14 - Design Patterns

March 13, 2023

COMP2404

Darryl Hill

Introduction to Design Patterns

Contents

Overview

What is a design pattern?

- A way of organizing code to address a common problem
- Organized via:
    - Inheritance and/or polymorphism
    - Delegation via composition, or composition alone
    - Certain operations to be implemented
    - etc.

Some problems recur over and over

- Over time, different programmers noticed they arrived at the same solutions.
    - ***Design patterns.***
- Design patterns are meant to address ***change***.
    - Making code easier to update.

Overview

Defacto authority on design patterns:

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994 - The "Gang of Four"

Types of design patterns

- Creational.
- Structural.
- Behavioural.
- Architectural. [1]

Design pattern ***Client Class***:

- Class using the classes in the design pattern.

[1] some put this in a separate category

Types of Design Patterns

Creational

- Specify how to create objects.
- Factory, Abstract Factory, Singleton, etc.
- Sometimes constructors are problematic.

Structural

- How objects relate.
  - Inheritance, composition.
- Facade, Bridge, Decorator, Proxy, etc

Types of Design Patterns

Behavioural

- Specify how objects communicate
  - Which objects call which.
- Observer, Strategy, Visitor, etc

Architectural (arguably not design patterns)

- How objects are grouped into subsystems.
  - Subsystem - group of classes that work together.
- Client-server, peer-to-peer, MVC, etc.

Facade

Facade is a structural design pattern.

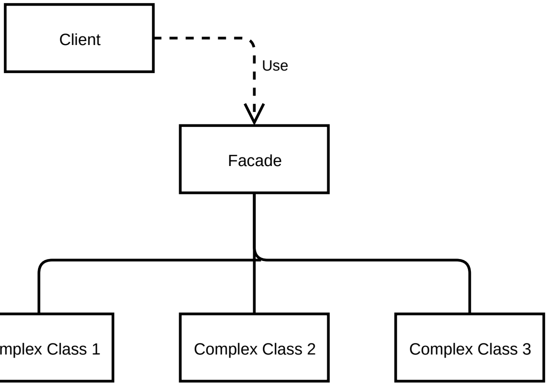
- I.e., how objects relate.

Facade provides a simplified interface for a complex class.

- Client class calls simple operations on Facade class.
- Facade class forwards these operations to complex class(es).
  - Taking care of extra details, multiple function calls, bounds checking, etc
  - Delegate, but sometimes in a complex way
- Python can be thought of as a Facade language
  - Encapsulating complex C calls

Facade UML

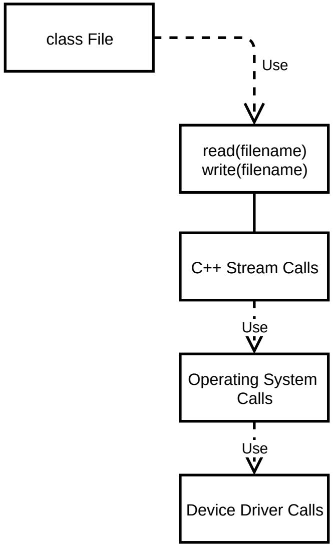
- The dotted line is a ”uses” association - a general relationship.
  - Could be composition



Facade UML

A simple "read" or "write" call hides

- C++ stream objects
- OS calls, which is a facade for
  - device driver calls



Observer

Observer is a behavioural design pattern.

Observer classes are informed of changes in the subject class.

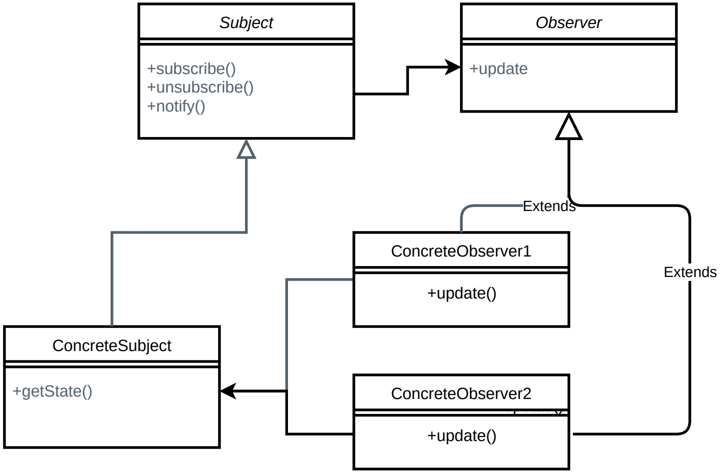
- Subject class:
    - Maintains a collections of observers.
    - Notifies observers of relevant changes.
        - Whatever changes they are subscribed to.
- Observer class
    - Subscribes to notifications.
    - Updates itself upon notification.
        - Usually a callback function.

