Mobile Application Development

Higher Diploma in Science in Computer Science

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

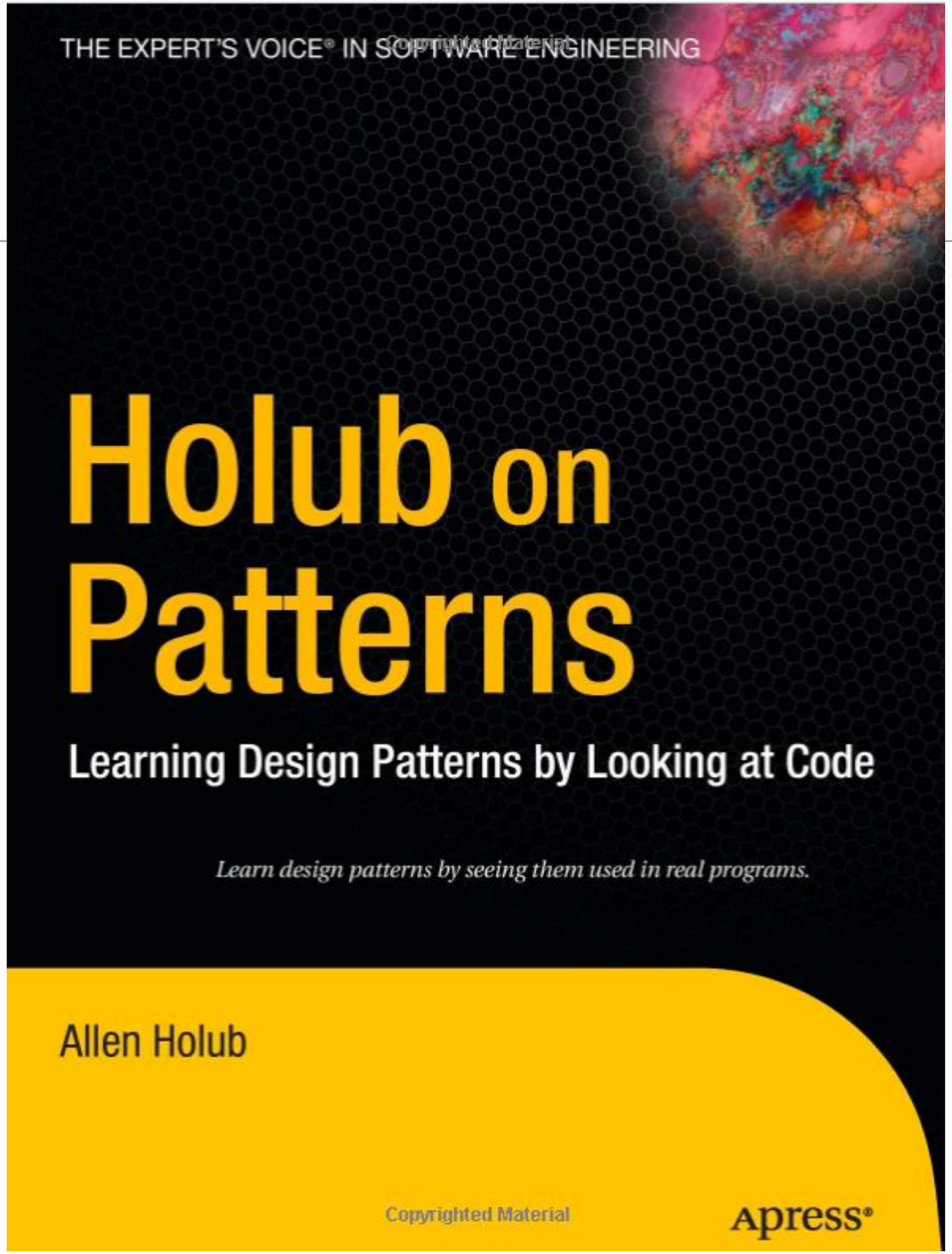
http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

