# Imperative Programming 3

## Undoing

Peter Jeavons

Trinity Term 2019

# Agenda

- The need for undo
- Storing information to allow undo
- The machinery of abstract classes
- Saving state: the Memento pattern
- Adding behavior: the Decorator pattern
- The machinery of inner classes

# Undo

- We want the editor to provide the facility to **undo** the changes made to the text
- To do this we need to store the sequence of changes:
    1. insert 'a' at 3
    2. delete character at 4
    3. insert 'c' at 17
- The undo command will remove the most recent change from the list and undo its effect

# The list of changes

**Question:** What exactly do we need to store in the list of changes?

Reminder:

1. insert 'a' at 3
2. delete character at 4
3. insert 'c' at 17

**Question:** After an undo, what do we need to store to redo the changes?

# One kind of change…

- After a deletion we must record the position *and* the character that was deleted so that it can be re-inserted when we undo the command

```
class Deletion(pos: Int, deleted: Char)
{
   def undo() { insert(pos, deleted) }
   def redo() { deleteChar(pos) }
}
```

# Another kind of change…

- After inserting a sequence of characters we must record the position *and* the characters that were inserted so they can be re-inserted when we redo the command

```
class Insertion(pos: Int, text: Text.Immutable)
{
   def undo() { deleteRange(pos, text.length) }
   def redo() { insert(pos, text) }
}
```

# An abstract superclass

At least some methods are undefined, so this class is **incomplete**....

```
abstract class Change {

  /* Reset the subject to its previous state. */
  def undo() //abstract

  /* Reset the subject to the state after the change. */
  def redo() //abstract
}
```

- Every change recorded by the editor will be a concrete subclass of this abstract class Change

# Abstract classes

- An abstract class can contain methods which are not implemented – they are declared as abstract and have no body
- It may also contain instance variables and ordinary methods
- No *instances* of an abstract class can be created (i.e., we can't call `new` on it )
- But *subclasses* which implement all the abstract methods can have instances

# Abstract classes and traits

| Traits | Abstract Classes | Classes |
|---|---|---|
| *May have* instance variables | Can have instance variables | Can have instance variables |
| *May have* methods | Can have methods | Can have methods |
| Have abstract methods with no bodies | Have abstract methods with no bodies | Define all methods so can have **instances** |
| Cannot have instances but can be **extended** by subclasses (**with** others) | Cannot have instances but can be **extended** by subclasses | Can extend another class (including abstract class) or several traits, and can be extended by subclasses |
| Cannot have constructor arguments | Can have constructor arguments | Can have constructor arguments |

Exact rules for traits affect interoperability with Java.

# Abstract classes in Java

### Interfaces

Have no instance variables

Have method headers
but no bodies

Cannot have instances
but can be **implemented**
by classes

### Abstract Classes

Can have
instance variables

Can have methods

Have abstract methods
with no bodies

Cannot have instances
but can be **extended**
by subclasses

### Classes

Can have
instance variables

Usually have methods

Can have **instances**

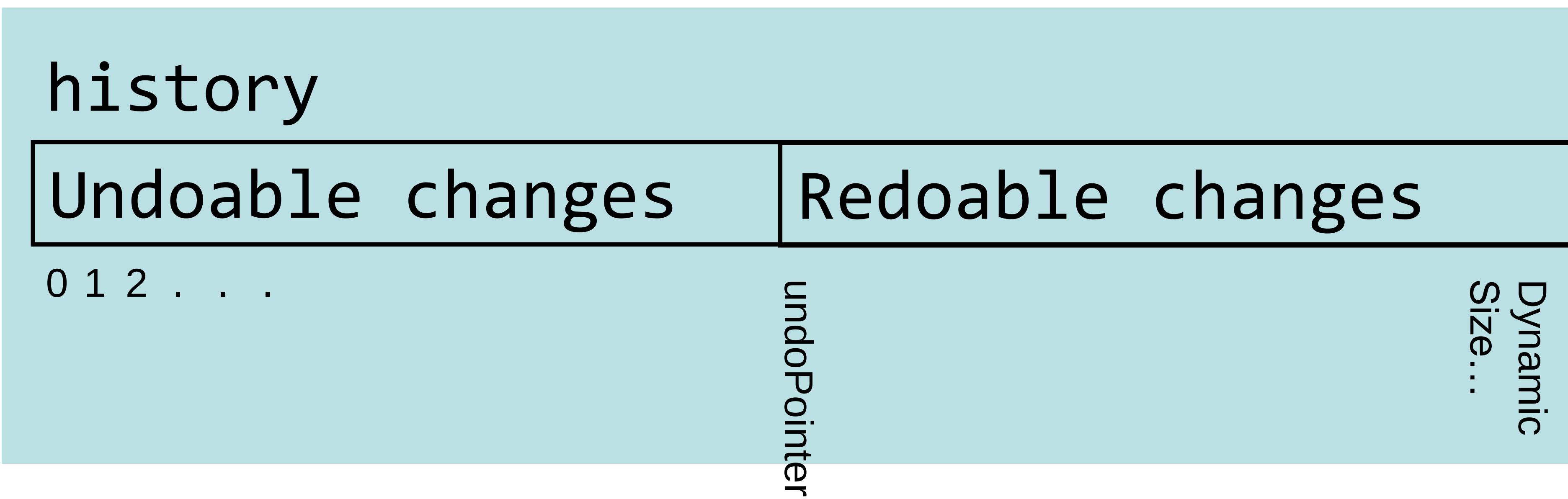
Can **extend** another class
(including abstract class)

Can **implement**
several interfaces

# Undo history

Each command that modifies text should

- Make a concrete `Change` object
- Add it to the undo history
- History is recorded as a stack (actually in a dynamically sized array)



# Undo history

```
trait UndoHistory {
   private val history = new ArrayBuffer[Change]
   private var undoPointer = 0
   def updatehistory(change: Change) {
         if (change != null) {
            history.reduceToSize(undoPointer)
            history.append(change); undoPointer += 1
         }
    }
    def undo(): Boolean = {
            if (undoPointer == 0) { beep(); return false }
            undoPointer -= 1
            val change = history(undoPointer)
            change.undo()
            return true
    }
    def redo(): Boolean = {...}
}
```

# Using UndoHistory in the Editor

- Use UndoHistory by mixing it into the Editor

```
class UndoableEditor extends Editor with UndoHistory {
    private var lastChange : Change = null

    override def insertCommand(ch: Char) {
        super.insertCommand(ch)
        lastChange = new Insertion(ed.point-1, ch)
    }

    override def obey(cmd: Editor.Command) {
            super.obey(cmd)
            if (lastChange != null) updateHistory(lastchange)
            lastChange = null
        }

We get undo and redo commands directly from UndoHistory
}
```

# Problem

The `Change` objects capture changes to the *text* but not the *point* (i.e., the position of the cursor)

Why is this important?

1. Must maintain the invariant that the point is within the text
   – A deletion when the point is near the end might break this

2. It's easier for the user to understand if the cursor is next to the text that has just changed

# Solution

We need to record the current value of the `point` somewhere (and any other relevant state) so that it can be restored after undoing the command

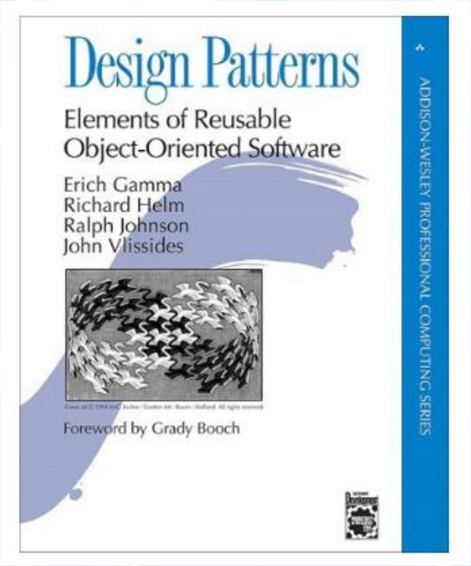
… but we don't want to break encapsulation

***Creational Patterns***

Abstract Factory
Builder
Factory Method
Factory Object
Lazy Initialization
Prototype
Singleton

***Structural Patterns***

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy




***Behavioral Patterns***

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

***Architectural***

Model-View-Controller
Service-oriented Architecture

***Concurrency Patterns:*** Active Object
Monitor
Thread Pool

# Solution

We deploy the Memento pattern:

- We record the value of the `point` (cursor position) in an object within `EdBuffer`
  - This object has a restore method that resets `point` to its former value
  - Other state components can similarly be recorded and restored within `EdBuffer` (see practical)
  - But we aren't breaking encapsulation and exposing this state

# A Memento for `EdBuffer`

The Memento pattern means taking a “memory” of the state so that we can return to it later (e.g., saving the state of a game)

```
class EdBuffer
    class Memento {
        private val pt = point
        def restore() { point = pt }
    }
```

this is an inner class “attached” to the EdBuffer object

- On *construction* the state of `EdBuffer` is saved
- The sole method restores the state to what it was when the memento was created
- “One thing well” => high cohesion

# Problem

We need to have a more sophisticated `Change` class that restores the value of the `point` after each undo/redo

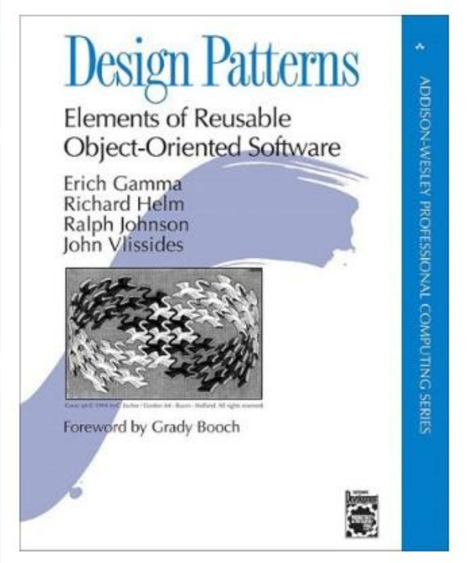
*… but we have lots of different **`Change`** classes to upgrade*

### *Creational Patterns*

Abstract Factory
Builder
Factory Method
Factory Object
Lazy Initialization
Prototype
Singleton

### *Structural Patterns*

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy




### *Behavioral Patterns*

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

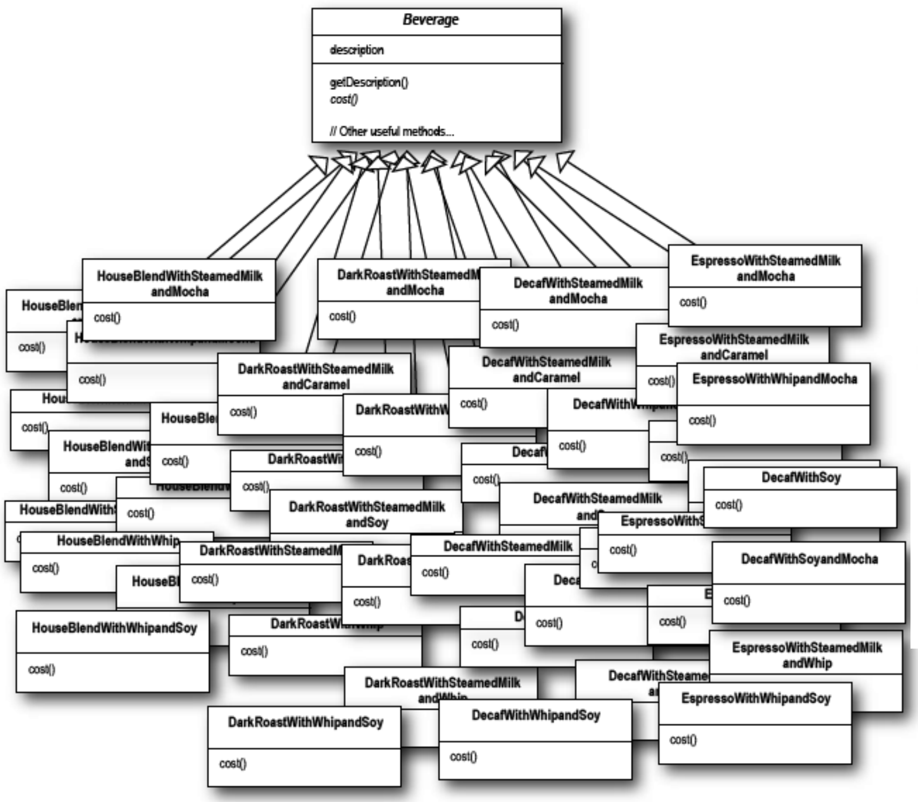
### *Architectural*

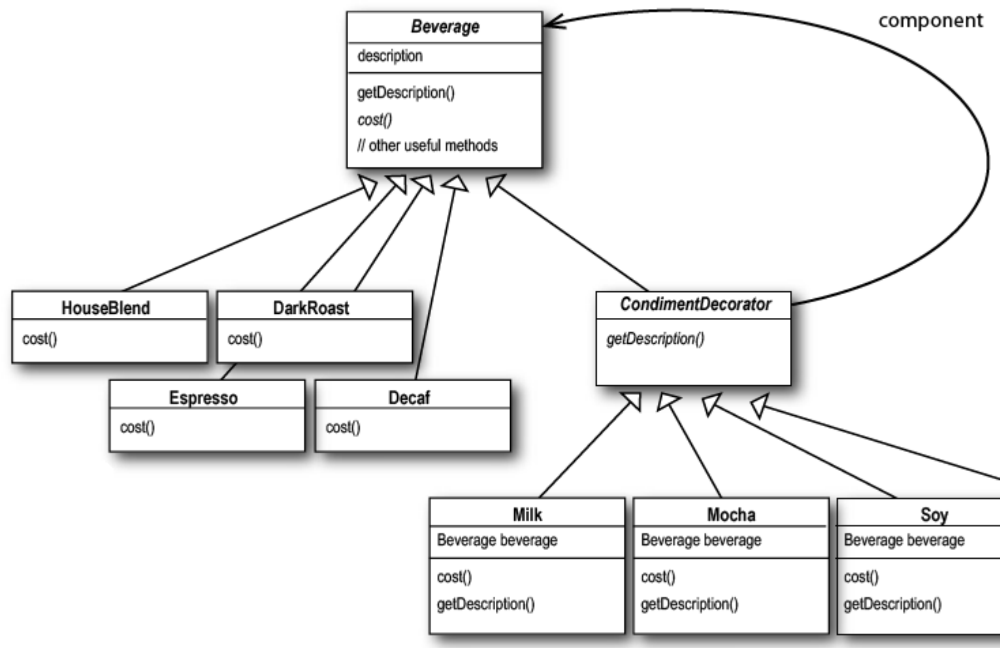
Model-View-Controller
Service-oriented Architecture

***Concurrency Patterns:*** Active Object
Monitor
Thread Pool

# Decorator

The Decorator pattern means using a mixin or a subclass to add functionality to a component class - may avoid an explosion of subclasses.




(From “Head First Design Patterns”)

# Solution

We deploy the Decorator pattern

- We wrap each `Change` in an `EditorChange` object in such a way that it can restore the state of the `EdBuffer` with each undo/redo operation

# A Decorator for Change

```
class EditorChange(before: ed.Memento,
                        change: Change, after: ed.Memento) extends Change {
        def undo() { change.undo(); before.restore() }
        def redo() { change.redo(); after.restore() }
}
```

```
class UndoableEditor extends Editor with UndoHistory {
    private var lastChange : Change = null

     override def obey(cmd: Editor.Command) {
            val before = ed.getState()
            super.obey(cmd)
            val after = ed.getState()
            if (lastChange != null)
                    updateHistory(new EditorChange(before,
                                                lastChange, after))
            lastChange = null
        }
}
```

# Ewoks: the whole story

- When a key is pressed the following things happen in the main loop of the editor:
  - The `key` value is requested from the `display` …
  - a `cmd` is found by looking up the `key` in the `keymap` …
  - `obey(cmd)` is invoked, which

    carries out tasks common to all editing commands like updating the display…
  - it also calls `cmd(editor)` to carry out the actions specific to this command, such as…
  - `editor.deleteCommand(RIGHT)` which actually performs the changes in the current text buffer

# Ewoks: the whole story (with undo)

- When a key is pressed the following things happen in the main loop of the editor:
  - The `key` value is requested from the `display` …
  - a `cmd` is found by looking up the `key` in the `keymap` …
  - `obey(cmd)` is invoked, which invokes the superclass `obey(cmd)`, and stores the resulting `Change` (if any) in the undo `history`, decorated with the state before and after the command…
  - The superclass `obey(cmd)` carries out tasks common to all editing commands like updating the display…
  - it also calls `cmd(editor)` to carry out the actions specific to this command, such as…
  - `editor.deleteCommand(RIGHT)` which actually performs the changes in the current text buffer and stores a `Change` in the `lastChange` field (if the command changes the text).

# Summary

- Extending `Editor` to `UndoableEditor`
  - Using an abstract superclass
  - Using a trait as a mixin
  - Using the Memento pattern
  - Using the Decorator pattern
  - Using inner classes

See also *Programming in Scala*: Chapter 12

See also *Head First Design Patterns*: Chapter 2