Undo

Some examples

Counter

counter.increase();     //increase counter by 1
counter.decrease();     //decrease counter by 1
Undo

Some examples

Text editing

Replace "Should" with "Could" at start of 3rd sentence in 5 paragraph
Undo - Some Issues

Redo

Multiple undo
Memento
Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

undo, rollbacks

Only originator:

Can access Memento’s get/set state methods
Create Memento
Example

package Examples;
class Memento{
    private Hashtable savedState = new Hashtable();

    protected Memento() {}; //Give some protection

    protected void setState( String stateName, Object stateValue ) {
        savedState.put( stateName, stateValue );
    }

    protected Object getState( String stateName) {
        return savedState.get( stateName);
    }

    protected Object getState(String stateName, Object defaultValue ) {
        if ( savedState.containsKey( stateName ) )
            return savedState.get( stateName);
        else
            return defaultValue;
    }
}
Sample Originator

package Examples;
class ComplexObject {
    private String name;
    private int someData;
    private Vector objectAsState = new Vector();

    public Memento createMemento() {
        Memento currentState = new Memento();
        currentState.setState( "name", name );
        currentState.setState( "someData", new Integer(someData) );
        currentState.setState( "objectAsState", objectAsState.clone() );
        return currentState;
    }

    public void restoreState( Memento oldState ) {
        name = (String) oldState.getState( "name", name );
        objectAsState = (Vector) oldState.getState( "objectAsState" );
        Integer data = (Integer) oldState.getState( "someData" );
        someData = data.intValue();
    }
}
Why not let the Originator save its old state?

class ComplexObject {
    private String name;
    private int someData;
    private Vector objectAsState = new Vector();
    private Stack history;

    public createMemento() {
        Memento currentState = new Memento();
        currentState.setState( "name", name );
        currentState.setState( "someData", new Integer(someData) );
        currentState.setState( "objectAsState", objectAsState.clone() );
        history.push(currentState);
    }

    public void restoreState() {
        Memento oldState = history.pop();
        name = (String) oldState.getState( "name", name );
        objectAsState = (Vector) oldState.getState( "objectAsState" );
        Integer data = (Integer) oldState.getState( "someData" );
        someData = data.intValue();
    }
}
Some Consequences

Expensive
Space

Narrow & Wide interfaces - Keep data hidden

Class Memento {
    public:
        virtual ~Memento();
    private:
        friend class Originator;
        Memento();
        void setState(State*);
        State* GetState();
}

class Originator {
    private String state;
    private class Memento {
        private String state;
        public Memento(String stateToSave) {
            state = stateToSave;
        }
        public String getState() {
            return state;
        }
        public Object memento() {
            return new Memento(state);
        }
    }
}
Using Clone to Save State

interface Memento extends Cloneable {
}

class ComplexObject implements Memento {
    private String name;
    private int someData;

    public Memento createMemento() {
        Memento myState = null;
        try {
            myState = (Memento) this.clone();
        }
        catch (CloneNotSupportedException notReachable) {
        }
        return myState;
    }

    public void restoreState(Memento savedState) {
        ComplexObject myNewState = (ComplexObject)savedState;
        name = myNewState.name;
        someData = myNewState.someData;
    }
}
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

Shallow Copy
Shallow Copy Verse Deep Copy

Original Objects

Deep Copy

Deeper Copy

Thursday, March 3, 16
class Door {
    public:
        Door();
        Door( const Door&);  
        virtual Door* clone() const;
        
        virtual void Initialize( Room*, Room*);
        // stuff not shown
    private:
        Room* room1;
        Room* room2;
    }

Door::Door ( const Door& other ) //Copy constructor {  
    room1 = other.room1;
    room2 = other.room2;
}

Door* Door::clone() const {
    return new Door( *this );
}
Cloning Issues - Java Clone

**Shallow Copy**

```java
class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

**Deep Copy**

```java
public class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        Door thisCloned = (Door) super.clone();
        thisCloned.room1 = (Room) room1.clone();
        thisCloned.room2 = (Room) room2.clone();
        return thisCloned;
    }
}
```
What if Protocol

When there are complex validations or performing operations that make it difficult to restore later

Make a copy of the Originator

Perform operations on the copy

Check if operations invalidate the internal state of copy

If so discard the copy & raise an exception

Else perform the operations on the Originator
Memento & Functional Programming

Immutable data
  Data that can not change
  Functional languages have primarily immutable data

If data can not change
  Don’t need memento pattern
Command
Command

Encapsulates a request as an object

Example
Invoker be a menu
Client be a word processing program
Receiver a document
Action be save
Sample Command

