Using Patterns

- What is a Pattern?
  - Reusable solution template (schema).
  - Captures proven solution in a canonical form.
  - Recognizes that certain solution structures recur in different problem domains.
  - Provides for reusable design.

Seminal Texts on Patterns

  - Commonly referred to as “Gamma” or the “Gang of Four” text.

  - Commonly referred to as the “POSA” text
Patterns--A More Precise Definition
(from POSA)

- A Pattern is a three part schema:
  - Context--a situation diving rise to the problem.
  - Problem--The recurring problem arising in that context.
    - Requirements
    - Constraints
    - Desirable Properties
  - Solution--A proven resolution of the problem.
    - Structure--components and relationships.
    - Behavior--dynamic aspects.

Patterns--An Example

- Consider a software system with a human-computer interface (HCI).
  - Prone to change requests.
    - New functionality
    - Adapted for specific customers
  - May need to be ported to different platforms with different “look-and-feel” conventions.
  - In general, the HCI of a long-lived system is a constantly moving target.
Pattern Example (continued)

- The Problem:
  - Changes to user interface should be easy and possible at run-time.
  - Adapting or porting the user interface should not impact the design or code in the functional part of the application.

The MVC Pattern

- **Model**
  - Core data
  - set_of_observers
  - attach(observer)
  - detach(observer)
  - notify
  - getData
  - service
  - attach request service

- **Controller**
  - myModel
  - myView
  - initialize(Model, View)
  - handleInputEvent
  - update

- **View**
  - myModel
  - myController
  - Initialize(Model)
  - makeController
  - activate
  - display
  - update
  - create
  - manipulate
  - display

- **Observer**
  - update

Diagram:

```
Pattern Example (continued)

• The Problem:
  – Changes to user interface should be easy and possible at run-time.
  – Adapting or porting the user interface should not impact the design or code in the functional part of the application.

The MVC Pattern

- **Model**
  - Core data
  - set_of_observers
  - attach(observer)
  - detach(observer)
  - notify
  - getData
  - service
  - attach request service

- **Controller**
  - myModel
  - myView
  - initialize(Model, View)
  - handleInputEvent
  - update

- **View**
  - myModel
  - myController
  - Initialize(Model)
  - makeController
  - activate
  - display
  - update
  - create
  - manipulate
  - display

- **Observer**
  - update
```
MVC Pattern--Structure

- **Class: Model**
  - **Responsibility**
    - Provides functional core of the application
    - Registers dependent views and controllers
    - Notifies dependent components about data changes
  - **Collaborators**
    - View
    - Controller

- **Class: View**
  - **Responsibility**
    - Creates and initializes its associated controller
    - Displays information to the user
    - Implements the update procedure
    - Retrieves data from the model
  - **Collaborators**
    - Controller
    - Model

MVC Pattern Structure--Continued

- **Class: Controller**
  - **Responsibility**
    - Accepts user input as events
    - Translates events to service requests for the model and/or display requests for the view.
    - Implements the update procedure (if required)
  - **Collaborators**
    - View
    - Model
MVC Pattern--Behavior

Scenario: Change-propagation resulting from user input (simplified view):

MVC Pattern--Summary

• Benefits:
  – Allows multiple, independent views of the same model
  – Views can be synchronized
  – “Pluggable” views and controllers

• Liabilities:
  – Increased complexity
  – Potential for excessive number of updates
  – Intimate connection between view and controller
  – Close coupling of view and controller to model
MVC Pattern Summary (Continued)

- Liabilities (continued)
  - Inefficiency of data access in view
  - View and controller are not inherently portable
  - Not particularly compatible with modern user-interface tools.

Applying Patterns--A Quick Example

- Consider the development of an interactive text editor
  - Typical operation: select text with mouse and change it from regular to bold type by selecting a menu item or clicking in icon.
- Is MVC an appropriate architecture for this application?
Text Editor based upon MVC

- Text selection--controller action--no change to model
- Selected text is input to another controller responsible for changing the face of the selected text.
- But text selection also has a visual appearance--i.e. selected text must be highlighted.
- Does controller implement this “view-like” behavior somehow interact with the view in which selected text appears?

