CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2013
Doc 9 Assignment2, Java GUI, Memento, Command
Feb 26, 2013

Assignment 2 - Sample Solution

Source Code

https://bitbucket.org/rogerwhitney/cs635spring13assignment2/overview

Used Mercurial

So can see commits & Versions

Commit history | Feature branches

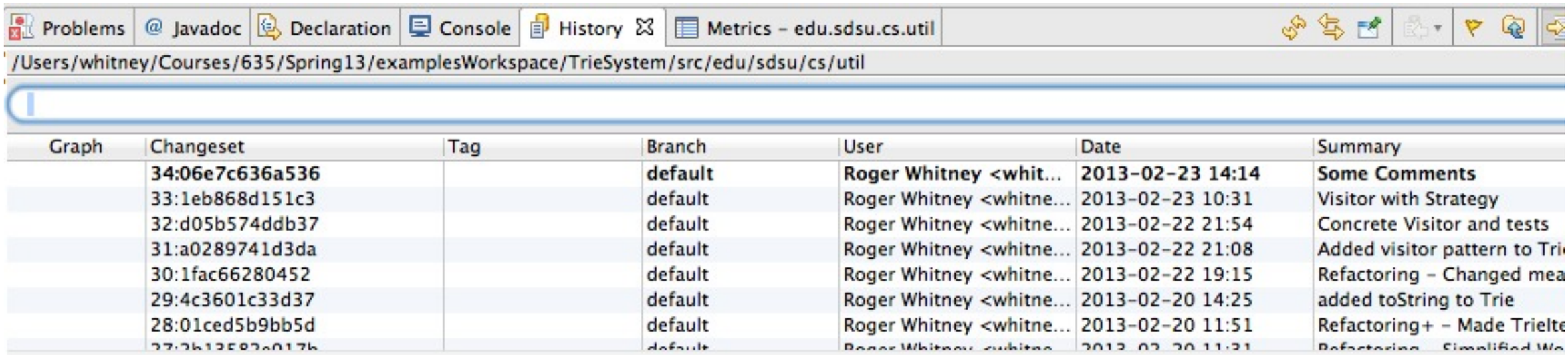
Showing 1–35 of 35

Find commits

Author	Commit	Message	Date
rogerwhitney	06e7c63	Some Comments	3 days ago
rogerwhitney	1eb868d	Visitor with Strategy	3 days ago
rogerwhitney	d05b574	Concrete Visitor and tests	4 days ago
rogerwhitney	a028974	Added visitor pattern to Trie and nodes, renamed WordNode to TrieWordNode	4 days ago
rogerwhitney	1fac662	Refactoring - Changed meaning of nextLevel in WordParameter	4 days ago
rogerwhitney	4c3601c	added toString to Trie	6 days ago
rogerwhitney	01ced5b	Refactoring+ - Made TrieIterator an inner class to make it easier to check for modifications of	6 days ago

Mercurial & Eclipse

https://code.google.com/a/eclipselabs.org/p/mercurialeclipse/



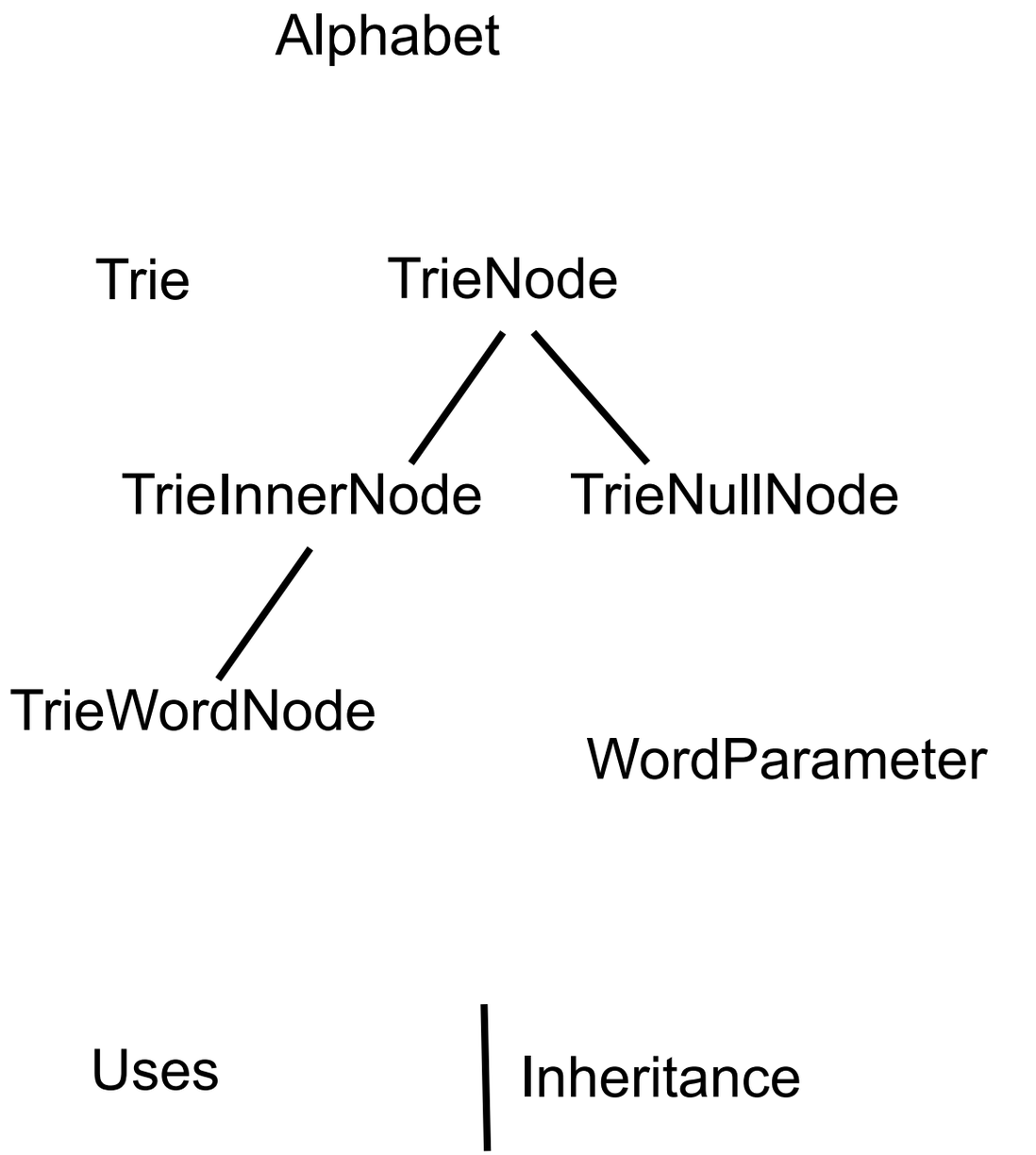
Some Metrics (Non test Classes)

