Gang of Four Patterns

S. Thiel[1]

[1]Department of Computer Science
Concordia University

July 20, 2018

Outline

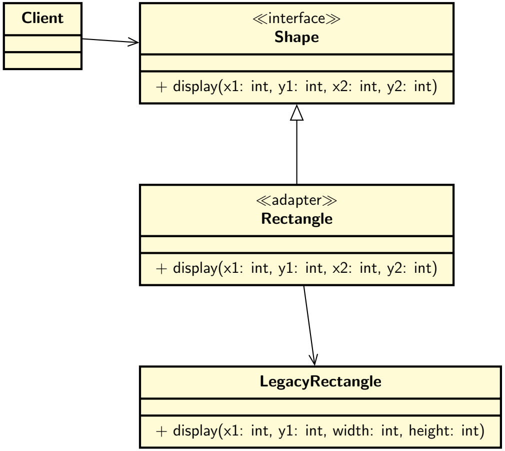
[1, p.280,435,436]

- Named after Gamma, Helm, Johnson and Vlissides, who wrote the mid-90s book "Design Patterns"
- Covered 23 core design patterns, arguably the most popular software design pattern language in use today.
- "15 are common use and most useful."[1, p.436]
- To fully understand the patterns takes more than what we cover here (read their book, etc)
- `https://sourcemaking.com/design_patterns`

Adapter [1, p.436-440]

- **Problem:** incompatible interface, similar components with differing interfaces
- **Solution:** Convert the original interface into another interface through an intermediate adapter object.
- Polymorphism from GRASP uses the adapter pattern, the adapter being the common interface.
- Often a common interface will be made, and adapters implementing that interface will delegate to the varying objects, hiding the differences.

Adapter Example [1]



Client
≪interface≫
Shape
+ display(x1: int, y1: int, x2: int, y2: int)
≪adapter≫
Rectangle
+ display(x1: int, y1: int, x2: int, y2: int)
LegacyRectangle
+ display(x1: int, y1: int, width: int, height: int)

[1] sourcemaking.com

Factory [1, p.440-442]

- GoF pattern is actually "Abstract Factory"
- We're talking about Simple Factory or Concrete Factory here, but widespread, so let's pretend it's one of the GoF[2]
- **Problem:** Need to create object with special considerations like complex creation logic, the need to separate concerns and hide variation in the created objects, the need to generally maintain high cohesion.
- **Solution:** Create a Pure Fabrication object called a Factory that handles the creation

[2] It is arguably a variation on Abstract Factory

Factory in Action

- In my Masters thesis I advised the use of a Factory to take care of *Domain Object* creation because it impacted database, caching and *Unit of Work* concerns that had no business in most areas that would otherwise take care of creating those *Domain Object*

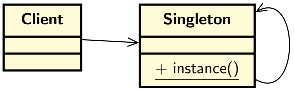
Singleton [1, p.442-446]

- **Problem:** Exactly one instance of a class is allowed. Other objects need a golbal and single point of access
- **Solution:** Define a static method of the class that returns the singleton
- Concurrency control is common around Singletons.

Singleton in Action

- In my Masters thesis I advised the use of a variant on this pattern to allow the trivial creation of a single-access point within a web request, the ThreadLocal Singleton. Each request was guaranteed a Thread for its sole use for the duration of the request. Request Attributes, standard fare for web applications, were readily stored in this ThreadLocal Singleton so did not have to be passed around through various Commands and Views, thus greatly reducing coupling between Commands and Views, while extending flexibility.

Singleton Example [3]



[3] sourcemaking.com

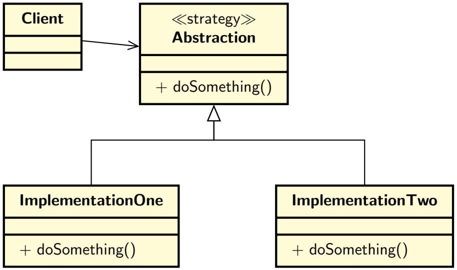
Why not make it all static? [1, p.445]

- instance-methods allow subclassing, refinement (overloading) of the Singleton class for effective reuse. Static methods aren't polymorphic, and generally don't support overwriting in languages where such things are allowed (part of the definition of the class)
- Remote access methods generally support only instance-methods. (e.g. Java's RMI)
- A class is not always needed as a singleton in all applications. Sometimes you figure out it's not a singleton later. Instance-based, this is an easy fix (generally you don't need to do anything).

Strategy [1, p.447-452]

- **Problem:** Algorithms or policies can or will change, need to be decided at run-time
- **Solution:** Define each algorithm/policy/strategy in a separate class with a common interface
- Often this leads to the instance of the strategy being created by a Factory that is passed enough info to know what to do.
- Polymorphism and Protected Variation at work!

Strategy Example [4]



[4] sourcemaking.com

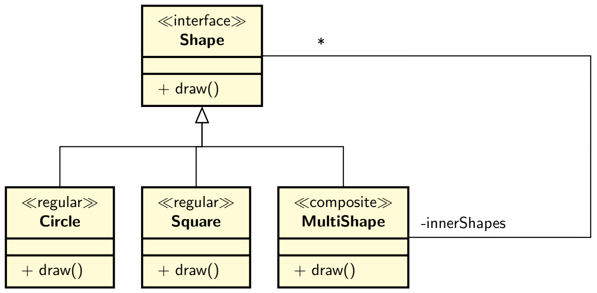
Strategy in Action

- In my Masters thesis I. . . don't think I did anything with the Strategy pattern
- When could this come up with pokemon?

Composite [1, p.452-461]

- **Problem:** A number of similar objects need to be treated atomically
- **Solution:** Define an interface for composite objects that is shared with the atomic objects it composes
- works well with the strategy pattern, as certain strategies may just be several simpler strategies applied together (e.g. discounts)
- Composite of related elements starts looking like proper object-oriented design? Larmen gives a strange example called *IDs to Objects*
- when using composites, it's generally a good practice to pass around the composites instead of digging out the children (components). Composite Pattern done well supports this.

Composite Example



Composite in Action

- In my Masters thesis I. . . don't think I did anything with the Composite pattern
- Think about applying this with complex pokemon abilities?

Facade [1, p.461-463]

- **Problem:** There are many components in a subsystem, each with some behaviour that is needed, but knowledge of this subsystem or access to the other behaviour is not
- **Solution:** Define a single point of access that provides an interface on the desired behaviours and encapsulates knowledge of the weird subsystem so nothing else has to know about it.

Facade in Action

- I once wrote a system for doing Video Annotations in a popular annotation tool that was a plugin for Chrome.
- We just needed to display and allow the selection of pieces of clips of video on the web
- Quicktime produced a massive and complex set of libraries that allowed you to do all this, but it was spread out and a mess
- The solution was to write a facade that allowed for the 5-10 behaviours that we needed and ignored the literally hundreds of other things available.

Observer/Subscriber [1, p.463-471]

- **Problem:** Different things want to know about different events and will behave in different ways, but low coupling is desired
- **Solution:** define subscribers via an interface that listen, and then have them register to listen to publishers of events that they care about. The publishers will notify the correct subscribers at the correct times.

Observer/Subscriber in Action

- Frequently use in UIs
- Many of you have implemented this in your pokemon game!

References I

[1] Craig Larman.
Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
Addison Wesley, 3rd edition, 2013.