Review of last class
# Design patterns

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Command</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Iterator</td>
</tr>
<tr>
<td>Factory</td>
<td>Composite</td>
<td>Observer</td>
</tr>
<tr>
<td>Singleton etc.</td>
<td>Decorator</td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td>Façade</td>
<td>Interpreter</td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>
Patterns to be covered

Builder
Factory Method
Singleton
Command
Iterator
Observer
Strategy
Creational Patterns

- Creational patterns abstract the instantiation process. They help make a system independent of how its objects are created.

- There are two recurring themes in creational patterns.
  - First, they all encapsulate knowledge about which concrete classes the system uses.
  - Second, they hide how instances of these classes are created and put together.

- Abstract Factory
- Singleton
- Builder
- Factory Method
Problem 1

- A reader for the RTF document exchange format should be able to convert RTF to many text formats.
- Conversion of documents from RTF to ASCII text.

Diagram:

- RTFtoASCII_Converter
  - Parse a RTF document and convert to ASCII

- RTFReader
  - Read and parse RTF

- ASCIIConverter
  - Convert to ASCII
Problem 1

• Idea: A reader for RTF format
  – Convert RTF to many text formats (EMACS, Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, HTML, SGML, WordPerfect 3.5, WordPerfect 7.0, ....)
  – Problem: The number of conversions is open-ended.

• Solution
  – Configure the RTF Reader with a “builder” object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market
Problem 1

While (t = GetNextToken()) {
    Switch t.Type {
        CHAR: builder->ConvertCharacter(t.Char)
        FONT: builder->ConvertFont(t.Font)
        PARA: builder->ConvertParagraph
    }
}

How to get the final product?
Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Where are the *new* operations?
Inside the `BuildPart()`.

Why isn’t the `GetResult()` in the `Builder` interface?
That would require a common interface for `ProductA` and `ProductB`.

Different directors may use the same `Builder`, and end up with different products.
Collaboration

• How does the *Director* get hold of a concrete *Builder*
• Draw a sequence diagram. – Page 99.

Note that the builder, not the director, has the final product. This totally isolates the code for construction and representation.
Applicability

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they’re assembled.
- The creation process must allow different representations for the object that is constructed.
Consequences

• It lets you vary a product’s internal representation.
• It isolates code for construction and representation. It improves modularity by encapsulating the way a complex object is constructed and represented.
  – Each concrete builder contains all the code to create and assemble a particular kind of product. The code is written once; the different directors can reuse it to build product variants from the same set of parts.
• Examples:
  – A house with 3 rooms, 2 hallways, 1 garage and 3 doors.
  – Who will carry the floor plan, the director or the builder?
  – Who will have the finished house?
  – A basic builder, a fancy builder.
  – How about a house with 4 rooms and 2 garage?
  – In this case, different directors but the same builders.
Some Implementation Issues

- Assembly and construction interface – the builder class interface must be general enough to allow the construction of products for all kinds of concrete builders.
  - What should be in builder’s interface in the previous house example?
- Sometimes you might need access to parts of the product constructed earlier. For example, building a door connecting two existing rooms. In that case, the builder would return built parts (rooms) to the director, which then would pass them back to the builder to build the new part (door).
- But does this means the director needs to know the room? – we already said that the internal representation of the product should be hidden.
- The book provided an example where indexes are used. For example: BuildRoom(int room), BuildDoor (int roomFrom, int roomTo);
- Empty methods as default in Builder.
- Why not abstract class for product?
  - The products produced by the concrete builders differ greatly thus there is little to gain from giving different products a common parent class.
  - Because the client configures the concrete builder, the client is in a position to know and to handle its products accordingly.
Abstract Factory vs Builder

• Abstract Factory
  – Focuses on product family
    • The products can be simple ("light bulb") or complex ("engine")

• Builder
  – Focuses on constructing a complex object (e.g., an intelligent home system) step by step.
  – The underlying product needs to be constructed as part of the system, but the creation is very complex
  – The construction of the complex product changes from time to time

• Builder returns the product as a final step, but for Abstract factory, the product get returned immediately.

