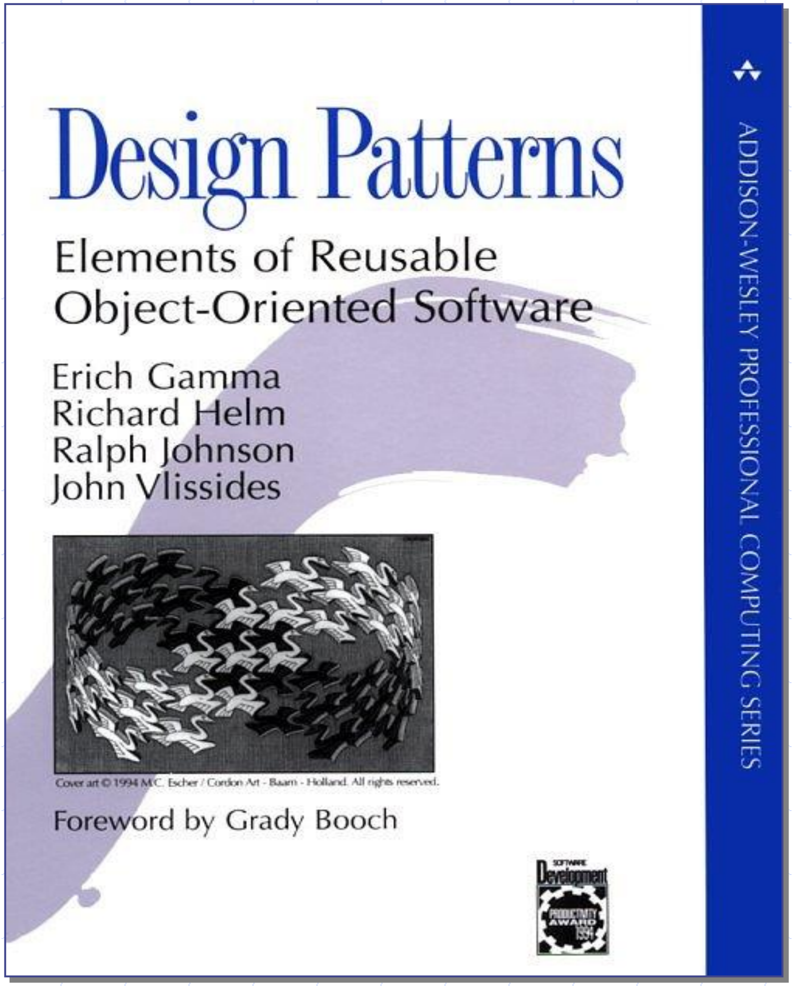
Introduction to Design Patterns



Overview

- Design Patterns in Software
  - Why?
  - What are design patterns?
  - What they're good for
  - How we develop / classify them
- The Iterator Pattern

Recurring Design Structures

OO systems exhibit recurring structures that promote

- abstraction
- flexibility
- modularity
- elegance

Therein lies valuable design knowledge

Problem: Capturing, communicating, and applying this knowledge

Design Patterns

- "Each *pattern* describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

  Christopher Alexander, *A Pattern Language: Towns/Buildings/Construction*, 1977

- A *object-oriented design pattern* systematically names, explains and evaluates an important and recurring design in object-oriented systems

Patterns in engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch
    - Reuse standard designs with successful track records
  - *Should software engineers make use of patterns? Why?*
    - Developing software from scratch is expensive
    - Patterns support reuse of software architecture and design

Design Patterns

- The landmark book on software design patterns is:

  Design Patterns: Elements of Reusable Object-Oriented Software

  Erich Gamma, Richard Helm,
  Ralph Johnson, John Vlissides

  Addison-Wesley, 1995

- This is also known as the GOF ("Gang-of-Four") book.

A *Design Pattern*…

- abstracts a recurring design structure
- comprises class and/or object
  - dependencies
  - structures
  - interactions
  - conventions
- names / specifies the design structure explicitly
- distills design experience

Goals

- Codify good design
  - distill / generalize experience
  - aid to novices / experts alike
- Give design structures explicit names
  - common vocabulary
  - reduced complexity
  - greater expressiveness
- Capture and preserve design information
  - articulate design decisions succinctly
  - improve documentation
- Facilitate restructuring / refactoring
  - patterns are interrelated
  - additional flexibility

Design Pattern Template (1st half)

NAME scope purpose

Intent
- short description of the pattern & its purpose

Also Known As
- Any aliases this pattern is known by

Motivation
- motivating scenario demonstrating pattern’s use

Applicability
- circumstances in which pattern applies

Structure
- graphical representation of the pattern using modified UML notation

Participants
- participating classes and/or objects & their responsibilities

Design Pattern Template (2nd half)

...

Collaborations

how participants cooperate to carry out their responsibilities

Consequences

the results of application, benefits, liabilities

Implementation

pitfalls, hints, techniques, plus language-dependent issues

Sample Code

sample implementations in C++, Java, C#, Smalltalk, C, etc.