Text Editor Example--Better Solution

- Use a variant of MVC that unifies view and controller functionality into a single component.
  - Called Document-View Pattern
Pattern Categories

• Architectural Patterns
  – Express fundamental (generic) structural organization schema for software systems
  – Specify a predefined set of subsystems, their relationships, and interactions.
  – Specify rules and guidelines for organizing relationships among subsystems.

Pattern Categories (Continued)

• Design Patterns
  – Provides a scheme for refining subsystems or components or the relationship between them.
  – Describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
Pattern Categories (continued)

- Implementation Idioms
  - Low-level patterns, often specific to a particular programming language
  - Describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Design Patterns--Example of a “Gang of Four” Pattern

- The Command Pattern
  - Allow an object to issue requests without knowing anything about the operation being requested or the receiver of the request.
    - E.g connecting an action to a button on a GUI.
  - Encapsulates requests as objects
  - An abstract Command Class declares the interface for executing operations
  - Concrete Command Subclasses specify receiver-action pairs.
  - Receivers carry out the requests.
Command Pattern Structure

- **Client**
- **Invoker**
- **Command**: declares an interface for executing an operation
- **ConcreteCommand**: defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operations on Receiver
- **Receiver**: Action()

Receiver -> Action()

Command Pattern--Participants

- **Command**
  - declares an interface for executing an operation
- **Concrete Command**
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operations on Receiver
- **Client**
  - creates a ConcreteCommand object and sets its receiver.
Command Pattern Participants--Continued

- **Invoker**
  - asks the command to carry out the request.
- **Receiver**
  - knows how to perform the operations associated with carrying out a request. Any class can serve as a Receiver.

---

Command Pattern--Collaboration

![Diagram of Command Pattern Participants]

1. ref := CreateCommand(receiver:Receiver)
2. StoreCommand(ref:ConcreteCommand)
3.1 Action()
3. Execute()
Command Pattern Example

A menu interface for a document processing system:

Command Pattern--Sample Code

// The abstract Command Class:

class Command {
public:
    virtual ~Command;
    virtual void Execute() = 0;
protected:
    Command();
};
Command Pattern Code-Continued

//The Concrete Class PasteCommand:

class PasteCommand : public Command {
public:
    PasteCommand(Document*);
    virtual void Execute( );
private:
    Document* _document;
};
PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}
void PasteCommand::Execute ( ) {
    _document->Paste( );
}

Command Class Code--Continued

//A template for simple commands

template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action) ( );
    SimpleCommand (Receiver* r, Action a) :
        _receiver (r), _action (a) {  }
    virtual void Execute ( );
private:
    Action _action;
    Receiver* _receiver;
};
template <class Receiver>
void SimpleCommand<receiver>::Execute ( ) {
    (_receiver->*_action)( );
}
Command Class Code--Continued

Client code to create a command that calls Action on an instance of MyClass:

MyClass* receiver = new MyClass;
// …
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// …
aCommand->Execute();
// ...

Another “Gang of Four” Pattern--The Proxy Pattern

- Allows a proxy (or surrogate) object to serve as a stand-in for a “real” object
- Applicability:
  - Remote proxy: provides local representative for an object in a different address space.
  - Virtual proxy--creates expensive objects on demand
  - Protection proxy--controls access to the original object
  - Smart reference--replaces bare pointer to an object with a reference that performs additional actions.
### Proxy Pattern Structure

- **Client**
  - \( \text{Request}(\cdot) \)
- **Subject**
  - \( \text{Request}(\cdot) \)
- **Proxy**
  - \( \text{Request}(\cdot) \)
  - `realSubject->Request`
- **RealSubject**
  - \( \text{Request}(\cdot) \)

### Proxy Pattern Participants

- **Proxy**
  - maintains a reference that allows proxy to access the real subject.
  - Provides interface identical to the subject’s.
  - controls access to the real subject--may be responsible for creation/deletion of real subject.
  - Other responsibilities depending on application:
    - remote proxy--encodes requests and arguments and sends them to real subject.
    - Virtual proxy--caches information about real subject
    - protection proxy--verifies that caller has appropriate access permissions.
Proxy Pattern Participants--Continued

- **Subject**
  - defines common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject**
  - defines the real object that the proxy represents.

