Software Patterns

An Example of Pattern

Motivation

- Change and Continuous Evolution
- Not the most powerful,but those who adopt to changes survive!

For an IT solution, survival is a major challenge. It must evolve continuously as the external dynamics changes

Software Evangelists are preaching good practices for years

Encapsulate what varies

Identify the aspects of your application that vary and separate them from what stays the same

- Less effect on rest of code
- Fewer unintended consequences from code changes
- More flexibility in the system
- Later, we can alter or extend the part that varies without affecting the part that doesn't
- Basis for almost every design pattern
- All patterns provide a way to let one part vary independently of another

Prefer interfaces, not classes

Program to interfaces, not implementations

- Provides higher level of decoupling
- Helps to be less aware of internal details of each other
- Interface here means more of a supertype
- Exploit polymorphism
  - Allows us to change the concrete implementations at runtime

HAS-A can be better than IS-A

Favour composition over Inheritance

- More flexibility in system
- Encapsulate family of algorithm into their own set of classes
- Change behavior at runtime
- Used in many patterns

The power of loose coupling

Strive for loosely coupled designs between objects that interact

- Minimize the interdependency between objects
- Build flexible OO systems

The open-closed principle

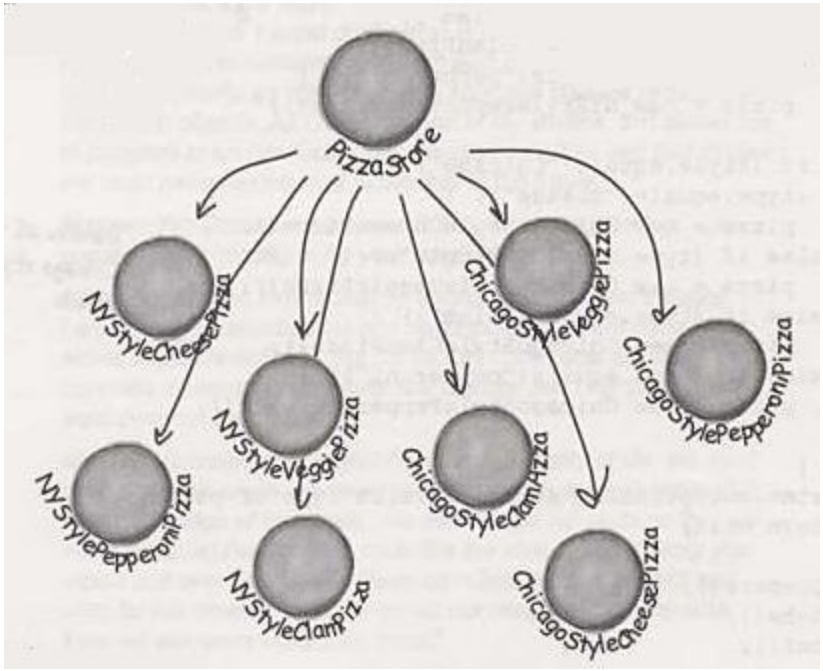
Classes should be open for extension but closed for modification

- If requirements change, extend the classes
- However don’t modify the existing code for that
- Design is resilient to change and flexible to take on new functionality
- Used in decorator, observer and many other patterns
- Over use of this principle leads to complex and hard to understand code

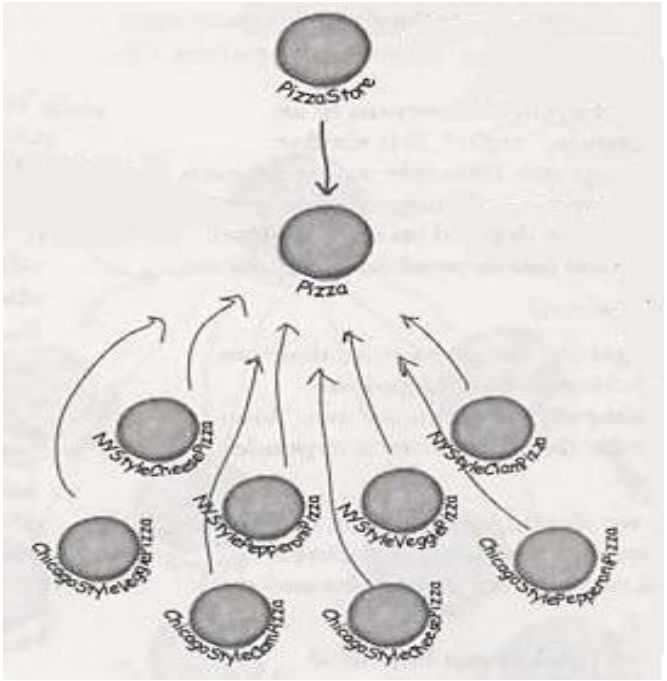
The dependency inversion principle

Depend upon abstractions. Do not depend upon concrete classes

- High level components should not depend on low level components, rather they should depend on abstraction



High dependency



low dependency

Principle of least knowledge

Talk only to your immediate neighbours

- Reduce interactions between objects to very few close classes
- Prevent designs in which large number of classes are coupled with each other
- From a method in an object, only invoke methods that belong to:
  - Object itself
  - Objects passed in as parameter to the method
  - Objects created by the method
  - Any components of the object

The Hollywood principle

Don’t call us, we will call you

- Prevents dependency rot
- When lots of components depend on each other, rot sets in
- In this principle
  - Low level components hook themselves into a system
  - High level components decide when and how the low level components are needed
- Similar to the Dependency inversion

Single responsibility

A class should have only one reason to change

- Closely related to cohesion
- Cohesion implies a class has a set of related functions
- Each responsibility is liable to change
- More than one responsibility means more than one area of change

Benefits of Pattern

- Communicates best practice in standards
  - Say more with less
- Solution reuse – no re-invention of wheel
  - Stay in the design longer
- Reduction of development time and therefore cost saving
- A common vocabulary of solution across the IT community
  - In design meetings
  - With other developers
  - In architecture documentation
  - In code comments and naming conventions
  - To groups of interested developers

Pattern is not a silver bullet but a guideline to reach to a solution. Pattern does not substitute experienced heads!

Difference between Dependency Inversion principle and Hollywood principle

- Dependency Inversion principle teaches us to avoid the use of concrete classes and instead work as much as possible with abstractions. Stronger statement on avoiding dependency.

Difference between Dependency Inversion principle and Hollywood principle

- Hollywood principle is for building frameworks or components so that lower level components can be hooked into computation without creating dependency between low and high level components. Design that allows low level structures to interoperate while preventing others becoming too dependent on them.
- Common Goal : Decoupling.

Single Resposibility Principle

A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

Open / Closed Principle

A software module (it can be a class or method) should be open for extension but closed for modification.

Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Interface Segregation Principle

Clients should not be forced to depend upon the interfaces that they do not use.

Dependency Inversion Principle

Program to an interface, not to an implementation.