```java
public abstract class Command {
    public abstract void execute();
    public abstract void undo();
}

public class IncreaseCommand extends Command {
    private Counter subject;

    public IncreaseCommand(Counter toIncrease) {
        subject = toIncrease;
    }

    public abstract void execute() { subject.increase(); }
    public abstract void undo() { subject.decrease(); }
}
```
Sample Command - Text Editing

Requires more details

Text that is being edited
Location in text to changed
Replacement text

Undo requires

Text that is being edited
Location in text that was changed
Text that was replaced
When to Use the Command Pattern

- Need action as a parameter (replaces callback functions)
  - Lambda's replace this use

- Specify, queue, and execute requests at different times

- Undo

- Logging changes

- High-level operations built on primitive operations
  - A transaction encapsulates a set of changes to data
  - Systems that use transaction often can use the command pattern

- Macro language
Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it

It is easy to add new commands, because you do not have to change existing classes

You can assemble commands into a composite object
Pluggable Commands

Can create one general Command using reflection

Don’t hard code the method called in the command

Pass the method to call an argument
import java.util.*;
import java.lang.reflect.*;

public class Command
{
    private Object receiver;
    private Method command;
    private Object[] arguments;

    public Command(Object receiver, Method command,
                   Object[] arguments )
    {
        this.receiver = receiver;
        this.command = command;
        this.arguments = arguments;
    }

    public void execute() throws InvocationTargetException,
                                 IllegalAccessException
    {
        command.invoke( receiver, arguments );
    }
}
Using the Pluggable Command

```java
public class Test {
    public static void main(String[] args) throws Exception {
        Vector sample = new Vector();
        Class[] argumentTypes = { Object.class };
        Method add =
            Vector.class.getMethod( "addElement", argumentTypes);
        Object[] arguments = { "cat" };

        Command test = new Command(sample, add, arguments );
        test.execute();
        System.out.println( sample.elementAt( 0));
    }
}
```

Output

```
cat
```

27
Pluggable Commands using Lambdas

```java
public interface Command {
    void execute();
}

public class PluggableCommand {
    Command do;
    Command undo;

    public PluggableCommand(Command do, Command undo) {
        this.do = do;
        this.undo = undo;
    }

    public void execute() { do.execute(); }

    public void undo() { undo.execute(); }
```

Pluggable Commands using Lambdas

```java
final Counter example = new Counter();
PluggableCommand increase;

increase = new PluggableCommand(
    () -> example.increase(),
    () -> example.decrease());

increase.execute();
```

Note
Java's lambdas put restrictions on the variable example
Command Pattern & Lambda

Lambda’s can replace command objects for
  Callbacks
  Batch processing
  Logging
  Macro language
Functional Programming & Command

Simple cases - can just use function

But what if function needs
  State
  Receiver
Closures

function counter()
  n = 0
  return () -> n += 1
end

counter_a = counter()
counter_b = counter()
counter_a()    # 1
counter_a()    # 2
counter_a()    # 3
counter_b()    # 1

So functions can maintain state
With Multiple Functions

```javascript
function counter(start = 0)
    n = start
    return () -> n += 1, () -> n = start
end

(plus_a, reset_a) = counter(10)
(plus_b, reset_b) = counter()

plus_a()     # 11
plus_a()     # 12
reset_a()    # 10
plus_b()     # 1
```
General Command

type Command
   execute::Function
   undo::Function
end

function execute(command::Command)
   command.execute()
end

function undo(command::Command)
   command.undo()
end

function counter(start)
   n = start
   return Command((()-> n += 1, ()-> n -= 1))
end

count = counter(5)
execute(count)  # 6
undo(count)     # 5
Command Processor Pattern
Command Processor Pattern

Command Processor manages the command objects

The command processor:

Contains all command objects

Schedules the execution of commands

May store the commands for later unto

May log the sequence of commands for testing purposes

Uses singleton to insure only one instance
Dynamics
Benefits

Flexibility in the way requests are activated

Different user interface elements can generate the same kind of command object

Allows the user to configure commands performed by a user interface element

Flexibility in the number and functionality of requests

Adding new commands and providing for a macro language comes easy

Programming execution-related services

Commands can be stored for later replay
Commands can be logged
Commands can be rolled back

Testability at application level

Concurrency

Allows for the execution of commands in separate threads
Liabilities

Efficiency loss

Potential for an excessive number of command classes

Try reducing the number of command classes by:

  Grouping commands around abstractions
  Unifying simple commands classes by passing the receiver object as a parameter

Complexity

  How do commands get additional parameters they need?
Singleton
Warning

Simplest pattern
  But has subtle issues particularly in Java

Most controversial pattern
Intent

Ensure a class only has one instance

Provide global point of access to single instance
public class Counter {
    private int count = 0;
    private static Counter instance;
    private Counter() { }

    public static Counter instance() {
        if (instance == null)
            instance = new Counter();
        return instance;
    }

    public int increase() { return ++count; }
}
Some Uses

Java Security Manager

Logging a Server

Null Object
Globals are Evil
Why Singletons Are Controversial(Evil)

Singletons provide global access point for some service

Hidden dependencies

Is there a different design that does not need singletons

Pass a reference
Why Singletons Are Controversial (Evil)

Singletons allow you to limit creation of objects of a class

Should that be the responsibility of the class?

Class should do one thing

Use factory or builder to limit the creation
Why Singletons Are Controversial(Evil)

Singletons tightly couple you to the exact type of the singleton object

No polymorphism

Hard to subclass
Why Singletons Are Controversial(Evil)

Singletons carry state with them that last as long as the program lasts

Persistent state makes testing hard and error prone
Why Singletons Are Controversial(Evil)

A Singleton today is a multiple tomorrow

Singleton pattern makes it hard to change to allow multiple objects
Why Singletons Are Controversial(Evil)

In Java Singletons are a lie

More on this later
Singleton Implementation
public class Counter {
    private static int count = 0;

    public static int increase() {return ++count;}
}

Why Not Use This?
public class Counter {
    private int count = 0;
    private Counter() {
    }

    public static Counter instance = new Counter();

    public int increase() { return ++count; }
}

Very subtle the error here
Two Useful Features

Lazy
  Only created when needed

Thread safe
Recommended Implementation