Known Uses

examples drawn from existing systems

Related Patterns

discussion of other patterns that relate to this one

OBSERVER object behavioral

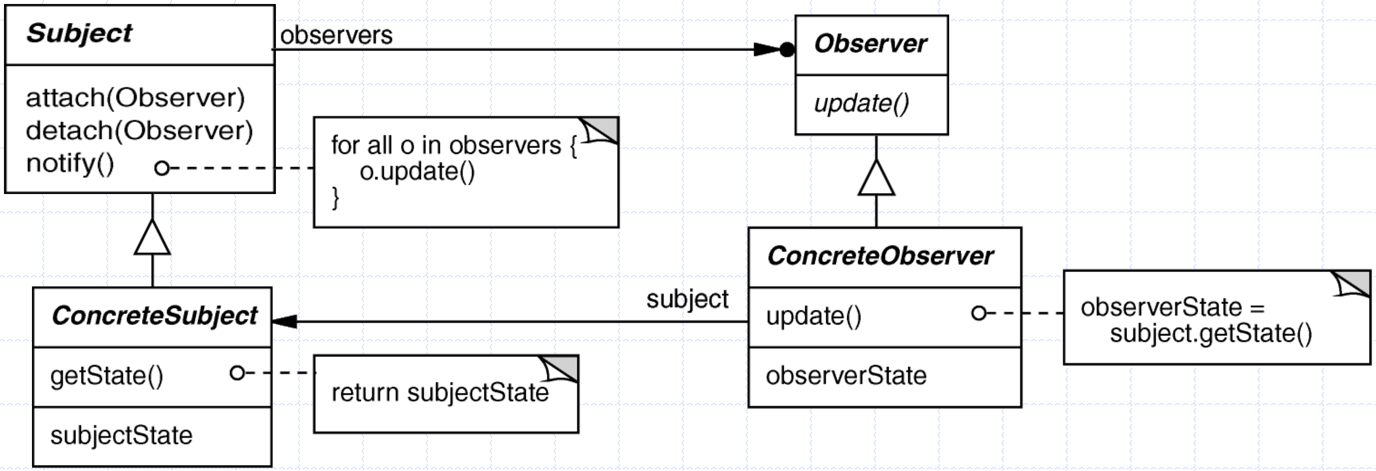
Intent

define a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated

Applicability

- an abstraction has two aspects, one dependent on the other
- a change to one object requires changing untold others
- an object should notify unknown other objects

Structure



OBSERVER (cont'd) object behavioral

Consequences

+ modularity: subject & observers may vary independently
+ extensibility: can define & add any number of observers
+ customizability: different observers offer different views of subject
– unexpected updates: observers don't know about each other
– update overhead: might need hints or filtering

Implementation

- subject-observer mapping
- dangling references
- update protocols: the push & pull models
- registering modifications of interest explicitly

Known Uses

- Smalltalk Model-View-Controller (MVC)
- InterViews (Subjects & Views, Observer/Observable)
- Andrew (Data Objects & Views)
- Pub/sub middleware (e.g., CORBA Notification Service, Java Messaging Service)
- Mailing lists

Iterator Design Pattern

ITERATOR object behavioral

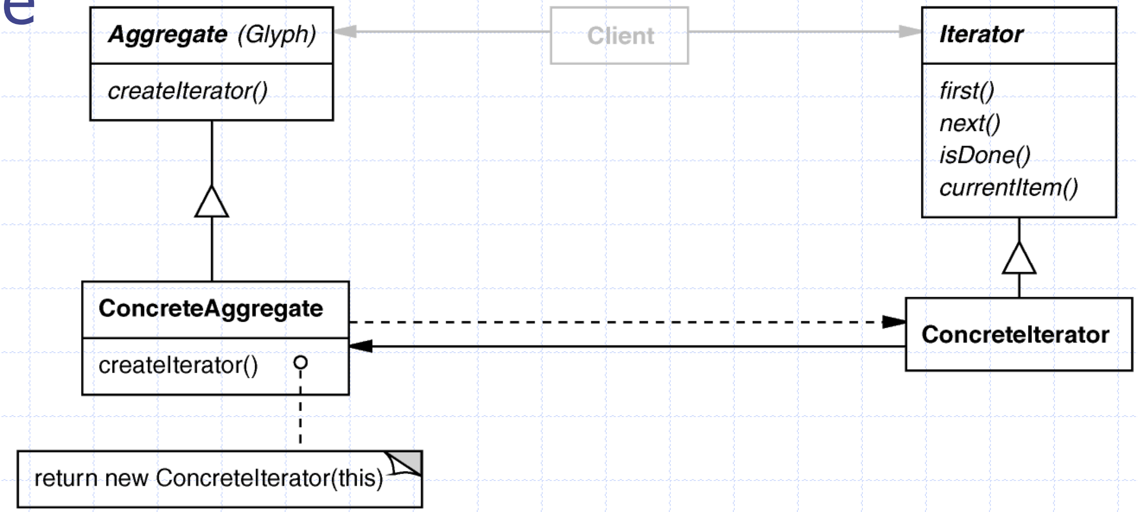
Intent

access elements of a container without exposing its representation

Applicability

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

Structure



ITERATOR (cont'd) object behavioral

Iterators are used heavily in the C++ Standard Template Library (STL)

```
int main (int argc, char *argv[]) {
  vector<string> args;
  for (int i = 0; i < argc; i++)
          args.push_back (string (argv[i]));
  for (vector<string>::iterator i (args.begin ());
     i != args.end ();
     i++)
   cout << *i;
  cout << endl;
  return 0;
}
```

