Design Patterns

V22.0474-001 Software Engineering
Lecture 9, Spring 2007

Clark Barrett, New York University
How Do We Design Software?

• We all understand
  - Algorithms
  - Data structures
  - Classes

• When describing a design, algorithms/data structures/classes form the vocabulary

• But there are higher levels of design
Design Patterns: History

• Christopher Alexander
  - An architect
  - A Berkeley professor
  - The father of design patterns
    • As applied to architecture
    • “Pattern Languages” (1977)

• The Gang of Four
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Application of design patterns to object-oriented programming
  - Book: Design Patterns: Elements of Reusable Object-Oriented Software
What are Design Patterns?

“A pattern describes a problem that occurs often, along with a tried solution to the problem”
- Christopher Alexander, 1977

- Descriptions of **communicating objects** and classes that are **customized** to solve a general **design problem** in a particular context
  - Not individual classes or libraries
    - Such as lists, hash tables
  - Not full designs
Elements of a Design Pattern

1. Pattern name
   - Useful part of design vocabulary
2. Problem solved and applicability
   - When to apply the pattern
3. Solution
   - Participants and their relationships
4. Consequences
   - Costs of applying the pattern, space and time trade-offs
Improved Communication

One of the main benefits of design patterns is that they name common (and successful) ways of building software.
More Specifically

• Teaching and learning
  - It is much easier to learn architecture from descriptions of design patterns than from reading code

• Teamwork
  - Members of a team have a way to name and discuss the elements of their design
Example: A Text Editor

- Describe a text editor using patterns
  - A running example

- Introduces several important patterns

- Gives an overall flavor of pattern culture

Note: This example is from the "Design Patterns" book.
Text Editor Requirements

- A WYSIWYG editor
- Text and graphics can be freely mixed
- Graphical user interface
  - Toolbars, scrollbars, etc.
- Multiple windowing systems
- Traversal operations: spell-checking, hyphenation
The Game

• I describe a design problem for the editor

• I ask “What is your design?”
  - This is audience participation time

• I give you the wise and insightful pattern
Problem: Document Structure

A document is represented by its physical structure:

- **Primitive glyphs**
  - characters, rectangles, circles, pictures, ...
- **Lines**
  - A sequence of glyphs
- **Columns**
  - A sequence of lines
- **Pages**
  - A sequence of columns
- **Documents**
  - A sequence of pages

*What is your design?*
Alternative Designs

• **Classes for Character, Circle, Line, Column, Page, ...**
  - Not so good
  - A lot of code duplication

• **One (abstract) class of Glyph**
  - Each element realized by a subclass of Glyph
  - All elements present the same interface
    - How to draw
    - Compute bounding rectangle
    - Mouse hit detection
    - ...
  - Makes extending the class easy
  - Treats all elements uniformly

Adapted from Prof. Necula  CS 169, Berkeley
### Example of Hierarchical Composition

<table>
<thead>
<tr>
<th>A</th>
<th>c</th>
<th>a</th>
<th>r</th>
</tr>
</thead>
</table>

**Character glyph**

**Line glyph** (composite)

**Column glyph** (composite)

**Picture glyph**
Diagram

Glyph
- Draw(Window)
- Intersects(Point p)
- ...

Character
- Draw(Window)
- Intersects(Point p)
- ...

Picture
- Draw(Window)
- Intersects(Point p)
- ...

Line
- Draw(Window)
- Intersects(Point p)
- ...

children

Adapted from Prof. Necula  CS 169, Berkeley
Notes

• Drawing
  - Each primitive element draws itself
    • At its assigned location
  - Each compound element recursively calls draw on its elements
    • But doesn’t care what those elements are

```cpp
Line::Draw(Window w) {
    for each c in children do
        c->Draw(w);
}
```
Composites

- This is the *composite* pattern
  - Goes by many other names
    - Recursive composition, structural induction, tree walk, ...
    - Predates design patterns

- Applies to any hierarchical structure
  - Leaves and internal nodes have same functionality
  - Composite implements the same interface as the contained elements

Adapted from Prof. Necula  CS 169, Berkeley
Problem: Formatting

• A particular physical structure for a document
  - Decisions about layout
  - Must deal with e.g., line breaking

• Design issues
  - Layout is complicated
  - No best algorithm
    • Many alternatives, simple to complex

