OO Design Patterns

Ref: *Design Patterns, Elements of Reusable Object-Oriented Software*
by Gamma, Helm, Johnson, & Vlissides
[aka The Gang of Four]
Addison-Welsley, 1995.

Basic idea: Reusable metaphors for designing OO software systems.
[But not all patterns are truly OO.]

I highly recommend you buy and read this book.
[Easy to read a few pages at a time.]

Why Design Patterns?

- A mature engineering discipline *codifies* its experience:

  - When confronted with endless possibilities, rely on those who have gone before you:

    - We software professionals have done a poor job of writing down what we’ve learned.

- When this book came out, it caused a lot of excitement:
  - Resonance with professional software developers.
  - “Oh yeah, I’ve seen that before. We call it XXX in our group.”
  - Put into writing a lot of shared knowledge.
  - Some people complained: “What’s the big deal? Everyone knows that!”

- Book describes 23 patterns:
  - name and description
  - abstract representation
  - examples (high-level and code-level)
  - applicability and usefulness
  - trade offs, caveats, experiences,
  - relative (dis)advantages
  - relationship to other patterns
  - but this list isn’t meant to be the last word

- There are thousands of patterns “out there” and thousands more waiting to be discovered:
  - Some are “domain specific”.
  - Some are a lot like others, special cases, etc.

- There is no official person or group who decides what is/isn’t a design pattern:
  - many DP web pages
  - newsgroups discussions
  - more books
  - academic conferences
  - special term: “patterns geek”
What Use Are Design Patterns?

Think of as a low-level software architecture
  or a high-level programming abstraction.
First, you learn the basics (data structures,
  algorithms, tool and lang details).
Then, you learn modules/interfaces/information
  hiding, classes/OO programming.
Design patterns are at the next level of abstraction:
  read, read, read, think, think, think, apply!
[How do chess players become “world class”?]

Design patterns help you:
  • design new systems using higher-level
    abstractions than variables, procedures, and
    classes.
  • understand relative tradeoffs, appropriateness,
    (dis)advantages of patterns
  • understand essence of what you’re constructing

communicate about systems with other
  developers
  • give guidance in resolving non-functional
    requirements and trade-offs:
      – portability
      – extensibility
      – maintainability, re-usability, scalability
  • avoid known traps, pitfalls, and temptations
  • ease restructuring, refactoring
  • coherent, directed system evolution and
    maintenance

Graphical Notation
  [similar to Rumbaugh’s OMT]

```
<table>
<thead>
<tr>
<th>className</th>
<th>methodSig();</th>
<th>dataSig:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>class B inherits from class A or class B implements abstract class A</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>class B instantiates from class A</td>
<td></td>
</tr>
</tbody>
</table>
```

The Adapter Pattern

**Intent:**
→ Convert the interface of a class into another
  interface clients expect.
→ Adapter lets classes work together that
  couldn’t otherwise because of incompatible
  interfaces.

**Motivation:**
→ When we want to reuse classes in an
  application that expects classes with a
  different and incompatible interface, we do
  not want to (and often cannot) change the
  reusable classes to suit our application.
Example of the Adapter Pattern

Structure of the Adapter Pattern

Using Multiple Inheritance

Using Object Composition

Participants of the Adapter Pattern

- **Target**: Defines the application-specific interface that clients use.
- **Client**: Collaborates with objects conforming to the target interface.
- **Adaptee**: Defines an existing interface that needs adapting.
- **Adapter**: Adapts the interface of the adaptee to the target interface.

Class- vs. Object-level Adapter

- Class-level (might) require real MI
  ⇒ all trad. MI problems
- Is Is-a the right relationship?
- Object-level makes for messy interface/wrapping (but can be selective):
  → extra syntax, wrapping
  → how much adapting will you do?
  → lose polymorphism
The Bridge Pattern

**Intent:**

→ Decouple an abstraction from its implementation so the two can vary independently.

