The principles of OO design :

SRP - Single Responsibility Principle

-> Principle: A class should have only one reason to change.

-> Responsibility = ”a reason to change”

-> If a class has more than one responsibility, then the responsibilities become coupled.

-> The cohesion is low if the module does several things

-> Cohesion should be high

OCP - Open-Closed Principle

--> Open for Extension (new functions are added)

--> Closed for Modification (existing code is

unchanged)

-> What this really means is that you should (re)design so that change leads to extending, not modifying existing code

LSP - Liskov Substitution Principle

-> The key of OCP: Abstraction and Polymorphism

--> Implemented by inheritance

--> How do we measure the quality of inheritance?

-> "If for each object ob1 of type S there is an object ob2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when ob1 is substituted for ob2 then S is a subtype of T." ~ B. Liskov, 1988

-> Subtypes must be substitutable for their base types.

ISP - Interface-Segregation Principle

-> ISP deals with the disadvantages of “fat” interfaces.

--> Clients should not be forced to depend on methods that they do not use.

-> Avoid classes with too many responsibilities (classes whose interfaces are not cohesive).

-> Break interfaces up into cohesive groups of methods, each serving a certain kind of clients.

Example

```
// Bad example

interface IWorker {
```

```
public void work();
public void eat();
}
class Worker implements IWorker{
public void work() { // ....working }
public void eat() { // ..... eating in launch break}
}
class SuperWorker implements IWorker{
public void work() { //.... working much more }
public void eat() { //.... eating in launch break }
}
class Manager {
IWorker worker;
public void setWorker(IWorker w) { worker=w; }
public void manage() { worker.work(); }
}

// Good example
interface IWorker extends Feedable, Workable { }
interface IWorkable { public void work();}
interface IFeedable { public void eat();}
class Worker implements IWorkable, IFeedable{
public void work() { // ....working }
public void eat() { //.... eating in launch break }
}
class Robot implements IWorkable{
public void work() { // ....working }
}
class SuperWorker implements IWorkable, IFeedable{
public void work() { //.... working much more }
```

```
public void eat() { //.... eating in launch break}

}

class Manager {

Workable worker;

public void setWorker(Workable w) { worker=w; }

public void manage() { worker.work();}

}
```

DIP - Dependency-Inversion Principle

-> High-level modules should not depend on low-level modules. Both should depend on abstractions.

-> Abstractions should not depend on details. Details should depend on abstractions.

-> Structured Analysis and Design tend to create software structures in which high- level modules depend on low-level modules, and in which policy depends on detail.

-> It is the high-level modules that contain the important policy decisions and business models of an application.

DIP – Violating DIP

-> If the business depends on concrete services in the service layer and the services depends on concrete utilities in the utility layer, the business depends

transitively on the utilities.

-> This is very unfortunate because changes in low level modules have effect on high-level modules.

-> High-level modules will be difficult to reuse in other contexts

DIP – Conforming to DIP

-> You should invert the dependencies by using interfaces declared in the upper layer (the client “owns” the interface).

-> Now, the business no longer depends on a concrete service and can be reused with different implementations of the service.

-> NOTE: The book uses a different naming convention for the interfaces.

Example

```
//Bad example

class Worker {
```

```
public void work() { // ....working }
}
class Manager {
Worker m_worker;
public void setWorker(Worker w) { m_worker=w; }
public void manage() { m_worker.work(); }
}
class SuperWorker {
public void work() { //.... working much more }
}

//Good example
interface IWorker { public void work(); }
class Worker implements IWorker{
public void work() { // ....working}
}
class SuperWorker implements IWorker{
public void work() { //.... working much more }
}
class Manager {
IWorker m_worker;
public void setWorker(IWorker w) { m_worker=w;}
public void manage() { m_worker.work();}
}
interface IWorker { public void work(); }
class Worker implements IWorker{
public void work() { // ....working}
}
class SuperWorker implements IWorker{
public void work() { //.... working much more}
```

```
}

class Manager {

IWorker m_worker;

public void setWorker(IWorker w) { m_worker=w;}

public void manage() { m_worker.work();}

}
```

Package Design:

Cohesion:

The **Reuse-Release Equivalence Principle**

-> The granule of reuse is the granule of release.

The **Common-Reuse Principle**

-> The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

The **Common-Closure Principle**

-> The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package and no other packages.

Coupling:

The **Acyclic-Dependencies Principle**

-> Allow no cycles in the package-dependency graph

The **Stable-Dependencies Principle**

-> Depend in the direction of stability.

The **Stable-Abstractions Principle**

-> A package should be as abstract as it is stable.

-> Abstractness = The number of classes in the package / The number of abstract classes in the package

4 + 1 Model:

Logical View -> Development View

\|/ Scenario \|/

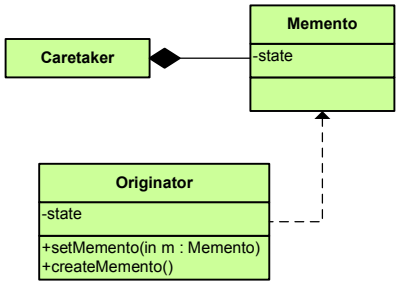
->Process View -> Physical View

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

Memento

Type: Behavioral

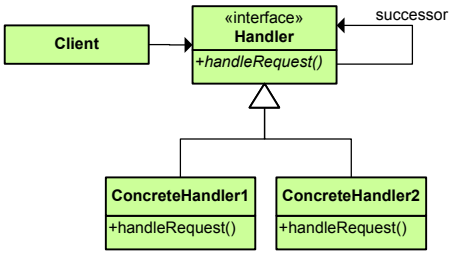
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

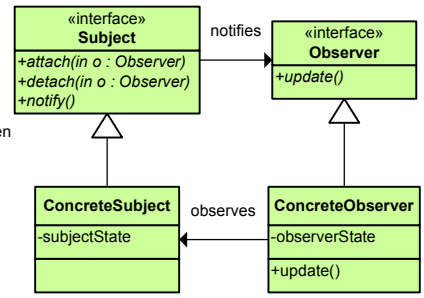
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Observer

Type: Behavioral

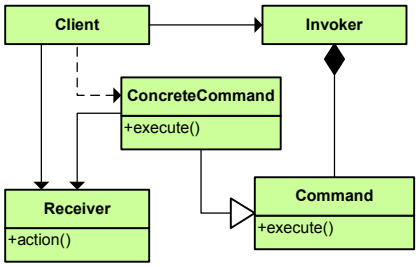
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Command

Type: Behavioral

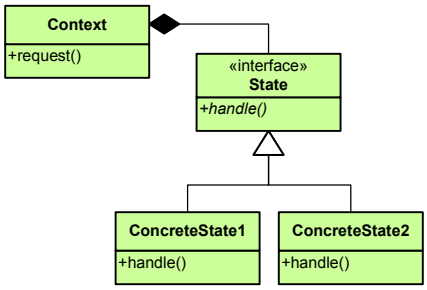
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



State

Type: Behavioral

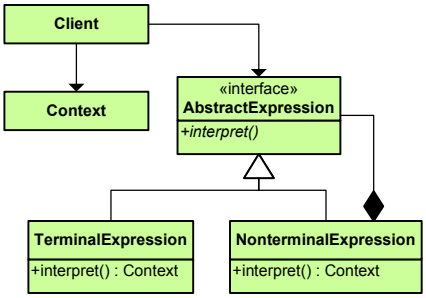
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Interpreter

Type: Behavioral

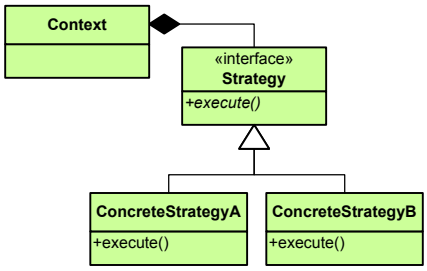
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Strategy

Type: Behavioral

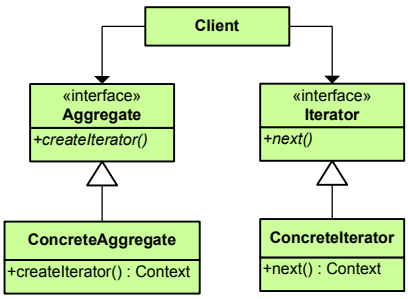
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Iterator

Type: Behavioral

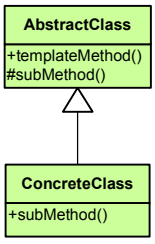
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Template Method

Type: Behavioral

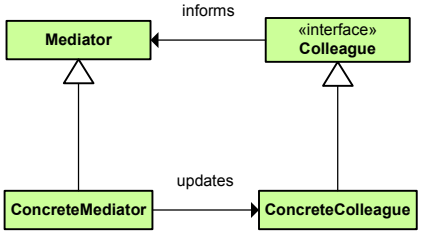
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Mediator

Type: Behavioral

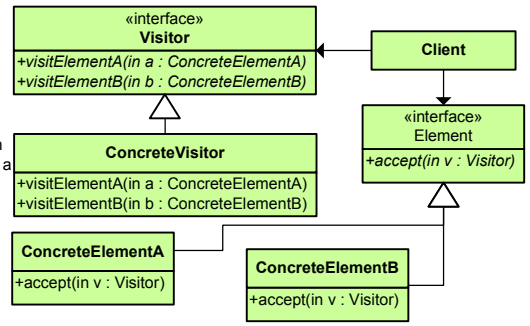
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.



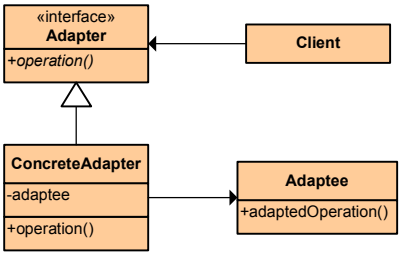
Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



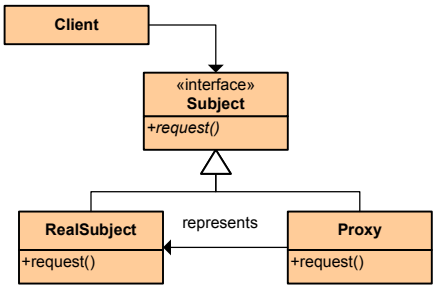
Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..



Adapter

Type: Structural

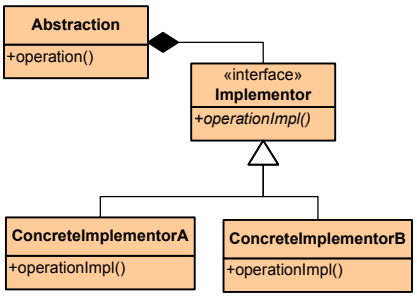
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.



Proxy

Type: Structural

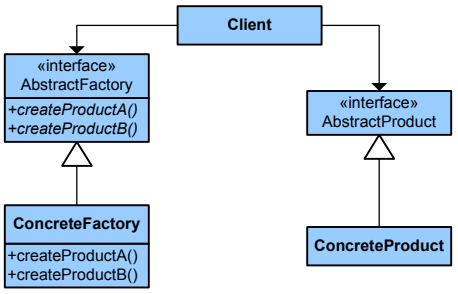
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Bridge

Type: Structural

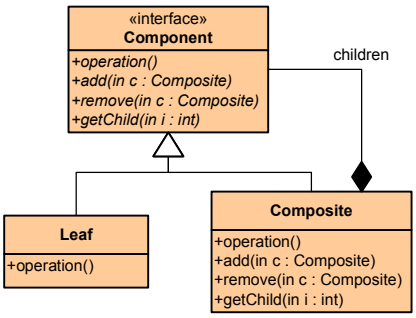
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.



Abstract Factory

Type: Creational

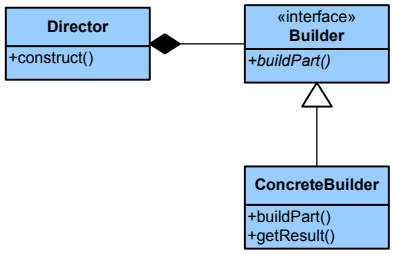
What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Composite

Type: Structural

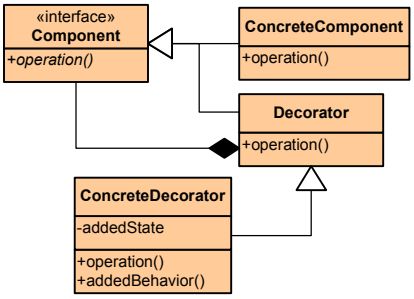
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Builder

Type: Creational

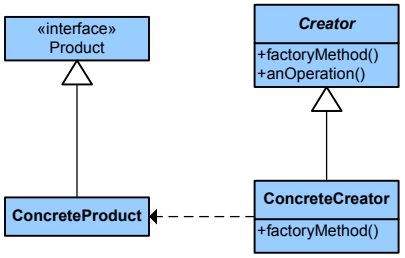
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Decorator

Type: Structural

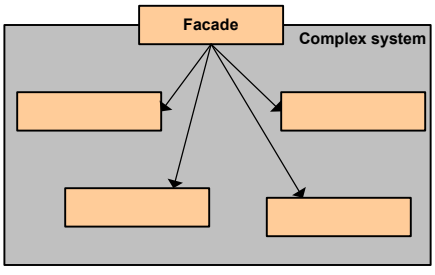
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.



Factory Method

Type: Creational

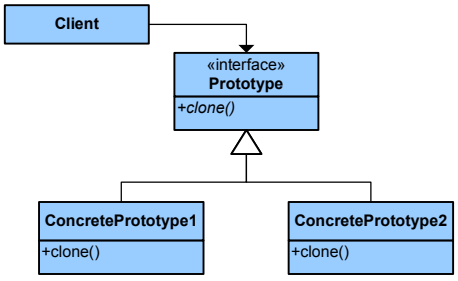
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Facade

Type: Structural

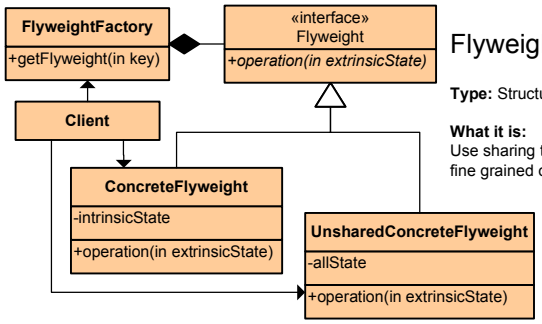
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



Prototype

Type: Creational

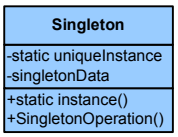
What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



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.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..