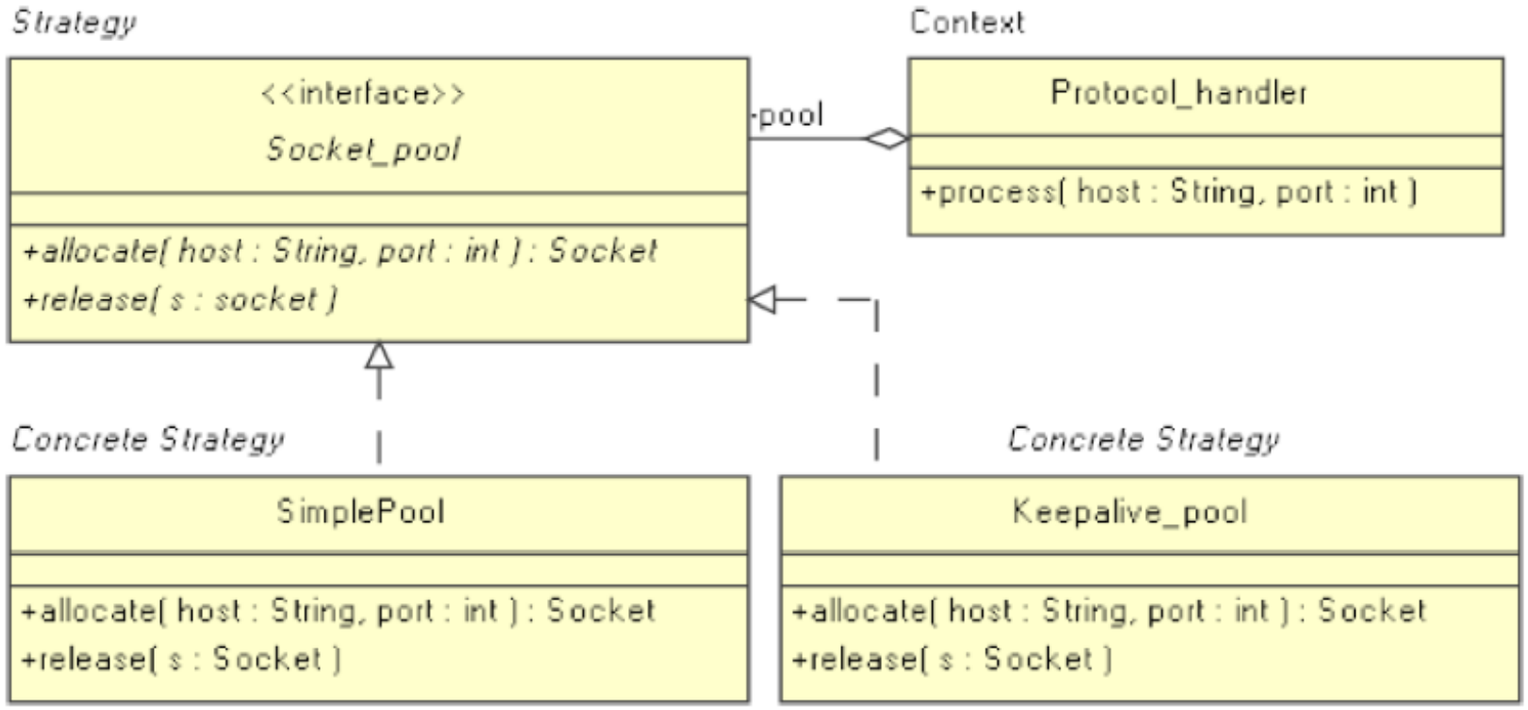
Catalogues

Holub


THE EXPERT'S VOICE® IN SOFTWARE ENGINEERING
Holub on Patterns
Learning Design Patterns by Looking at Code
Learn design patterns by seeing them used in real programs.
Allen Holub
Copyrighted Material
Apress®

Strategy

Define an interface that defines a strategy for performing some operation. A family of interchangeable classes, one for each algorithm, implements the interface.



Strategy: an interface that allows access to an algorighm.
Concrete Strategy: Implements a particular algorithm to conform to the Strategy interface.
Context: Uses the algorithm through the Strategy interface.

What Problem Does It Solve?

Sometimes, the only difference between subclasses is the strategy that's used to perform some common operation. For example, a frame-window might lay out its components in various ways, or a protocol handler might manage sockets in various ways. You can solve this problem with derivation—several Frame derivatives would each lay out subcomponents in different ways, for example. This derivation-based solution creates a proliferation of classes, however. In *Strategy*, you define an interface that encapsulates the strategy for performing some operation (like layout). Rather than deriving classes, you pass the "Context" class the strategy it uses to perform that operation.

Pros (✓) and Cons (✖)

✓ Strategy is a good alternative to subclassing. Rather than deriving a class and overriding a method called from the base class, you implement a simple interface.

✓ The strategy object concentrates algorithm-specific data that's not needed by the "Context" class in a class of its own.

✓ It's very easy to add new strategies to a system, with no need to recompile existing classes.

✖ There's a small communication overhead. Some of the arguments passed to the strategy objects might not be used.

Often Confused With

Command objects are very generic. The invoker of the command doesn't have a clue what the command object does. A *Strategy* object performs a specific action.

See Also

Command

Implementation Notes and Example

```
interface Socket_pool
{ Socket  allocate( String host, int port )
  void    release ( Socket s )
}
class Simple_pool implements Socket_pool
{ public Socket allocate(String host,int port)
  { return new Socket(host, port);
  }
  public void release(Socket s)
  { s.close();
  }
};
class Keepalive_pool implements Socket_pool
{ private Map connections = new HashMap();
  public Socket allocate(String host,int port)
  { Socket connection =
       (Socket)connections.get(host+":"+port);
    if(connection == null)
      connection = new Socket(host,port);
    return connection;
  }
  public void release(Socket s)
  { String host =
            s.getInetAddress().getHostName();
    connections.put( host+":"+s.getPort(),s );
  }
  //...
}
class Protocol_handler
{ Socket_pool pool = new Simple_pool();
  public void process( String host, int port )
  { Socket in = pool.allocate(host,port);
    //...
    pool.release(in);
  }
  public void set_pooling_strategy( Socket_pool p)
  { pool = p;
  }
}
```

The code at left implements a skeleton protocol handler. Some of the hosts that the handler talks to require that sockets used for communication are closed after every message is processed. Other hosts require that the same socket be used repeatedly. Other hosts might have other requirements. Because these requirements are hard to predict, the handler is passed a socket-pooling strategy.