**Motivation:**

→ Usually, when an abstraction has several possible implementations, use an abstract base class with several concrete derived classes:

```
AbsClass
  B
  C
  D
```

But sometimes, we want a hierarchy for A on its own, independent of these concrete classes:

```
Window
  XWindow
  IconMSWindow
  IconXWindow
  TransientWindow
  IconWindow
  TransientWindow

→ Could add flag to all Windows
   Window::WindowKind enum {X, Win, MacOS};
   and add switch statement to all routines, ...
   but this defeats OO programming and polymorphism!
   • Lots of code duplication
     [cloning, not “good” reuse]
   • less maintainable

The basic problem is that we have two dimensions:

1. Abstract windows
   *i.e., Window, IconWindow, TransientWindow*
2. Window implementations
   *i.e., X, MS, MacOS*

**Bridge solution:** use two inheritance hierarchies and add a level of indirection.

```
Bridge
  Window
    ShowText
    ShowRect
  WindowImpl
    XWindow
    IconWindow
    MSWindow
    TransientWindow

ConcreteImplementor
  Impl
    ImplImpl
    ImplImpl
    ImplImpl

ConcreteImplementorB
  Impl
    ImplImpl
    ImplImpl
```

**Structure of the Bridge Pattern**

```
Bridge
  Abstraction
    Impl
      OperationImpl
  Implementor
    OperationImpl
      ImplImpl
        ImplImpl
        ImplImpl
```

CS211 — ooDesignPatterns
Participants

Abstraction (Window)
- defines interface of abstraction
- maintains a reference to an object of Implementor

RefinedAbstraction (Icon Window)
- extends interface of Abstraction

Implementor (WindowImpl)
- defines interface for implementation classes
- often quite different from Abstraction
- usually defines primitives which Abstraction combines usefully

ConcreteImplementor (XWindowImpl)
- implements primitives defined in Implementor

The Composite Pattern

Intent:

→ Compose objects into tree structures to represent part-whole hierarchies.

→ Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation:

→ If pattern not used, clients must treat primitives and containers differently.

→ Usually, clients shouldn’t need to know if object is primitive or composite.
Notes on the Composite Pattern

- Gamma et al. think that the management routines should be declared at the top level, even if leaves (e.g., Rectangle) have no use for them.
- Their motivation is “transparency”: only one interface which everyone implements. This is WRONG OO-wise.
- If you use a second interface (i.e., Container), can still use the Graphic interface most of the time.
- Why did they do this? Confusion of C++ RTTI with proper dynamic type information. This is completely safe in Java.

Participants of Composite Pattern

Component (Graphic)
- Declares the interface for objects in the composition.
- Implements default behavior for the interface common to all classes.
- BUT should NOT declare an interface for accessing and managing its child components (Gamma et al. says it should).

Leaf (Rectangle)
- Represents leaf objects in the composition. A leaf has no children.
- Defines behavior for primitive objects in the composition.

ComponentContainer (GraphicContainer)
- Defines interface for components having children.
- Stores child components.
- Implements some child-related operations in the component interface; defers others to its derived classes.

ConcreteCompContainer (Picture)
- Defines remaining behavior (for components having children) not defined by ComponentContainer.

Client
- Manipulates objects in the composition through the component interface.

The Facade Pattern

Intent:
→ Provide a single, unified interface to a set of interface/modules in a subsystem.
→ Abstracts a whole subsystem into one interface.

Motivation:
→ Reduce complexity of a system.
→ Increase cohesion within a subsystem and reduce coupling with other parts of the system.
→ Fewer, simpler, interactions within system.
→ Easier to use, reuse.
→ Easier to replace components.
Example of the Facade Pattern

<table>
<thead>
<tr>
<th>Compiler</th>
<th>CodeGenerator</th>
<th>RISCCG</th>
<th>StackMachineCG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compiler Subsystem Classes

<table>
<thead>
<tr>
<th>Scanner</th>
<th>Token</th>
<th>ProgNodeBuilder</th>
<th>ProgNode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another Example of the Facade Pattern

<table>
<thead>
<tr>
<th>FILE_SYS</th>
<th>USER_SYS</th>
<th>GLOBALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>User</td>
<td>System</td>
</tr>
<tr>
<td>Directory</td>
<td>Family</td>
<td>Panic</td>
</tr>
</tbody>
</table>

Structure of the Facade Pattern

Client Classes

Subsystem Classes

Participants of the Facade Pattern

Facade (Compiler)
- Knows which subsystem classes are responsible for a request.
- Delegates client requests to appropriate subsystem objects.
- Subsystem still does the work; facade just directs traffic.

Subsystem Classes (Other SS components)
- Implement subsystem functionality.
- Handle work assigned by the facade object.
- Have no knowledge of the facade; that is, they keep no references to it.
The Mediator Pattern

**Intent:**

→ Define an object that encapsulates how a set of objects interact, rather than having the objects refer to each other explicitly.

**Motivation:**

→ Simpler model; components send messages to the “manager” who decides how to reroute them.

→ Easier to evolve components, less “special knowledge” scattered around subsystem

→ Allows for different kinds of dependencies/communication between components.

---

An Object-Level Example

---

A Class-Level Example

---

Structure: Object-Level
The Mediator Pattern

