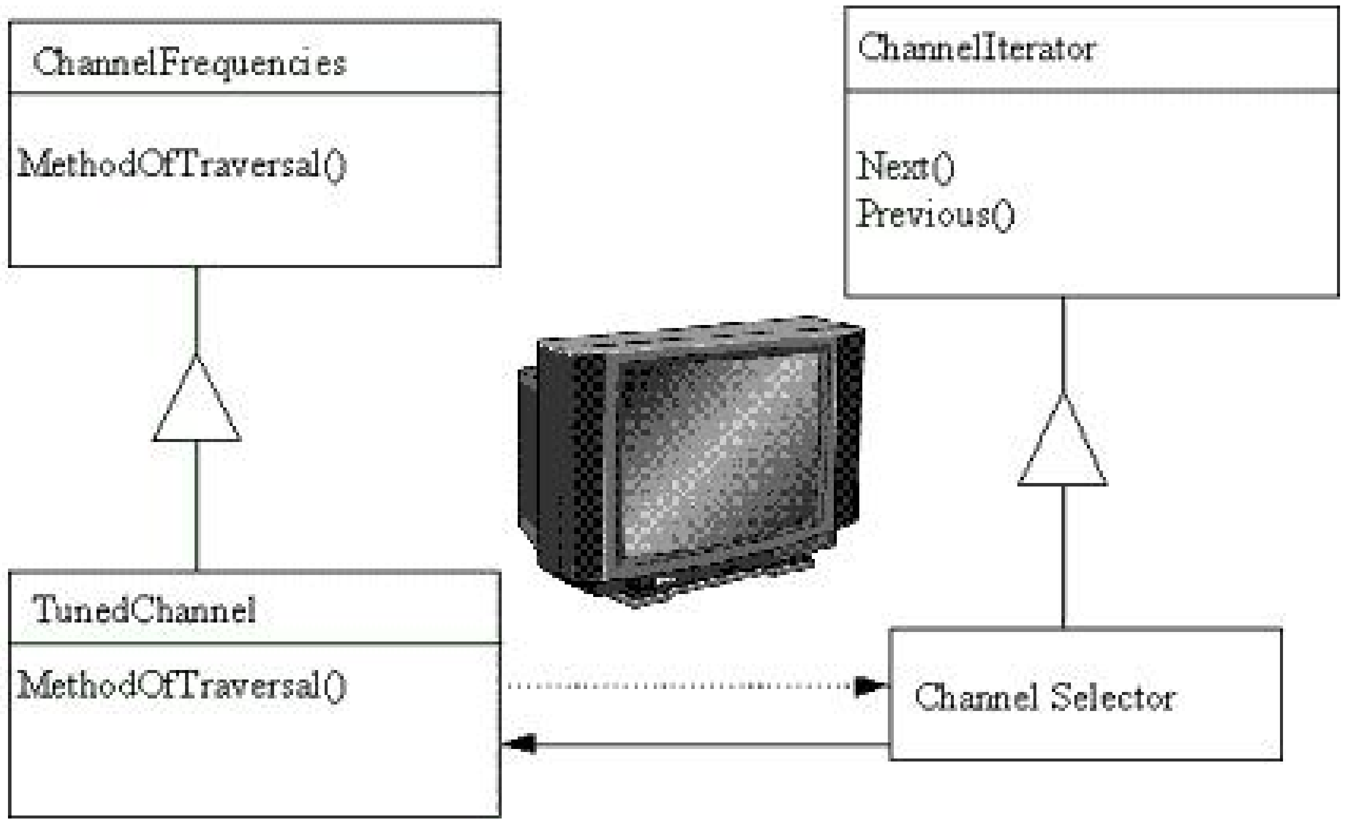
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Example:

Consider watching television in a hotel room in a strange city.

