Design Patterns

- Why is an experienced designer more productive than a novice?
- From experience, a designer builds up a repertoire of general solutions to problems that occur repeatedly.
- If experienced designers write down their solution structures then they can share them with others.
- If we agree on names for these solutions then we can use them to communicate ideas with other designers.

In the OOD context, a *pattern* is a named problem/solution pair that can be applied in new contexts.
Ideas originate in architectural patterns for buildings.

Pattern Notes

- Patterns are reusable **solutions**, not code.
- **Not** about designs that can be encoded in a class and used as is (e.g., linked list, hash table).
- **Not** complex, domain-specific designs for an entire application or subsystem.
- No catalog of patterns is complete — there is always a possibility to define new patterns.

Pattern Descriptions

Gamma *et al* [1] (a.k.a., "the gang of four" or "GoF") have identified a collection of common patterns and a template for describing them.

Essential components of pattern description:

Name — One or two words that identify the problem & solution.

Problem — When to apply the pattern.

Solution

- Elements that make up the design,
- their relationships,
- responsibilities, and
- collaborations.

Consequences — Results and trade-offs of applying the pattern.

Pattern Classification

Two criterion:

1. Purpose

   Creational — concerned with process of creating objects.
   Structural — about the composition of classes or objects.
   Behavioural — the way in which classes or objects interact and distribute responsibility.

2. Scope

   Class — relationships between classes and their sub-classes (static, compile-time relationships).
   Object — relationships between objects (dynamic, run-time).

GoF Patterns

	Creational	Structural	Behavioural
Class	Factory Method	Adapter (class)	Interpreter Template method
Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Pattern Description Template

1. Name & Classification
2. Intent — what does the pattern do?
3. Also known as
4. Motivation — scenario that illustrates the problem and how it is solved by the pattern.
5. Applicability — situations in which the pattern can be applied.
6. Structure — class and/or interaction diagrams.
7. Participants — classes/objects and their responsibilities.
8. Collaborations
9. Consequences
10. Implementation — pitfalls, hints, or techniques relevant to the solution implementation.
11. Sample code.
12. Known uses — from real systems.
13. Related Patterns.

Model-View-Controller

Intent Separate issues of graphical presentation, user input interpretation, and application data/state.

Motivation Previously the GUI code was mixed with the application, which made it difficult to change either. GUI code can be quite complcated and is full of issues that are quite independent of the application.

Applicability *The* standard pattern for systems with a GUI.

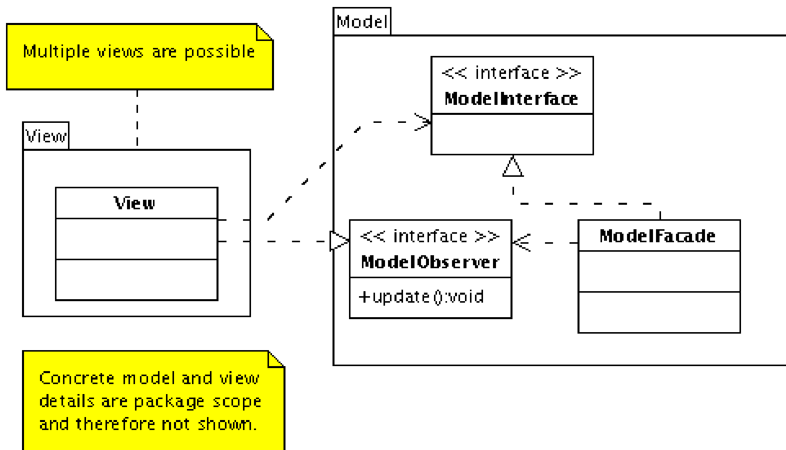
Model-View-Controller (cont'd)

Participants

1. Model — represents the application data in a manner independent of its presentation to the user. Normally doesn't know about the view or controller.
2. View — presents the data to the user. Knows how to get the information it needs from the Model.
3. Controller — interprets user input and passes changes etc. to Model.

- Not a GoF pattern, but has been around longer.
- Uses many GoF patterns: Observer, Composite, Strategy, (Factory Method, Decorator).
- Often view is combined with controller — the Model-View pattern.