The default strategy (`Simple_pool`) simply opens a socket when asked and closes the socket when the `Protocol_handler` releases it.

The `Keepalive_pool` implements a different management strategy. If a socket has never been requested, this second strategy object creates it. When this new socket is released, instead of closing it, the strategy object stores it in a `Map` keyed by combined host name and port number. The next time a socket is requested with the same port name and host, the previously created socket is used. A more realistic example of this second strategy would probably implement notions like “aging,” where a socket would be closed if it hadn’t been used within a certain time frame.

In the interest of clarity, I’ve left out the exception handling.

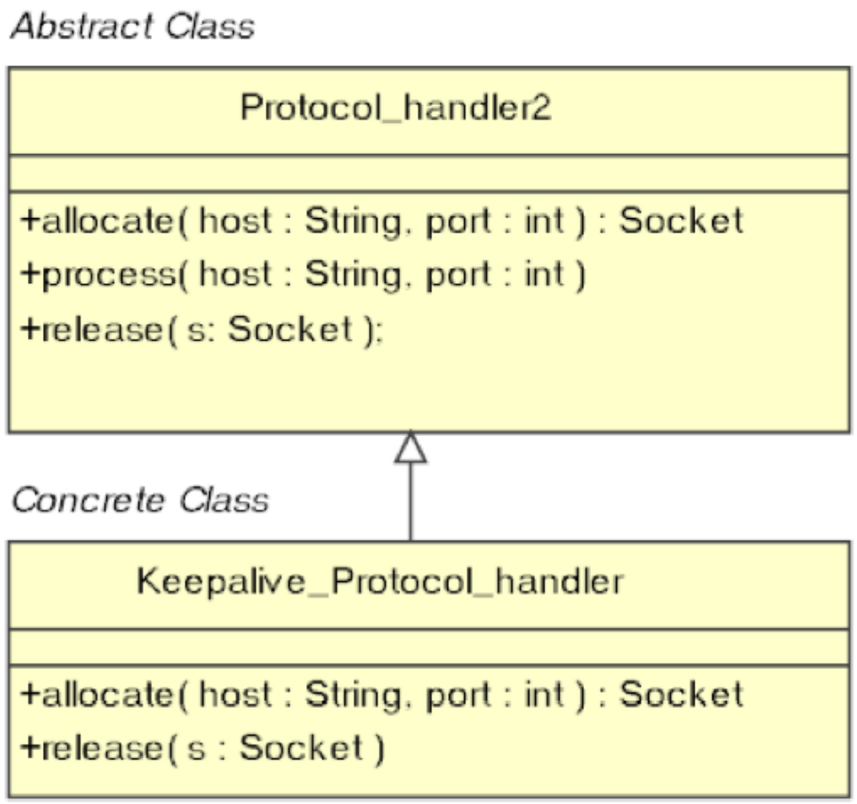
Usage

`JFrame frame = new JFrame();`
`frame.getContentPane().setLayout`
`( new FlowLayout() );`
`frame.add( new JLabel("Hello World");`
	The `LayoutManger` (`FlowLayout`) defines a strategy for laying out the components in a container (`JFrame`, the "Context");

`String[] array = new String[]{ ... };`
`Arrays.sort`
`( array,`
`new Comparator`
`{ int Compare( Object o1, Object o2 )`
`{ return ((String)o1).compareTo((String)o2);`
`}`
`}`
`);`
	The **`Arrays.sort(...)`** method is passed an array to sort and a **`Comparator`** that defines a strategy for comparing two array elements. This use of *Strategy* makes **`sort(...)`** completely generic—it can sort arrays of anything..

Template Method

Define an algorithm at the base-class level. Within the algorithm, call an abstract method to perform operations that can't be generalized in the base class. This way you can change the behavior of an algorithm without changing its structure.



Abstract Class: Defines an algorithm that uses "primitive" operations that are supplied by a derived class.
Concrete Class: Implements the "primitive" operations.

What Problem Does It Solve?

Template method is typically used in derivation-based application frameworks. The framework provides a set of base classes that do 90% of the work, deferring application-specific operations to abstract methods. You use the framework by deriving classes that implement this application-specific behavior.

Pros (✓) and Cons (✖)

✖ Template method has little to recommend it in most situations. *Strategy*, for example, typically provides a better alternative. Well done class libraries work "out of the box." You should be able to instantiate a framework class and it should do something useful. Generally, the 90/10 rule applies (10% of the functionality is used 90% of the time, so that 10% should be define the default behavior of the class). In template method, however, the framework defines *no* default behavior, but rather, you are required to provided derived classes for the base class to do anything useful. Given the 90/10 rule, this means that you have to do unnecessary work 90% of the time.

Template method does not prohibit the class designer from providing useful default functionality, expecting that the programmer will modify the behavior of the base class through derived-class overrides. In an OO system, though, using derivation to modify base-class behavior is just run-of-the-mill programming, hardly worth glorifying as an official pattern.

✓ One reasonable application of *Template Method* is to provide empty "hooks" at the base-class level solely so that a programmer can insert functionality into the base class via derivation.

Often Confused With

Factory Method is nothing but a *Template Method* that creates objects.

