Design patterns and where to find them

Nitin Kishore
Jul 29, 2018 · 9 min read

A pattern is a visual stimulus that evokes a response. A repeated design cryptically crafted into any mundane object that can be identified after a keen observation. Everything has a pattern to it and nothing is truly random. Naturally there is a pattern to software coding practices. Knowledge of these patterns will help you understand, locate and design your code to be better and consistent.

I believe the best way to learn this topic would be to identify a pattern in well structured code and understand which category a particular coding pattern falls under. Design patterns are broadly classified into 3 categories -

1. **Structural** — Used to establish relationships between software components Eg: ***Inheritance***

2. **Behavioral** — Involves the best practices/protocols for object interactions Eg: ***Methods and their signatures***

3. **Creational** — Used to create objects in a systematic way. Eg:***Polymorphism***

What you need to understand about these patterns is that they are language neutral, dynamic and incomplete by design, to promote customization, making them extremely important to software engineering. However, they might not be that relevant to a non-object oriented programming language like C. If you have ever worked in a big company with an established code base, you would have undoubtedly seen these patterns.

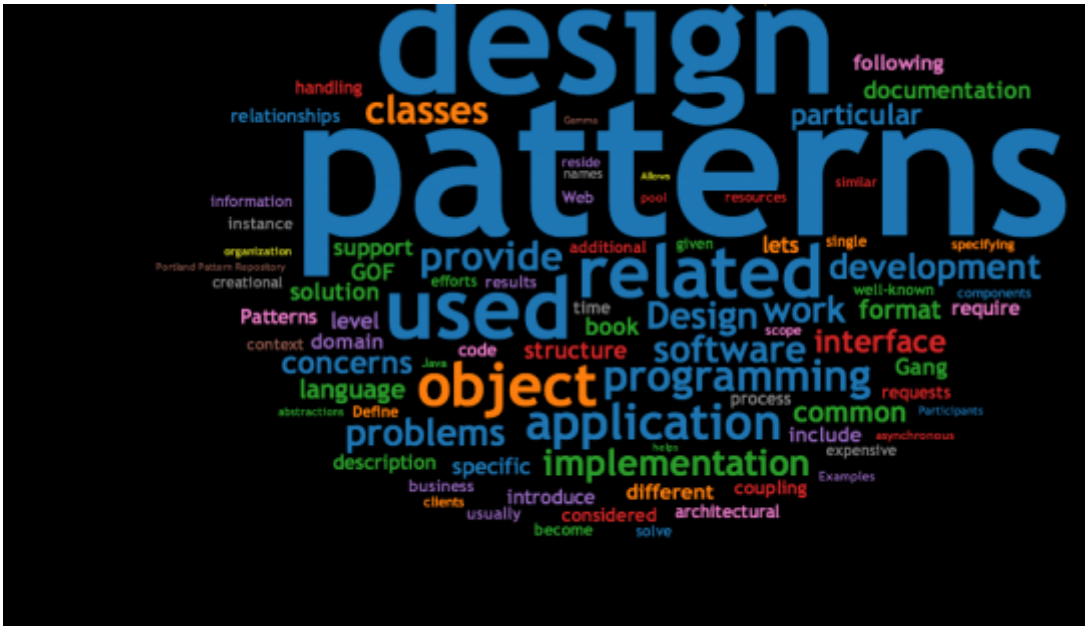
Classes in object oriented programming are like a template to create objects,whenever you need them and define these objects by their attributes(properties and state) and methods(behavior). ***Inheritance*** establishes a parent-child relationship between two classes where attributes can be shared. ***Polymorphism*** relies on this inheritance and allows the parent class to be manifested into any of the child classes. For instance, the parent *Pet* class can be instantiated as either a *dog* or a *cat* child class.