• Abstract Factory and Builder work well together for a family of multiple complex products.
  – How do they work together? Consider the example of constructing a GUI for Mac OS or Window95 OS.
Abstract Factory and Builder work well together for a family of multiple complex products.

Consider the example of constructing the above GUI for Mac OS or Window95 OS.

- The GUI needs to be constructed in a specific procedure, e.g., creating the three buttons and then add them to the button panel...
- The buttons, panels, and frames for Mac OS and Window95 OS have different look-and-feel style.
Sample code

P 101
Problem 2

- Frameworks use abstract classes to define and maintain relationships between objects. Clients have to subclass them to realize their application-specific implementations.
- A framework is often responsible for creating objects as well.
- The application only knows *when* a new document should be created, but not *what kind* of document to create.
- This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.

```
MyApplication
  +-- CreateDocument()
  |    +-- NewDocument()
  |    +-- OpenDocument()
  +-- Document doc = CreateDocument();
      doc.Open();
```

```
MyDocument
  +-- Document

Application
  +-- CreateDocument()
      +-- Document doc = CreateDocument();
          doc.Open();

Document
  +-- Open()
  +-- Close()
  +-- Save()
  +-- Revert()
```
Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

Also known as virtual constructor.
Applicability

Use the Factory Method when

- A class can’t anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
Structure

Product

Creator

ConcreteProduct

ConcreteCreator

FactoryMethod()

AnOperation()

... Product = FactoryMethod();
... Return new ConcreteProduct;
Consequences

• Factory Methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined concreteProduct classes.

• Provides hooks for subclasses. Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory method gives subclasses a hook for providing an extended version of an object – the subclass can override the parent’s factory method. Note that in this case, the factory method is not abstract.
Some Implementation Issues

• Two major varieties
  – The case when the Creator class is an abstract class. This requires subclasses to define an implementation.
  – The case when the Creator class is a concrete class and provides a default implementation for the factory method. In this case, the concrete creator uses the factory method primarily for flexibility.

• Parameterized factory methods
Sample code

P 114
Discussion of creational patterns

There are two common ways to parameterize a system by the classes of objects it creates.
One way is to subclass the class that creates the objects – this corresponds to using the Factory Method pattern.

The other way to parameterize a system relies more on object composition: define an object (a “factory object”) that’s responsible for knowing the class of the product objects – this is a key aspect of the Abstract Factory, Builder.
Behavioral Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.
Problem 4

- Sometimes it’s necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
- You want to build a user interface toolkit
- You want to provide menus
- You want to make the user interface reusable across many applications
  - You cannot hardcode the meanings of the menus for the various applications
  - Only the applications know what has to be done when a menu is selected.
How about implementing the Menu class to have multiple methods, each of which is corresponding to a command, such as open, close?

This is hard coded, not a toolkit. The number of menu items should be able to be customized.

The MenuItem class makes it possible to customize the Menu with a variety number of menu commands

How to implement the Clicked()? 

How about defining the MenuItem as an abstract class, and ask user to implement their specific MenuItems?

This means the meaning of each menu item cannot be changed. Furthermore, how to support redo, undo, save the command? How about changing the meaning of the MenuItem dynamically? How about macro commands?