See Also

Factory Method.

Implementation Notes and Example

```
class Protocol_handler2
{ public Socket allocate(String host,int port)
  { return new Socket(host, port);
  }
  public void release(Socket s)
  { s.close();
  }

  public void process( String host, int port )
  { Socket in =
        socket_pool.allocate(host,port);
    //...
    socket_pool.release(in);
  }
}

class Keepalive_Protocol_handler
{
  private Map connections = new HashMap();

  public Socket allocate(String host,int port)
  { Socket connection =
       (Socket)connections.get(host+":"+port);
    if(connection == null)
      connection = new Socket(host,port);
    return connection;
  }
  public void release(Socket s)
  { String host=
       s.getInetAddress().getHostName();
    connections.put( host+":"+s.getPort(),s);
  }
}
```

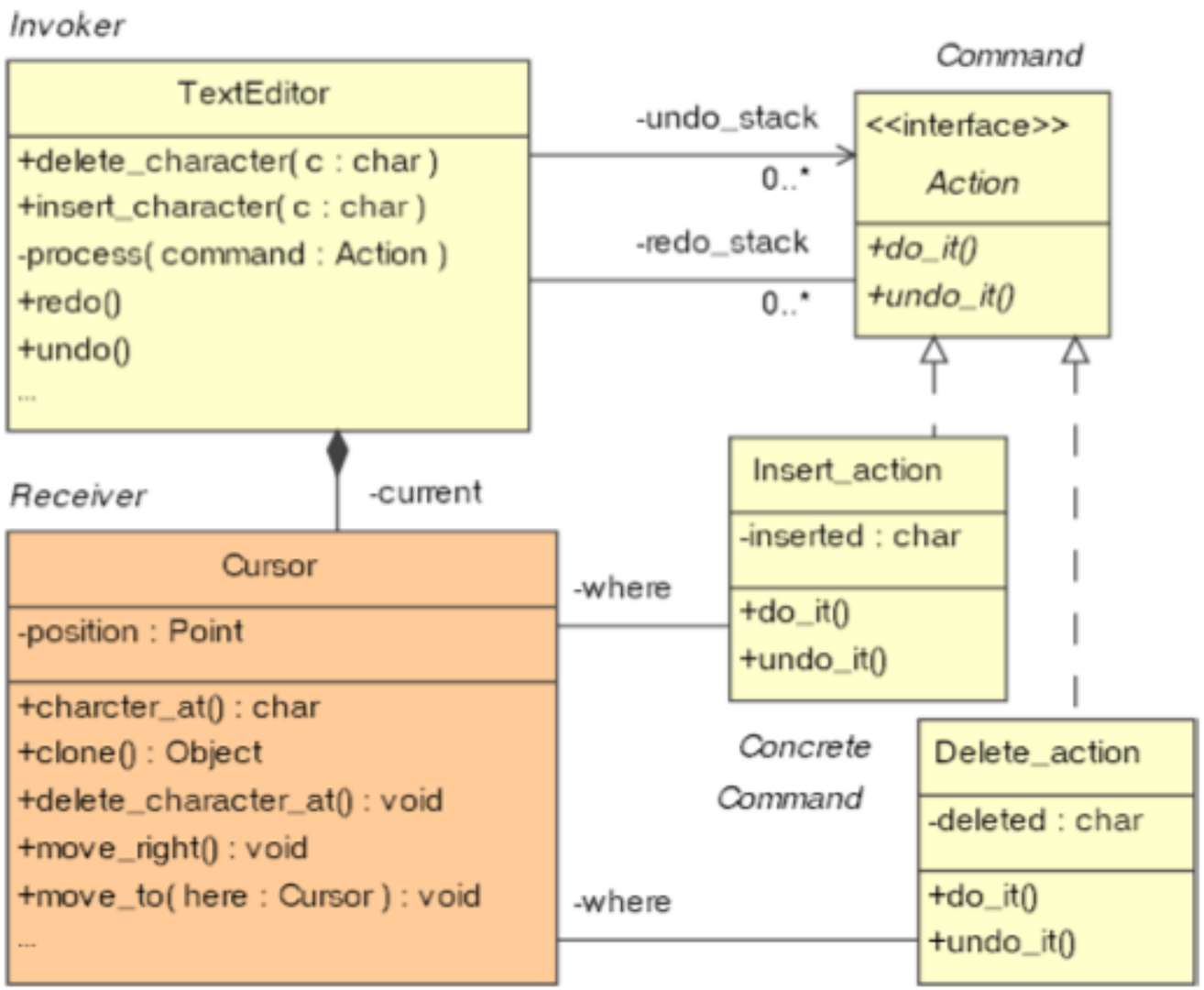
This example is the example from *Strategy* rewritten to sue *Template Method*. Rather than provide a strategy object, you derive a class that modifies the base-class behavior. Put differntly, you modify the behavior of the protocol-processing algorithm with respect to socket management.

Usage

`class my_Panel extens JPanel`
`{  public void paint( Graphics g )`
`  { g.drawString("Hello World", 10, 10);`
`  }`
`}`
	Define painting behavior by overriding the `paint(...)` method. You could easily do the same thing by passing a `Panel` a "paint" strategy.