Model-View Structure



Information Transfer in M-V

When the model changes state the views must update. Three choices

1. Object that updated the model tells the view(s) to refresh. Disadvantage: Every object that might change the model must know about all the views.
2. Model tells the view(s) that it has changed. Disadvantage: Model must know every view.
3. As above, but Model calls through a listener interface.

Information Transfer in M-V (cont'd)

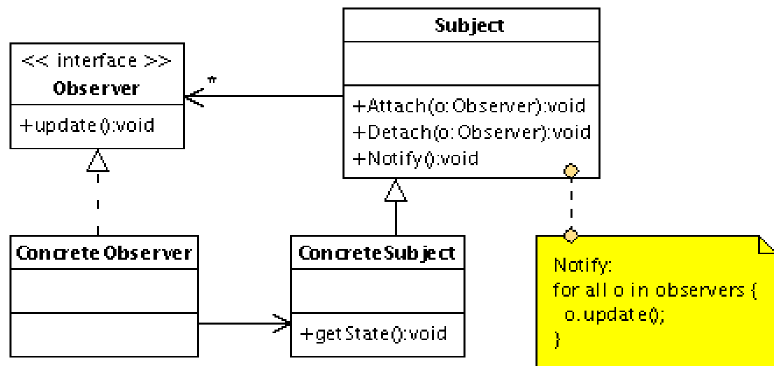
Once the view knows about a change how does it get the data to update its presentation?

1. Push: The message that informs the view of the change also contains the updated information.
   Disadvantage: The information may not be relevant to this view.
2. Pull: The view asks the model for what it needs.

M-V Interaction

- Often the view objects update the model.
- The update notification may arrive at the view in the middle of another call to the model.
- Be careful that the model object is in a consistent state before updating.

Observer Pattern



Observer Pattern (cont'd)

Applicable whenever multiple objects must be kept consistent.

Subject Knows observers.

- Any number of observers may observe a subject.
- Provides and interface for attaching and detaching observers.

Observer (interface) defines updating interface for objects that should be notified of changes.

ConcreteSubject Stores state (model). Notifies observers of changes.

ConcreteObserver

- Maintains reference to ConcreteSubject.
- Implements Obsever update interface to keep its state consistent with subject.

Observer Consequences (+)

- ConcreteSubject may be reused without reusing observers.
- Observer classes may be added or removed without modifying ConcreteSubject or other observer classes.
- Observers may belong to higher level in a layered system.
- Supports broadcast to many observers

Observer Consequences (-)

- Cost of update is hidden from subject.
- No indication of how subject has changed — may lead to costly unneeded updates.
- Subject must be consistent when it calls notify.
- Too many notifications — every change causes notify.

Observer Implementation Issues and Variations

Dangling references Deleting either a subject or observer may leave dangling references — need to ensure that referring objects are informed of delete (de-register self before delete).

Update triggering Whose responsibility is it to call update?

- state-setting methods in subject — may lead to too many updates.
- clients — error prone.

Observing more than one subject — update needs a parameter to identify itself to the observer.

Observer Implementation Issues and Variations (con'd)

Update protocols How is the information about he change communicated to the observer?

- Push model — subject sends observers detailed information about the change. Subject needs to know more about observers.
- Pull model — subject sends minimal information, observer goes and gets it. May be less efficient.

Specifying changes of interest explicitly Observers register as being interested in specific kinds of changes.

Change manager Encapsulates particularly complex update semantics (example of *Mediator* pattern).

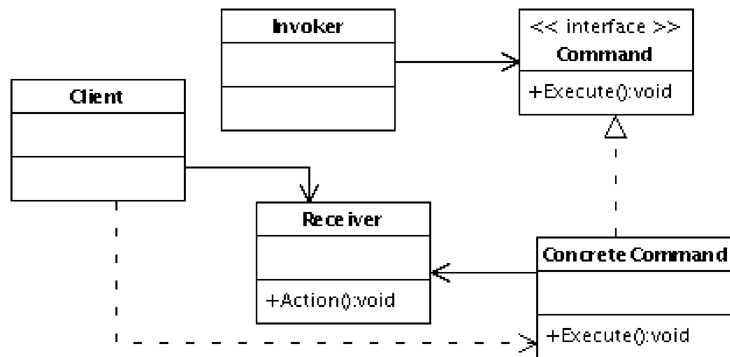
Command Pattern (Behavioural)

Intent Encapsulate a request as an object, enabling parameterized clients, queuing, logging and undoing.