Proxy Pattern Example

- **Document Processing System**
  - Can embed graphical objects in document.
  - Some graphical objects can be expensive to create.
  - Want to avoid creating all expensive objects when document is opened.
  - Will create images on demand--i.e., when they become visible.
  - Will use an *image proxy* as a stand-in for the real object.
Proxy Pattern Example--Continued

General Idea:

:TextDocument
  image

:ImageProxy
  fileName

:Image
  data

in memory

on disk

DocumentEditor

* Graphic
  Draw()
  GetExtent()
  Store()
  Load()

Image
  imageImp
  extent
  Draw()
  GetExtent()
  Store()
  Load()

ImageProxy
  fileName
  extent
  Draw()
  GetExtent()
  Store()
  Load()

if (image == 0)
  image = loadImage(fileName);
  image->Draw();
else
  if (image == 0)
    return extent;
  else
    return image->GetExtent();
// Graphic class defines interface for graphical objects

class Graphic {
public:
    virtual ~Graphic ( ) = 0;
    virtual void Draw (const Point& at) = 0;
    virtual void HandleMouse (Event& event) = 0;
    virtual const Point& getExtent ( ) = 0;
    virtual void Load (istream& from) = 0;
    virtual void Save (ostream& to) = 0;
protected:
    Graphic ( );
};

// Image class implements the Graphic interface to display
// image files. Image overrides HandleMouse to let users
// resize the image interactively

class Image : public Graphic {
public:
    Image (const char* file); // loads image from a file
    virtual ~Image ( );
    virtual void Draw (const Point& at);
    virtual void HandleMouse (Event& event);
    virtual const Point& GetExtent ( );
    virtual void Load (istream& from);
    virtual void Save (ostream& to);
private:
    ...;
};
Proxy Pattern Example--Continued

// Image Proxy has same interface as Image:

class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy ( );
    virtual void Draw (const Point& at);
    virtual void HandleMouse (Event& event);
    virtual const Point& GetExtent ( );
    virtual void Load (istream& from);
    virtual void Save (ostream& to);
protected:
    Image* GetImage ( );
private
    Image* _image;
    Point _extent;
    char* _filename;
};

Proxy Pattern Example--Continued

// Constructor saves local copy of filename of the file that stores
// the image and initializes _extent and _image.

ImageProxy : : ImageProxy (const char* fileName)  {
    _fileName = strdup (fileName );
    _extent = Point : : Zero;   //don't know the extent yet
    _image = 0;
}

Image* ImageProxy : : GetImage ( ) {
    if (_image = = 0)
        _image = new Image (_fileName);
    return _image;
}
Proxy Pattern Example--Continued

//GetExtent returns the cached extent if possible.
// Draw loads the image.
// HandleMouse forwards the event to the real image.

Const Point& ImageProxy : : GetExtent ( )  {
  if (_extent = = Point : : Zero
    _extent = GetImage ( ) -> GetExtent ( );
    return _extent;
  }

void ImageProxy : : Draw (const Point& at)  {
  GetImage ( ) -> Draw (at);
}

void ImageProxy : : HandleMouse (event)  {
  GetImage ( ) -> HandleMouse (event);
}

Proxy Pattern Example--Continued

// The Save operation saves the cached image extent and image
// filename to a stream. Load retrieves this information and
// initializes the corresponding members.

void ImageProxy : : Save (ostream& to)  {
  to << _extent << _fileName;
}

void ImageProxy : : Load (istream& from)  {
  from >> _extent >> _fileName;
}
Proxy Pattern Example--Continued

Now, suppose we have a class TextDocument that can contain Graphic objects:

```cpp
class TextDocument {
public:
    TextDocument ( );
    void Insert (Graphic*);
    ... 
};
```

We can insert an ImageProxy into a text document like this:

```cpp
TextDocument* text = newTextDocument;
...
text ->Insert (new ImageProxy ("an ImageFileName"));
```

Proxy Pattern Example--Continued

C++ implementation of a virtual proxy using overloading of the member access operator (operator->)

```cpp
class Image;
extern Image* LoadAnImageFile(const char*);

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr ( );
    virtual Image* operator->( );
    virtual Image& operator*( );
private:
    Image* LoadImage ( );
    Image* _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr
( const char* theImageFile ) {
    _imageFile = theImageFile;
    _image = 0;
}

Image* ImagePtr::LoadImage ( ) {
    if (_image == 0)
        _image = LoadAnImageFile(_imageFile)
    return _image;
}
Proxy Pattern Example--Continued

The overloaded -> and * operators use LoadImage to return _image to callers (loading the image if necessary):