Participants of the Mediator Pattern

**Mediator** (DialogDirector)
- defines an interface for communicating with colleague observers

**ConcreteMediator** (FontDialogDirector)
- implements co-operative behaviour by co-ordinating Colleague objects
- knows and maintains its list of colleagues

**Colleague classes** (Listbox)
- each colleague knows who its mediator object is
- each colleague communicates with its mediator when it wants to send messages

The Observer Pattern

**Intent:**

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

**Motivation:**

→ A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.

→ You don’t want to achieve consistency by making the classes tightly coupled, because that reduces their re-usability.
Example of the Observer Pattern

Structure of the Observer Pattern

- The key objects in this pattern are subject and observer.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state.

Participants of the Observer Pattern

Subject
- Knows its numerous observers.
- Provides an interface for attaching and detaching observer objects.
- Sends a notification to its observers when its state changes.

Observer
- Defines an updating interface for concrete observers.

ConcreteSubject
- Stores state of interest to concrete observers.

ConcreteObserver
- Maintains a reference to a concrete subject object.
- Stores state that should stay consistent with the subject’s.
- Implements the updating interface.

When to use the observer pattern:
- when a change to one object requires changes to others, but you don’t know (or ought to know) which ones those are
- when an the objects should not be tightly coupled

Notes:
- after being informed of a change, Observer may query subject for more information
- lets you vary subject and observer independently
- decouples structure between objects
- can use abstract classes for Subject and Observer
  → more flexible, easier variation and composition...
There are two basic models of updating:

1. **push** — all observers notified whether they want the information or not

2. **pull** — subjects are sent little or no information ("There has been a change; contact me for more details.")

Might not want to "push" all of the information around to all observers:

- Not all objects need or ought to know everything.
- Message passing takes time, space, overhead.
- Some changes are undone or superfluous
- Subjects may register for only some kinds of information
  → Have to specialize kinds of messages, more detail to manage.

---

### Flyweight Pattern

**Intent:**

→ Use sharing to support large numbers of finely-grained objects efficiently.

**Motivation:**

→ Some applications use lots of finely-grained *(i.e., little)* objects.

→ BUT objects entail overhead (wrapping).

→ If LOTS of such objects are required AND if such objects are really *values* at heart, then make a single pool of these objects for everyone to share.

---

### Examples of the Flyweight Pattern

- Java scalar wrapper classes *(e.g., Integer)* + GC
- Java Strings + GC
- Word processing system would like to treat all entities as objects for sake of system uniformity, even *chars*

**Solution:** Treat *chars* as object-values

→ Global pool of constant objects; likely stored in a table of some kind for look up, but clients can link to these objects directly.

→ only one object instantiated to represent, say, letter “m”. All uses of “m” are links to the global object representing this letter.

→ each letter-object represents a constant *value* rather than an “interesting” independent object with mutable state.
Notes:

- letter-objects cannot (easily) have back references to the “containing” object

- When it comes time to draw the character, will have to pass in appropriate context that would normally be part of the object state
  
  e.g., \((x, y)\) screen locations, font information, text colour

- Could have separate object pool for each font family, pass in locations, size, slant, text colour to drawChar

  → Allows for flexible font family hierarchies rather than monolithic character class

- Call flyweight because each object stores minimal information about its state

Structure of the Flyweight Pattern

**Flyweight** (Glyph)

- declares an interface through which flyweights can receive and act on extrinsic state

**Concrete Flyweight** (Character)

- implements the Flyweight interface and adds storage for intrinsic state (if any).
  - An instance of this class must be *sharable*
  - Any state it stores must be *intrinsic*
Flyweight Factory

- creates and manages Flyweight objects as needed
- ensures sharing is done properly, provides (and sometimes creates) instances as requested
- might create all flyweight objects at start or just as needed

Client

- maintains references to flyweights
- computes/stores extrinsic state of flyweight objects

Applicability of Flyweight Pattern

All of the following should be true:

- Applications uses LARGE number of objects
- most object state can be made extrinsic practically
- proliferation of objects will be greatly reduced by using a shared pool
- (non-extrinsic) object identity doesn’t matter

Notes on Flyweights

- creation may be done once at initialization or as needed on the “fly”
- need to manage context (extrinsic state) separately from the objects
  → This is unnatural and awkward!
  → Watch out for errors, funny assumptions, odd “glue”
- commonly done to manage system resources
  e.g., system fonts, colour maps

Consider how Java handles strings:

- **String** instances are immutable once created
  [Use **StringBuffer** for mutable strings]