To use design patterns effectively you need to know the context in which each one works best. This context is :

- Participants — Classes involved
- Quality attributes — usability, modifiability, reliability, performance
- Forces — Various factors or trade-offs to avoid unintended consequences
- Consequences — Side effects like better security causing worse or slow performance

The consequences in a context help you decide what trade-offs you are willing to make in your quality attributes for classes, to decide on a suitable design pattern. The patterns are specified in terms of their structure(relationship among elements) and behavior(interactions). This leads us to what is called a **"Pattern Language".**

1. Name — captures the gist of the pattern and is meaningful
2. Context — provides insights on when to use the pattern
3. Problem — describes a design challenge
4. Solution
5. Related Patterns — used in conjunction with the current pattern

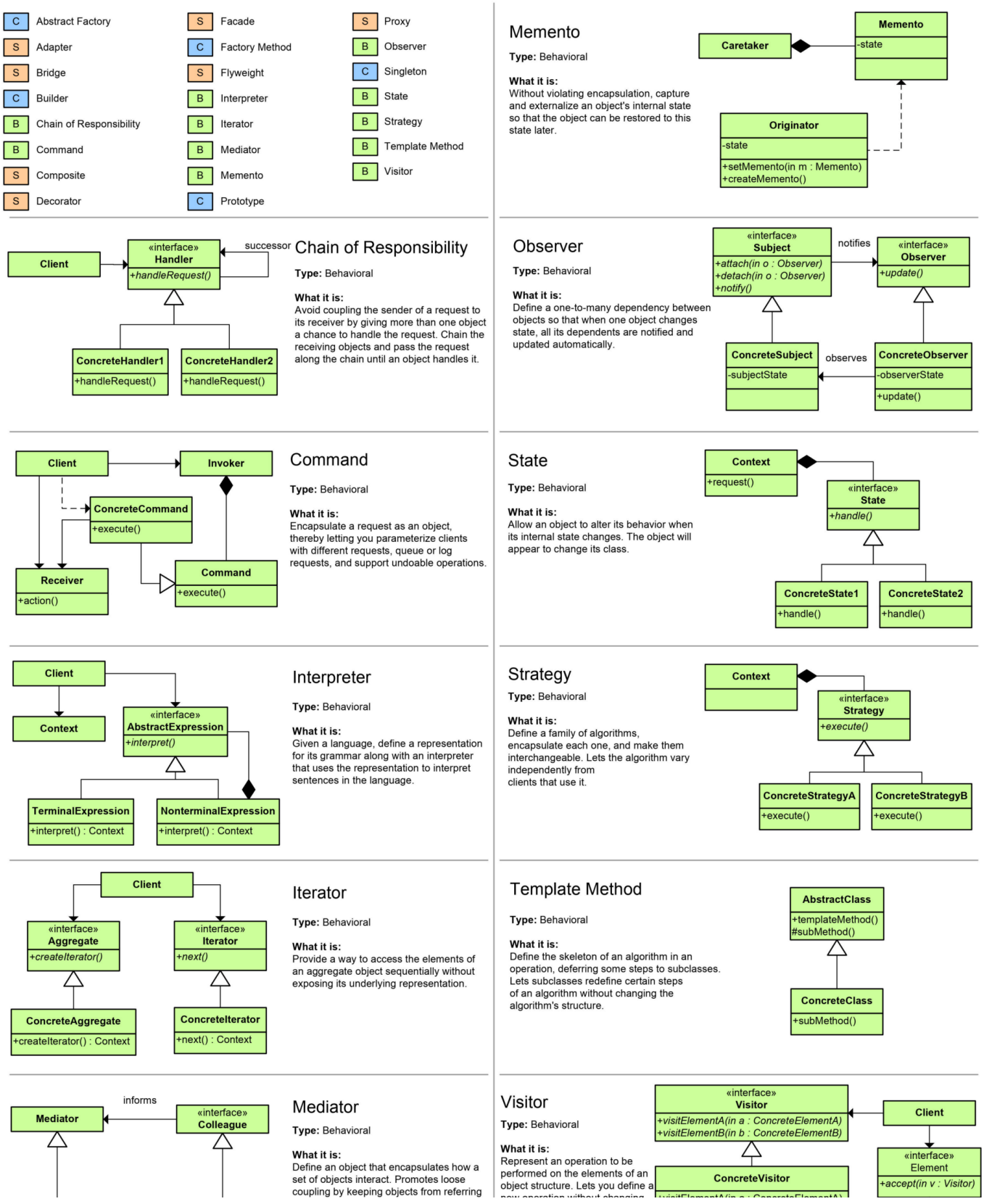


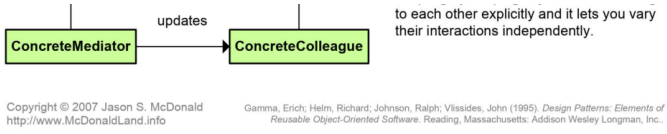
