Design Patterns

Lecturer: Kevin Jackson
Group: Work Insurance

Vu Xuan Bach – s3298809
Le Hoang Hai – s3298775
Nguyen Ba Dao – s3296796
Nguyen Thanh Luan – s3312335

11/23/2012

Table of Contents

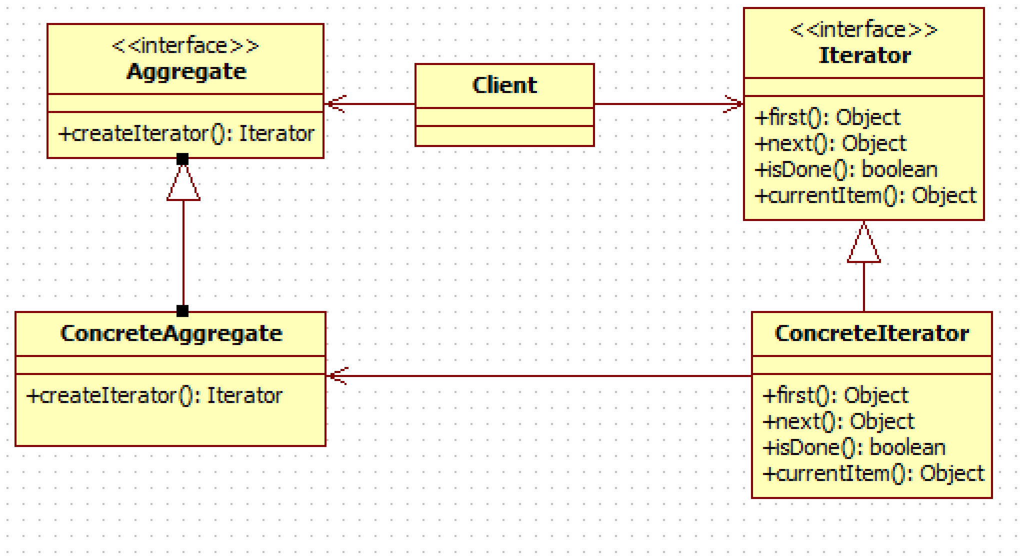
I. Iterator pattern 1
1. Description 1
2. Implementation 1
3. Advantages and disadvantages 2
4. Example 2
5. Alternative 3
II. Flyweight Pattern 4
1. Description 4
2. Implementation 4
3. Advantages and Disadvantages 5
4. Example: Text formatting and editing 6
5. Related patterns 6
III. Abstract Factory pattern 7
1. Description 7
2. Implementation 7
3. Advantages and disadvantages 8
4. Example: Look and Feel 9
5. Detailed Implementation 10
6. Alternative 10
IV. Chain of Responsibility pattern 11
1. Description 11
2. Implementation 11
3. Advantages and disadvantages 12
4. Example 12
5. Detailed implementation 13
V. Reference 14

I. Iterator pattern

1. Description

Iterator pattern provides an interface called Iterator to access the aggregate and still keeps the encapsulation of its representation.

2. Implementation



The classes that participate to the Iterator pattern are:

Aggregate - defines the Aggregate.

ConcreteAggregate – define the implementation of the collection.

Iterator - defines the iterator.

ConcreteIterator - defines implementation of the iterator.

Client - uses Aggregate and Iterator interfaces to traverse the collection in ConcreteIterator.

3. Advantages and disadvantages

Advantages:

- Access the content but still encapsulate the aggregate object.
  - Client calls the iterator to traverse through the collection
  - Depend on the type of collection, the iterator will use certain methods.
- Take the responsibility of traversing different aggregate structures.
  - Client does not need to know the actual implemented collection.
- Support multiple traversals of aggregate objects.
  - Each aggregate object can have many iterator for different traversals

Disadvantages:

- Do not have some useful method (add, set,...)
  - The iterator only provides a way to access, not modify an aggregate object.
- The order of the iteration is fixed in the implementation.
  - Each iterator has its own traversal algorithm.
  - If we need a new traversal, we should have a different iterator