A.K.A. Action, Transaction

Motivation Need to issue requests to objects without knowing about the operation being requested or the receiver of the request (e.g., buttons etc. on GUIs).

Command Pattern: Structure



Command Pattern: Participants

- **Command** — interface for executing an operation (e.g., java.awt.event.ActionListener)
- **ConcreteCommand** — implements Command interface. Defines binding between receiver and action.
- **Client** — creates ConcreteCommand object (and sets its receiver and invoker).

```
ActionListener loadAction = new LoadAction();
loadMenuItem.addActionListener(loadAction);
loadToolBarButton.addActionListener(loadAction);
```

- **Invoker** — asks the command to carry out the request (e.g., MenuItem, ToolBarButton).
- **Receiver** — knows how to perform the operations associated with carrying out the request.

Command Pattern: Applicability

- Parameterize objects by an action to perform (e.g., MenuItem).
- Specify, queue and execute requests at different times.
- Support undo — commands store state for reversing effects.
- Transaction processing.

Composite Pattern (Structural)

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

Motivation GUI frameworks build complex GUIs out of components which contain other components . . . Key is abstract class (Component) that represents both primitives and their containers.

Participants

- **Component** — declares interface, implements default behaviour.
- **Leaf** — is not a container
- **Composite** (container) — stores child components, implements child-related operations.
- **Client** — manipulates objects in the composition through the Component interface.

Composite Pattern: Consequences

- Clients can deal with only the root — existence of children can be hidden.
- Extension is easier. Newly defined subclasses work automatically.
- Can make design overly general. Can't restrict (at compile time) contents of containers to certain types of components.

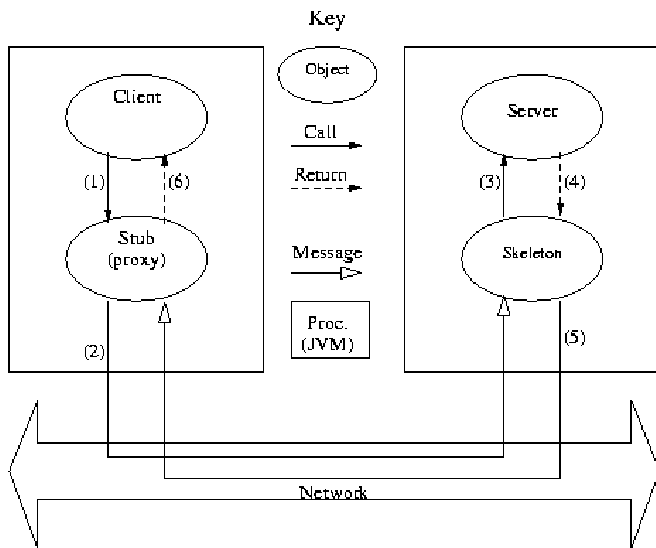
Proxy Pattern (Structural)

Intent Provide a surrogate or placeholder for another object to control access to it.

Applicability Whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

- Remote proxy — local representative of remote object.
- Virtual proxy — delay creation of actual (expensive) object until needed.
- Protection proxy — control access to object.
- Smart reference — a replacement for pointer that performs additional actions (e.g., counting references, locking, loading etc.)

Proxy Pattern Example: RMI in Java



Factory Method Pattern (Creational)

Intent Define an interface for creating an object, but let subclasses decide which class to instantiate.

Motivation

- We want to design a text editor framework that can edit a variety of document types (Rich Text Format, HTML, plain old text, etc.)
- In swing.text the objects that make up a text editor know an EditorKit object.
- Implementing the new menu item, we ask the EditorKit object to create a new document.

Old Fashioned Solution

- One way to do this involves a single EditorKit class containing:

```
EditorKit( String textType ) {
  this.textType = textType ;
}

Document makeDocument() {
  if (textType.equals(rtf) {
    return new RTFDocument() ;
  } else if (textType.equals(html) {
    return new HTMLDocument() ;
  } ...
}
```

- Problem: To add new document kind, we must edit this class. Thus it is not reusable.

Factory Method Solution

- EditorKit is an abstract class with method

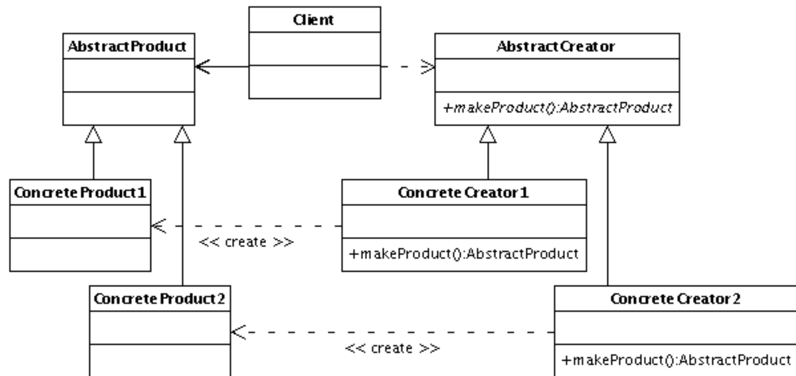
```
// Factory Method
abstract Document makeDocument() ;
```

- EditorKit has a number of subclasses

```
class RTFDocument {
    ...
    // Factory Method
    Document makeDocument() { new RTFDocument(); }
    ... }
```

- Adding a new document type means creating a new subclass of Document and a new subclass of EditorKit. It does not require editing EditorKit, Document, or the code of the text editor.
- EditorKits are factories for Documents.

Factory Method Structure



Example 2: Factory Methods in Collection Classes

- Package java.util defines a number of
  - interfaces (Collection, List, Set) and
  - classes (ArrayList, LinkedList, HashSet, TreeSet)
- It also defines a number of Iterator classes for iterating over these various classes. E.g.

```
LinkedList list = new LinkedList();
Iterator it = list.iterator();
while(it.hasNext()) {
  Object item = it.getNext();
  ... // do something with item
}
```

Collection Classes (cont'd)

- We can also write generic code. E.g.

```
int sum( Collection col ) {
  Iterator it = col.iterator();
  int s = 0;
  while(it.hasNext()) {
    Integer item = (Integer) (it.getNext());
    s += item.getValue();
  }
  return s;
}
```

Aside: Java 1.5+ method:

```
int sum(Collection col) {
  int s = 0;
  for (Integer item : col) {
    s += item.getValue();
  }
  return s;
}
```

Collection Classes (cont'd)

- Each class that implements Collection must implement iterator() to return an iterator object of the appropriate class.
- Collection classes are factories for iterators.
- We can add new subclasses of Collection and reuse generic code.

Factory Pattern Consequences

(+) Client code can create objects of any of a variety of classes without depending on any of those classes. Hence it is generic and reusable.

(+) Connects parallel class hierarchies (e.g. the Container hierarchy and the Iterator hierarchy).

(+) Whether or not an object is created can be hidden from client. E.g. creator could return a previously created object. (Consider immutable objects)

Builder Pattern (Creational)

Intent Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation By example:

- XML is a common file format for structured documents. However different applications require different internal representations of XML documents.
  - E.g. Xylia is a generic editor for XML documents built on top of the swing.text package. Thus its internal representation extends swing.text.AbstractDocument.
  - But an editor for the Chemical Markup Language (a specific XML language) might require a much different internal representation.
- We would like to use a single parser to read XML files for both these applications.

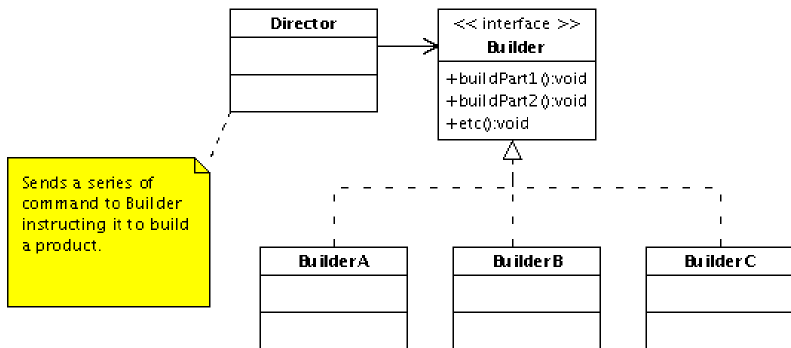
Example Solution

- Normally a parser converts a sequence of characters into an object (while checking for syntax errors).
- Instead we write a parser that converts a sequence of characters into a sequence of calls to a pluggable object.
- `<xhtml><body><p>hi</p></body></xhtml>`
- Is converted to calls

```
obj.startTag( xhtml );
obj.startTag( body );
obj.startTag( p );
obj.content( hi );
obj.endTag() ;
...
```

- Here "obj" is a pluggable object.

Builder Structure



Builder Example 2: SAX

- SAX (Simple API for XML) uses the Builder strategy to specify a common standard for XML parsers
  - A number of different parsers all use SAX.
  - A large number of XML based applications use SAX.
- SAX allows multiple handler (builder) objects.
- Allows different handlers to pay attention to different commands.
- Content models in XML describe constraints on the sequence of children an element can have. E.g. `(p | table | img)*` describes the (simplified) content model for an xhtml body.
- This is converted to a sequence of calls to a pluggable object via an interface.

Builder Example 2 (cont'd)

The interface is optimized for building trees:

```
Object finishContentModel( int kind );
Object finishContentModel( int kind, Object regExp );
Object mkPCDATA( );
Object mkName( String name );
Object mkAlternation( Object right, Object left );
Object mkSequence( Object right, Object left );
Object mkOptional( Object operand );
Object mkKleeneStar( Object operand );
Object mkKleenePlus( Object operand );
```

Patterns Survey: Abstract Factory (Creational)

Use factory methods to create a family of related objects. (E.g. In java.awt a factory object creates a set of widgets that work together (scrollbars, windows, menus)).

Patterns Survey: Singleton (Creational)

Ensure that only one instance of a class is created

```
class Quangle {
  private static Quangle singleton = null;

  // Constructor is private
  private Quangle() { ... };

  public static Quangle getInstance() {
    if( singleton==null )
      singleton = new Quangle();
    return singleton;
  }
  ...
}
```

Use with caution. Forcing client code to call static methods prevents reuse with subclass.

Survey of some other patterns (cont'd)

Adapter (Structural) (a.k.a., Wrapper) Use a class to adapt another class to conform to an expected interface.

- Whereas the Proxy adds functionality while keeping the same interface, the Adapter changes the interface while leaving the same functionality.

Façade (Structural) Provide a single interface to a set of objects.

Iterator (Behavioural) Give sequential access to the items of a collection without exposing its representation.

- Allows generic algorithms to operate on a variety of collections (sets, lists, maps, etc)
- java.util.Iterator

```
public boolean hasNext()
public Object next() ; // Mutator
```

- Issues abound when you consider insertions and deletions from the underlying data structure.

Survey of some other patterns (cont'd)

Mediator (Behavioural) Use an object to mediate all interaction between two or more other objects.

- The mediator class calls on the other classes so that neither has to call (and hence depend on) the other.

Strategy (Behavioural) Define a family of algorithms, encapsulate each one, and make them interchangeable.

- More flexible than template method in that it allows run-time configuration.

Patterns Survey (cont'd): Template Method (Behavioural)

Vary details of algorithm by filling in abstract methods.

```
abstract class AbstractParent {
  public void templateMethod() {
    ... // part of algorithm
    hookMethod() //  down call
    ... // another part of algorithm
  }
  protected abstract void hookMethod();
  ...
}

class Child1 extends AbstactParent {
  protected void hookMethod() {
    // implementation 1
  } }
```

References

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1994.