Command

Encapsulate a request or unit of work into an object. *Command* provides a more capable alternative to a function pointer because the object can hold state information, can be queued or logged, and so forth.



Command: Defines an interface for executing an operation or set of operations.
Concrete Command: Implements the *Command* interface to perform the operation. Typically acts as an intermediary to a *Receiver* object.
Invoker: Asks the command to carry out a request.
Receiver: Knows how to carry out the request. This functionality is often built in to the Command object itself.

What Problem Does It Solve?

You can't have a function pointer in an OO system simply because you have no functions, only objects and messages. Instead of passing a pointer to a function that does work, pass a reference to an object that knows how to do that work.

A Command object is effectively a transaction encapsulated in an object. Command objects can be stored for later execution, can be stored as-is to have a transaction record, can be sent to other objects for execution, etc.

Command is useful for things like "undo" operations. It's not possible to undo an operation simply by rolling the program back to a previous state, because the program might have had an effect on the outside world while transitioning from the earlier state to the current one. Command gives you a mechanism for actively rolling back state by actively reversing side effects like database updates.

By encapsulating the work in an object, you can also define several methods, and even state information, that work in concert to do the work. For example, a single object can encapsulate both "undo" and "redo" operations and the state information necessary to perform these operations.

Command also nicely solves "callback" problems in multithreads systems. A "client" thread creates a command object that performs some operation, then notifies that client when the operation completes. The client then gives the *Command* object to a second thread on which the operation is actually performed.

Pros (✓) and Cons (×)

✓ Command decouples operations from the object that actually performs the operation.

Often Confused With

Strategy. The invoker of a *Command* doesn't know what the *Command* object will do. A *Strategy* object encapsulates a method for doing a specific task for the invoker.

See Also

Memento, Strategy

Implementation Notes and Example

```
abstract class Cursor extends Cloneable
{   public Object clone();
    public char  character_at();
    public void  delete_character_at();
    public void  insert_character_at(
                            char new_character);
    public void  move_to(Cursor new_position);
    public void  move_right();
    //...
}
class Text_Editor
{   private Cursor current=Cursor.instance();
    private LinkedList undo_stack =
                            new LinkedList();
    private LinkedList redo_stack =
                            new LinkedList();
    public void insert_character(char c)
    {   process( new Inserter(c) );
    }
    public void delete_character()
    {   process( new Deleter() );
    }
private void process( Action command )
{   command.do_it();
    undo_stack.addFirst(command);
}
public void undo()
{   Action action =
          (Action) undo_stack.removeFirst();
    action.undo_it();
    redo_stack.addFirst( action );
}
public void redo()
{   Action action =
          (Action) redo_stack.removeFirst();
    action.do_it();
    undo_stack.addFirst( action );
}
private interface Action
{   void do_it();
    void undo_it();
}
private class Inserter implements Action
{   Cursor where = (Cursor) current.clone();
    char   inserted;
    public Insert_action(char new_character)
    {  inserted = new_character;
    }
    public void do_it()
    {  current.move_to( where );
       current.
           insert_character_at(inserted);
       current.move_right();
    }
    public void undo_it()
    {  current.move_to( where );
       current.delete_character_at();
    }
}
private class Deleter implements Action
{
    Cursor where = (Cursor) current.clone();
    char   deleted;
    public void do_it()
    {  current.move_to( where );
       deleted = current.character_at();
       current.delete_character_at();
    }
    public void undo_it()
    {  current.move_to( where );
       current.
           insert_character_at( deleted );
       current.move_right();
    }
  }
  //...
}
```

Most of the work is done by the **Cursor**, which reifies *Iterator* (see). The Text editor is driven by a Client class (not shown) that interprets user input and tells the editor to do things like insert or delete characters. The **Text_Editor** performs these request by creating *Command* objects that implement the **Action** interface. Each **Action** can both do something and also undo whatever it did. The editor tells the **Action** to do whatever it does, and then stacks the object. When asked to undo something, the editor pops the Action off the undo stack, asks it to undo whatever it did, and then puts it on a redo stack.. Redo works in the a similar way, but in reverse.