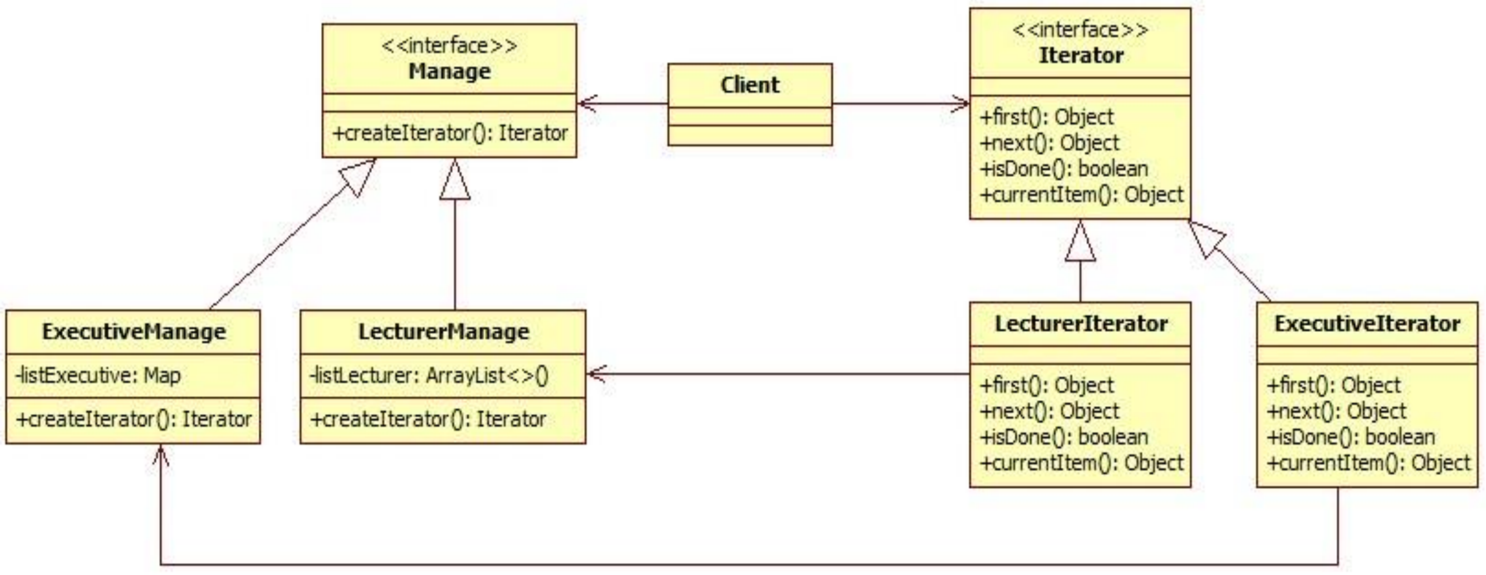
Usages:

- When the system needs to be independent of how its products are created composed and represented.
- When the system needs to be configured with one of multiple families of products.

4. Example

We have two classes for managing executives and lecturers in the university. We use ArrayList for lecturer management because it is easy to extend. On the other hand, the executive position is usually not duplicated, so we can use Map to store the position and information of executive people.

As a result, we have two different kinds of collection and we can use Iterator pattern to traverse both of them.