```cpp
Image* ImagePtr::operator-> ( )  {
    return LoadImage( );
}

Image& ImagePtr::operator* ( )  {
    return *LoadImage( );
}
```

Image Operations can be invoked through ImagePtr objects without making the operations part of the ImagePtr interface:

```cpp
ImagePtr image = ImagePtr ("an ImageFileName");
image->Draw(Point (50, 100));
  // (image.operator->( ) ) -> Draw(Point (50, 100) )
```

The “Gang of Four” Design Patterns--A Summary

• Creational Patterns:
  – Abstract factory--Provide an interface for creating families of related or dependent objects without specifying their concrete class.
  – Builder--Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  – Factory Method--Define an interface for creating an object, but let subclasses decide which class to instantiate--allow a class to defer instantiation to subclasses.
  – Prototype--Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  – Singleton--Ensure that a class has only one instance, and provide a global point of access to it.
<table>
<thead>
<tr>
<th>GOF Design Pattern Summary--Continued</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural Patterns:</strong></td>
</tr>
<tr>
<td>– <strong>Adapter:</strong> Convert the interface of a class into another interface that clients expect--allows classes work together that would otherwise have incompatible interfaces.</td>
</tr>
<tr>
<td>– <strong>Bridge:</strong> Decouple an abstraction from its implementation so that the two can vary independently.</td>
</tr>
<tr>
<td>– <strong>Composite:</strong> Compose objects into tree structures to represent part-whole hierarchies.--i.e Allow clients to treat individual objects and compositions of objects uniformly.</td>
</tr>
<tr>
<td>– <strong>Decorator:</strong> Attach additional responsibilities to an object dynamically. Provides an alternative to subclassing for extending functionality.</td>
</tr>
<tr>
<td>– <strong>Façade:</strong> Provide a unified, higher-level interface to a set of interfaces in a subsystem.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GOF Design Pattern Summary--Continued</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural Patterns (continued):</strong></td>
</tr>
<tr>
<td>– <strong>Flyweight:</strong> Use sharing to support large numbers of fine-grained objects efficiently.</td>
</tr>
<tr>
<td>– <strong>Proxy:</strong> Provide a surrogate for another object to control access to it.</td>
</tr>
<tr>
<td><strong>Behavioral Patterns:</strong></td>
</tr>
<tr>
<td>– <strong>Chain of Responsibility:</strong> Avoid coupling of sender of a request to its receiver by giving more than object a chance to handle the request. Receiving objects are chained and request is passed along the chain until an object handles it.</td>
</tr>
<tr>
<td>– <strong>Command:</strong> Encapsulate a request as an object, thereby allowing clients to be parameterized with different requests.</td>
</tr>
</tbody>
</table>
GOF Design Pattern Summary--Continued

• Behavioral Patterns (continued):
  – Interpreter--Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  – Iterator--Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  – Mediator--define an object that encapsulates the manner in which a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and allows their interaction to be varied independently.
  – Memento--Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.
  – Observer--Define a one-to-many dependency between objects so that when one object changes state the others are updated automatically.

GOF Design Pattern Summary--Continued

• Behavioral Patterns (continued):
  – State--Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  – Strategy--Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy allows the algorithm to be varied independently of the clients that use it.
  – Template Method--Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. This allows subclasses to redefine certain steps of the algorithm.
  – Visitor--Represent an operation to be performed on the elements of an object structure in a way that allows a new operation to be defined without changing the classes of the elements on which it operates.
Dealing with Aggregations--The Iterator Pattern

• Problem: How do you access the elements of an aggregate object without exposing its internal structure.
  – Suppose that we had used a list object as the container class for the group-item aggregation in our graphical editor example
  – How can we provide various types of access to this list without extending the interface of the list object.

The Iterator Pattern

• Separates the traversal mechanism from the storage mechanism for aggregate objects.
• Allows multiple traversal methods to be defined.
• Provides a uniform interface for traversing different aggregate structures (polymorphic iteration).
Iterator Pattern--Structure

- **Iterator**--defined an interface for accessing and traversing elements.
- **ConcreteIterator**--implements iterator interface and keeps track of current position in traversal of aggregate.
- **Aggregate**--defines interface for creating Iterator object
- **ConcreteAggregate**--implements the iterator creation interface to return an instance of the proper ConcreteIterator
Iterator Pattern Example

Traversing a list:

```
List
Count()
Append()
Remove(Element)
```

```
ListIterator
list
Index
First()
Next()
IsDone()
CurrentItem()
```

Issues:
- List must be instantiated before ListIterator
- Iterator and list are coupled.
- Client must know that it is traversing a List

Iterator Pattern Example--Continued

Removing the coupling between client and aggregate structure:

```
AbstractList
CreateIterator()
Count()
Append(Item)
Remove(Item)
```

```
Client
```

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

```
List
```

```
ListIterator
```

AnotherType of List

AnotherListIterator
Iterator Example--C++ Implementation

The List interface (from C++ Foundation Class Library):

template <class Item>
class List {
  public:
    List (long size = DEFAULT_LIST_CAPACITY);
    List(List&);
    ~List();
    List& operator=(const List&);
    long Count( ) const;
    Item& Get(long index) const;
    Item& First( );
    Item& Last( );
    void Append(const Item&);
    void remove(const Item&);
    ...
};

Iterator Example Code--Continued

The Iterator Class:

template <class Item>
class Iterator {
  public:
    virtual void First( ) = 0;
    virtual void Next( ) = 0;
    virtual bool IsDone( ) const = 0;
    virtual Item CurrentItem( ) = 0;
  protected:
    Iterator( );
};
Iterator Example Code--Continued

The ListIterator subclass:

template <class Item>
class ListIterator : public Iterator<Item> {
    public:
        ListIterator (const List<Item>* aList);
        virtual void First();
        virtual void Next();
        virtual bool isDone() const;
        virtual Item CurrentItem() const;
    private:
        const List<Item>* _List;
        long _current;
};

Implementation of the ListIterator subclass

template <class Item>
ListIterator<Item> : : ListIterator (const List<Item>* aList):
    _list(aList), _current(0) {
}

template <class Item>
void ListIterator<Item> : : First() {
    _current = 0;
}

template <class Item>
void ListIterator<Item> : : Next() {
    _current++;
}

template <class Item>
bool ListIterator<Item> : : IsDone() const {
    return _current >= _list -> Count();
}

template <class Item>
Item ListIterator<Item> : : CurrentItem() const{
    if (IsDone( ))
        throw IteratorOutOfBoundsException;
    return _list -> Get(_current);
}

We could also define a ReverseListIterator that iterates the list in reverse order. Try this as an exercise.
Iterator Example Code--Continued

Using the Iterator to print a list of employees:

```cpp
void printEmployees (Iterator<Employee*>& i) {
    for (i.first(); i.IsDone; i.Next())
        i.CurrentItem()->Print();  //Employee class has a Print method
}
...

List<Employee*>* employees;
...
ListIterator<Employee*> list_iterator(employees);
...
PrintEmployees(list_iterator);
```

Note: This implementation is not truly polymorphic with respect to List. We will discuss this issue in our next exciting episode.

The Iterator Pattern--Continued

The previous implementation of the iterator is not polymorphic with respect to the List implementation. For instance, if we wanted to substitute a new list class called Another List for List, we would need to do the following:

```cpp
AnotherList<Employee*>* employees;
...
// AnotherList Class must provide AnotherListIterator
...
AnotherListIterator<Employee*> iterator(employees);

// AnotherList Iterator conforms to Iterator interface
// so PrintEmployees doesn’t have to be modified.
PrintEmployees(iterator);
```
Making the Iterator Polymorphic

1. Introduce an abstract class `AbstractList` to standardize the interface for `List` and `AnotherList`.

2. `AbstractList` will define a **factory method** `CreateIterator`. Subclasses will override this method to return their corresponding iterator.

   ```cpp
   template <class Item>
   class AbstractList : {
   public:
     virtual Iterator<Item>* CreateIterator() const = 0;
     ...
   };
   ```

Making the Iterator Polymorphic--Continued

3. Now the code for printing employees can be independent of the list implementation.

   ```cpp
   AbstractList<Employee*>* employees;
   ...
   Iterator<Employee*>* iterator = employees->CreateIterator();
   PrintEmployees(*iterator);
   
   // Since we create a new Iterator object each time we do an iteration, we should be sure to delete Iterators when we are through using them:
   delete iterator;
   ...