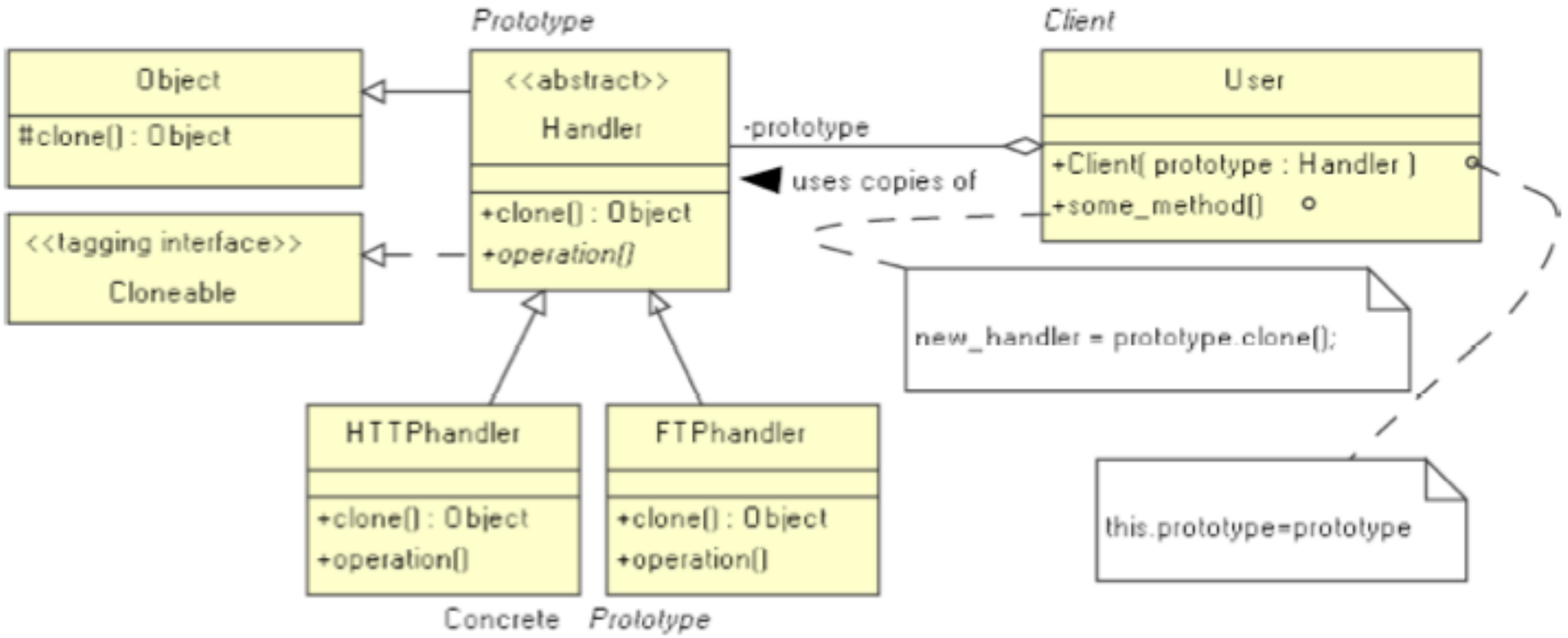
Usage

`new Thread()`
`{ public void run(){ /*...*/ }`
`}.start();`
	`Thread` is passed a `Runnable` *Command* object that defines what to do on the thread.

`java.util.Timer t = new java.util.Timer();`
`t.schedule( new java.util.TimerTask()`
`{ public void run()`
`{System.out.println(□hello world□);}`
`}, 1000);`
	Print "hello world." one second from now. The `TimerTask` is a *Command* object. Several `TimerTask` objects may be queued for future execution.

Prototype

Create objects by making copies of ("cloning") a prototypical object. The prototype is usually provided by an external entity or a Factory, and the exact type of the prototype (as compared to the interfaces it implements) may not be known.



Prototype: Interface of object to copy, must define a mechanism for cloning itself.
ConcretePrototype: (Object that's copied, implements cloning mechanism.
Client: Creates a new object by asking the Prototype for a clone.

What Problem Does It Solve?

(1) In *Abstract Factory*, information needed to initialize the Concrete Product (constructor arguments, for example) must be known at compile time. Most Abstract Factory reifications use the default, "no-arg," constructor. When you use Abstract Factory to make objects that must be in a non-default state, you must first create the object, then modify it externally, and this external modification may happen in many places in the code. It would be better to create objects with the desired initial (non-default) state and simply copy those objects to make additional ones. You might use Abstract Factory to make the prototype object.

(2) Sometimes. objects will be in only a few possible states, but you have many objects in each state. (The GoF describe a Note class in a music-composition system; there are many instances of whole-note, half-note, and quarter-note objects—but there all whole notes are in an identical state.

(3) Sometimes classes are specified at runtime and are created with "dynamic loading" [e.g. `Class.forname("class.name")`] or a similarly expensive process (when initial state is specified in an XML file, for example). Rather than repeatedly going through the expense of creating an object, create a single prototype and copy it multiple times.

Pros (✓) and Cons (✖)

✓ You can install a new concrete product into a Factory simply by giving it a prototype at run time. Removal is also easy.

✓ Prototype can reduce object-creation time.

✓ Abstract factory forces you to define classes with marginally different behavior using subclassing. Prototype avoids this problem by using state. When an object's behavior changes radically with state, you can look at the object as a dynamically specifiable class, and Prototype is your instantiation mechanism.

✖ You must explicitly implement `clone()`, which can be quite difficult. Worry about the memory-allocation issues discussed below. Also think about deep-vs.-shallow copy issues (should you copy a reference, or should you clone the referenced object?). Finally, sometimes the clone method should act like a constructor and initialize some fields to default values. A clone of a list member cannot typically be in the list, for example.