5. Alternative

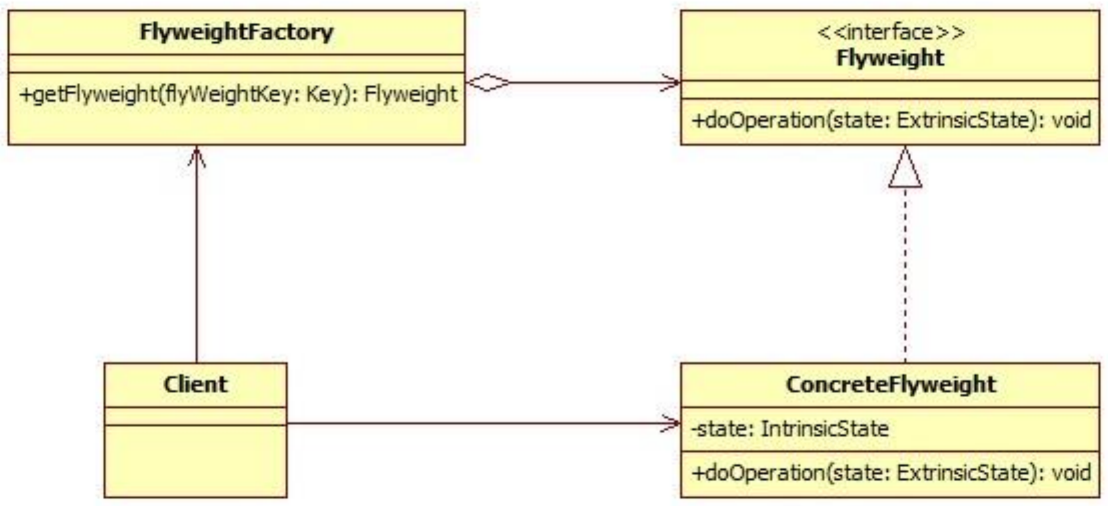
Visitor pattern can be used to traverse the object structure. The Iterator pattern is ussually used on collections with same type objects. On the other hand, the Visitor pattern can be used to traverse complex structures (hierarchical structures or composite structures). The Visitor pattern also helps to perform operations on the objects which have different interfaces.

II. Flyweight Pattern

1. Description

Flyweight is one of the structural design patterns. The intent of this pattern is to use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary. Flyweight was invented in order to solve the problems that some programs require a large number of objects that have shared state among them and they often consume lots of memory and may incur unacceptable runtime overhead.

2. Implementation


FlyweightFactory
+getFlyweight(flyWeightKey: Key): Flyweight
<<interface>>
Flyweight
+doOperation(state: ExtrinsicState): void
Client
ConcreteFlyweight
-state: IntrinsicState
+doOperation(state: ExtrinsicState): void

Flyweight – Declares an interface through which flyweights can receive and act on extrinsic state

ConcreteFlyweight – Implements the Flyweight interface and stores intrinsic state. A ConcreteFlyweight object must be sharable. The Concrete flyweight object must maintain state that it is intrinsic to it, and must be able to manipulate state that is extrinsic.

FlyweightFactory – The factory creates and manages flyweight objects. In addition the factory ensures sharing of the flyweight objects. The factory maintains a pool of different flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.

Client – A client maintains references to flyweights in addition to computing and maintaining extrinsic state

A client needs a flyweight object; it calls the factory to get the flyweight object. The factory checks a pool of flyweights to determine if a flyweight object of the requested type is in the pool. If there is, it returns that object. If there is no object of the required type, the factory creates a flyweight of the requested type, adds it to the pool, and returns it. The flyweight maintains intrinsic state which is shared among the large number of objects that we have created the flyweight for. It also provides methods to manipulate external state which vary from object and is not common among objects that we have created the flyweight for.

3. Advantages and Disadvantages

Advantages

- Reuse effectively objects
- Memory saving

Disadvantages

- Introduce runtime costs with
  - Transferring
  - Finding
  - Computing extrinsic state

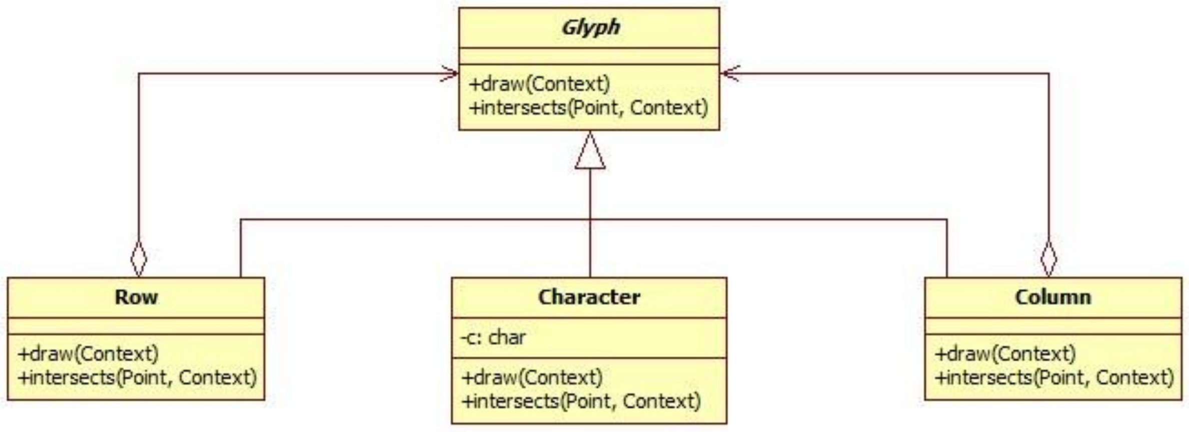
Usage:

- The more flyweights are shared, the greater the storage savings.
- The saving increase with the amount of share state.
- The greatest savings occur when the objects use substantial quantities of both intrinsic state, and the extrinsic state can be computed rather than stored.

4. Example: Text formatting and editing

Object-oriented document editors typically use objects to represent embedded elements like tables and figures. However, they usually stop short of using an object for each character in the document in order to increase flexibility. The characters could then be formatted and drawn. The application could be extended to support new character sets without disturbing other functionalities.

If we just normally create object for each character, it will consume lots of memory and huge overhead. A moderate sized document may require hundreds of thousands of character objects. On the other hands, using Flyweight pattern will save lots of memory. A flyweight is created for each letter of the alphabet. Each flyweight stores a character code, but its coordinate position in the document and its style can be determine from the text layout algorithms and formatting commands. We can say that the character code is intrinsic state, while other information is extrinsic state. Basically, there is one shared flyweight per character, and it appear in different contexts in the document structure. Each occurrence of particular character object refers to the same instance pool of flyweight objects.


Glyph
+draw(Context)
+intersects(Point, Context)
Row
+draw(Context)
+intersects(Point, Context)
Character
-c: char
+draw(Context)
+intersects(Point, Context)
Column
+draw(Context)
+intersects(Point, Context)

5. Related patterns

Factory and Singleton patterns: Flyweights are usually created using a factory and the singleton is applied to that factory so that for each type or category of flyweights, a singleton instance is returned.

State and Strategy patterns: State and Strategy objects are usually implemented as Flyweights because State and strategy increase number of objects if an application. We can reduce this overhead by implementing them as stateless objects that contexts share.