We need a more flexible design!
Problem 4

- The application configures each MenuItem with an instance of a concrete Command subclass.
- MenuItems doesn’t know which subclass of Command they use.

```
• The application configures each MenuItem with an instance of a concrete Command subclass.
• MenuItems doesn’t know which subclass of Command they use.
```
Problem 4

![Diagram showing the relationship between Command, Execute(), PasteCommand, Execute(), and Document.paste()](image-url)
Problem 4

```
Application
Add(document)

OpenCommand
Execute()
askUser()

Command
Execute()

fileName = askUser()
Doc = new Document(fileName)
Application.add(doc)
Doc.open()
```
Problem 4 – Macro Command

`Command`  
`Execute()`  

For all `c` in commands  
`c.execute()`
Problem 4 -- undo

Document
- Open()
- Close()
- Cut()
- Copy()
- Paste()

Command
- Execute()
- unExecute()

PasteCommand
- Execute()
- unExecute()

Document.paste(text)
Document.delete(text)

Dr. Xiaolin Hu
Flexibility

- Decouple the object that invokes the operation from the one having the knowledge to perform it.
- An application can provide both a menu and a push button interface for the same feature by sharing the same command subclass.
- We can replace command dynamically, for implementing context-sensitive menus.
- Support command scripting by composing commands into larger ones.
- Save executed commands in a queue and supports undo and redo.
Command Pattern

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

• Also known as Action, Transaction
Applicability

“Encapsulate a request as an object, thereby letting you

– parameterize clients with different requests,
– queue or log requests (by augmenting the command interface with load and store operations)
– support undoable operations (the command interface must have an added unexecute operation that reverses the effects of a previous call)

Support logging changes so that they can be reapplied in case of a system crash

Structure a system around high-level operations built on primitives operations – support transactions.
• **Client** *(the Application)* creates a **ConcreteCommand** and binds it with a **Receiver** *(the Document)*.

• **Client** hands the **ConcreteCommand** over to the **Invoker** *(the MenuItem)* which stores it.

• The **Invoker** has the responsibility to do the command including “execute” or “undo”.

Dr. Xiaolin Hu
Collaborations

Sequence diagram P 237
Consequences

- Command decouples the object that invokes the operation from the one that know how to perform it
- Commands are first-class objects. They can be manipulated and extended like any other object
- You can assemble commands into a composite command.
- It is easy to add new commands, or change to other commands in runtime.
Some Implementation Issues

• How intelligent should a command be?
• Supporting undo and redo. –a command should provide a way to reverse their execution. A command may need to store additional state to do so.
• An application can have a history list of commands that have been executed.
Sample code

P. 239
Problem 5

- You have a list
- You want to traverse the list (current, next, previous).
- How about adding methods such as current(), next(), and previous() in the List interface?
- You want to have a different traverse such as filteringTraverse.
- You want to have multiple traverses at the same time.
Problem 5

• An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.

• Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish.

• But you don’t want to bloat the list interface with operations for different traversals.

• You might also need to have more than one traversal pending on the same list.
The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object.

How to create the ListIterator?

- The client instantiates the ListIterator (by providing the List as a parameter). Or more commonly, the List instantiates the ListIterator.
Iterator Pattern

• Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
• Also known as cursor
Applicability

Use the iterator pattern

To access an aggregate object’s contents without exposing its internal representation.

To support multiple traversals of aggregate objects

To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration) – the skipList example in the book.
Structure

**Aggregate**
- `CreateIterator()`

**ConcreteAggregate**
- `CreateIterator()`

**Factory Method**

**Client**

**Iterator**
- `First()`
- `Next()`
- `IsDone()`
- `CurrentItem()`

**ConcretelIterator**

Return new ConcretelIterator(this)
Consequences

• It supports variations in the traversal of an aggregate
• Iterators simplify the aggregate interface
• More than one traversal can be pending on an aggregate.
Some Implementation Issues

• Who controls the iteration? Client controls: external iterator; iterator control: internal iterator

• How robust is the iterator – it can be dangerous to modify an aggregate while you’re traversing it.