See Also

Abstract Factory, State

Implementation Notes and Example

```
abstract class Handler implements Cloneable
{   // Derived classes of Handler must be placed
    // in the com.holub.tools.handlers package,
    // and must have the name ProtocolHandler,
    // where Protocol is the handled protocol.

    public Object clone()
    {   Handler copy =
                (Handler)(super.clone());
        // initialize fields of copy here...
        return copy;
    }
    abstract public operation();
}

class User
{   private Handler prototype;
    public User( Handler prototype )
    {   this.prototype = prototype;
    }
    public some_method()
    {   Handler new_handler=prototype.clone();
        //...
    }
}

class User_of_User
{   private User my_client;
    public User_of_user(String protocol)
    {   StringBuffer name = new StringBuffer
              ("com.holub.tools.handlers");
        name.append(protocol);
        name.append("Handler");
        my_client = new Client(
              Class.forname(name.toString()));
    }
}
```

In this example, we need to create many copies of a Handler derivitive for a particular protocol. The prototcol is specified using a string, and the Handler is created dynamically using `Class.forname()`. Prototype is used to avoid the overhead of multiple calls to `Class.forname()`.

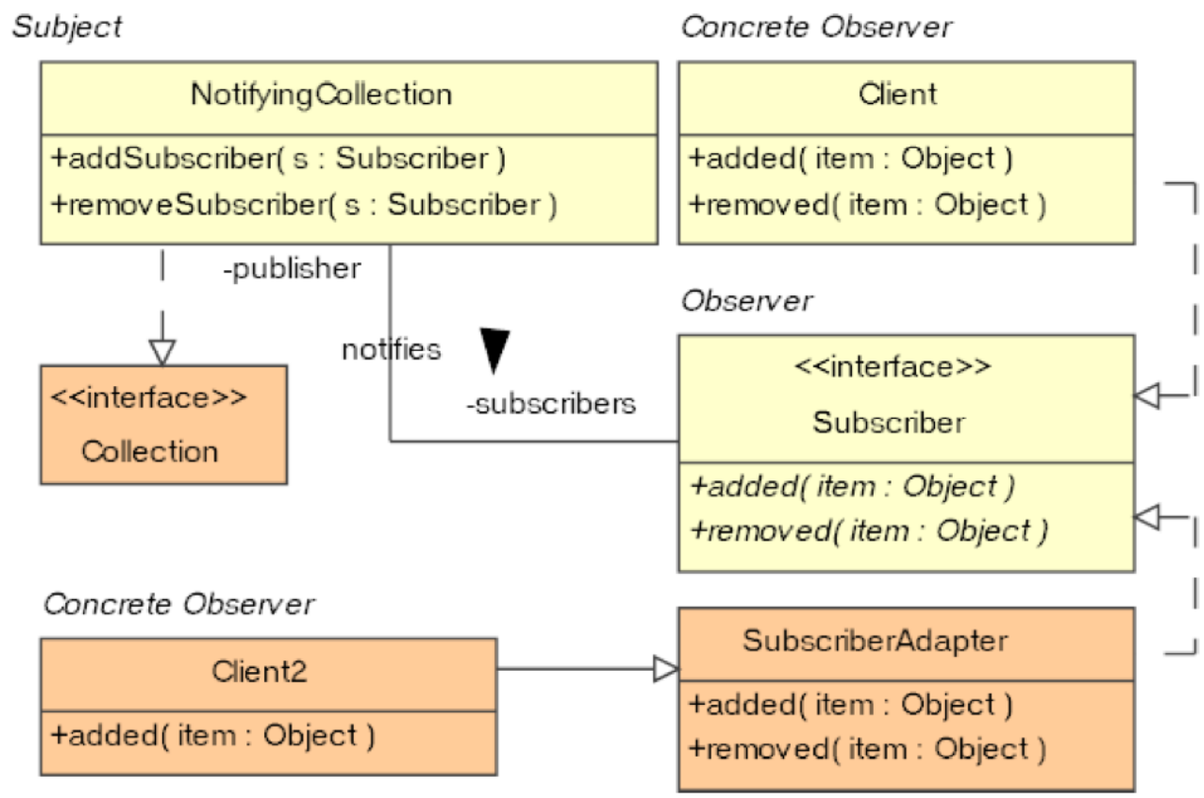
You cannot use `new` to implement a "clone" method. The following code won't work:

```
Class Grandparent
{   public Grandparent(Object args){/*...*/}
    Base my_clone(){ return new Base(args);}
}
Class Parent
{   public Parent(){   super("arg");}
    Derived my_clone(){return new Parent(args)}
}
Class Child
{   public Child(){ super(); }
    /* inherit the base-class my_clone */
}
//...
Grandparent g = new Child();
//...
g.my_clone(); // Returns a Parent, not Child!
```

Using Java's `clone()` solves this problem by getting memory from `super.clone()`, but `clone()` is `protected` for some reason, so can't be used directly in Prototype. Expose `clone()` with a `public` pass-through method [`create()`];

Observer (Publish/Subscribe)

When an object changes states, it notifies other objects that have registered their interest at run time. The notifying object (publisher) sends an event (publication) to all its observers (subscribers).



Subject: The Publisher—notifies Observers that some event has occurred. Keeps a subscription list and a means for modifying the list. Sometimes *Subject* is an interface implemented by a *Concrete Subject*.

Observer: Defines an interface for notifying observers.

Participant: Implements the *Observer* interface to do something when notified..

What Problem Does It Solve?