> *A short history nugget: Christopher Alexander proposed 253 designs/guidelines for architecture in his book* ***"A Pattern Language",*** *some of which were adopted in* ***Design***

> ***patterns, Elements of Reusable Object Oriented Software*** *written by "Gang of Four"(1994) comprising of 23 proposed patterns.The number has risen up to 40 now.*

Cheat sheet

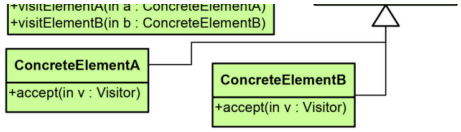
Here is a great cheat sheet by Jason S. McDonald covering those 23 fundamental design patterns for a quick reference or introduction.

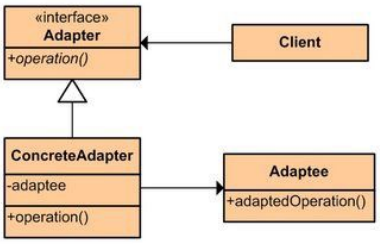




to each other explicitly and it lets you vary their interactions independently.

new operation without changing the classes of the elements on which it operates.





Adapter

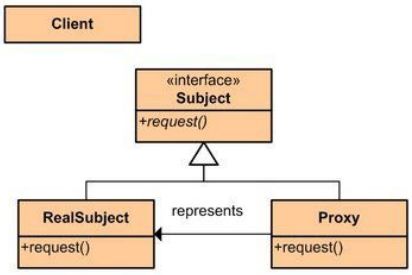
Type: Structural

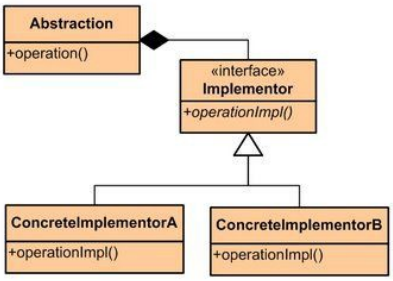
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.





Bridge

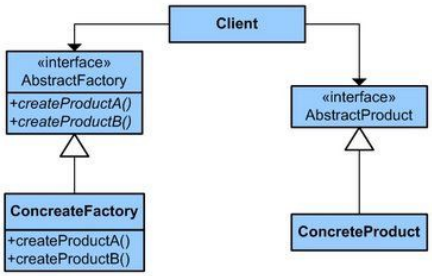
Type: Structural

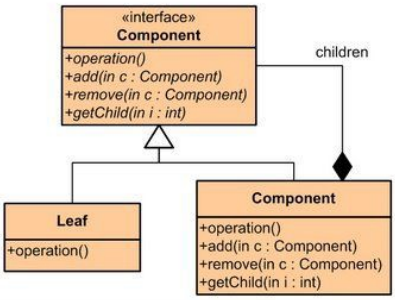
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.