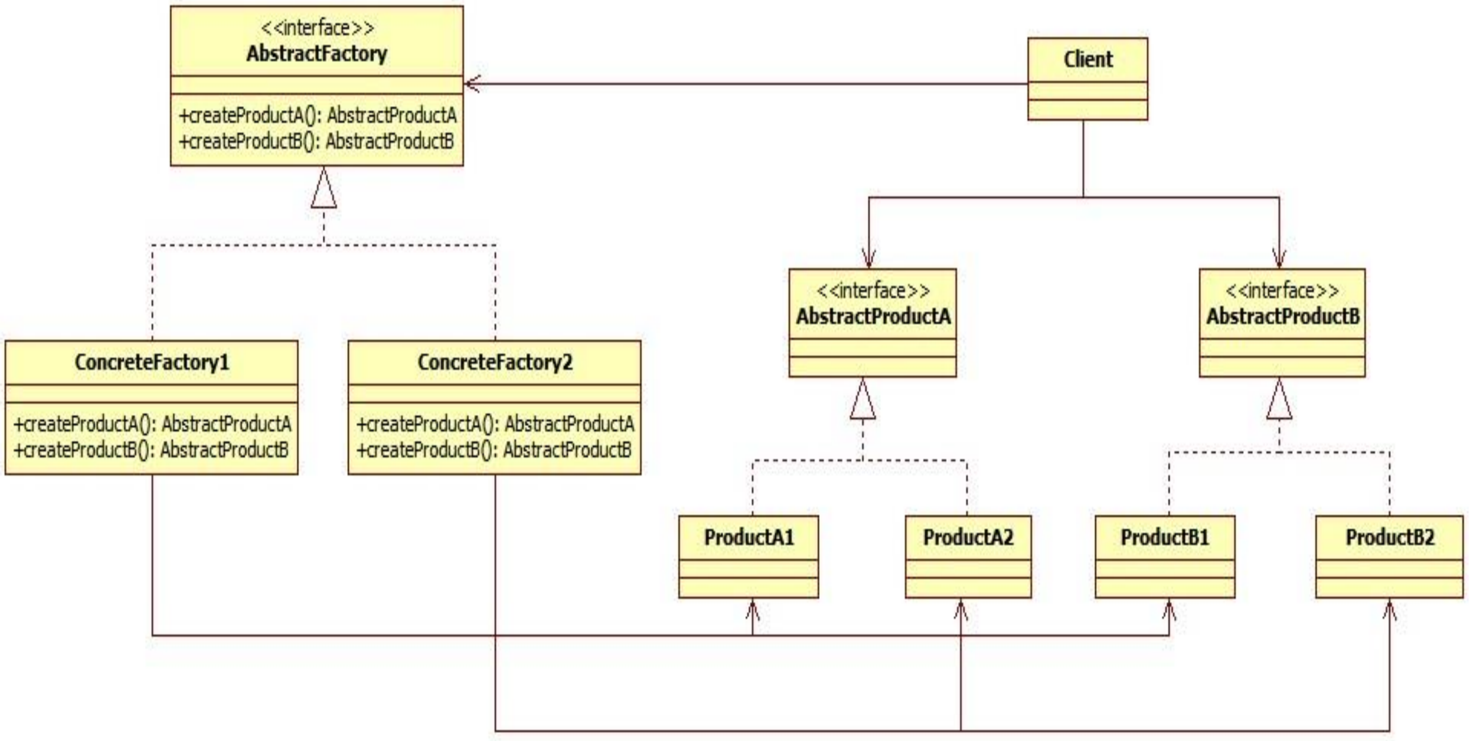
Composite pattern: The flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.

III. Abstract Factory pattern

1. Description

Abstract Factory pattern provides an interface for creating a family of related and dependent objects without specifying their concrete classes.

2. Implementation



<<interface>>
AbstractFactory
+createProductA(): AbstractProductA
+createProductB(): AbstractProductB
Client
ConcreteFactory1
+createProductA(): AbstractProductA
+createProductB(): AbstractProductB
ConcreteFactory2
+createProductA(): AbstractProductA
+createProductB(): AbstractProductB
<<interface>>
AbstractProductA
<<interface>>
AbstractProductB
ProductA1
ProductA2
ProductB1
ProductB2

The classes that participate to the Abstract Factory pattern are:

AbstractFactory - declares a interface that returns abstract products.

ConcreteFactory - implements AbstractFactory and is responsible for creating concrete products.

AbstractProduct - declares an interface for a type of product family.

Product - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.

Client - uses AbstractFactory and AbstractProduct interface to do get the product.

The AbstractFactory interface is the one that decides and creates the actual type of the concrete object and returns an abstract pointer to that object. This helps establish an encapsulation between client and objects, hiding the client from knowing anything about how the object is created and what factory it is going to use. This implies that there is no need for including any class declarations relating to the concrete objects which are accessed by the client only through the abstract interface.

Another objective of Abstract Factory pattern is that when a new concrete object type is needed, it is only required to make the client code use a different factory. This makes it much easier than instantiating a new type, which requires changing the code wherever a new object is created.

3. Advantages and disadvantages

Advantages

- It isolates construction classes from the client.
  - AbstractFactory is used to control objects creation.
  - Client gets the object through AbstractFactory interface.
- Exchanging product families is easy.
  - None of the client code breaks because it is encapsulated by AbstractFactory interface and composition usage.
  - Only one product family changes when the concrete factory is changed.
- It promotes consistency among products.
  - It is the concrete factory's job to make sure that the right products are used together.

Disadvantages

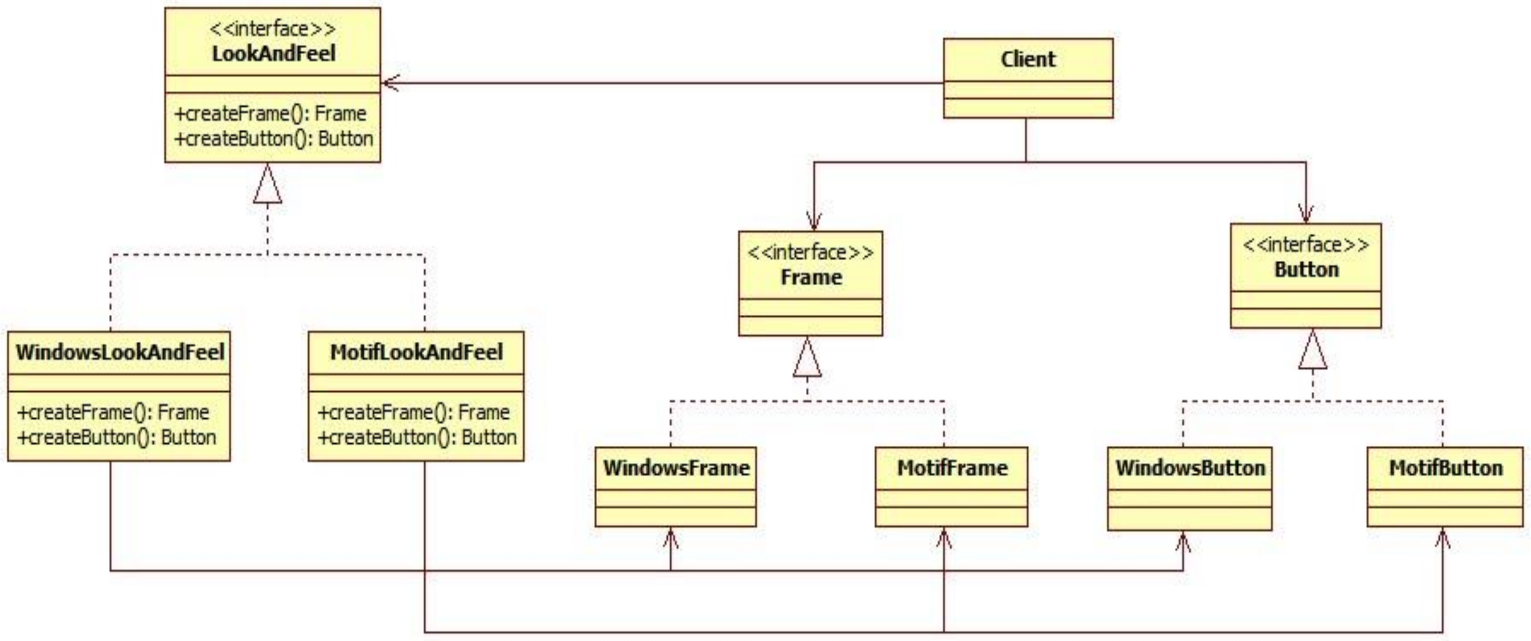
- Adding new products requires extending the abstract interface which causes all of its concrete classes to change:
  - New abstract product class is added
  - New product implementation is added
  - Abstract factory interface is extended
  - Derived concrete factories must implement the extensions
  - Client has to be extended to use the new product

Usage

- When the system needs to be independent of how its products are created composed and represented.
- When a family of product exists and need to be used together.
- The system need to expose product family interface not implementation.

4. Example: Look and Feel

A GUI framework often contains different look and feel themes such as Motif and Windows look. Each theme specifies different looks and behaviors of controls like frame and button. . In order to avoid the confusion between each type of control we define an abstract class **LookAndFeel** interface which has concrete classes like **WindowsLookAndFeel** and **MotifLookAndFeel.** These classes goal is to create controls that are configured to their look and flavor.



5. Detailed Implementation

As we only need a single ConcreteFactory object; people often combine it with Singleton to ensure only one instance of a family of products is created.

In addition, Prototype pattern can be used in each family of product. This help increase performance and extensibility. One reason is prototype creates new objects by cloning one of a few stored prototypes. This makes instantiate new object quickly. Another reason is when a new product is needed instead of creating it, the existing prototype is cloned. This approach eliminates the need for a new concrete factory for each new family of products.

6. Alternative

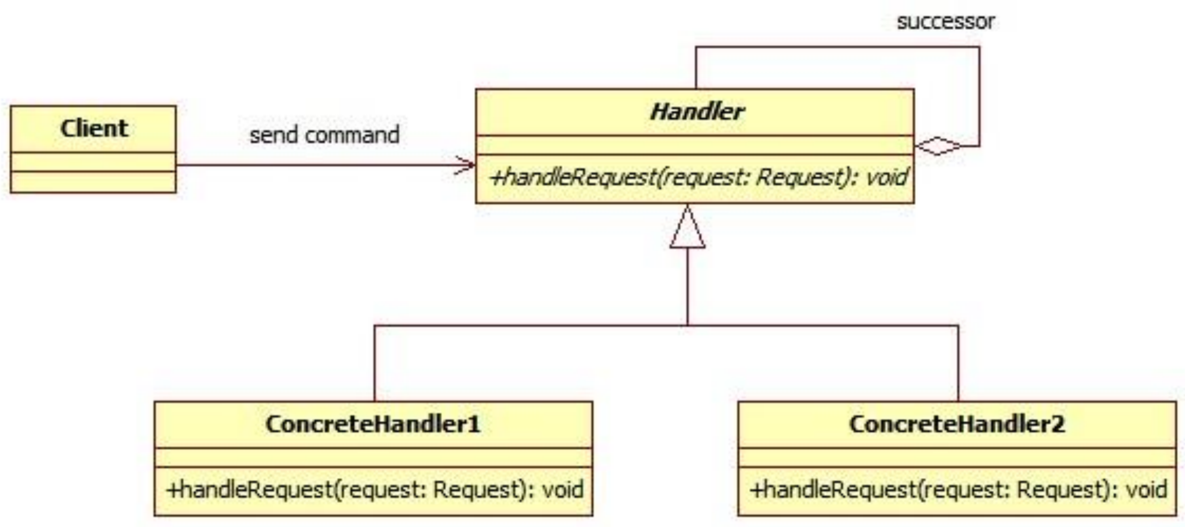
One can use Builder pattern instead of Abstract Factory because it has more flexibility. Instead of hiding all operation of how an object is created, Builder let client decide which attributes and behaviors an object should have when it is built. This offers the creation of complex objects and different representations of new objects. Builder pattern is suitable when it comes to encapsulate the instantiation of objects. Also, it is less error-prone as user will know what they are passing because of explicit method call. However, Builder pattern might become a very complex design for solving small problem. To sum up, it depends on what kind of system and construction that whether a design is suitably applied.

IV. Chain of Responsibility pattern

1. Description

Chain of responsibility pattern is combined by a group of processing objects and command objects. Each processing object will handle one type of command objects. Other types will be passed to the next processing object.

2. Implementation


successor
Client
send command
Handler
+handleRequest(request: Request): void
ConcreteHandler1
+handleRequest(request: Request): void
ConcreteHandler2
+handleRequest(request: Request): void

- **Handler** - defines an abstract class for handling requests.
- **ConcreteHandler** - handles the requests when in turn.
- **Client** - sends commands to the first object in the chain that may handle the command.

