Imperative Programming 3

MVC and *Command*

Peter Jeavons

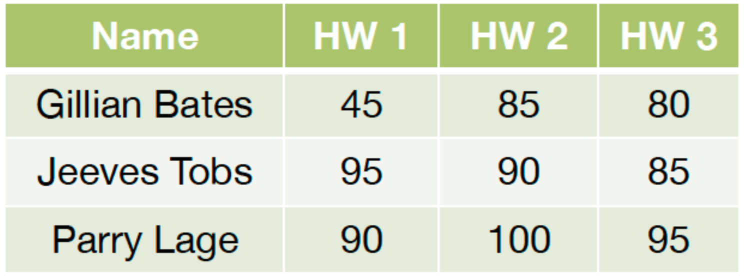
Trinity Term 2019

Agenda

- Observer pattern
- MVC architecture
- Applying MVC in the text editor
- Command pattern
- Applying Command in the text editor

Example: Student Database

- Database of grades
  - centralized storage of grade data



Name	HW 1	HW 2	HW 3
Gillian Bates	45	85	80
Jeeves Tobs	95	90	85
Parry Lage	90	100	95

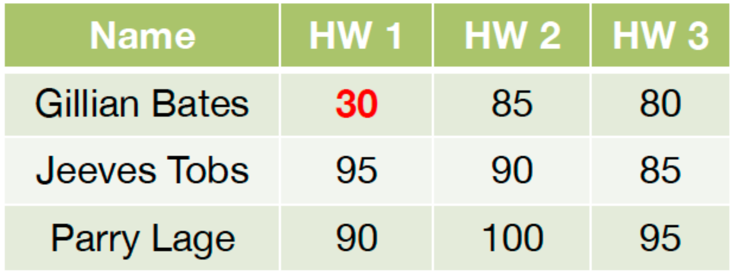
- Spreadsheet viewer
  - provides up-to-date view of grades
  - is notified of changes by the database

```
val ssv = new SpreadsheetView()
```

Change notification

- Regrade HW 1 of G. Bates
  - change score from 45 to 30

- Propagate change to view

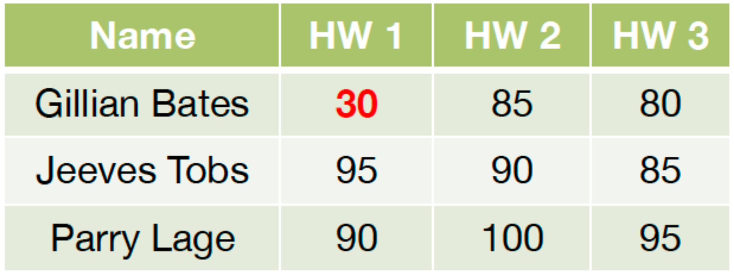


Name	HW 1	HW 2	HW 3
Gillian Bates	**30**	85	80
Jeeves Tobs	95	90	85
Parry Lage	90	100	95

```
val ssv = new SpreadsheetView()

// ...
ssv.update(“OOP”, “Gillian Bates”, “HW 1”, 30)
```

Multiple views

- Multiple views for same data
  - spreadsheet for instructor, bar chart for dean, pie charts for president, …
- Affects code on DB server
  - tight coupling



Name	HW 1	HW 2	HW 3
Gillian Bates	30	85	80
Jeeves Tobs	95	90	85
Parry Lage	90	100	95

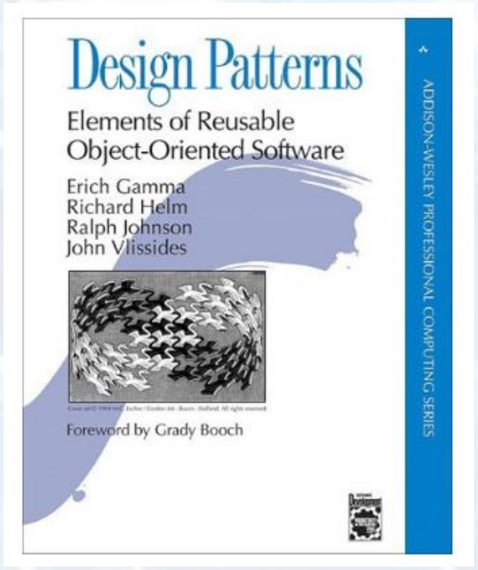
```
val ssv = new SpreadsheetView()
val bcv = new BarchartView()

// ...
ssv.update("OOP", "Gillian Bates", "HW 1", 30)
bcv.update("OOP", "Gillian Bates", "HW 1", 30)
```