```java
public class Counter {
    private int count = 0;
    private Counter() { }

    private static class SingletonHolder {
        private final static Counter INSTANCE = new Counter();
    }

    public static Counter instance() {
        return SingletonHolder.INSTANCE;
    }

    public int increase() { return ++count; }
}
```

public class Counter {
    private int count = 0;
    protected Counter() { }

    private final static Counter INSTANCE = new Counter();

    public static Counter instance() {
        return INSTANCE;
    }

    public int increase() { return ++count; }
}
Lazy, Thread safe with Overhead

```java
public class Counter {
    private int count = 0;
    private static Counter instance;
    private Counter() { }

    public static synchronized Counter instance() {
        if (instance == null)
            instance = new Counter();
        return instance();
    }

    public int increase() { return ++count; }
}
```
Java Templates & Singleton

Does not compile

```java
public class TemplateSingleton<Type> {
    Type foo;

    public static TemplateSingleton<Type> instance = new TemplateSingleton<Type>();
}
```
When is a Singleton not a Singleton?
When Java Garbage Collects Classes

Singleton class can be garbage collected
Singleton loses any value it had

Solution

Turn off garbage collection of classes (-Xnoclassgc)

Make sure there is always a reference to the class-instance
When Multiple Java Class Loaders are Used

When loaded by two different class loaders there will be two versions of the class

Some servlet engines use different class loader for each servlet

Using custom class loaders can cause this
Purposely Reloading a Java Class

Servlet engines can force a class to be reloaded
Serialize and Deserialize Singleton Object

Serialize the singleton
Deserialize the singleton
You now have two copies

One way to serialize a Java object is using ObjectOutputStream

Ruby Marshal.dump() will not marshal a singleton
Proxy
Proxy (Surrogate)

a person authorized to act on behalf of another
class Proxy {
    AbstractSubject realSubject;

    public Foo service(Bar x ) {
        return realSubject(x);
    }
}
Why do it?
Remote Proxy

Machine A

HelloClient
SayHello()
Hello
Server
Proxy

Machine B

HelloServer
SayHello()
Hello
Client
Proxy

String server = getHelloHostAddress( args);
Hello proxy = (Hello) Naming.lookup( server );
String message = proxy.sayHello();
System.out.println( message );
More General Proxy

class Proxy {
    AbstractSubject realSubject;

    public Foo service(Bar x) {
        some preprocessing
        result = realSubject(x);
        some postprocessing
    }
}
Virtual Proxy

Creates/accesses expensive objects on demand

O-R Mapping Layers
Java's Synchronized List

ArrayList notSafe = new ArrayList();
List threadSafe = Collections.synchronizedList(notSafe);

static class SynchronizedList {
    List list;
    public Object get(int index) {
        synchronized(mutex) {return list.get(index);} 
    }
}
Java's Unmodifiable List

ArrayList notSafe = new ArrayList();
List noChange = Collections.unmodifiableList(notSafe);

static class UnmodifiableList {
    List list;
    public Object get(int index) { return list.get(index);}

    public Object set(int index, Object element) {
        throw new UnsupportedOperation();
    }
}
Proxy or Decorator?

ArrayList notSafe = new ArrayList();
List noChange = Collections.unmodifiableList(notSafe);
List threadSafe = Collections.synchronizedList(noChange);
Proxy verses Decorator

"Decorators can have similar implementations as proxies"

Proxy controls access to an object

Decorator adds one or more responsibilities to an object