What is your design?
Not So Good

• Add a *format* method to each *Glyph* class

• Problems
  - Can’t modify the algorithm without modifying *Glyph*
  - Can’t easily add new formatting algorithms
The Core Issue

• Formatting is complex
  - We don’t want that complexity to pollute Glyph
  - We may want to change the formatting method

• Encapsulate formatting behind an interface
  - Each formatting algorithm an instance
  - Glyph only deals with the interface
Diagram

**Glyph**
- Draw(Window)
- Intersects(Point p)
- Insert(Glyph)

**Document**
- Draw(Window)
- Intersects(Point p)
- Insert(Glyph g)

**Formatter**
- Format()
- ...

**FormatWord**
- Format()
- ...

**FormatTex**
- Format()
- ...

Adapted from Prof. Necula  CS 169, Berkeley
Strategies

• This is the *strategy* pattern
  - Isolates variations in algorithms we might use
  - Formatter is the strategy, Compositor is context

• General principle
  *encapsulate variation*

• In OO languages, this means defining abstract classes for things that are likely to change
Problem: Enhancing the User Interface

• We will want to decorate elements of the UI
  - Add borders
  - Scrollbars
  - Etc.

• How do we incorporate this into the physical structure?

What is your design?
Not So Good

• Object behavior can be extended using inheritance
  - Major drawback: inheritance structure is static
• Subclass elements of Glyph
  - BorderedComposition
  - ScrolledComposition
  - BorderedAndScrolledComposition
  - ScrolledAndBorderedComposition
  - ...
• Leads to an explosion of classes
Decorators

- Want to have a number of decorations (e.g., Border, ScrollBar, Menu) that we can mix independently
  \[ x = \text{new ScrollBar(new Border(new Character))} \]

  - We have \( n \) decorators and \( 2^n \) combinations
Transparent Enclosure

• Define Decorator
  - Implements Glyph
  - Has one member Glyph decorated
  - Border, ScrollBar, Menu extend Decorator

```cpp
Border::Draw(Window w) {
    decorated->draw(w);
    drawBorder(decorated->bounds());
}
```
Decorators

• This is the *decorator* pattern

• A way of adding responsibilities to an object

• Commonly extending a composite
  - As in this example
Problem: Supporting Look-and-Feel Standards

• Different look-and-feel standards
  - Appearance of scrollbars, menus, etc.

• We want the editor to support them all
  - What do we write in code like
    ScrollBar scr = new ?

What is your design?
The Not-so-Good Strawmen

- Terrible
  
  ScrollBar scr = new MotifScrollBar

- Little better
  
  ScrollBar scr;
  if (style == MOTIF) then scr = new MotifScrollBar
    else if (style == ...) then ...
  - will have similar conditionals for menus, borders, etc.
Abstract Object Creation

- Encapsulate what varies in a class

- Here object creation varies
  - Want to create different menu, scrollbar, etc
  - Depending on current look-and-feel

- Define a `GUIFactory` class
  - One method to create each look-and-feel dependent object
  - One `GUIFactory` object for each look-and-feel
  - Created itself using conditionals
Diagram

```
GuiFactory
CreateScrollBar()
CreateMenu()

MotifFactory
CreateScrollBar() {
    return new MotifScrollBar();
}
CreateMenu() {
    return new MotifMenu();
}

MacFactory
CreateScrollBar() {
    return new MacScrollBar();
}
CreateMenu() {
    return new MacMenu();
}
...```

Adapted from Prof. Necula  CS 169, Berkeley
Diagram 2: Abstract Products

Glyph

ScrollBar
scrollTo(int);

MotifScrollBar
scrollTo(int);

MacScrollBar
scrollTo(int);

...
Factories

- This is the abstract factory pattern

- A class which
  - Abstracts the creation of a family of objects
  - Different instances provide alternative implementations of that family

- Note
  - The “current” factory is still a global variable
  - The factory can be changed even at runtime
Problem: Supporting Multiple Window Systems

- We want to run on multiple window systems

- Problem: Wide variation in standards
  - Big interfaces
    - Can’t afford to implement our own windowing system
  - Different models of window operations
    - Resizing, drawing, raising, etc.
  - Different functionality

What is your design?
A First Cut

- Take the intersection of all functionality
  - A feature is in our window model if it is in every real-world windowing system we want to support

- Define an abstract factory to hide variation
  - Create windowing objects for current window system using the factory

- Problem: intersection of functionality may not be large enough
Second Cut

- Define our own abstract window hierarchy
  - All operations we need represented
  - Model is tuned to our application

- Define a parallel hierarchy
  - Abstracts concrete window systems
  - Has all functionality we need
    - I.e., could be more than the intersection of functions
    - Requires writing methods for systems missing functionality
wimpl->DrawLine;
wimpl->DrawLine;
wimpl->DrawLine;
wimpl->DrawLine;

Diagram

Window
   DrawRect()
   ...

WindowImp
   DrawLine()
   ...

AppWindow

MacWindowImp
   DrawLine()

XWindowImp
   DrawLine()

IconWindow

...
Bridges

- This is the *bridge* pattern

- Note we have two hierarchies
  - Logical
    - The view of our application, tuned to our needs
  - Implementation
    - The interface to the outside world
    - Abstract base class, with multiple implementations

- Logical, implementational views can evolve
  - independently,
  - So long as the “bridge” is maintained
User Commands

• User has a vocabulary of operations
  - E.g., jump to a particular page
  - Operations can be invoked multiple ways
    • By a menu
    • By clicking an icon
    • By keyboard shortcut
  - Want undo/redo/command line option/menu option

• How do we represent user commands?

What is your design?
A Good Design

- Define a class of user operations
  - Abstract class
  - Presents interface common to all operations
    - E.g., undo/redo

- Each operation is a subclass
  - Jump to a page, cut, paste, ...
Diagram

Command

Execute()
Undo()

CutCommand

Execute()
Undo()

SaveCommand

Execute()
Undo()

…
Commands

• This is the *command* pattern

• Note the user has a small “programming language”
  - The abstraction makes this explicit
  - In this case the language is finite
    - Class structure can represent all possibilities explicitly

• Other patterns for richer languages
  - E.g., the Interpreter Pattern
Problem: Spell Checking

• Considerations
  - Spell-checking requires traversing the document
    • Need to see every glyph, in order
    • Information we need is scattered all over the document
  - There may be other analyses we want to perform
    • E.g., grammar analysis

What is your design?
One Possibility

- **Iterators**
  - Hide the structure of a container from clients
  - A method for
    - pointing to the first element
    - advancing to the next element
    - getting the current element
    - testing for termination

```java
iterator i = CreateIterator(composition);
for(i = i->first(); !(i->isdone()); i = i->next())
    { ... do something with Glyph i->current() ...; }
```
Notes

• Iterators work well if we don’t need to know the type of the elements being iterated over
  - E.g., send kill message to all processes in a queue
• Not a good fit for spell-checking

  for(i = i->first(); !(i->isdone()); i = i->next())
  {
    ... do something with Glyph i->current() ...;
  }

• Must cast i->current() to spell-check it . . .

  if(i instanceof Char) { ... } else { ... }

Adapted from Prof. Necula  CS 169,
Berkeley
Visitors

• The visitor pattern is more general
  - Iterators provide traversal of containers
  - Visitors allow
    • Traversal
    • And type-specific actions

• The idea
  - Separate traversal from the action
  - Have a “do it” method for each element type
    • Can be overridden in a particular traversal
### Diagram

**Glyph**
- Draw(Window)
- Scan(Visitor)
- ...

**Visitor**
- visitChar(Character)
- visitPicture(Picture)
- visitLine(Column)
- ...

**Character**
- Draw(Window)
- Scan(Visitor v) {
  v->visitChar(this);
}

**Picture**
- Draw(Window)
- Scan(Visitor v) {
  v->visitChar(this);
  v->visitPicture(this);
}

**Line**
- Draw(Window)
- Scan(Visitor v) {
  v->visitLine(this);
  for each c in children
  c->Scan(v)
}

Adapted from Prof. Necula CS 169, Berkeley
Visitor Comments

• The dynamic dispatch on Glyph::Scan achieves type-safe casting
  - dynamic dispatch to Char::Scan, Picture::Scan, ...

• Each of the Glyph::Scan
  - calls the visitor-specific action (e.g., Visitor::visitChar)
  - implements the search (e.g., in Line::Scan)

• Have a visitor for each action (e.g., spell-check, search-and-replace)
Design Patterns

• A good idea
  - Simple
  - Describe useful “micro-architectures”
  - Capture common organizations of classes/objects
  - Give us a richer vocabulary of design

• Relatively few patterns of real generality

• Watch out for the hype . . .