Number of classes - 15

Total Lines of Code - 465

Metric	Total	Mean	Std. Dev.	Max
Lines of code/method	180	1.9	2.1	13
Number of Methods	96	6.4	3.8	17

Basic Structure



Alphabet

Characters in alphabet

Order of characters

Maps characters to index

first, last, next character

TrieNode

parent pointer

alphabet used by trie

character the node represents

Array of TrieNode for subtries

Initial each character points to same TrieNullNode

WordParameter

Used in recursive add and recursive contains method

Represents word and current location of word in add/contains

Adding Word

Trie add

```
public boolean add(String wordToAdd){
        if (wordToAdd == null) return false;
        if (wordToAdd.isEmpty()) return false;
        if (!characterSet.validCharacters(wordToAdd))
              throw new IllegalArgumentException(wordToAdd +
          " contains characters not in alphabet: " + characterSet);
        modificationCount++;
        return root.add(new WordParameter(wordToAdd.toLowerCase()));
}
```

InnerNode Add

```
public boolean add(WordParameter wordToAdd) {
    if (wordToAdd.atEnd()) {
        becomeWordNode();
        return true;
    }
    char whereToAdd = wordToAdd.nextChar();
    return getNode(whereToAdd).add(wordToAdd.nextLevel());
}
```

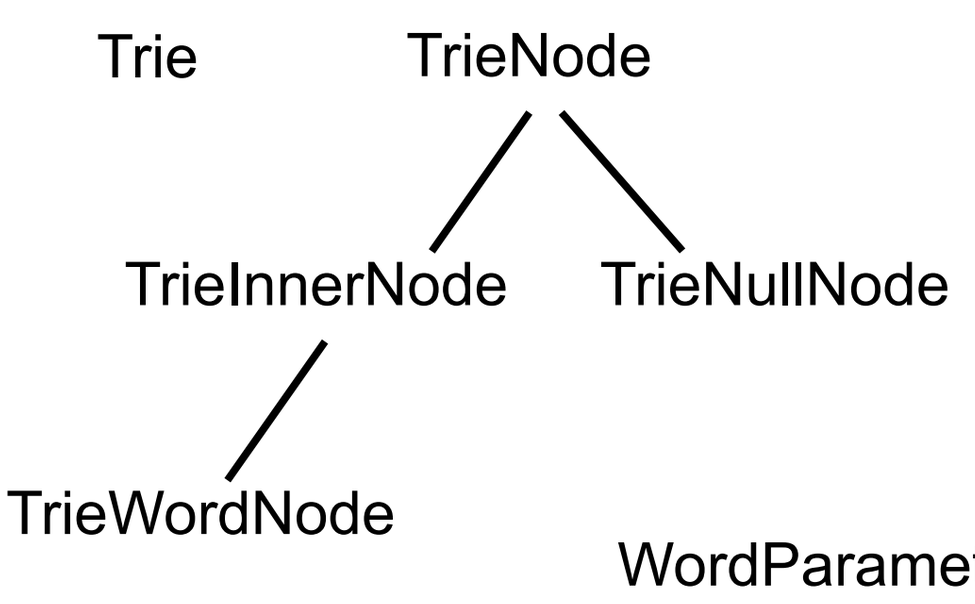
becomeWordNode in TrieInnerNode

```
/*
 * Replace current node with TrieWordNode object
 */
protected void becomeWordNode() {
      //Point all references to current node to myReplacement
      //WordNode constructor points all my subtrie parent references to
myReplacement

      TrieWordNode myReplacement =
    new TrieWordNode(characterSet, parent, character, subtries);
      parent.replaceNodeWith(myReplacement);
}
```

TrieNullNode - add

```
public  boolean  add(WordParameter wordToAdd) {
        TrieInnerNode newMe = this.becomeInnerNode(wordToAdd.currentChar());
        return newMe.add(wordToAdd);
}


/*
 * Replace NullNode in Trie with an InnerNode
 */
        private TrieInnerNode becomeInnerNode(char letter) {
                //change all references to self to point to myReplacement
                TrieInnerNode myReplacement = new
TrieInnerNode(characterSet,parent,letter );
                parent.replaceNodeWith(myReplacement);
                return myReplacement;
        }
```

Basic Iterator Structure



TrieIterator

TrieNodeIterator

IteratorPathMarker

Uses | Inheritance

IteratorPathMarker

Represents state of traversal in a Trie at a node

Knows

- Its node
- Last subtrie visited
- Next subtrie to visit
- If there is next subtrie to visit

Used by TrieNodeIterator

TrieNodeIterator

Returns nodes
For use inside Trie only
Simplifies TrieIterator

```
public class TrieNodeIterator implements Iterator<TrieNode> {
    //Class Invariant - always prefetch next node

    private Stack<IteratorPathMarker> path = new Stack<IteratorPathMarker>();
    private TrieNode next;
```

HasNext

```
public TrieNodeIterator(TrieNode root) {
        path.push(new IteratorPathMarker(root));
        next = root;
}

@Override
public boolean hasNext() {
        return next != null;
}
```

next

```
@Override
public TrieNode next() {
        if (next == null)
                throw new NoSuchElementException();
        TrieNode current = next;
        next = getNext();
        return current;
}
```

GetNext()

```
private TrieNode getNext() {
        if (path.empty())
                return null;
        IteratorPathMarker currentNode = path.pop();
        while (!path.empty() && (currentNode.finished())) {
                currentNode = path.pop();
        }
        if (currentNode.finished())
                return null;
        if (currentNode.hasNext())
                path.push(currentNode);
        TrieNode next = currentNode.next();
        path.push(new IteratorPathMarker(next));
        return next;
}
```

TrieIterator

```
private class TrieIterator implements Iterator<String> {
        //Class invariant = always prefetch next element to simplify hasNext logic

        private TrieNodeIterator nodeIterator;
        private TrieWordNode next;
        int initialModificationCount;

        public TrieIterator(TrieNode root) {
            nodeIterator = new TrieNodeIterator(root);
            next = getNextWordNode();
            initialModificationCount = modificationCount;
    }
```

next

```
public boolean hasNext() {
        if (initialModificationCount != modificationCount)
      throw new UnsupportedOperationException();
        return next != null;
    }

    @Override
    public String next() {
        if (initialModificationCount != modificationCount)
      throw new UnsupportedOperationException();
        if (next == null) throw new NoSuchElementException ();
        TrieNode current = next;
        next = getNextWordNode();
        return current.getWord();
    }
```

getNextWordNode

```
private TrieWordNode getNextWordNode() {
        if (!nodeIterator.hasNext()) return null;
        TrieNode nextNode = nodeIterator.next();
        while (nodeIterator.hasNext() && !nextNode.isWord()) {
                nextNode = nodeIterator.next();
        }
        if (!nextNode.isWord()) return null;
        return (TrieWordNode) nextNode;
}
```

Java GUI

Java UI & Patterns

Observer
Template Method
Composite
Bridge
Strategy

Parts of Java UI

Components

Basic UI things like

Buttons, Checkboxes, Choices, Lists, Menus, and Text Fields

Containers

Contain components and containers

Window

LayoutManagers

Position items in window

Graphics

Drawing on the screen

Events

AWT and Swing

AWT

- First Java UI
- Simple
- Frame, Button, etc

Swing

- Replacement for AWT
- Lot more features, but more complex
- JFrame, JButton, etc

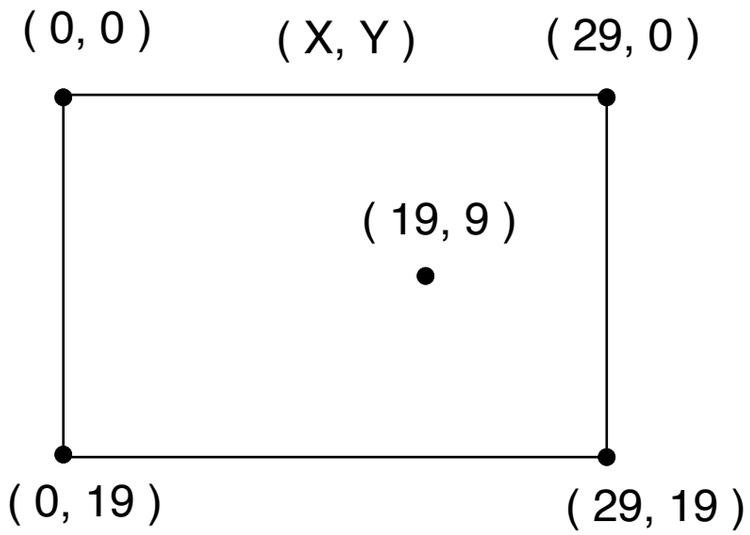
Topics to Master

Layouts

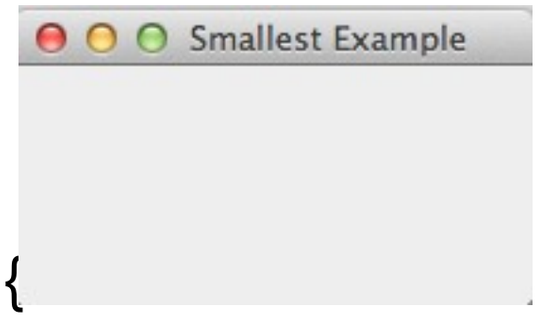
Getting GUI events

No main driver program

Coordinate System



Small Example



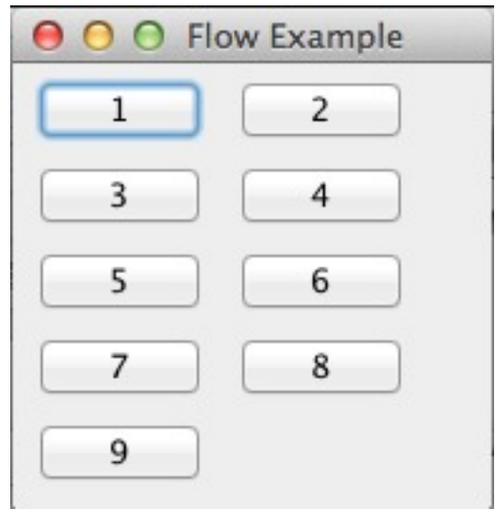

```
class  JavaUIExample  extends JFrame  {

      public JavaUIExample( int  widthInPixels, int heightInPixels ) {
            setTitle( "Smallest Example" );
            setSize( widthInPixels, heightInPixels );
      }

      public  static  void  main( String  args[] ) {
            JavaUIExample application = new  JavaUIExample(150, 100);
            application.setVisible(true);
      }
}
```


setTitle, setSize are public methods so can be called outside of the object

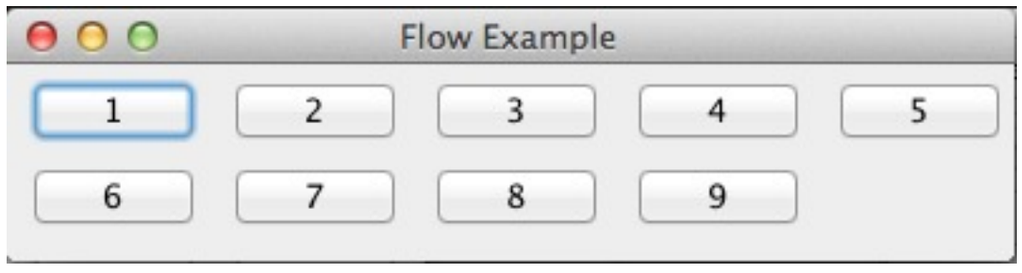
Hello World


```
class  HelloLabel  extends  Frame
      {
      public HelloLabel( int left, int top, int width, int height, String title )
            {
            super( title );
            setSize( width, height );
            setLocation( left, top);
            setLayout(new BorderLayout());

            Label hello = new Label( "Hello World", Label.CENTER);
            add( hello, BorderLayout.CENTER );

            show();
            }
}

HelloLabel  mainWindow = new  HelloLabel( 20, 40, 150, 80, "Hi Dad");
```

Layouts - Strategy Pattern




```
class  JavaUIExample  extends JFrame  {

      public JavaUIExample( int  width, int height ) {
            setTitle( "Flow Example" );
            setSize( width, height );
            setLayout( new FlowLayout( FlowLayout.LEFT) );

            for ( int label = 1; label < 10; label++ )
                  add( new Button( String.valueOf( label ) ) );
            setVisible(true);
      }

      public  static  void  main( String  args[] ) {
            new  JavaUIExample( 400, 100 );
            new  JavaUIExample( 190, 200 );
      }
}
```



Layouts - Strategy Pattern

Layouts - algorithm that arranges items on the screen

Layout Tutorial
http://docs.oracle.com/javase/tutorial/uiswing/layout/layoutlist.html

Some common & easy to use layouts

BoxLayout
- A row or a column

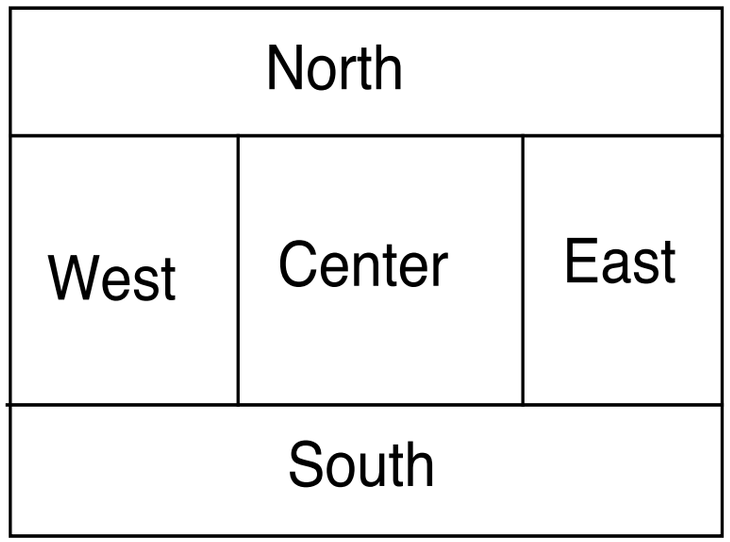
BorderLayout

FlowLayout
- Displays components left to right, top to bottom

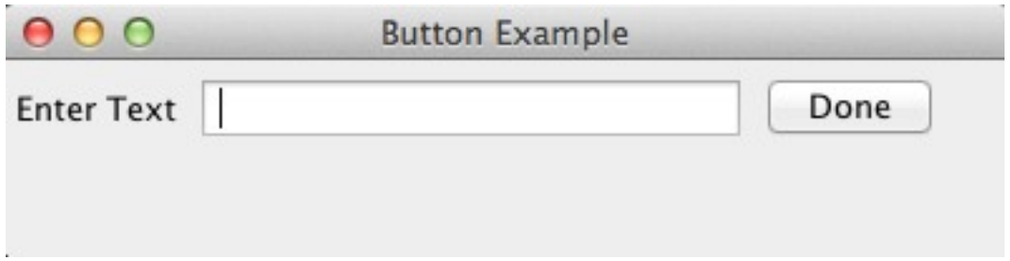
GridLayout
- Displays components in a grid of rows and columns

BorderLayout

Divides its components into regions



Buttons & Text




```
class  JavaUIExample  extends JFrame  {

    private final TextField textField = new TextField("",20);
    private final Button button = new Button("Done");

    public JavaUIExample( int  width, int height ) {
        setTitle( "Button Example" );
        setSize( width, height );
        setLayout( new FlowLayout( FlowLayout.LEFT) );
        Label text = new Label("Enter Text");
        add(text);
        add(textField);
        add(button);
        setVisible(true);
    }

    public  static  void  main( String  args[] ) {
        new  JavaUIExample( 400, 100 );
    }
}
```

Button action - Observer

```
class  JavaUIExample  extends JFrame implements ActionListener  {

    private final TextField textField = new TextField("",20);
    private final Button button = new Button("Done");

    public JavaUIExample( int  width, int height ) {
        setTitle( "Button Example" );
        setSize( width, height );
        setLayout( new FlowLayout( FlowLayout.LEFT) );
        Label text = new Label("Enter Text");
        add(text);
        add(textField);
        add(button);
        button.addActionListener(this);
        setVisible(true);
    }
```

Button action - Observer

```
@Override
public void actionPerformed(ActionEvent e) {
        String textEntered = textField.getText();
        System.out.println("You entered: " + textEntered);
        textField.setText("This is the new text");
}
```

```
public  static  void  main( String  args[] ) {
        new  JavaUIExample( 400, 100 );
}
```

Major Issue

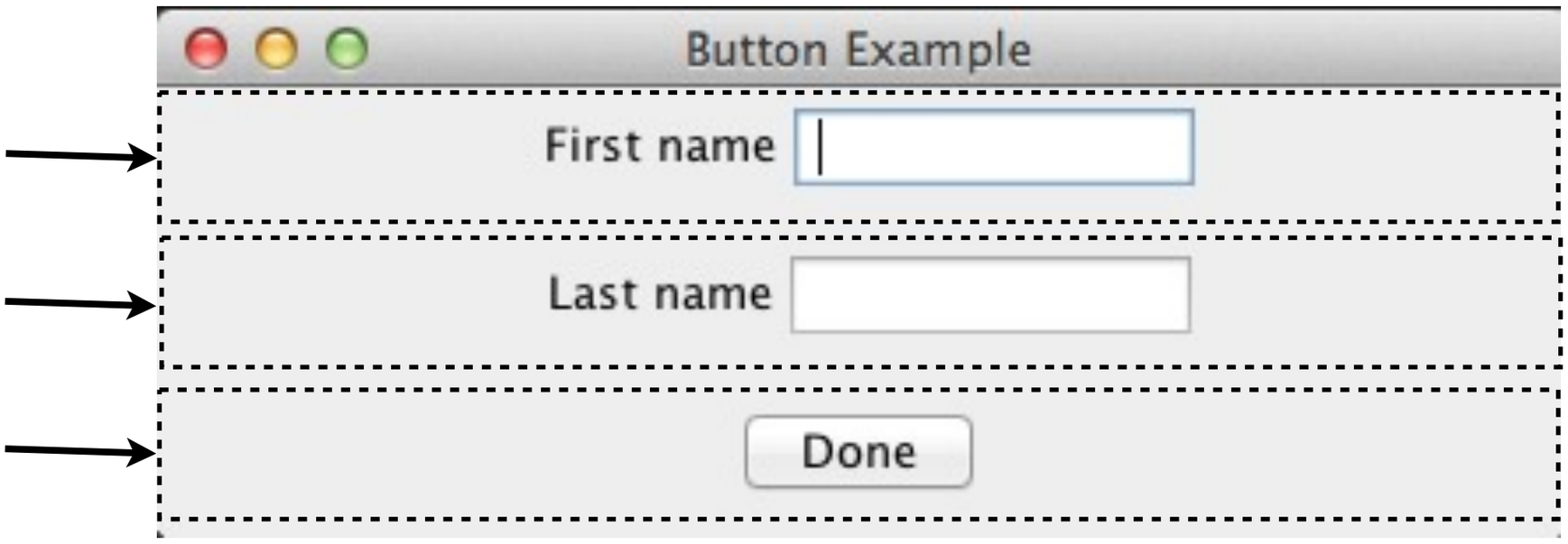
No main driver for program

GUI callbacks drive program

Panels - Composite Pattern

A container for GUI widgets and containers

Using 3 panels to help organize window elements

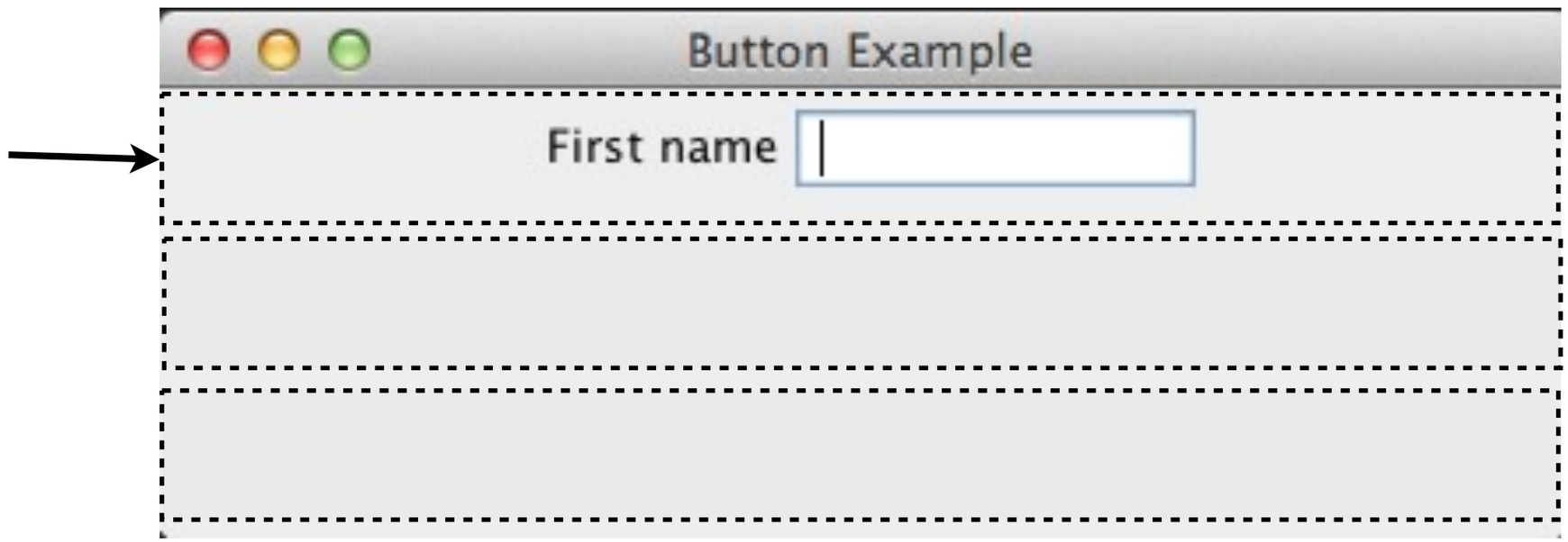


Example Using Panels

```
class  JavaUIExample  extends JFrame  {

      TextField firstName;
      TextField lastName;
      Button done = new Button( "Done" );

      public JavaUIExample( int  width, int height ) {
            setTitle( "Button Example" );
            setSize( width, height );
            setLayout( new GridLayout(3,1) );       3 rows, 1 column
```

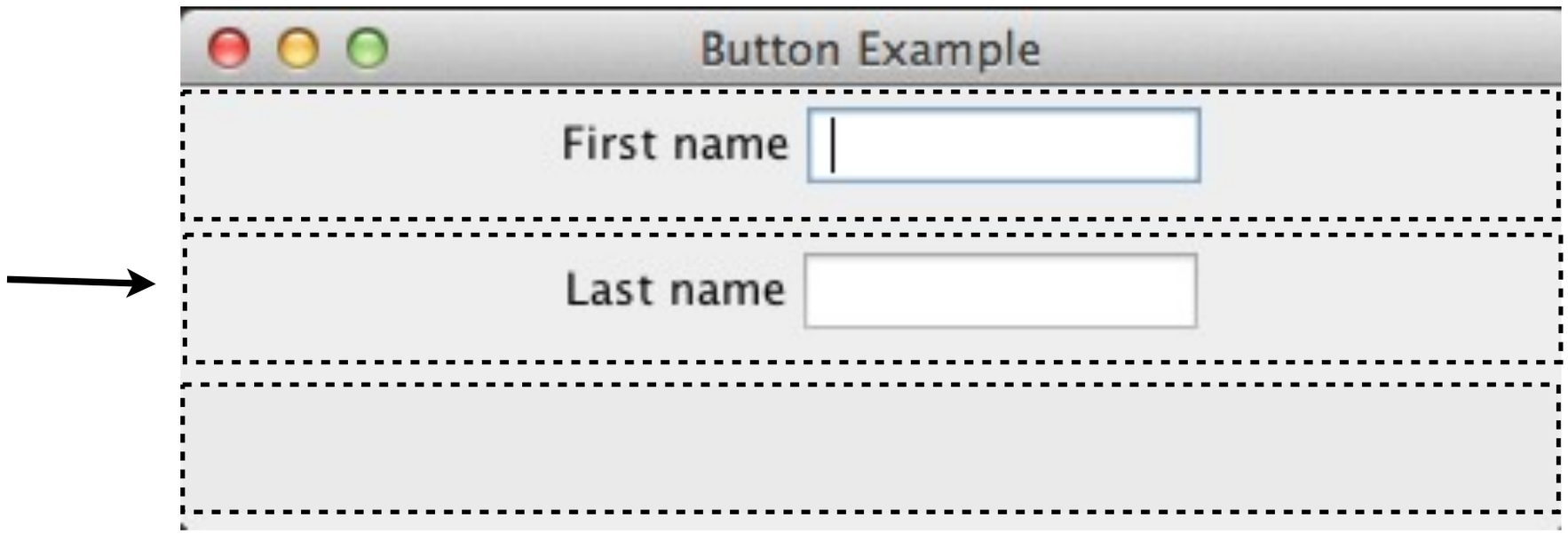
Example Using Panels

```
    int numberOfColumns = 10;


        Panel first = new Panel( new FlowLayout() );
    firstName = new TextField(  numberOfColumns );
        first.add( new Label( "First name", Label.RIGHT ));
        first.add( firstName );
add(first);
```



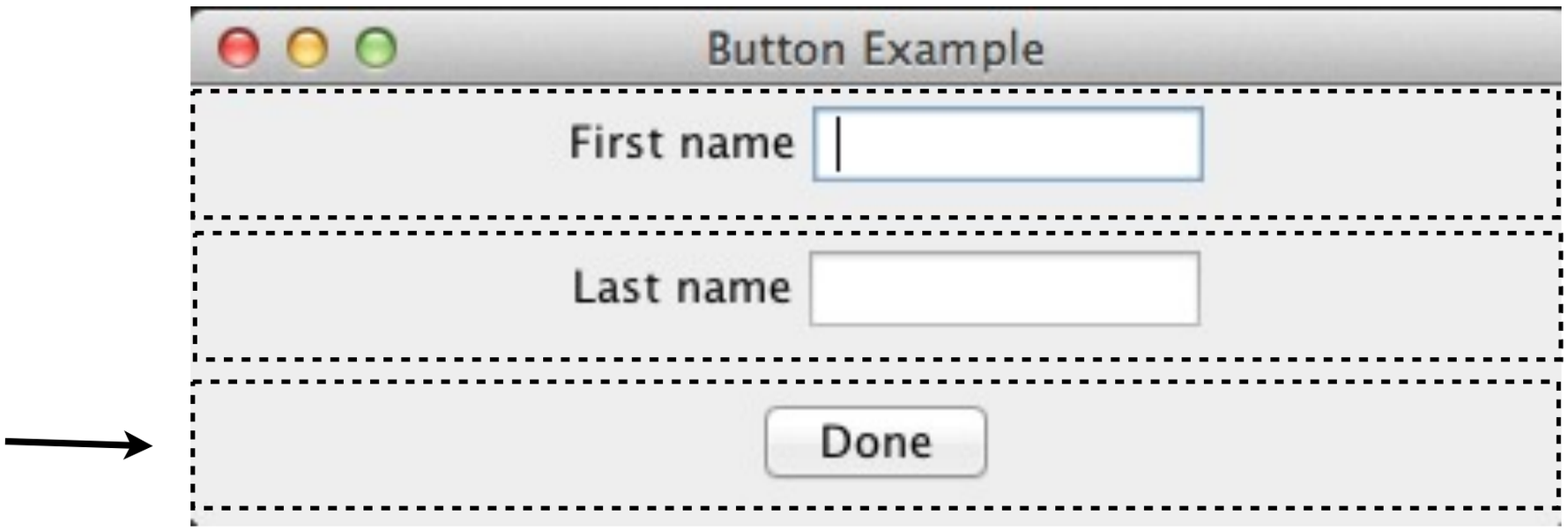
Example Using Panels

```
lastName = new TextField(  numberOfColumns );
    Panel last = new Panel( new FlowLayout() );
    last.add( new Label( "Last name", Label.RIGHT ) );
    last.add( lastName );
    add(last);
```



Example Using Panels

```
Panel buttons =
        new Panel( new FlowLayout( FlowLayout.CENTER));
    buttons.add( done );

    add(buttons);
    setVisible(true);
```



Complete Example

```
class  JavaUIExample  extends JFrame  {

        TextField firstName;
        TextField lastName;
        Button done = new Button( "Done" );

        public JavaUIExample( int  width, int height ) {
                setTitle( "Button Example" );
                setSize( width, height );
                setLayout( new GridLayout(3,1) );
                int numberOfColumns = 10;
                firstName = new TextField(  numberOfColumns );
                Panel first = new Panel( new FlowLayout() );
                first.add( new Label( "First name", Label.RIGHT ));
                first.add( firstName );

                lastName = new TextField(  numberOfColumns );
                Panel last = new Panel( new FlowLayout() );
                last.add( new Label( "Last name", Label.RIGHT ) );
                last.add( lastName );

                Panel buttons =
                        new Panel( new FlowLayout( FlowLayout.CENTER));
                buttons.add( done );
                add(first);
                add(last);
                add(buttons);
                setVisible(true);
        }
```

```
new  JavaUIExample( 400, 150 );
```

Tutorial

http://docs.oracle.com/javase/tutorial/uiswing/index.html

Java GUI Builder

WindowBuilder Pro

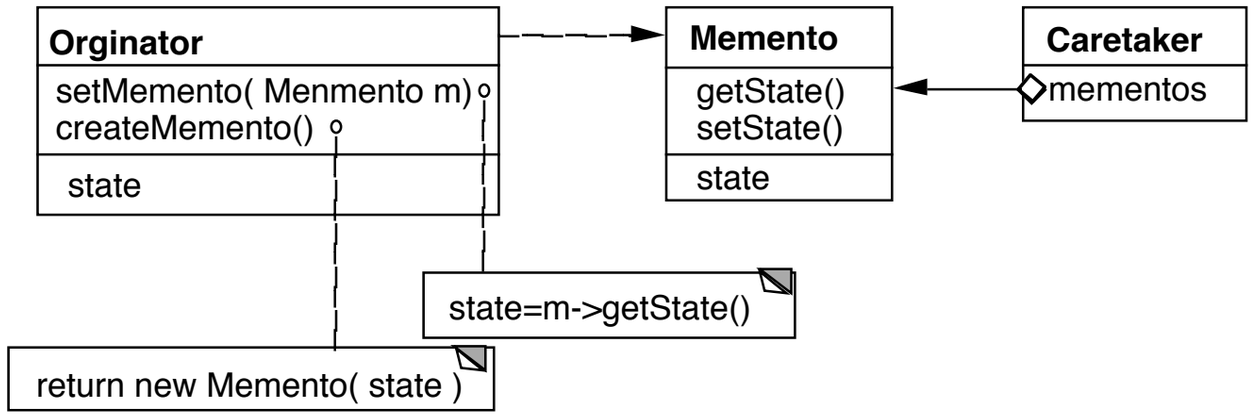
https://developers.google.com/java-dev-tools/download-wbpro

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

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;
        }
}
```

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

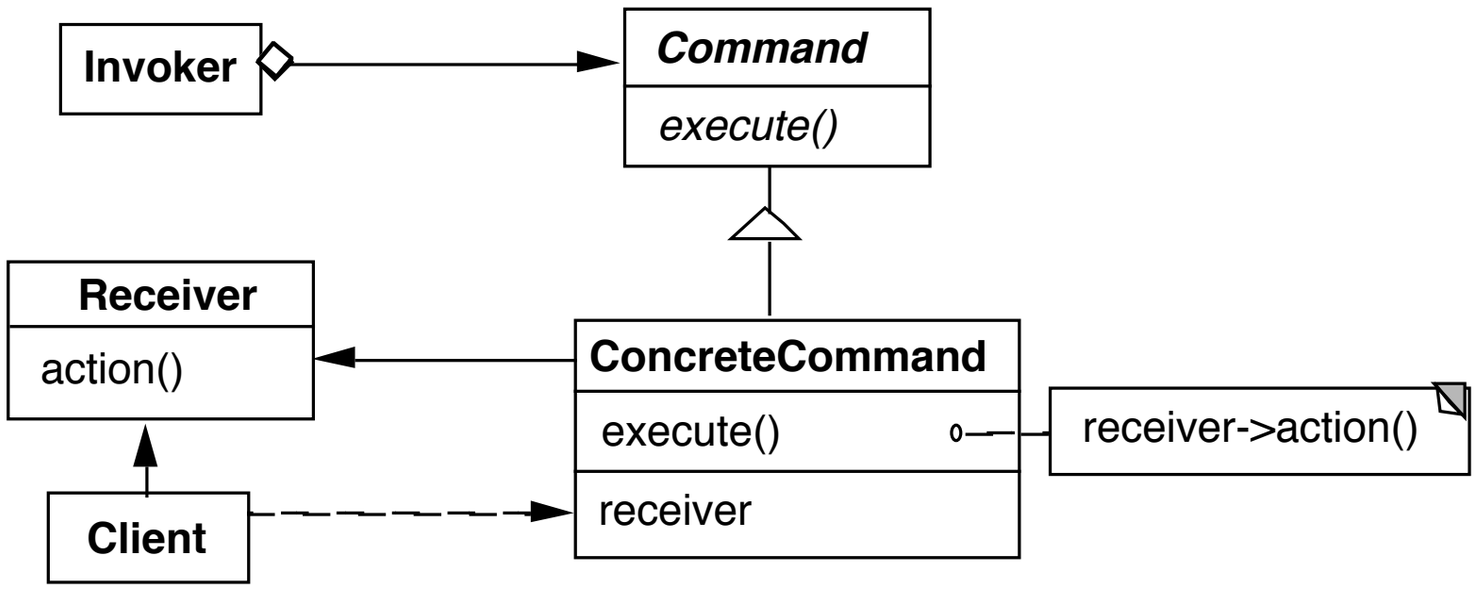
If so discard the copy & raise an exception

Else perform the operations on the Originator

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

When to Use the Command Pattern

Need action as a parameter (replaces callback functions)

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

Refactoring: Replace Conditional Dispatcher with Command

```
public class SDSUChatServer {
    public void processClientRequest(String request) {
        blah
        if (command.equals("quit"))
            quit();
        else if  (command.equals("register"))
            registerNewUser(commandData);
        else if  (command.equals("login"))
            login(commandData);
        else if  (command.equals("nickname"))
            checkNickname(commandData);
        blah
    }
```

```
action = actions.get(command);
action.execute(commandData);
```


Refactoring to Patterns, Kerievsky, pp 191-201

Sample Command

```
public class RegisterCommand extends Command {
    private SDSUChatServer target;

    public RegisterCommand(SDSUChatServer aServer) {
        target = aServer;
    }
                                                bad example do not use
    public void execute(String commandData) {
        target.registerNewUser(commandData);
    }
}
```

The actions table

```
public class SDSUChatServer {
    private HashMap<String, Command> actions;

    private populateActions() {
        actions = new HashMap<String, Command>();
        actions.put("quit", new QuitCommand(this));
        actions.put("register", new RegisterCommand(this));
        actions.put("login", new LoginCommand(this));
        actions.put("nickname", new NicknameCommand(this));
    }
```

When to do this?

Need runtime flexibility

Conditional Dispatcher is bloated

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

Java Example of Pluggable Command

```
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

```
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
```

Command Processor Pattern

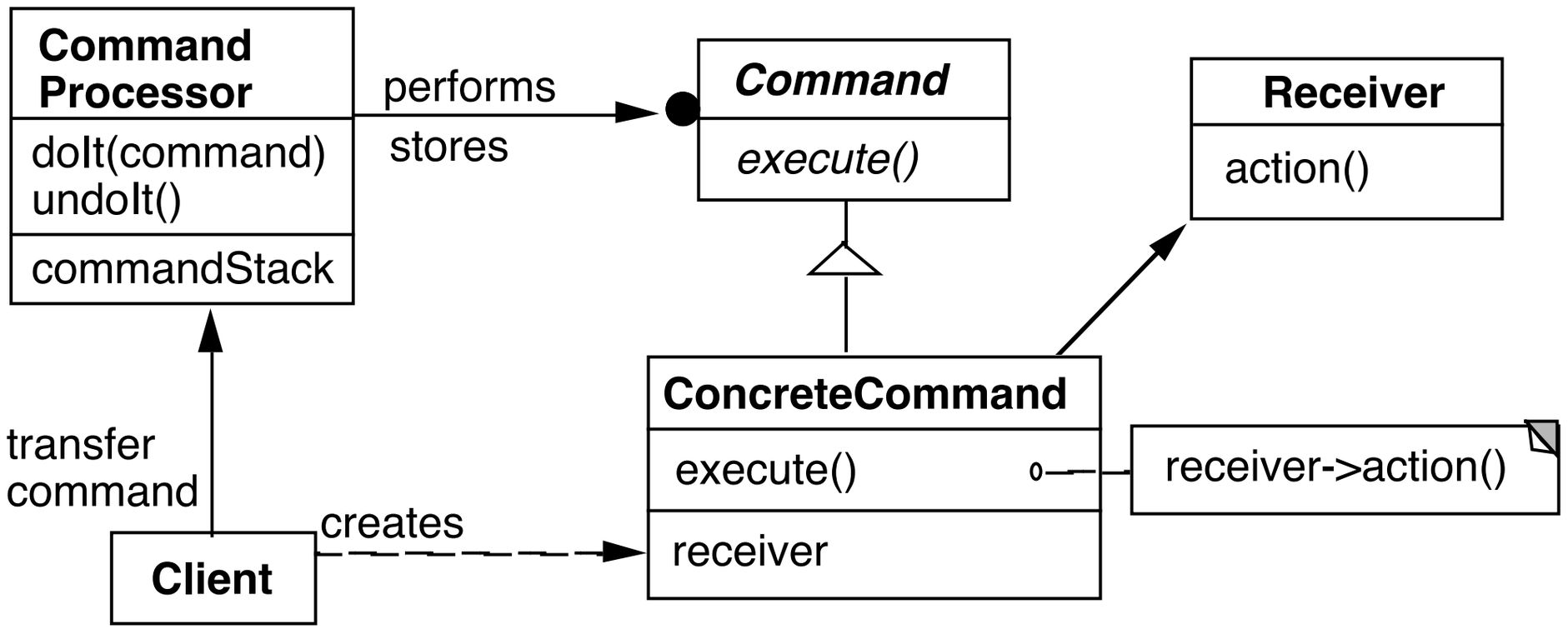
Command Processor Pattern

Command Processor manages the command objects

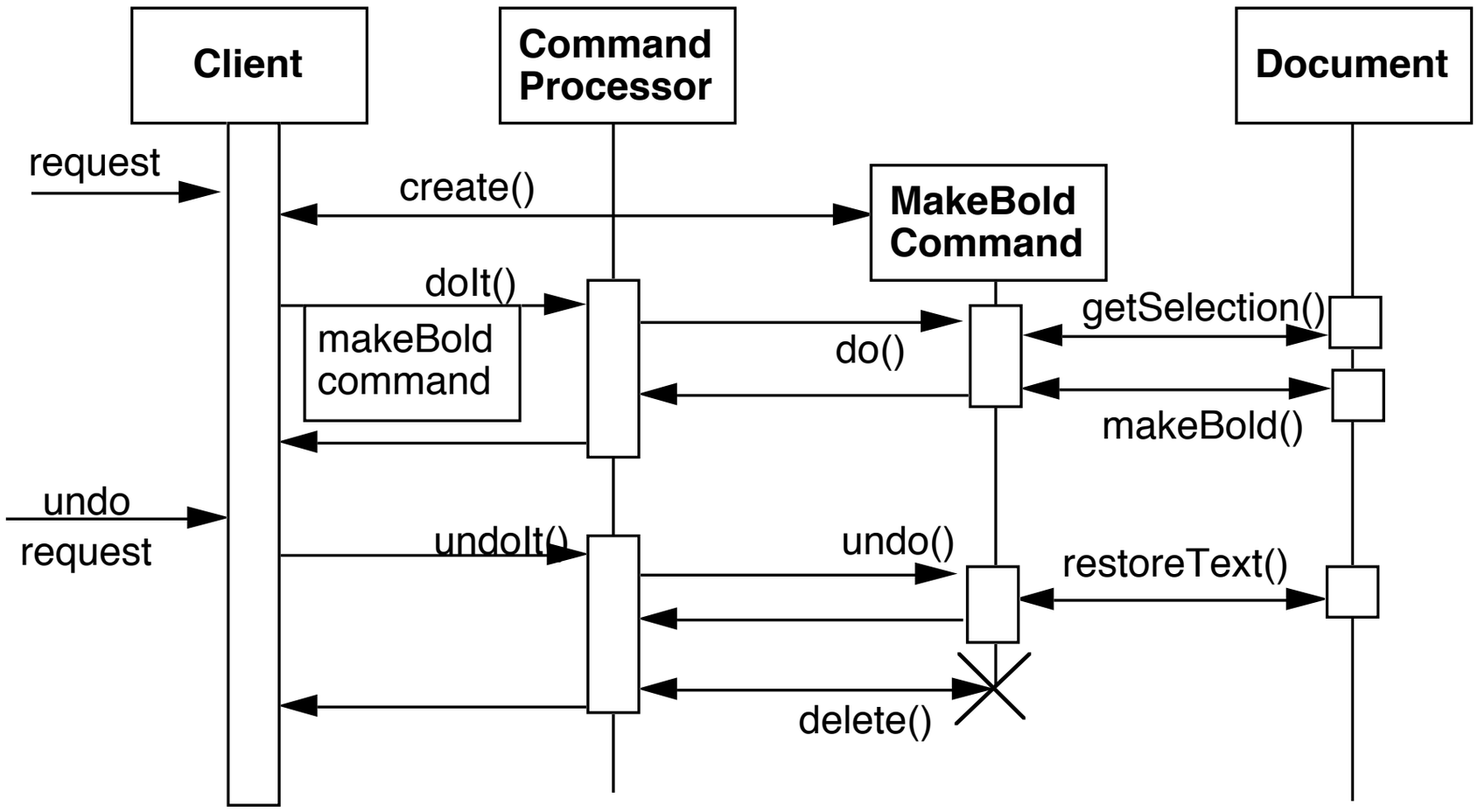
The command processor:

- 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

Structure


Command Processor
doIt(command)
undoIt()
commandStack
performs
stores
Command
execute()
Receiver
action()
transfer command
ConcreteCommand
execute()
receiver
receiver->action()
creates
Client

Dynamics


Client
Command Processor
Document
request
create()
MakeBold Command
doIt()
makeBold command
do()
getSelection()
makeBold()
undo request
undoIt()
undo()
restoreText()
delete()

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?