Observer

- Observer pattern is also know as publisher/subscriber.
- Observer is often used with MVC
  - updates on the model are relayed to the view

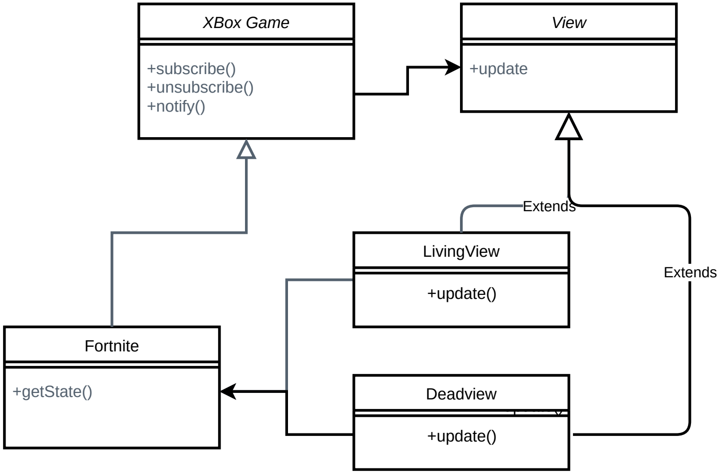
Observer UML

- Italics is abstract class
- notify() loops over observers and calls their update()
- The Observers update their state based on the state of the Subject



Observer UML

- Italics is abstract class
- notify() loops over observers and calls their update()
- The Views get the game state and update accordingly



Visitor UML

The **Visitor** design pattern is often used to solve the multiple dispatch problem. Functions of the type

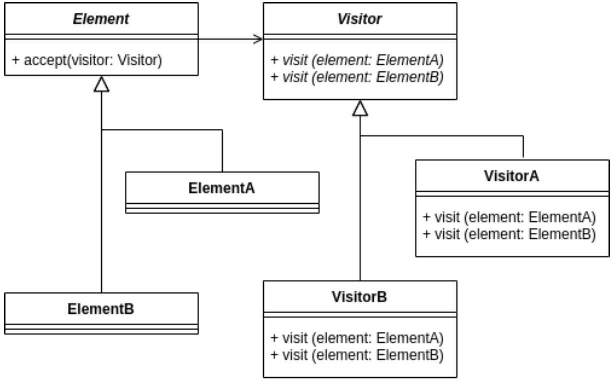
- `fun(A, B)` where both A and B have subclasses.
- We write `fun(A, B)` slightly differently:

  `VisitorA::visit(ElementA)`

  `VisitorA::visit(ElementB)`

  `VisitorB::visit(ElementA)`

  `VisitorB::visit(ElementB)`

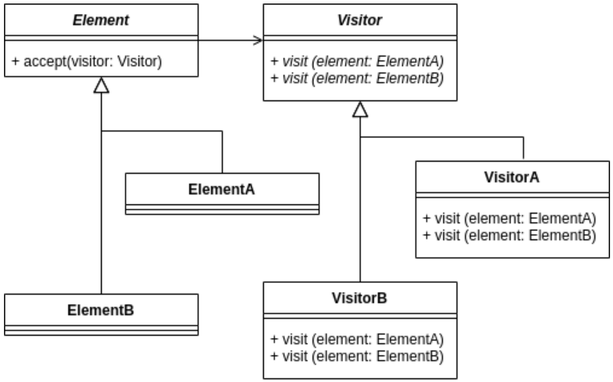

Element
+ accept(visitor: Visitor)
Visitor
+ visit (element: ElementA)
+ visit (element: ElementB)
ElementA
ElementB
VisitorA
+ visit (element: ElementA)
+ visit (element: ElementB)
VisitorB
+ visit (element: ElementA)
+ visit (element: ElementB)

Each of these functions has a different behaviour depending on the exact types of `Visitor` and `Element` involved.