3. Advantages and disadvantages

Advantages:

- Reduce coupling of the request handler by dividing the process to many objects.
  - The receiver and the sender do not have knowledge of each other
  - The object in the chain does not know about the structure of the chain
- Add flexibility in assigning responsibilities to objects.
  - Add or change responsibilities for handling a request by adding or changing the chain at run-time

Disadvantage:

- Request is not guaranteed
  - A request can have no explicit receiver, so there is no guarantee it will be handled
  - A request can also be unhandled when the chain is not configured properly

Usage:

- Has many objects to handle a request
- Issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

4. Example

Shipping through electronic orders:

The methods to process an order online differ from one to another depending on each customer. The number of products, time, money transfer, way of shipment, special cases causes the system to be able to handle all requests. Chain of Responsibility pattern can accept requests with different information and pass them to a chain of order handlers until the suitable handler with the request is found. When special case occurs, new handler is simply added to the system.

5. Detailed implementation

One of the worst parts of Chain of Responsibility pattern is that it does not guarantee all requests are handled even if they go through all handlers. This makes the system performed poorly. One way to avoid this is to check whether a request is executed before. If not, then we ignore it and implement handler for that request later.

Furthermore, in order to make Chain of Responsibility more flexible, we should make Request class, as shown in diagram above, an interface. Using this way, when new request appears, we only need to extend the Request class to handle it. Another solution is to use xml file or database to store all possible requests containing data and pass them to handler when the system operates.

Often, Chain of Responsibility is used in conjunction with Composite pattern. This is applied when the handlers or objects becomes complex and developers would like a flexible as well as maintainable system. Each object, representing a group, can contain many other objects, leaf. This creates a whole hierarchy so that the requests can pass through all this to find correct handler.

V. Reference

CodeProject 2009, ‘Chain of Responsibility Pattern’, codeproject.com, viewed 17 Nov 2012, <http://www.codeproject.com/Articles/41786/Chain-of-Responsibility-Design-Pattern>

Gamma, E, Helm, R, Johnson, R, Vlissides, J 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edn, M.C. Escher, Holland.

JavaRevisited 2012, ‘Builder Design pattern in Java - Example Tutorial’, 29 June, viewed 18 Nov 2012, <http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html>

Kulkarn V. 2009, ‘Understanding Factory Method and Abstract Factory Patterns’, 5 July, viewed 22 Nov 2012, <http://www.codeproject.com/Articles/35789/Understanding-Factory-Method-and-Abstract-Factory>

Microsoft 2012, ‘Creational Patterns: Prototype, Factory Method, and Singleton’, viewed 18 Nov 2012, <http://msdn.microsoft.com/en-us/library/orm-9780596527730-01-05.aspx>

OODesign 2012, ‘Abstract Factory’, OODesign.com, viewed 16 Nov 2012, <http://www.oodesign.com/abstract-factory-pattern.html>

OODesign 2012, ‘Builder Pattern’, OODesign.com, viewed 22 Nov 2012, <http://www.oodesign.com/builder-pattern.html>

OODesign 2012, ‘Composite Pattern’, OODesign.com, viewed 17 Nov 2012, <http://www.oodesign.com/composite-pattern.html>

OODesign 2012, ‘Chain of Responsibility Pattern’, OODesign.com, viewed 17 Nov 2012, <http://www.oodesign.com/chain-of-responsibility-pattern.html>

OODesign 2012, ‘Flyweight Pattern’, OODesign.com, viewed 20 Nov 2012, <http://www.oodesign.com/flyweight-pattern.html>

Object Oriented Design, *Iterator pattern*, Object Oriented Design, viewed 20 November 2012, <http://www.oodesign.com/iterator-pattern.html>

Sourcemaking 2012, ‘Abstract Factory Design Pattern’, Sourcemaking – Teaching IT professionals, viewed 16 Nov 2012, <http://sourcemaking.com/design_patterns/abstract_factory>