- When the viewer selects the "next“ button, the next tuned channel will be displayed.
- When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its

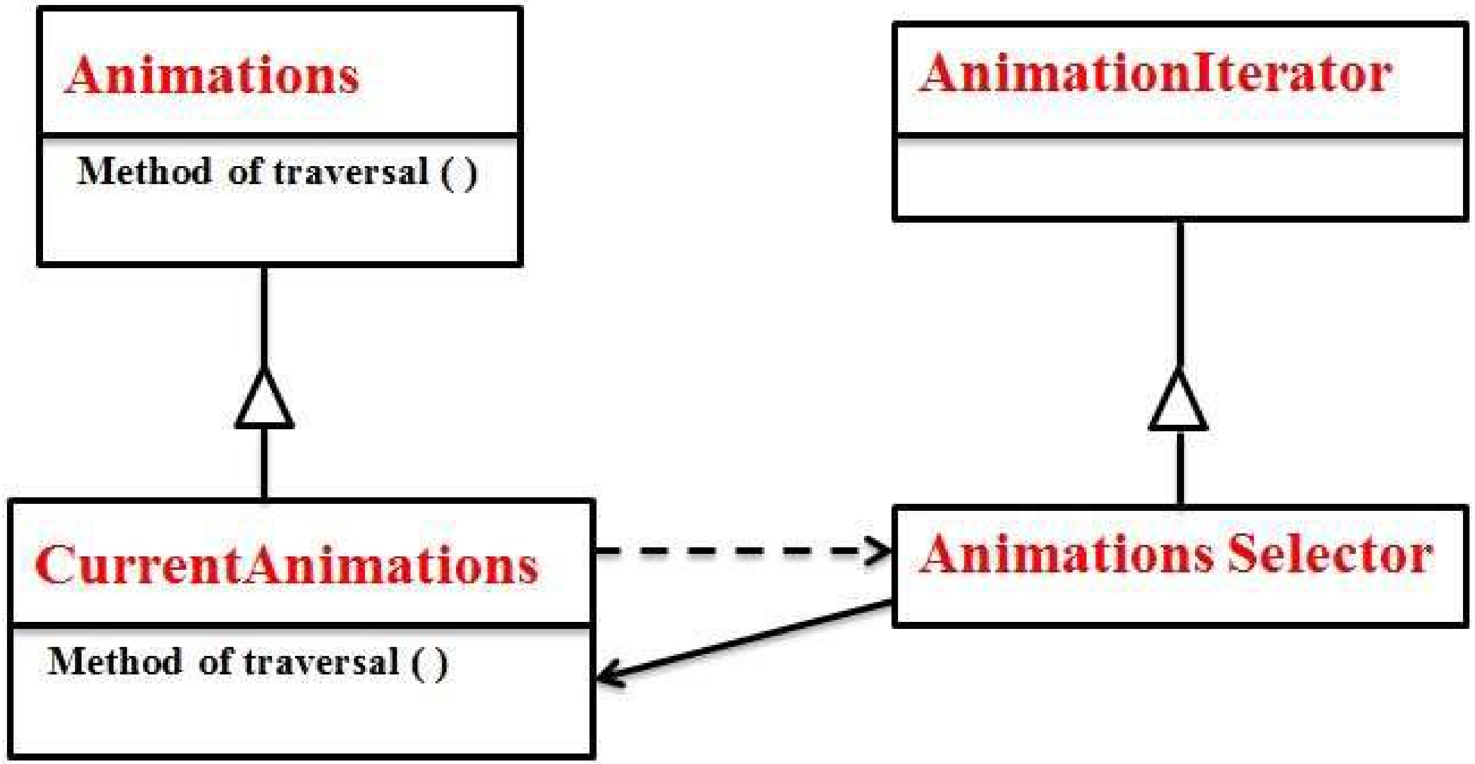
  number.


ChannelFrequencies
MethodOfTraversal()
ChannelIterator
Next()
Previous()
TunedChannel
MethodOfTraversal()
Channel Selector

Software example

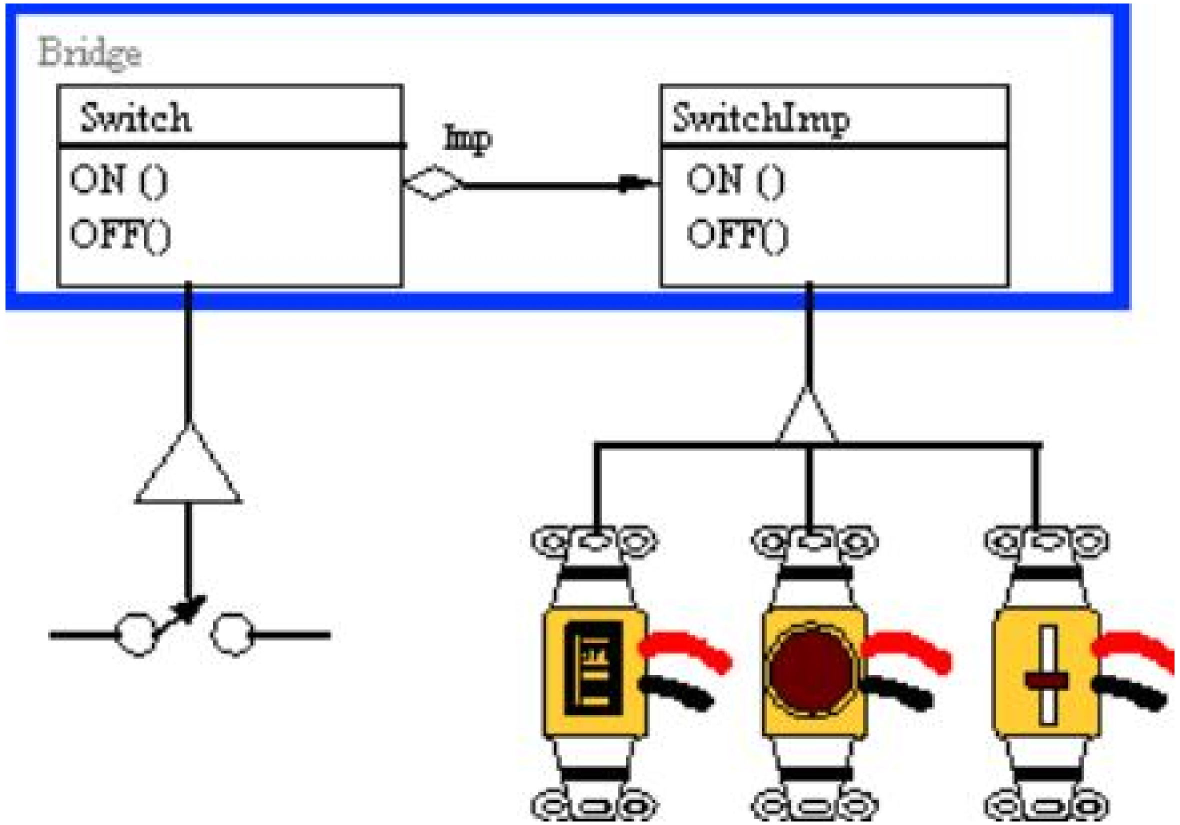
- In temple run game, the player has to run on the animated path which is designed in such a way that the different animations are repeated by selecting them randomly.
- Animation selector selects one of the following in an iterative manner throughout the game.

1. Normal rock road on mountains
2. Rope
3. wooden bridge
4. broken bridge
5. Tunnel
6. water falls
7. fallen trees and other obstacles


Animations
Method of traversal ( )
AnimationIterator
CurrentAnimations
Method of traversal ( )
Animations Selector

Bridge pattern

- The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.
- Example: A household switch controlling lights, ceiling fans, etc. is an example of the Bridge.
- The purpose of the switch is to turn a device on or off.
- The actual switch can be implemented as a pull chain, a simple two position switch, or a variety of dimmer switches.


Bridge
Switch
ON ()
OFF()
Imp
SwitchImp
ON ()
OFF()

Software example

- In temple run game we can have following options under the caption "play"
- 1. Swipe
- 2. Slide
- 3. Tilt
- 4. Duck
- 5. Double tap
- Using bridge pattern we are switching between one of the above operation without even knowing the actual implementation.

2030
0

81679
308
Swipe To Duck

Singleton pattern

- Ensure a class has only one instance, and provide a global point of access to it.