• Iterators for composites – internal iterator is better.
Sample code

P. 263
Problem 6

• A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
• For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data.
• Classes defining application data and presentations can be reused independently.
• The number of user interfaces is expandable.
• The different user interfaces do not know each other, but they behave as they do – when the user change the information in one user interface, the other UIs reflects the changes immediately. (demo using File Manager in Windows)
• The Observer pattern describes how to establish these relationships.
Observer pattern

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

- Also called “Publish and Subscribe”
Applicability

• Use the observer pattern
  – When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  – When a change to one object requires changing others and you don’t know how many objects need to be changed.
  – When an object should be able to notify other objects.
- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) *not* an interface.
Collaborations

AConcreteSubject

Attach(this)

setState("new state")

notify()

AConcreteObserver

AConcreteObserver

Attach(this)

anotherConcreteObserver

update()

update()

gateState()
Consequences

• Abstract coupling between Subject and Observer – the subject doesn’t know the concrete class of any observer. This provides support for a layered architecture.

• Support for broadcast communication – the notification that a subject sends needn’t specify its receiver.

• Unexpected updates – a seemingly harmless operation on the subject may cause a cascade of updates to observers and their dependent objects.
Some Implementation Issues

• Observing more than one subject – the subject can pass itself as a parameter in the update() to allow the observer to know which subject to examine.

• Who triggers the update?
  – Have state-setting operations (e.g., the setState() method) on Subject call Notify after they change the subject’s state
  – Make clients responsible for calling Notify at the right time – avoid needless intermediate updates

• Avoiding observer-specific update protocols
  – Push model
  – Pull model
Sample code

P. 300
Problem 7

• Many different algorithms exists for the same task
• Examples:
  – Breaking a stream of text into lines
  – Parsing a set of tokens into an abstract syntax tree
  – Sorting a list of customers
• The different algorithms will be appropriate at different times
  – Rapid prototyping vs delivery of final product
• We don’t want to support all the algorithms if we don’t need them
• If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm
Problem 7

Maintain and update the linebreaks of text displayed in a test viewer using a linebreaking algorithm.

Composition & Compositor

Composition  Compositor

Maintain and update linebreaks

Linebreaking algorithms

A big and hard-to-maintain client

Dr. Xiaolin Hu
Strategy pattern

• Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

• Also known as Policy
Applicability of Strategy Pattern

• Many related classes differ only in their behavior. Strategy allows to configure a single class with one of many behaviors
• Different variants of an algorithm are needed that trade-off space against time. All these variants can be implemented as a class hierarchy of algorithms
• An algorithm uses data that clients shouldn’t know about. Use the strategy pattern to avoid exposing complex, algorithm-specific data structures.
Policy decides which Strategy is best given the current Context
Applying a Strategy Pattern in a Database Application

```
<table>
<thead>
<tr>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search()</td>
</tr>
<tr>
<td>Sort()</td>
</tr>
</tbody>
</table>

Strategy

Sort()

- *Strategy
  - BubbleSort
    - Sort()
  - QuickSort
    - Sort()
  - MergeSort
    - Sort()
```
Another Example

Handout
Consequences

• Families of related algorithms
• Separating strategies out makes it easier to subclassing Context
• Clients must be aware of different Strategies
• Communication overhead between strategy and context – some simple concreteStrategy may not use all the information provided by the context.
Compare with the Decorator Pattern

Consider the example of a textView with different styles of Border. How to design this using the decorator design pattern? How to design this using the strategy design pattern? – P. 180.

The component doesn’t have to know anything about its decorators

The component itself knows possible extensions
Sample code

P. 320
Discussion of Behavior Patterns

Encapsulating variation is a theme of many behavioral patterns.

When an aspect of a program changes frequently, these patterns define an object that encapsulates that aspect.

Strategy
Iterator
Discussion of Behavior Patterns

Objects as argument
Several patterns introduce an object that is always used as an argument.
Command
Discussion of Behavior Patterns

Decoupling senders and receivers – when collaborating objects refer to each other directly, they become dependent on each other.

The command pattern supports decoupling by using a Command object to define the binding between a sender and receiver.

The observer pattern decouples senders (subject) from receivers (observers) by defining an interface for signaling changes in subjects.
http://javarevisited.blogspot.com/2012/06/20-design-pattern-and-software-design.html