Composite

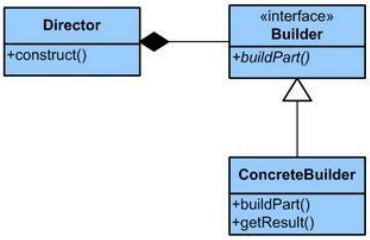
Type: Structural

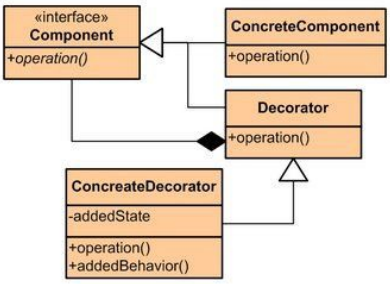
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.





Decorator

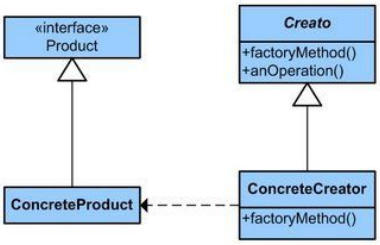
Type: Structural

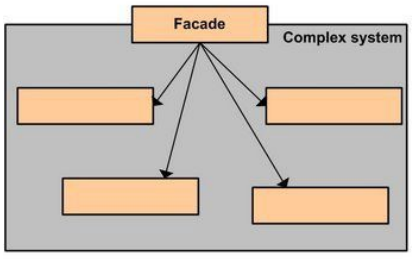
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.





Facade

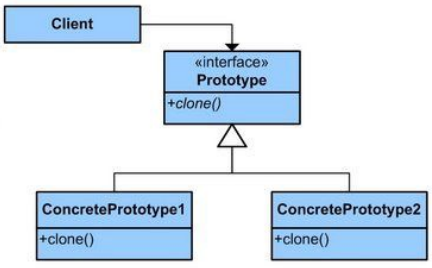
Type: Structural

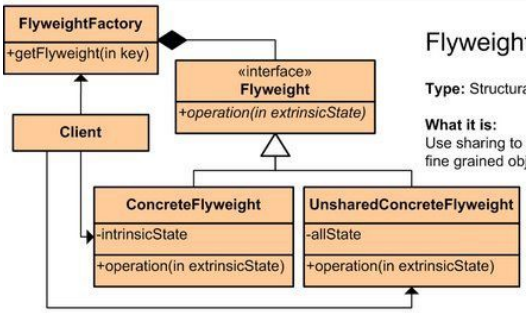
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.





Flyweight

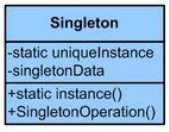
Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Creational Patterns

Factory

This is useful when you are unsure about what type of objects you'll need.In this pattern, you define an interface for creating an object, but the instantiation of said object is deferred to the run-time. This factory can produce any request given by the customer for a specific object. Let's say it's an Apple factory. I ask for an iPod, they build one. I ask for an iPhone, they build me one. I ask for a MacBook Air or an iPad, they build one. You get the point. All of these are objects requested by me, made by the factory (probably in China), only after the request. Perhaps not the most aptly named pattern.

Abstract Factory

This is useful when the expectation is to receive a family of related objects at a given time but don't have to know which family it is until run time.

This is what comes to mind when you think of a factory, unless you are thinking of those tall towers blowing smoke into the sky

Borg — Singleton

No single individual truly existed within the **Borg** Collective (with the possible exception of the **Borg** Queen), as all **Borg** were linked into a hive mind

This is the pattern you need when you want to allow instantiation of only one object from a class. Singleton is an object oriented way of providing global variables. Perhaps there is a need for keeping a cache of information to be shared by multiple objects. Modules in python act as singletons. The Borg design pattern is one way to implement a singleton

```
class Borg:
  """Borg class making attributes global"""

  _shared_state={}  # attribute dictionary

  def __init__(self):
     self.__dict__= self._shared_state

class Singleton(Borg): # Inherits from Borg
  """This class now shares all attributes among various instances"""

  def __init__(self,**kwargs):
     Borg.__init__(self) # update the dictionary by adding(k,v)pairs
     self._shared_state.update(kwargs)

  def __str__(self):
    """Prints the attribute dictionary"""
```

Each time you create a new singleton class it adds on to the global dictionary, preserving the values from the previous instances. ***Multiton pattern*** ensures a class has only named instances, and provide a global point of access to them.