Example:

- The office of the President of the United States is a Singleton.
- The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession.
- As a result, there can be at most one active president at any given time.
- Regardless of the personal identity of the active president, the title, "The President of the United
- States" is a global point of access that identifies the person in the office.

Creational Patterns:

- Creational patterns abstract the instantiation process.
- They help to make a system independent of how its objects are created, composed, and represented
- Creational patterns for classes use inheritance to vary the class that is instantiated.
- Creational patterns for objects delegate instantiation to another object.

Examples:

- Factory
- Singleton
- Builder
- Prototype

Structural patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Structural class patterns use inheritance to compose interfaces or implementations.
- Structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.

Examples:

- Adapter
- Proxy
- Bridge
- Composite

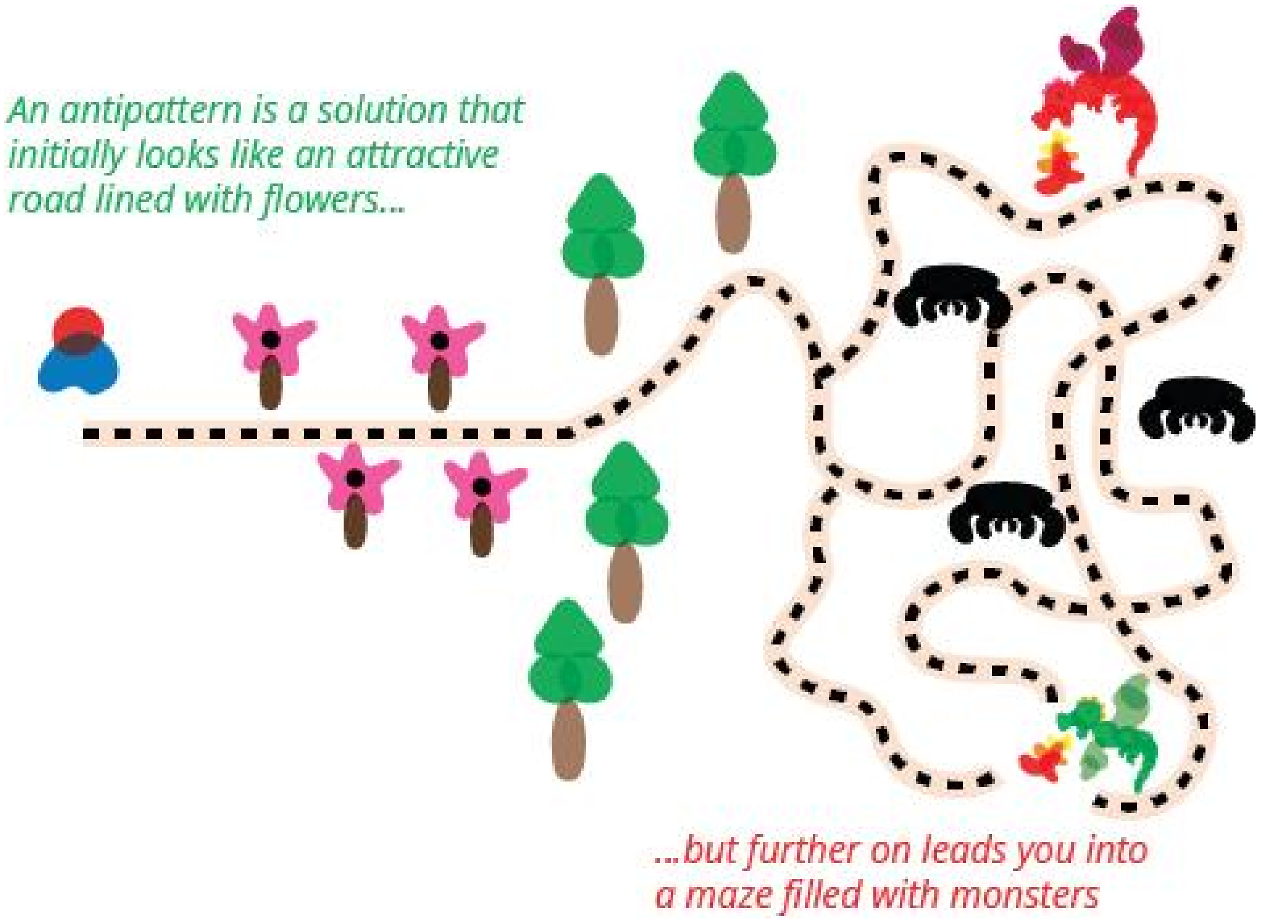
Behavioral patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects
- Behavioral class patterns use inheritance to distribute behavior between classes.
- Behavioral object patterns use composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.

