Design Pattern Summary

By Rune Rasmussen www.eyeweb.dk/rune

This is a short summary on many of the design patterns in the book "Design Patterns" by GoF[1]. For full understanding please read the book.

The summary was written to help myself study for an exam. I give no guaranty for the validity of the information here.

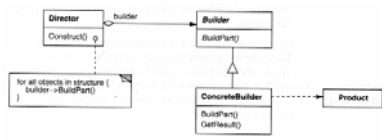
Content:

[1] Design Patterns, Elements of Reusable Object-Oriented Software. Authors: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

Creational patterns

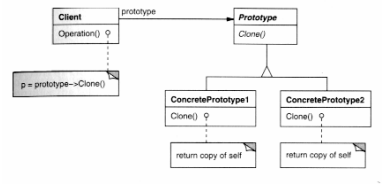
Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Construction is done step-by-step.
Example: Save Word doc to RTF doc.
The builder can be an interface or a class with empty methods (the concrete builder overwrites selected methods).
Related pattern: Abstract Factory creates a family of products (simple or complex). When the builder is done constructing it returns the product.



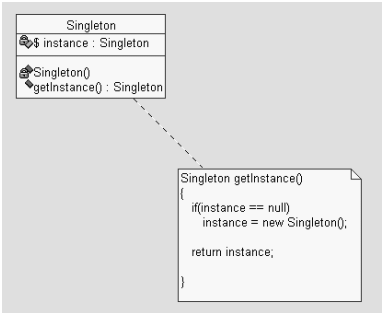
Prototype

Create a prototypical instance that can be copied – instead of creating an object from scratch.
The prototypes implements a prototype interface – so the rest of the class hierarchy does not have to distinct between prototype implementations when wanting a clone.
Each subclass of Prototype must implement the clone() method, which may be difficult (e.g. if it include objects that do not support copying).
The class to instantiate can be defined at run-time.
Prototypes avoid building a class hierarchy of factories that parallels the class hierarchy (done in Abstract Factory).
The prototype can be stored as a static variable on the class itself.
Related patterns: Like Abstract Factory and Builder it hides the concrete product classes from the client (so it only knows the interface).



Singleton

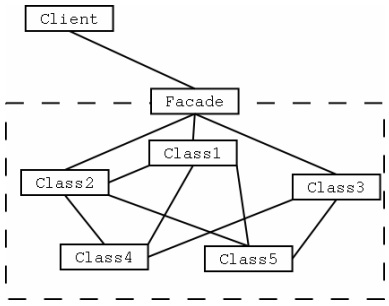
Ensure a max no (larger than zero) of instances of a class and provide a simple global point of access to it.
Let the class itself be responsible for instantiating it.
Is extensible by subclassing.
Implementation: Constructor is private. A static method calls constructor and stores a reference to the object on a private class level.
Alternative is a global variable that can not be controlled.
Related patterns: Singleton is used in Builder, Prototype, Abstract Factory.



Structural patterns

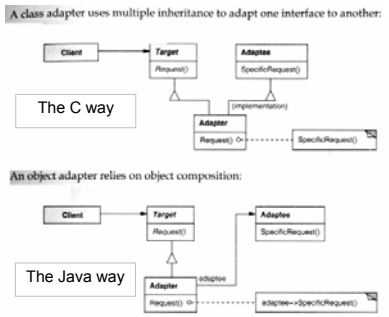
Facade pattern

Ease the interaction with a set of classes by using a façade-class. The subsystem is a closed system that can be changed safely. All access to these classes will be done through the façade-class.
The façade-class is adjusted to the subsystem and can e.g. use a platform specific subsystem internally.



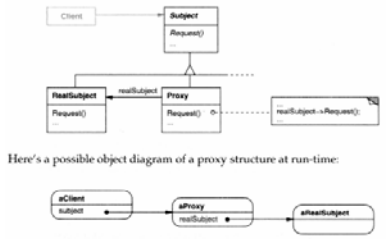
Adapter

Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
An intermediate object, the adapter, serves to adapt an interface to another existing class.
A class adapter won't work when we want to adapt a class *and* all its subclasses. Adapter can overwrite functionality (responsibility is spread).
Object adapter: The adapter can use multiple adaptees to achieve the match. If an adaptee is subclassed the Adapter must be changed manually to use it.



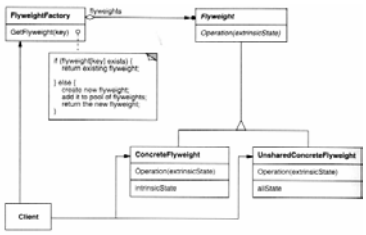
Proxy

A placeholder for another object to control access to it.
The proxy forwards requests to the real object.
Virtual proxy: Defer initialization until the object is needed. The placeholder can contain a minimum of information from the object (e.g. name).
Remote proxy: A local representative for an object in a different address space.
Protection proxy: Control access to the object. A protection proxy might refuse to perform an operation – being a subset of the subject.
Related patterns: The Adapter provides a different interface to the object it adapts. The proxy provides the same interface as its subject.

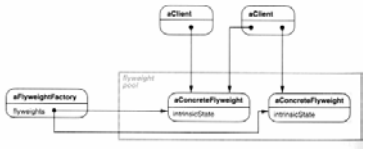


Flyweight

Reuse general objects. Saves space – uses little more cpu time.
Example: Document with flyweight chars.
Intrinsic state: General information. Sharable information.
Extrinsic state: Unique information for a flyweight's context. Can't be shared.
Useful when: Many objects, costly storage, most objects state can be made extrinsic, the object identity is not used by the application.
Clients must obtain ConcreteFlyweight objects from the FlyweightFactory object to ensure they are shared properly.
Related patterns: Often combined with the Composite pattern to represent a hierarchical structure as a graph with shared leaf nodes.
State and Strategy objects are often flyweights.

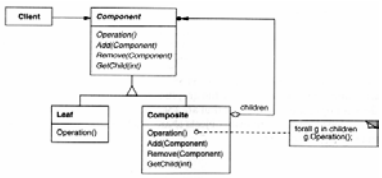


The following object diagram shows how flyweights are shared:



Composite

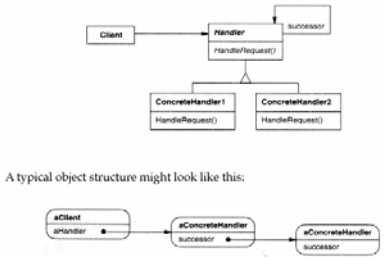
Describes how to use recursive composition.
An abstract class represents both primitives and their containers/composites, which means that the user does not have to distinct between the two. It has methods for managing the composites. It also contains methods for the primitives, the composites forwards these requests, possibly performing operations before/after forwarding.
The client can ignore the difference between compositions of objects and individual objects.
Disadvantage: Hard to restrict the components of a composite (there is no distinction).
Related patterns: Iterator can be used to traverse compositions. Flyweight lets you share components, but they can no longer refer to their parents. The component-parent link is often used for a Chain of Responsibility.



Behavioral patterns

Chain of Responsibility

Give more than one object a chance to handle a request, by avoid coupling the sender to a receiver. Like exception handling in Java.
A receiver of a request can decide to handle the request or to pass it up the chain. The pattern does not guarantee that the request is handled.
Use when: You want to send a request without knowing who handles it (reduced coupling). The set objects that can handle the request can be specified at runtime.
Related patterns: Often applied in conjunction with Composite.

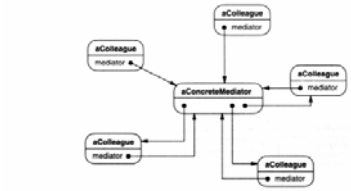


A typical object structure might look like this:

Mediator pattern

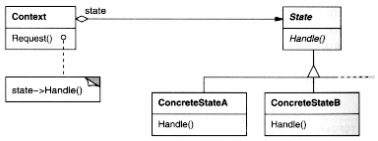
If multiple objects interact together e.g. in a dialog box it can be useful to move the responsibility of this interaction into one separate mediator object. Each object must inform the mediator if it changes state. The mediator can then change the other objects appropriately (e.g. enable/disable buttons).
MISSING CONTENT

A typical object structure might look like this:



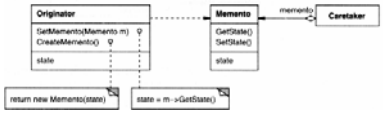
State

Allow an object to change its behavior when it's internal state changes. The object will appear to change its class.
This avoids long conditional statements that depend on the object state.
The pattern does not define who make the state transition – the Context or the state class itself. If creating individual state objects it must be considered to create them when needed or initially.
Related patterns: Flyweight explains when and how State objects can be shared. State objects are often Singletons.



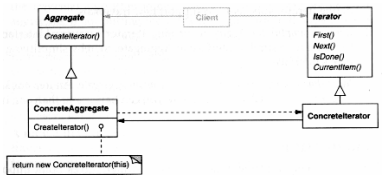
Memento

Without violating encapsulation, capture and externalize an objects internal state so it can be restored later.
Use when state must be saved and the saved state cannot be exposed outside the object.
The state object only stores the information necessary to restore the original object. The Caretaker stores the Memento state objects safely from other classes (may be difficult in some languages). The Caretaker deletes the Memento state objects and may only store a predefined no of objects. If stored in order it may only contain changed values.
Related patterns: Command pattern use mementos for undoable operations.



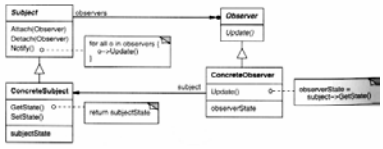
Iterator

A way to access elements sequentially in a collection without knowing the actual implementation of the collection.
Java does this through The Collections Framework (e.g. Iterator interface).
A uniform interface for traversing aggregate structures.
CreateIterator() is a Factory Method.
Related patterns: Iterators are often applied to recursive structures such as Composites.



Observer

When an object changes state, all its dependents (observers) are notified. The observer then decides whether to ignore or handle the notification.
The pattern ensures consistency without making the classes tightly coupled (reduces their reusability).
Abstract coupling between Subject and Observer: The subject doesn't know the concrete class of any observer.
Observers can be added and removed at any time.
Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.
Related patterns: Mediator is a intermediate object between many objects, whereas the Observer talks directly to it's observers.????????

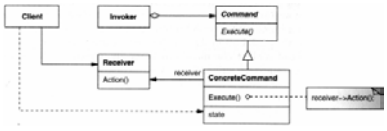


Command

Encapsulates a request as an object, thereby decoupling the object that invokes the method from the one that knows how to perform it.
A concrete command implements a command interface and may store a reference to the object it will execute on.
Multiple commands can be joined using the Composite pattern.
The pattern can:

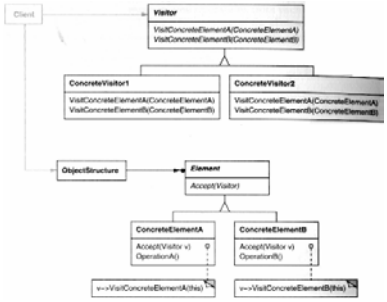
- Support undo by keeping a history of command and reversing commands using an Unexecute() method. Memento can be used to undo object state changes.
- Support logging, so the commands can be reapplied in case of system crash.
- Reflect commands like transactions

Related patterns: Composite and Memento as mentioned. A command that must be copied before being placed on the history list acts as a Prototype.



Visitor

Unrelated classes can share mutual methods in a separate visitor class. Instead of binding methods statically into the Element interface you can consolidate the methods in a Visitor and use an accept() method to do the binding at run-time.
How: The client creates a ConcreteVisitor and traverses the object structure calling accept(ConcreteVisitor) on each element. The element then calls the validate method in the ConcreteVisitor that corresponds to its class.
Adding methods to Element amounts to defining one new Visitor subclass rather than man new Element subclasses.
Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are portioned in their own visitor subclasses.
When to use: If the Element class hierarchy is stable but new methods are added the Visitor pattern will help. If the class hierarchy changes frequently it may be better to define methods in the classes in the class hierarchy.
Disadvantage: The pattern often forces you to provide public methods that access an elements internal state (compromise encapsulation).
Related patterns: Visitors can be used to apply an operation over an object structure defined by the Composite pattern. Visitor may be applied to do the Interpretation.



Strategy

Extract complex algorithm from a class. Enables you to switch algorithm easily and share algorithms with other classes.
Related patterns: Algorithms are often flyweights.