```java
String s1, s2;
s1 = "hello";
s2 = "hello";
if (s1 == s2) {  // Probably wrong
    ...
```

[You probably meant to say `if (s1.equals(s2))`]

- Also, all Java scalars have corresponding “wrapper” classes (e.g., **Integer**.)
The Iterator Pattern

Intent:

→ Provide a clean, abstract way to access all of the elements in an aggregate (container) without exposing the underlying representation.

→ Also, moves responsibility for access and traversal to a separate “iterator” object.

Motivation:

→ Often want to say to a container (e.g., tree, graph, table, list, graphic), “Apply f( ) to each of your objects.”

→ Don’t want to expose implementation details of the container AND want to allow multiple simultaneous traversals

⇒ Create separate interface/class that provides simple “hooks”.

How it’s done in various programming languages:

- Lisp-y functional languages provide mapcar-like function to do this. Pass in a function as an argument.

- C/C++ allow function-as-arguments, but C++ also provides an iterator class in the STL

- Java does not allow function-as-arguments, but does an iterator construct called an Enumeration.

Java Enumerations

java.util.Enumeration is an interface that may be implemented by any class.

```java
public interface Enumeration
{
    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
}
```

java.util.Vector (array with resizable bounds) defines a method called elements that returns an enumeration of the vector.

```java
public class Vector
{
    public final synchronized Enumeration elements();
}
```

Example in Java

```java
class CardPlayer {
    ...

    // CardPile extends java.util.Vector
    CardPile tricks;

    // Count up how many points I've earned in my tricks this round and add it to my running total.
    public void countAndAddPoints () {
        int numPointsThisRound = 0;
        Enumeration tricksEnum = tricks.elements();
        while (tricksEnum.hasMoreElements()) {
            Card c = (Card) tricksEnum.nextElement();
            numPointsThisRound += c.value();
        }
        points += numPointsThisRound;
    }
}
```
Structure of the Iterator Pattern

<table>
<thead>
<tr>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count()</td>
</tr>
<tr>
<td>Append(Element)</td>
</tr>
<tr>
<td>Remove(Element)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ListIterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>First()</td>
</tr>
<tr>
<td>Next()</td>
</tr>
<tr>
<td>IsDone()</td>
</tr>
<tr>
<td>CurrentItem()</td>
</tr>
<tr>
<td>index</td>
</tr>
</tbody>
</table>

Participants of the Iterator Pattern

Iterator
- Defines an interface for accessing and traversing elements.

ConcreteIterator
- Implements an iterator interface.
- Keeps track of the current position in the traversal of the aggregate.

Aggregate
- Defines an interface for creating an iterator object.

ConcreteAggregate
- Implements the iterator creation interface to return an instance of the proper concrete iterator.

Notes on the Iterator Pattern

- The iterator interface basically provides two hooks:
  - “Next, please.”
  - “All done?”
  
  Can provide a few others hooks too (how many left, go back, start over, etc.)

- Can define different iterator implementation that redefine what “next element” means.
  - walk through a tree, alphabetical order, LRU

- Can have multiple iterations ongoing simultaneously (enumerator state basically consists how far along it is in the “list”).

- Of course, if you change (or allow others to change) the aggregate (add/remove) or its contained objects, then chaos may result! May therefore want to
  - lock out changes during an iteration (concurrency control)
  - iterator may take fast snapshot of aggregate state and use that to iterate through
  - may want to buffer requested changes while an iterator is active, then commit

- Complicated structures may be difficult to iterate through
  - Store “path taken” or “nodes visited”
The Template Method Pattern

Intent:

→ Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

→ Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

Motivation:

→ By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering.

→ Provide the skeleton of a method, where parts are defined/overridden by derived classes.

→ Typically, do NOT override the template method itself.

Example (Cargill again)

```cpp
class Vehicle {
    private:
        char *plate;
    protected:
        virtual char *group () = 0;
    public:
        Vehicle () {plate=NULL;}
        Vehicle (char *p) {
            plate = new char [strlen(p)+1];
            strcpy (plate, p);
        }
        virtual ~Vehicle () {delete [] plate;}
        virtual void identify () {
            char *p = plate ? plate : "<none>";
            printf("%s with plate %s\n", group(), p);
        }
};
// Define similarly for Truck
class Car : public Vehicle {
    private:
        char *group () {return "Car";};
    public:
        Car () : Vehicle () {}
        Car (char *p) : Vehicle (p) {}
};
```

Structure of the Template Method Pattern

Participants of the Template Pattern

AbstractClass (Vehicle)

- Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- Implements a template method defining the skeleton of an algorithm.
- The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

ConcreteClass (Car, Truck)

- Implements the primitive operations to carry out subclass-specific steps to the algorithm.
Notes on the Template Method Pattern

- Very common to have in abstract base classes, esp. libraries intended for reuse and customization.
- A kind of polymorphism (think about it) practised within a class hierarchy.
  - “Whatever object I turn out to be, use my definition on primitiveOp(), then print, ...”
- This is a common pattern to recognize during secondary passes at design.
  - Refactor commonalities into the parent class; push as much as possible as high as possible.

The Memento Pattern

**Intent:**
- Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.

**Motivation:**
- Often want to be able to take “snapshots” of an object’s current state in case:
  - want to unroll changes
  - debugging
  - store sequence of abstract transactions by class that is not privy to underlying representation of transaction.
- BUT don’t want to break encapsulation/information hiding
  ⇒ state type must be opaque

The Memento Pattern

**Intent:**
- Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.

**Motivation:**
- Often want to be able to take “snapshots” of an object’s current state in case:
  - want to unroll changes
  - debugging
  - store sequence of abstract transactions by class that is not privy to underlying representation of transaction.
- BUT don’t want to break encapsulation/information hiding
  ⇒ state type must be opaque

Participants of the Memento Pattern

**Memento**
- Stores internal state of the Originator object. The memento may store as much or as little of the originator’s internal state as necessary at its originator’s discretion.
- Protects against unwanted access by objects other than the Originator.
- Effectively has two interfaces: narrow and wide

**Originator**
- Creates a memento containing a snapshot of its current internal state.
- Uses the memento to restore its internal state.
Notes on the Memento Pattern

- Simplifies Originator as others may now maintain collections of state, albeit without being able to “look under the hood”
- Don’t forget TNSTAAFL
  → Temptation to store all states for “unlimited undo” BUT don’t forget that everything takes storage.
  [Did you keep every mail message you ever received?]

Design Patterns — Summary

- Intended to provide a common:
  - vocabulary for inter-developer communication.
  - set of abstract building blocks for system designers
  - set of patterns to look for when refactoring / redesigning / reverse engineering existing systems.
- Not exclusively OO, despite the book title; really based on encapsulation and interacting objects.
- Seems particularly useful for concurrent/distributed systems where logic is hard to get right and possibilities are too numerous.
- Early days yet, but many are very excited.

Caretaker

- is responsible for the memento’s safekeeping.
- never directly operates on or examines the contents of a memento.