Creational Patterns

Abstract Factory
Builder
Factory Method
Factory Object
Lazy Initialization
Prototype
Singleton

Structural Patterns

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Behavioral Patterns

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

Architectural

Model-View-Controller
Service-oriented Architecture

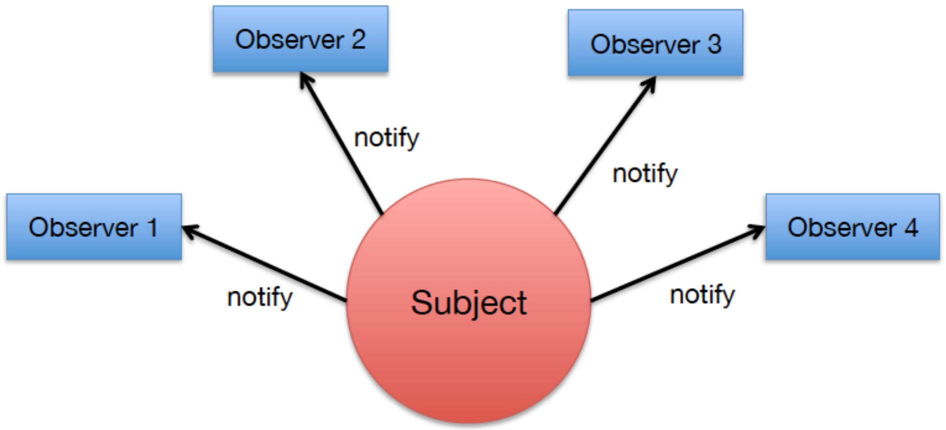
Concurrency Patterns: Active Object
Monitor
Thread Pool

Observer pattern

- Problem = several users of the same data…
  - can end up tightly coupled
  - adding new users means re-writing/recompiling
- Solution = make a modifiable list of observers
  - observers can be added and removed at runtime
  - data owner notifies all registered observers of changes
- Consequences
  - Interface not obvious - must choose either to simply notify that some change has happened or specify exactly what changed
  - Cannot rely on a specific order amongst observers

Observer pattern

- One-to-many dependency between Subject and any number of Observers
- Subject changes => all Observers are notified by invoking a fixed method in their interface
- Loose coupling
  - subject knows nothing about observers



All observers share the same interface

```
trait GradeDBObserver {
   def update(course: String, name: String, ...)
}
```

```
val observers = new ListBuffer[GradeDBObserver]()

def addObserver(obs: GradeDBObserver) =
  observers += obs
```

Add/remove observers dynamically

```
def removeObserver(obs: GradeDBObserver) =
  observers -= obs

observers.foreach(o => o.update("OOP", "G. Bates", …))
```

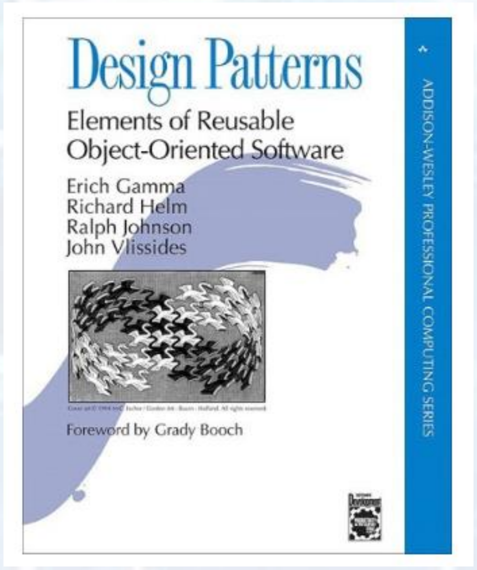
Notify all observers of changes to the DB

Creational Patterns

Abstract Factory
Builder
Factory Method
Factory Object
Lazy Initialization
Prototype
Singleton

Structural Patterns

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Behavioral Patterns

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

Architectural

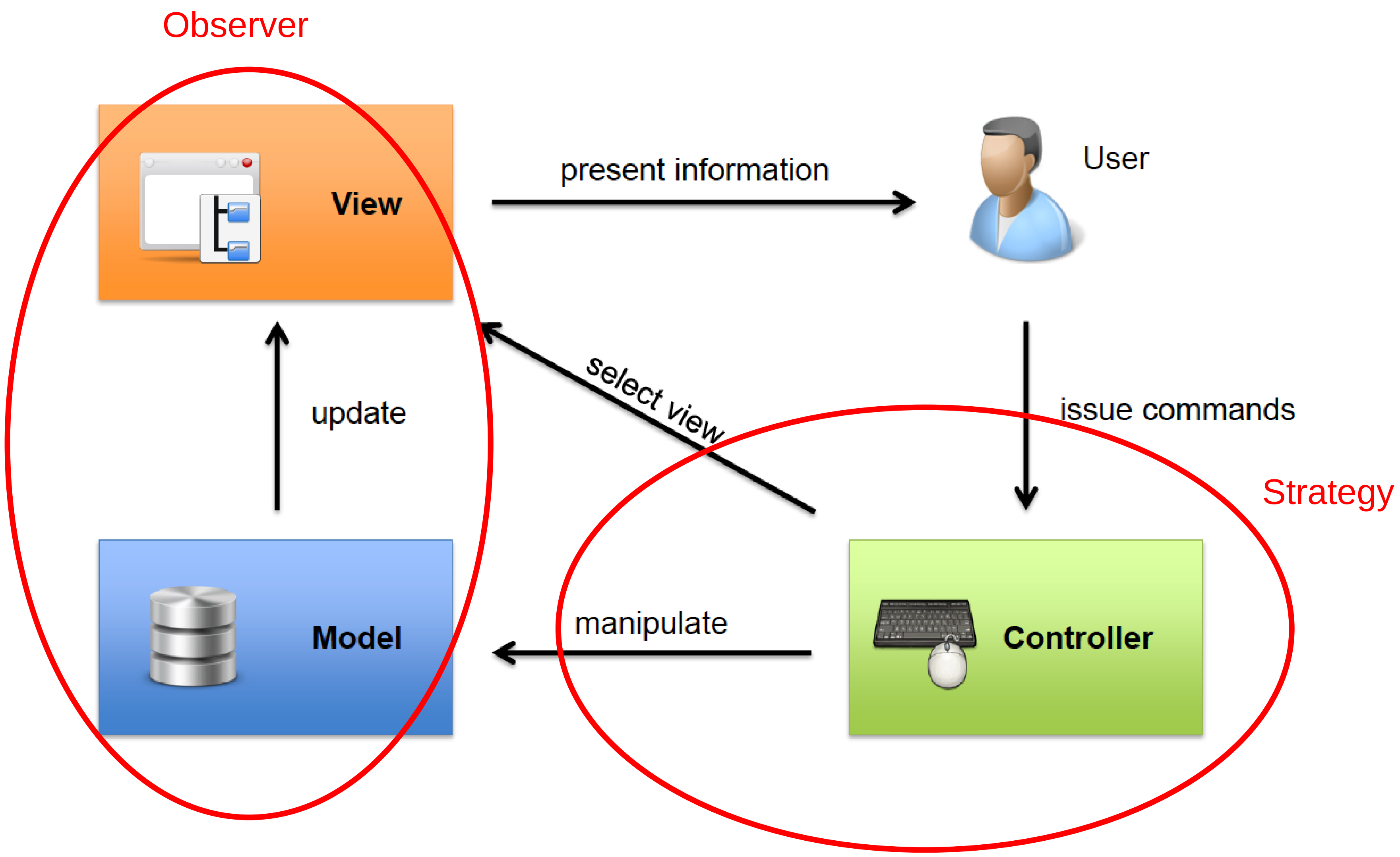
Model-View-Controller
Service-oriented Architecture

Concurrency Patterns: Active Object
Monitor
Thread Pool

Model-View-Controller Architecture

- Typical programs
  - process input, manipulate data, and display results
  - so have to co-ordinate a range of very different tasks
- Model-View-Controller
  - defines roles, *separates concerns*
  - introduced in SmallTalk at Xerox PARC in 1979
  - today’s standard design for GUIs and web applications
- Not (exactly) a pattern
  - an overall architecture combining several patterns

Model-View-Controller Architecture



MVC Architecture

- Separate Model, View, and Controller roles
  - each role can be spread over multiple modules
    - e.g., in web apps, part on client side, part on server side
  - but each module has only one role
  - can be fuzzy at times, choose cleanest solution with roles in mind
- Modularity
  - multiple views, possibly parallel and nested
  - controller can be substituted
  - logic can change without touching the user interface

Model-View-Controller in Ewoks?

Editor

Text

insert(int

currentP

insertOpera

deleteOperation()

saveOperation()

loadOperation()

Keymap

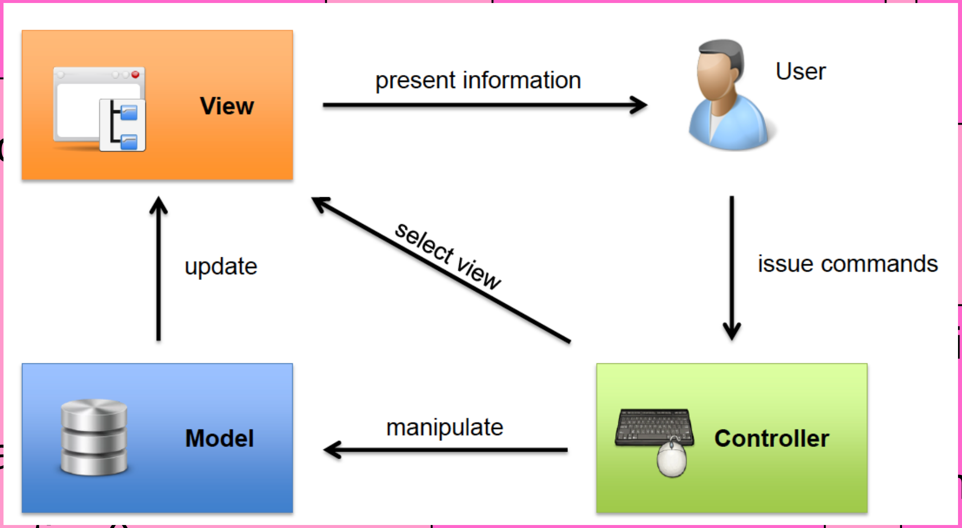
getLast()

InputStream

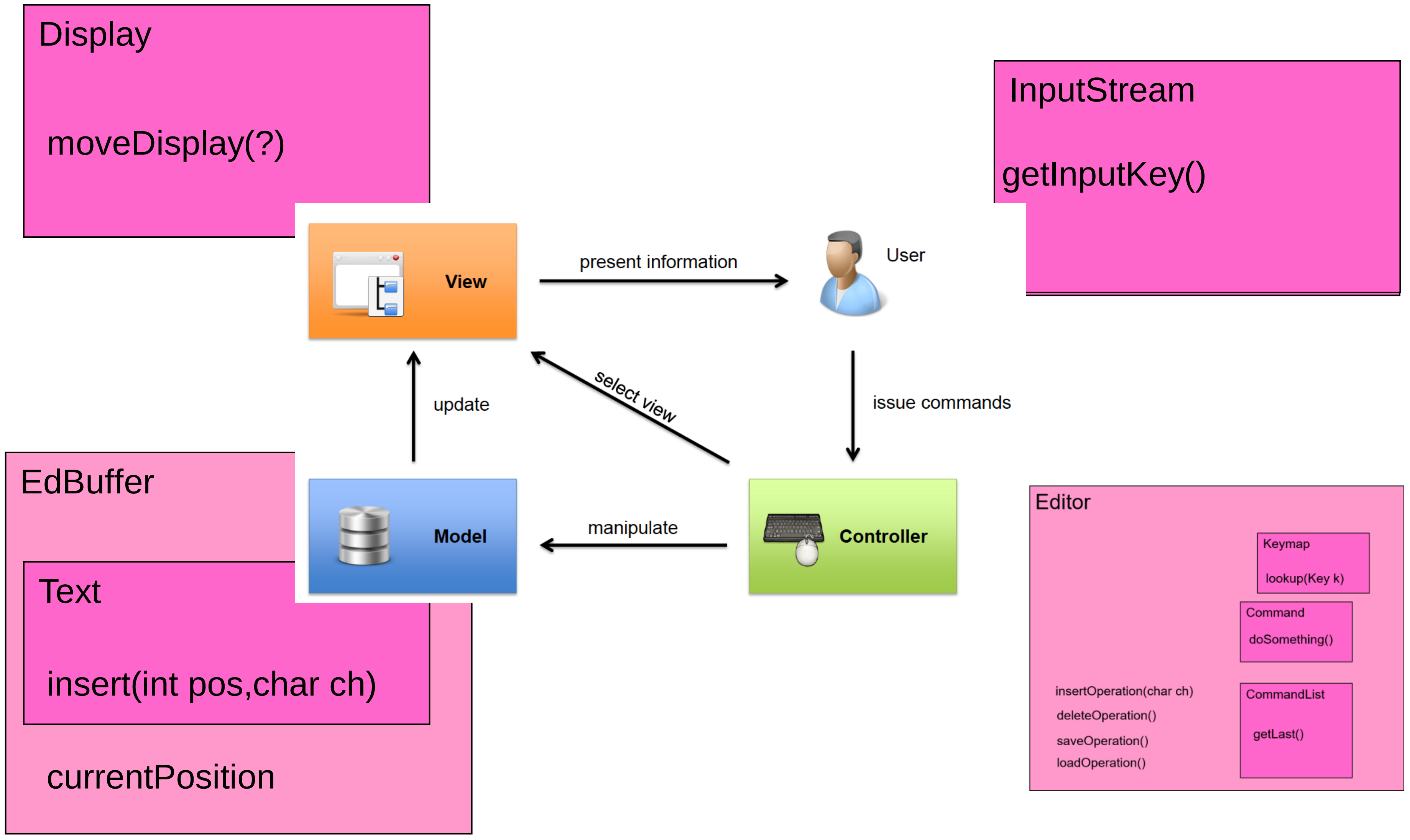
getInputKey()

isplay

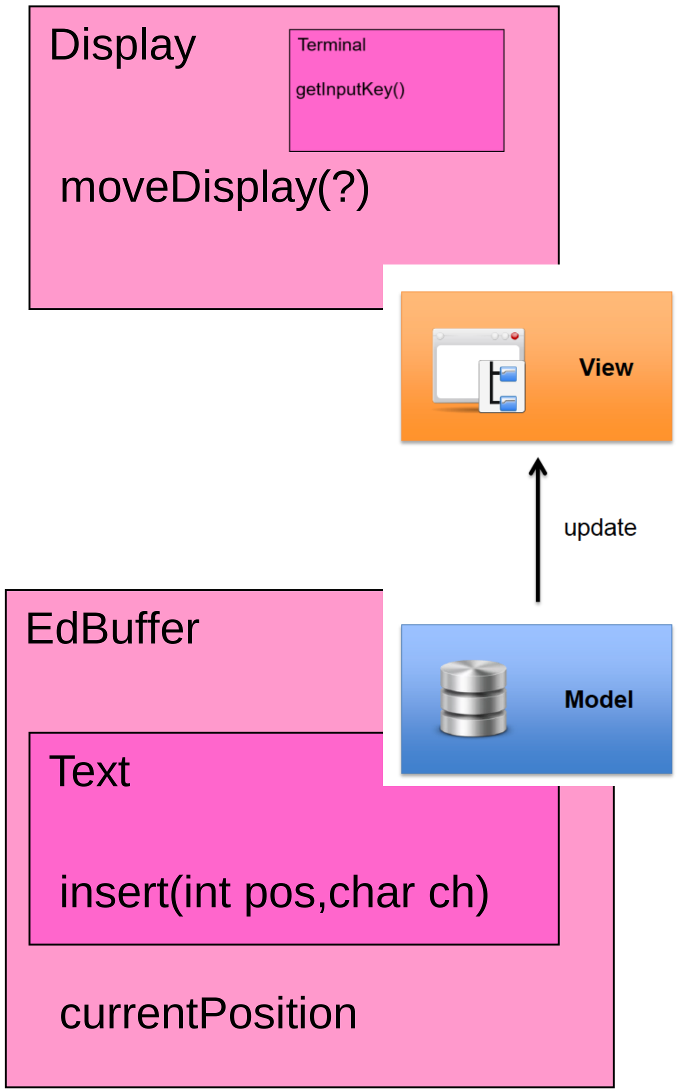
noveDisplay(?)



Model-View-Controller in Ewoks



Model-View in Ewoks



- It also has a `Display` object whose job is to provide a view of the text model
- When the state of the model changes, we need to update the display at some point
- The editor has an `EdBuffer` object whose job is to hold the current state of the text model

Updating the Display

- In our new architecture the controller simply invokes the `update` method of the `EdBuffer` object after processing each command

```
class Editor {
   protected val ed = new EdBuffer
   ...

   def obey(cmd: Command) = {
      cmd.execute(this)
      ed.update()
   }
}
```

Updating the Display

- When asked to `update`, the `EdBuffer` object will first check whether anything in the model has changed...

Question: How can we detect if the state of the model has changed?

Updating the Display

- Each command that changes the model sets an instance variable of the `EdBuffer` object to record the fact that change has occurred
- This variable can take several different values to indicate the degree of change…

The update method

```
class EdBuffer
    private val text = new PlaneText()
    private var point = 0

    private var damage = EdBuffer.CLEAN
    ...

    def update() {
        display.refresh(damage,
             text.getRow(point), text.getColumn(point))
        damage = EdBuffer.CLEAN
    }
```

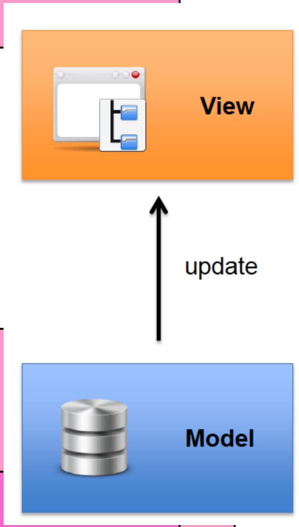
Model-View in Ewoks

Display

Terminal

getInputKey()

moveDisplay(?)



EdBuffer

Text

insert(int pos,char ch)

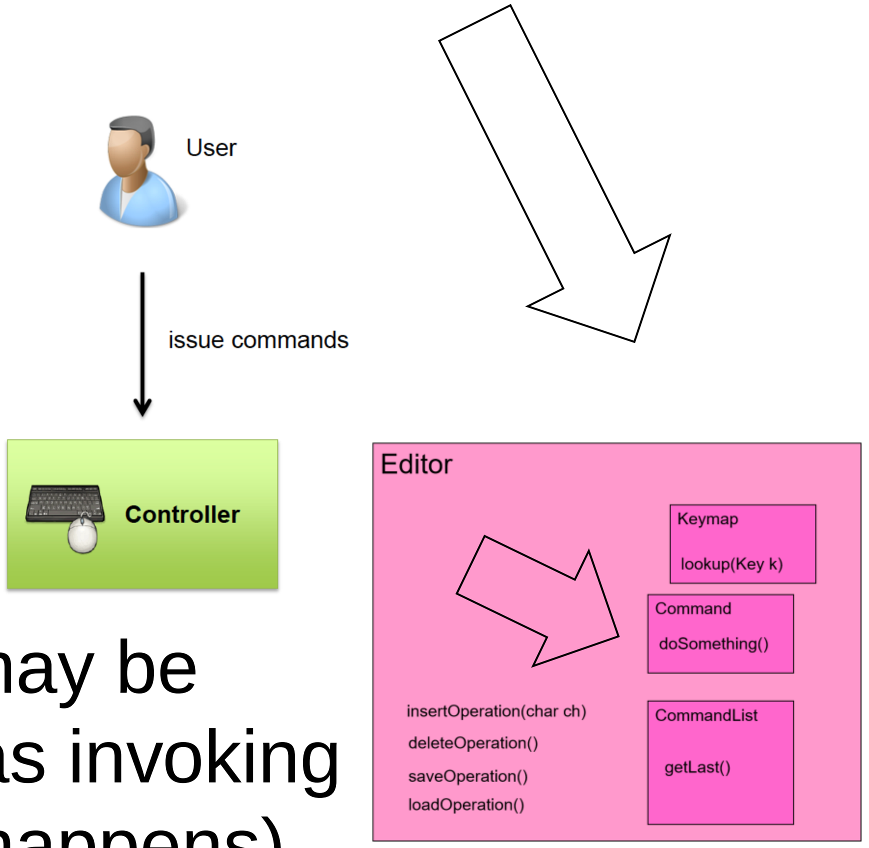
currentPosition

Controller in Ewoks

- The editor has an `Editor` object whose job as controller is to respond to user commands

- When the user presses a key, we will need to respond by carrying out some `Command`

- Each `Command` will do different things, but there may be common behaviour (such as invoking `update` or recording what happens)



Example: InsertCommand

```
class InsertCommand(val ch: Char) {

  def execute(editor: Editor) = {
    editor.insertOperation(ch)
  }

}
```