The same iterator pattern can be applied to any STL container!

```
for (Glyph::iterator  i = glyphs.begin ();
   i != glyphs.end ();
   i++)
  ...
```

ITERATOR (cont'd) object behavioral

Consequences

+ flexibility: aggregate & traversal are independent
+ multiple iterators & multiple traversal algorithms
– additional communication overhead between iterator and aggregate

Implementation

- internal versus external iterators
- violating the object structure's encapsulation
- robust iterators
- synchronization overhead in multi-threaded programs
- batching in distributed & concurrent programs

Known Uses

- C++ STL iterators
- JDK Enumeration, Iterator
- Unidraw Iterator

Design Patterns are NOT

- Data structures that can be encoded in classes and reused *as is* (i.e., linked lists, hash tables)
- Complex domain-specific designs (for an entire application or subsystem)
- If they are not familiar data structures or complex domain-specific subsystems, *what are they*?
- They are:
  - "Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary

Liabilities of Patterns

- Require significant tedious and error-prone human effort to handcraft pattern implementations *design* reuse
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leaves some important details unresolved

Iterating Through A `Vector`

- Suppose we want to “step through” a **`Vector`** object, retrieving each of the objects stored in the vector one at a time:

Iterating Through A `Vector`

```
import java.util.*;
public class IterationExample {
    public static void main(String args[]) {
        Vector v = new Vector();
        // Put some Complex number
        // objects in the vector.
        v.addElement(new Complex(3.2, 1.7));

        v.addElement(new Complex(7.6));
        v.addElement(new Complex());
        v.addElement(new Complex(-4.9, 8.3));

        Complex c;
        for (int i = 0; i < v.size(); i++) {
            c = (Complex)v.elementAt(i);
            System.out.println(c);
        }
    }
}
```

Iterating Through A **`Vector`**

- This approach works well for vectors because the elements of a vector can be retrieved by specifying their position:

```
v.elementAt(i);
```

- We start by retrieving element 0, then retrieve element 1, etc., until we retrieve the last element (at position

```
v.size()- 1)
```

What About Other Collection Classes?

- This approach doesn't work well for all collection classes:
  - what would a position (i.e. index) mean for a **`Set`**?
  - values stored in a **`Hashtable`** object are retrieved by keys, not by position
- Is it possible to come up with a more general approach, that can be used to iterate over any collection of objects?
- There must be a general design solution to this…

Collection Classes & Iterators

- The `Collection` interface provides an `iterator()`method to be implemented by all concrete collections:

  `Iterator iterator()`

  - `Collection` plays the role of the "Aggregate" in the GoF `Iterator` pattern
- A "ConcreteAggregate" (i.e. `Vector`, `LinkedList`…) implementing this method should create and return a iterator object

`Iterator` Objects

- An `Iterator` object provides two main methods:
  - `public boolean hasNext()`
    - returns true if the `Iterator` object has more elements (objects) that have not yet been returned by `next()`
  - `public Object next()`
    - returns the next element (object) from the iteration

Iterating Through a `Vector`

```
// Create a vector object and initialize
// it with Complex objects

Complex c;
Iterator i;
i = v.iterator();
while (i.hasNext())
 {
    c = (Complex)i.next();
    System.out.println(c);
 }
```

Iterating Any `Collection`

For instance, the code below shows how `AbstractCollection` implements `addAll()` without making any assumption on the nature of the collection that is passed as an argument:

```
public boolean addAll(Collection from)
{   Iterator iter = from.iterator();
    boolean modified = false;
    while (iter.hasNext())
       if (add(iter.next()))
         modified = true;
    return modified;
}
```

Building an Iterator

- Suppose we have a class called `CircularList` that provides these methods (other `CircularList` methods not shown here):

```
// Constructor.
public CircularList();
// Return the count of the number of
// elements in the list.
public int count();
// Return the element stored at the specified
// position in the CircularList.
public Object get(int index);
// Remove the element stored at the specified
// position in the CircularList.
public Object remove(int index);
```

Building an Iterator

- How do we build an iterator (i.e., an `Iterator` object) for this class?
- We want an object that lets us do this:

```
CircularList l = new CircularList();
...
//store integers in the list
...
Iterator i = l.iterator();
while (i.hasNext()) {
  System.out.println(i.next());
}
```

The `Iterator` Interface

- When we look through the documentation for the Java API, we discover that `Iterator` is an interface, not a class:

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
    public void remove();
}
```

- An `Iterator` object is an instance of a class that implements the `Iterator` interface; i.e., defines the `hasNext()`, `next()`, and `remove()` methods