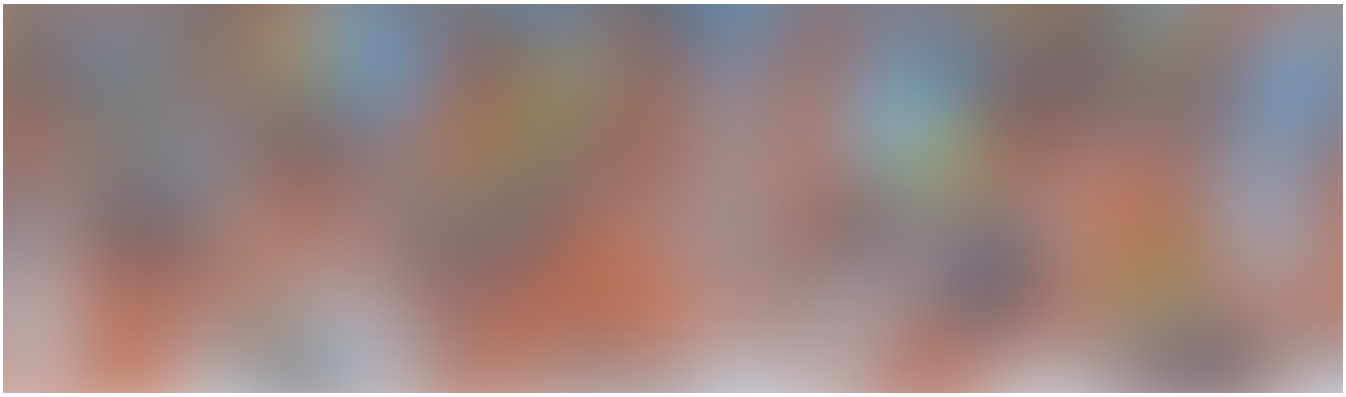
Builder

A Builder partitions the process of building a complex object into 4 different roles, using a Divide and conquer to make the process less complicated

- Director, incharge of building the product using Builder object
- Abstract Builder, class provides all interfaces required
- Concrete Builder, inherits from abstract builder and implements interfaces
- Product, is the object that is being built

Prototype

Prototype clones objects according to a prototypical instance and is especially useful when creating many individual objects that are identical. We can create clones of the object instead of creating one each time. This is somewhat related to abstract factory



Did anyone say clone?

Structural Patterns

Adapter

It converts the interface of a class into another one that the client is expecting, in case they were initially incompatible. It translates the method names between server and client.

Adapt to this

Bridge

This design pattern, related to Abstract Factory and Adapter patterns, helps untangle unnecessary or complicated class hierarchies, in cases where *implementation specific* classes are mixed with *implementation independent* ones. We want to separate these two unrelated parallel or orthogonal abstractions by creating two circle abstractions. The implementation specific circle abstraction involves, drawing a circle and the implementation independent circle involves how to scale it and define the properties of that circle

Two abstractions- Implementation specific and implementation independent

Composite

This design pattern maintains a recursive tree data structure, to represent part-whole relationships, so that elements can have their own sub elements. like a navigation menu with sub-items and sub-sub-items.