Examples:

- Command
- Iterator
- Observer
- Strategy

Anti-patterns


An antipattern is a solution that initially looks like an attractive road lined with flowers...
...but further on leads you into a maze filled with monsters

What Is an Anti-pattern?

- In real-world application development, you may follow approaches that are very attractive at first, but in the long run, they cause problems.
- For example, you try to do a quick fix to meet a delivery deadline, but if you are not aware of the potential pitfalls, you may pay a big price.
- Anti-patterns alert you about common mistakes that lead to a bad solution.
- Knowing them helps you take precautionary measures. The proverb “prevention is better than cure” very much fits in this context.

Examples of Anti-patterns

The following are some examples of the anti-patterns and the concepts/mindsets behind them.

- *Over Use of Patterns* : Developers may try to use patterns at any cost, regardless of whether it is appropriate or not.
- *God Class*: A big object that tries to control almost everything with many unrelated methods. An inappropriate use of the mediator pattern may end up with this anti-pattern.

- In software engineering, the **mediator pattern** defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.
- With the **mediator pattern**, communication between objects is encapsulated within a **mediator** object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

- *Not Invented Here*: I am a big company and I want to build everything from scratch. Although there is already a library available that was developed by a smaller company, I'll not use that. I will make everything of my own and once it is developed, I'll use my brand value to announce, "Hey Guys. The ultimate library is launched for you."*(* an unwillingness to acknowledge or value the work of others, jealousy, belief perseverance,*)*
- *Golden Hammer*: Mr. X believes that technology T is always best. So,if he needs to develop a new system (that demands new learning), he still prefers T, even if it is inappropriate. He thinks, "I do not need to learn any more technology if I can somehow manage it with T."

- *Management by Numbers*: The greater the number of commits, the greater the number of lines of code, or the greater the number of defects fixed are the signs of a great developer. Bill Gates said, “Measuring programming progress by lines of code is like measuring aircraft building progress by weight.”
- *Swiss Army Knife*: Demand a product that can serve the customer’s every need. Or make a drug that cures all illnesses. Or design software that serves a wide range of customers with varying needs. It does not matter how complex the interface is.

- A Swiss Army Knife, also known as Kitchen Sink, is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. In the attempt, he or she adds a large number of interface signatures in a futile attempt to meet all possible needs.

- *Copy and Paste Programming*: I need to solve a problem but I already

have a piece of code to deal with a similar situation.

- So, I can copy the old code that is currently working and start modifying it if necessary.
- But when you start from an existing copy, you essentially inherit all the potential bugs associated with it.
- Also, if the original code needs to be modified in the future, you need to implement the modification in multiple places.

- *Architects Don't Code*: I am an architect. My time is valuable. I'll only show paths or give a great lecture on coding. There are enough implementers who should implement my idea. *Architects Play Golf* is a sister of this anti-pattern.
- *Hide and Hover*: Do not expose all edits or delete links until he/she hovers the element.
- *Disguised Links and Ads*: Earn revenue when users click a link or an advertisement, but they cannot get what they want.

Types of Anti-patterns

Anti-patterns can belong in more than one category. Even a typical anti-pattern can belong in more than one category.

- The following are some common classifications.
- *Architectural anti-patterns*: The Swiss Army Knife anti-pattern is an example in this category.
- *Development anti-patterns*: The God Class and Over Use of Patterns are examples in this category.
- *Organizational anti-patterns*: Architects Don’t Code and Architects Play Golf are examples in this category.
- *User Interface anti-patterns*: Examples include Disguised Links and ads