Visitor UML

The **Visitor** design pattern is often used to solve the multiple dispatch problem.
Imagine a space game with `SpaceObjects`:

```
Spacecraft::collide(Asteroid)
Spacecraft::collide(Spacecraft)
Asteroid::collide(Spacecraft)
Asteroid::collide(Asteroid)
```


Element
+ accept(visitor: Visitor)
Visitor
+ visit (element: ElementA)
+ visit (element: ElementB)
ElementA
ElementB
VisitorA
+ visit (element: ElementA)
+ visit (element: ElementB)
VisitorB
+ visit (element: ElementA)
+ visit (element: ElementB)

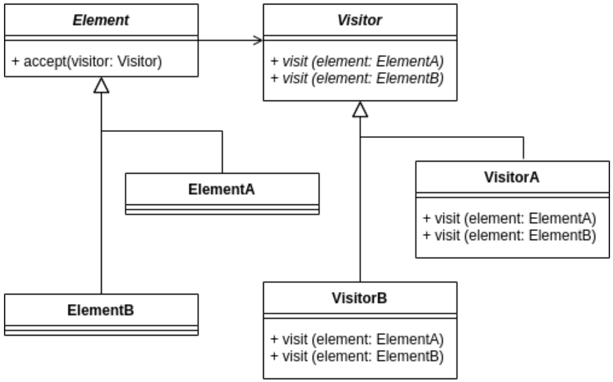
Each of these functions has a different behaviour depending on the exact types of `SpaceObjects` are colliding.

Visitor UML

The accept function is often simply

```
void accept(Visitor& v){
   v.visit(*this);
}
```

The exact `visit` function that is called is determined at runtime based on the subclasses of `Element` and `Visitor`.


Element
+ accept(visitor: Visitor)
Visitor
+ visit (element: ElementA)
+ visit (element: ElementB)
ElementA
ElementB
VisitorA
+ visit (element: ElementA)
+ visit (element: ElementB)
VisitorB
+ visit (element: ElementA)
+ visit (element: ElementB)

Factory

Factory is a creational design pattern.

- Useful for when ***how*** an object is created will change.
- Perhaps some information is retrieved from disk, or entered on a form.

Encapsulates creation of derived objects.

- Factory creates derived object and returns to client class.
- Client uses derived object as base class (is-a)
- Client class does not know specific type of derived object
  - Does not need to know

Base class is often abstract

- Provides generic interface (facade) to the client

Factory

StackOverFlow

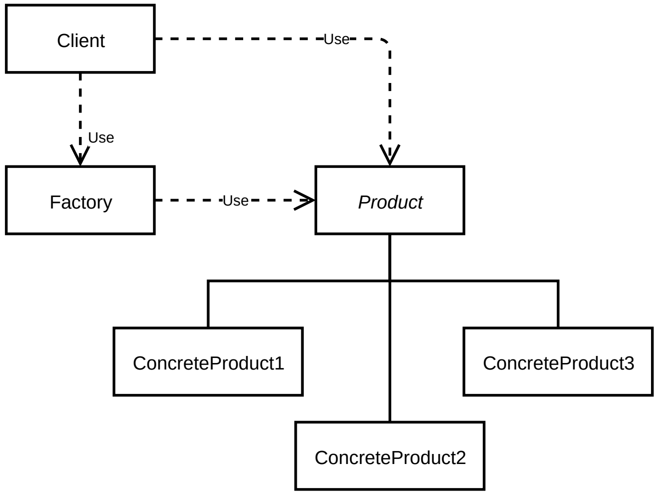
If you think of classes as people, `Factory` is a hiring agency.

- I need a class to do a particular job
- Know a few details about how I want it done
- Many details I don't need to know

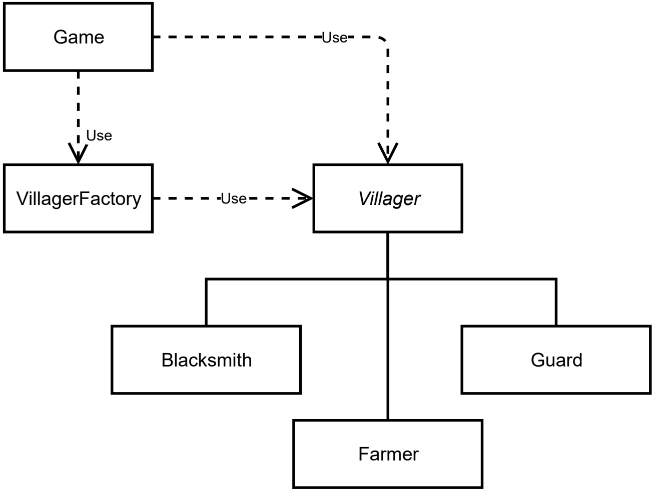
`Factory` will give me a class to do the job

- Handles things like dependencies, implementation

Factory UML


Client
Use
Use
Factory
Use
Product
ConcreteProduct1
ConcreteProduct3
ConcreteProduct2

Factory UML


Game
Use
Use
VillagerFactory
Use
Villager
Blacksmith
Guard
Farmer

Anti-Patterns

Common bad programming practice

- Too many to count

Very common: the Blob

- one class contains all functionality
- the God object
- can be a danger of Facade