In *Chain of Responsibility* a button notifies a parent of a press event like this:

```
class Window
{   void button_pressed() {/*...*/}
        //...
}

class Button implements Window
{   private Window parent;
    public Button(Window parent)
    {   this.parent = parent;  }
    on_mouse_click()
    { parent.button_pressed(); }
}
```

An abstraction-layer (business) object must learn about presses through a *Mediator* called a *Controller*—a `Window` derivative that overrides `button_pressed()` to send a message to the business object .The coupling relationships between the controllers, the abstraction layer, and the presentation (the button) are too tight. There's too much code affected if anything changes.

The Observer pattern addresses the problem by adding an interface between the "publisher" of an event (the button) and a "subscriber" (the business object that's actually interested in the button press. This interface decouples the publisher and makes it reusable in the sense that it's a stand-alone component, with no dependencies on the rest of the system. A publisher can notify any class that implements the subscriber interface.

Pros (✓) and Cons (✖)

✓ Observer nicely isolates subsystems, since the classes in the subsystems don't need to know anything about each other except that they implement certain "listener" interfaces. This isolation makes the code much more reusable.

✖ There's no guarantee that a subscriber won't be notified of an event after the subscriber cancels its subscription—a side effect of a thread-safety. (AWT and Swing both have this problem.)

✖ Publication events can propagate alarmingly when observers are themselves publishers. It's difficult to predict that this will happen.

✖ Memory leaks are easily created by "dangling" references to subscribers. (When the only reference to an object is the one held by a publisher, a useless might not be garbage collected. It's difficult in Java, where there are no "destructor" methods, to guarantee that publishers are notified when an object becomes useless.

Often Confused With

Command, Strategy

See Also

Chain of Responsibility

Implementation Notes and Example

```
public final class NotifyingCollection
                        implements Collection
{  private final Collection c;
   public NotifyingCollection(Collection wrap)
   {  c = wrap;   }
   private final Collection subscribers
                           = new LinkedList();
   public interface Subscriber
   {  void added  ( Object item );
      void removed( Object item );
   }
   synchronized public void addSubscriber(
                              Subscriber subscriber)
   {  subscribers.add( subscriber ); }
   synchronized public void removeSubscriber(
                              Subscriber subscriber)
   {  subscribers.remove( subscriber );
   }
   private void notify(boolean add, Object o)
   { Object[] copy;
     synchronized(this)
     {  copy = subscribers.toArray();
     }
     for( int i = 0; i < copy.length; ++i )
     { if(add)((Subscriber)copy[i]).added  (o);
       else   ((Subscriber)copy[i]).removed(o);
     }
   }
   public boolean add(Object o)
   { notify(true,o);  return c.add(o);    }
   public boolean remove(Object o)
   { notify(false,o); return c.remove(o);  }
   public boolean addAll(Collection items)
   {   Iterator i = items.iterator()
       while( i.hasNext() )
          notify( true, i.next() );
      return c.addAll(items);
   }
   public int size()    { return c.size();  }
   public int hashCode(){ return hashCode();}
   // pass-through implementations of other
   // Collection methods go here...
}
```

The example at left is a *Decorator* (see) that wraps a collection to add a notification feature. Objects that are interested in finding out when the collection is modified register themselves with the collection.
In the following example, I create an "adapter" (in the Java/AWT sense, this is *not* the *Adapter* pattern) that simplifies subscriber creation. By extending the adapter rather than implementing the interface, we're saved from having to implement uninteresting methods. I then add a subscriber:

```
class SubscriberAdapter implements
              NotifyingCollection.Subscriber
{   public void added(Object item){}
    public void removed(Object item){}
}

 NotifyingCollection c =
     new NotifyingCollection(new LinkedList());
c.addSubscriber
(   new SubscriberAdapter()
    {  public void added( Object item )
       { System.out.println("Added " + item);
       }
    }
)
```

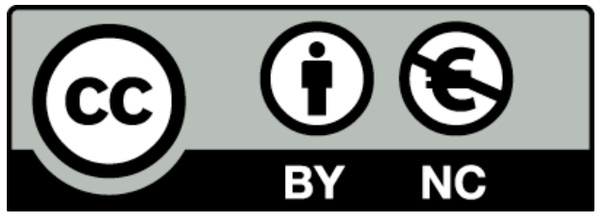
This implemenation of *Observer* is simplistc—copy is a very inefficent stragegy for solving the problem of one thread adding or removing a subcriber while notifications are in progress. A more realistic implementation was presented earlier in the book.

Observer encompases both one-to-many and many-to-one implementations. For example, one button could notify several observers when it's pressed, but by the same token, several buttons could all notify the same subscriber, which would use some mechanism (perhaps an event object passed as an argument) to deermine the publisher.

Usage

`JButton b = new JButton("Hello");`
`b.addActionListener(`
`  new ActionListener()`
`  { public void actionPerformed(ActionEvent e)`
`  { System.out.println("World");`
`  }`
`  } );`
	Print *World* when the button is pressed. The entire AWT event model is based on Observer. This model supercedes a Chain-of-Responsibility-based design that proved unworkable in an OO environment..

`Timer t = new java.util.Timer();`
`t.scheduleAtFixedRate( new TimerTask()`
`{ public void run()`
`  { System.out.println(new Date().toString());`
`  }`
`}, 0, 1000 );`
	Print the time once a second. The `Timer` object notifies all its observers when the time interval requested in the schedule method elapses.



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit