Command Pattern

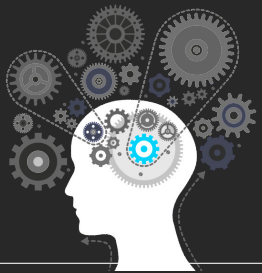
Dr. Isaac Griffith Idaho State University

Outcomes

Idaho State University | Computer Science

After today's lecture you will be able to:

- Implement Undo within an MVC Framework
- Use the Command Pattern to better operationalize commands within a UI



Undo Operation

CS 2263

Undo Operation

- Key issues in implementing Undo
  - Single-level undo vs. Multiple-level undo
    - **Single-level**: we undo a single operation. Relatively easy via cloning the model
  - Undo and redo are unlike the other operations
    - If treated like other operations, consecutive undos would cancel each other out
    - Thus they should be treated as special **meta-operations**
  - No all things are undoable
    - So operations are irreversible (i.e., print file)
    - Others are not worth the effort (i.e., save a file)
  - Blocking further undo/redo operations
    - To maintain system safety and reduce frivolous undo requests, we should block undo/redo during the execution of an operation.
    - Additionally, some operations will render undo or redo impossible.
  - Solution should be efficient

Simple Approach to Undo

1. Create a stack for storing the history of the operations
2. For each operation
   - define a data class that will store the information necessary to undo the operation
3. Implement code so that whenever any operation is carried out
   - the relevant information is packed into the associated data object
   - the object is pushed onto the stack.
4. Implement an `undo` method in the controller that:
   - simply pops the stack
   - decodes the popped data object
   - invokes the appropriate method to extract the information
   - performs the task of undoing the operation

Simple Approach to Undo

- An initial approach could start with creating a `StackObject` which stores objects using a `String` identifier

```
public class StackObject {
  private String name;
  private Object object;

  public StackObject(String string, Object object) {
    name = string;
    this.object = object;
  }

  public String getName() {
    return name;
  }

  public Object getObject() {
    return object;
  }
}
```

Simple Approach to Undo

- Each command then stores the info needed to undo it.
- For a line it would be the following:

```
public class LineObject {
  private Line line;

  public Line getLine() {
    return line;
  }

  public LineObject(Line line) {
    this.line = line;
  }
}
```

Simple Approach to Undo

- When the operation to add a line is complete the appropriate `StackObject` is created and pushed onto the stack.

```
public class Controller {
  private Stack history;

  public void makeLine(Point point1, Point point2) {
    Line line = new Line(point1, point2);
    model.addItem(line);
    history.push(new StackObject("line", new LineObject(line)));
  }

  // other fields and methods
}
```

Simple Approach to Undo

- Decoding is simply popping the stack and reading the `String`

```
public void undo() {
  StackObject undoObject = history.pop();
  String name = undoObject.getName();
  Object obj = undoObject.getObject();
  if (name.equals("line")) {
    undoLine((LineObject) obj);
  } else if (name.equals("delete")) {
    undoDelete((DeleteObject) obj);
  } else if (name.equals("select")) {
    undoSelect((SelectObject) obj);
  }
  // one else for each command
}
```

Simple Approach to Undo

- Finally, undo becomes a matter of retrieving the reference and removing it from the model

```
public class Controller {
  public void undoLine(LineObject object) {
    Line line = object.getLine();
    model.removeItem(line);
  }
}
```

Simple Approach Drawbacks

By now the drawbacks of this approach should be obvious:

1. the long conditional statement in the undo method of the controller
   - We can deal with this by subclassing and using the refactoring **Replace Conditional Logic with Polymorphism**
2. The need to rewrite the controller whenever we make changes such as adding or modifying the implementation of an operation

Refactoring

Idaho State University | Computer Science

- If we review the `undo` method in the controller, we note that it mainly deals with data from `StackObject`
  - So we should probably move it there first

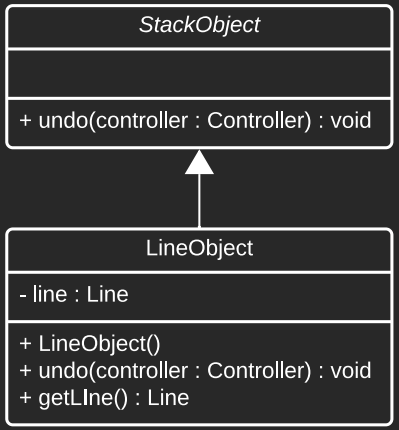
```
public class Controller {
  private Stack history;

  public void undo() {
    StackObject undoObject = history.pop();
    undoObject.undo(this);
  }
  // other fields and methods
}
```

```
public class StackObject {
  public void undo(Controller controller) {
    String name = getName();
    Object object = getObject();
    if (name.equals("line")) {
      controller.undoLine((LineObject) object);
    } else if (name.equals("delete")) {
      controller.undoDelete((DeleteObject) object);
    } else if (name.equals("select")) {
      controller.undoSelect((SelectObject) object);
    }
  }
  // other fields and methods
}
```

Refactoring

- We then can cleanup the undo operation by making it abstract and subclassing `StackObject`



This also simplifies controller operations such as `makeLine`

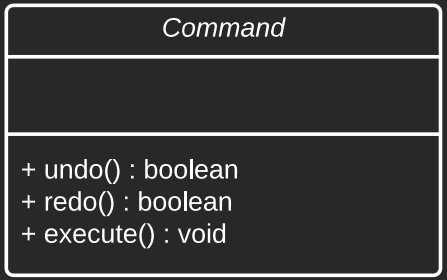
```
public void makeLine(Point point1, Point point2) {
  Line line = new Line(point1, point2);
  model.addItem(line);
  hisotry.push(new LineObject(line));
}
```

Command Pattern

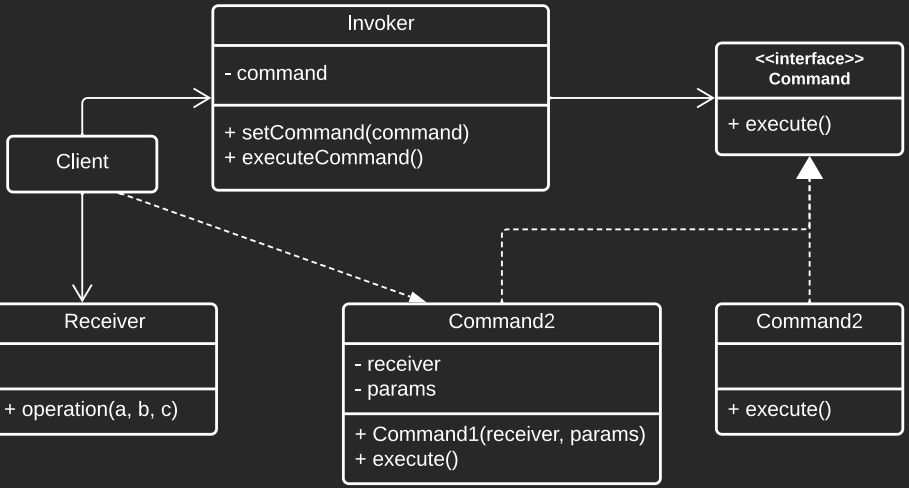
CS 2263

Command Pattern

- **Pattern Intent**:
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Provides a template by which our objects can be both a repository of data and include the operations which need access to that data.
  - This template is provided by the abstract `Command` class which contains the following methods
    - `execute`
    - `undo` – default is to return false
    - `redo` – default is to return false

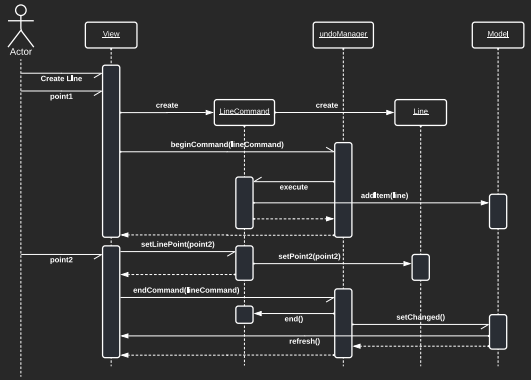


Command Pattern Structure



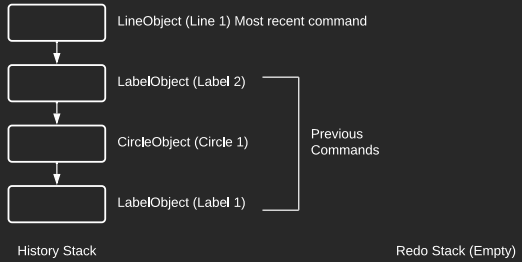
Adding a Line

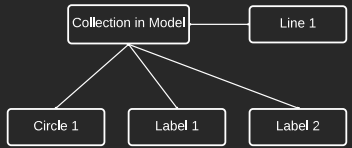
- Central idea: employ two stacks (stored in the `UndoManager`)
  - one for storing commands that can be undone (history stack)
  - one for storing commands that may be redone (redo stack)



Undoing an Operation

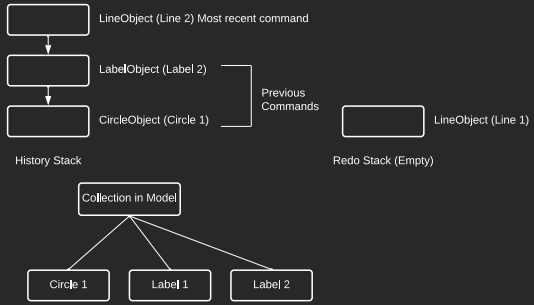
- General Idea: Undo
  - when an undo operation is requested
  - if the top of the undo stack is a command that can be undone
    - the command is undone and transferred to the redo stack
- General Idea: Redo
  - When a redo operation is requested
  - if the top of the redo stack is a command that can be re-executed
    - the command is re-executed and transferred to the top of the history stack
- Note: not all commands can be undone





Undoing an Operation

- General Idea: Undo
  - when an undo operation is requested
  - if the top of the undo stack is a command that can be undone
    - the command is undone and transferred to the redo stack
- General Idea: Redo
  - When a redo operation is requested
  - if the top of the redo stack is a command that can be re-executed
    - the command is re-executed and transferred to the top of the history stack
- Note: not all commands can be undone



Undoing an Operation

- We now reach the question of what to do with commands that we are in the middle of completing?
  - We could prevent users from aborting commands in the middle, by disabling other command buttons until the current command is finished.
    - though this opens up difficulties with managing data consistency
  - We could add an additional method in both the undo manager and the command class

Undoing an Operation

- We will pursue the second choice
  - keeping the current command away from the history stack until the command is completed
  - add a method which checks if the command is complete called `end`
    - which then fills in necessary missing data

```
public boolean end() {
  if item is incomplete
    attempt to complete using data already received
    if cannot be completed
       return false
    end if
  end if
  return true
}
```

- Only when the command is complete does the `UndoManager` add it to the history stack

Subclassing Command

- Subclasses of `Command` store all the data needed for undo and redo
- The implementation specifics of `Command` are as follows:

```
public void execute() {
  model.addItem(line);
}

public boolean undo() {
  model.removeItem(line);
  return true;
}

public boolean redo() {
  execute();
  return true;
}
```

```
public boolean end() {
  if (line.getPoint1() == null) {
    undo();
    return false;
  }
  if (line.getPoint2() == null) {
    line.setPoint2(line.getPoint1());
  }
  return true;
}
```

UndoManager

Idaho State University | Computer Science

- Here we declare two stacks for tracking the undo and redo operations (`history` and `redoStack`).
- We store the current command in `currentCommand`

```
public class UndoManager {
  private Stack history;
  private Stack redoStack;
  private Command currentCommand;
}
```

UndoManager

- If the previous command was not terminated properly, we need to arrange for `currentCommand` to be `null` when a new command is issued.
  - we do this with `beginCommand` method

```
public void beginCommand(Command command) {
  if (currentCommand != null) {
    if (currentCommand.end()) {
      history.push(currentCommand);
    }
  }
  currentCommand = command;
  redoStack.clear();
  command.execute();
}
```

UndoManager

Idaho State University | Computer Science

- `undo` and `redo` are fairly straight-forward

```
public void undo() {
  if (!(history.empty())) {
    Command command = (Command) (history.peek());
    if (command.undo()) {
      history.pop();
      redoStack.push(command);
    }
  }
}
```

```
public void redo() {
  if (!(redoStack.empty())) {
    Command command = (Command)(redoStack.peek());
    if (command.redo()) {
      redoStack.pop();
      history.push(command);
    }
  }
}
```

UndoManager

- Once complete the view calls `endCommand` in the `UndoManager`
  - this pushes the `currentCommand` onto the history stack and sets `currentCommand` to `null`

```
public void endCommand(Command command) {
  command.end();
  history.push(command);
  currentCommand = null;
  model.updateView();
}
```

For Next Time

- Review Chapter 11.7
- Review this lecture
- Read Chapter 11.8 - 11.10
- Watch Lecture 33

Are there any questions?