It is related to Decorator, Iterator and Visitor. The solution has three major elements

- Component — Abstract class
- Child — Concrete Class inheriting the component
- Composite — Concrete class also inheriting from component that maintains child objects by adding and removing them to a tree structure

Decorator

This pattern is closely related to Adapter,Composite and Strategy. It helps in adding additional features to existing objects, dynamically, without using sub-classsing. Functions are also objects in python and we can add additional features thanks to the in-built decorator.

```
@wraps(function)    - makes the effect of decorator transparent
```

Add to or modify(decorate) the return value of a particular function using a decorator

Proxy

Related to Adapter and Decorator, this pattern is helpful when there is a need to create a object that is very resource intensive. It allows you to postpone the object creation as long as possible by using a placeholder, that will in turn create the object if it is absolutely necessary. The proxy object is basically a middle man

A proxy server is a good example of having a middleman handle requests to access the expensive resources when necessary

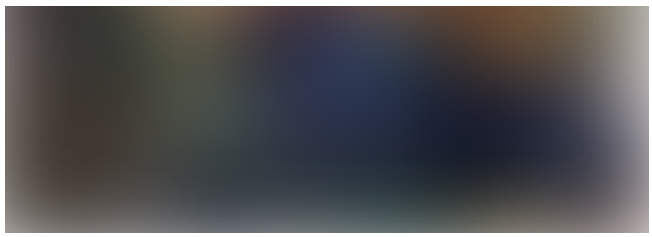
Behavioral Patterns

Command

Different commands given by one entity that are executed by another

Encapsulate a request as an object with parameters thereby decoupling the object that invokes an operation from the one that performs it.

Observer

This is a pool interaction model, which is less efficient than a push model but more flexible. Here you define a one-to-many dependency between objects and when the state of an object changes, all its dependents are notified and updated automatically. Distinct observer objects are registered to a subject. Changes in the subject are broadcast to all observers and this allows a number of objects to be configured dynamically instead of being statically specified at compile time. Instead of the subject pushing what has changed, to all observers, if each observer responsible for pulling its relevant information from that subject

Visitor

This allows adding new features to an existing class hierarchy, dynamically, with minimal changes. It represents new operations to be performed on various elements in a composite object.

Iterator

This pattern allows a client to have sequential access to the elements of an aggregate object, without exposing the underlying structure.It isolates access and traversal features, provides an interface for accessing, and keeps track of objects being traversed.

Strategy

To be able to dynamically change the behavior of an object, this pattern offers a family of interchangeable algorithms to a client. The abstract strategy class has a set of default behaviors and we replace this default method with the concrete strategy class based on our needs or requirements. This is done in python by importing the ***types*** module.

```
types.MethodType(function,self)
```

Chain of reponsibility

This pattern opens up various possibilities of processing a given request, by decoupling the request and its processing. Typically we use an abstract handler, that stores a Successor which handles the request, if it is not handled at the current handler and then concrete handlers check if they can handle it, returning a true value indicating that it has been handled. Composite pattern is related to this

MVC — Model View Controller

Model , View, Controller

This is a UI/Web development related pattern. It has 3 parts.

- **Model** — The lowest level responsible for maintaining data
- **View** — responsible for displaying all or a portion of the data to the user.
- **Controller** — Code that controls the interactions between these two

The idea is that domain objects or representation of data should be completely self contained without reference to presentation. Companies like Pinterest and LinkedIn use Flask as a micro web framework for python. Recently Django is becoming a more feature heavy alternative.

Anti-patterns

The other side of the coin we've been flipping. Some popular examples of this include ***Spaghetti*** code — where your code is all tangled(in hindsight, they should have called this the Pocket Earphones). Typically this has lot of goto statements that redirect you everywhere in the unstructured code. This is characterized by a small number of objects with large methods and multiple responsibilities. The ***Telescoping Constructor*** is another anti-pattern that occurs when a developer builds a complex object using excessive number of constructors.

You know why this is here

Please share this with your friends and hit that claps (✋) button below to spread it around even more. Also add any other patterns you customized or use often in the comments below!

. . .

References

1. http://www.mcdonaldland.info/2007/11/28/40/
2. https://en.wikipedia.org/wiki/Software_design_pattern
3. https://www.linkedin.com/learning/python-design-patterns/what-is-a-design-pattern
4. https://www.youtube.com/watch?v=MALfxsgobA4&list=PLTgRMOcmRb3PQEztkPnMvAeehReJPoSNP&index=6

Programming Python Software Design Object Oriented Design Patterns