Defines some method to be invoked

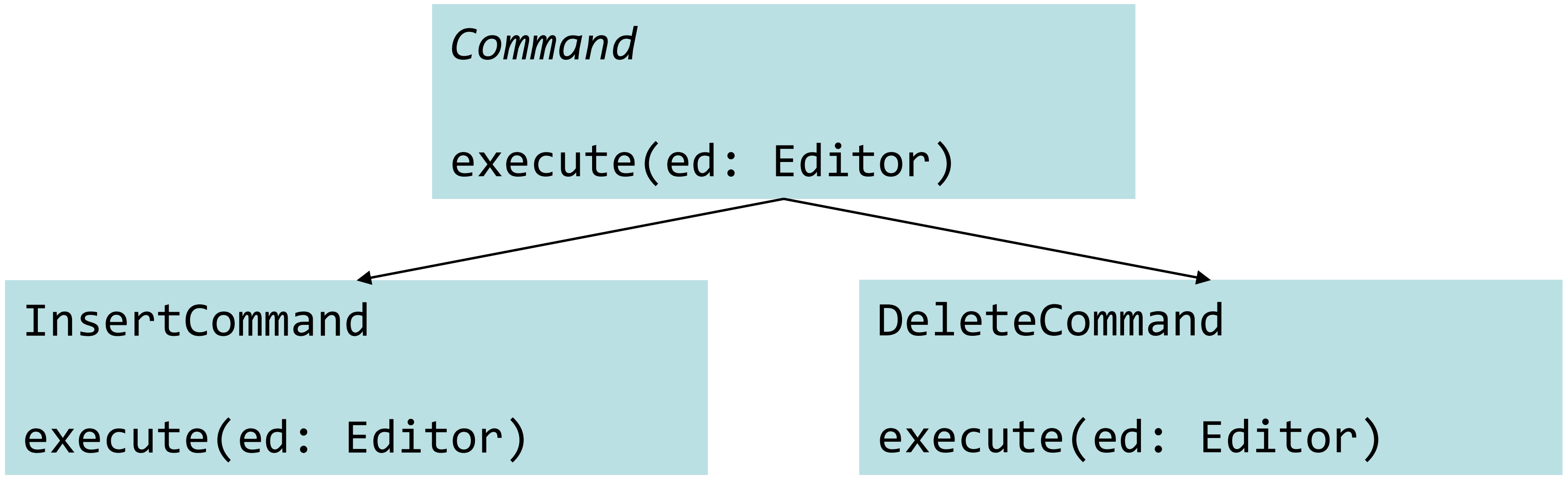
On some "receiver" object

Example: DeleteCommand

```
class DeleteCommand {

    def execute(editor: Editor) {
        editor.deleteOperation();
    }

}
```

The Command interface

- We now have two different kinds of command
  - `InsertCommand` and `DeleteCommand`
- They both provide an `execute` method



Example: InsertCommand

- Other classes can use the `InsertCommand` class in the following way:

```
val ed = new Editor()

val cmd: Command = new InsertCommand('a')

cmd.execute(ed)
```

Example: DeleteCommand

- Other classes can use the `DeleteCommand` class in the following way:

```
val ed = new Editor()

val cmd: Command = new DeleteCommand()

cmd.execute(ed)
```

- A common interface allows us to treat all commands uniformly:

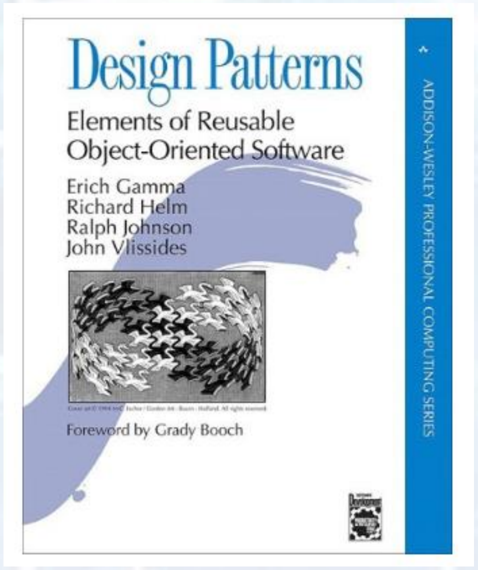
“Program to the interface”

Creational Patterns

Abstract Factory
Builder
Factory Method
Factory Object
Lazy Initialization
Prototype
Singleton

Structural Patterns

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Behavioral Patterns

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

Architectural

Model-View-Controller
Service-oriented Architecture

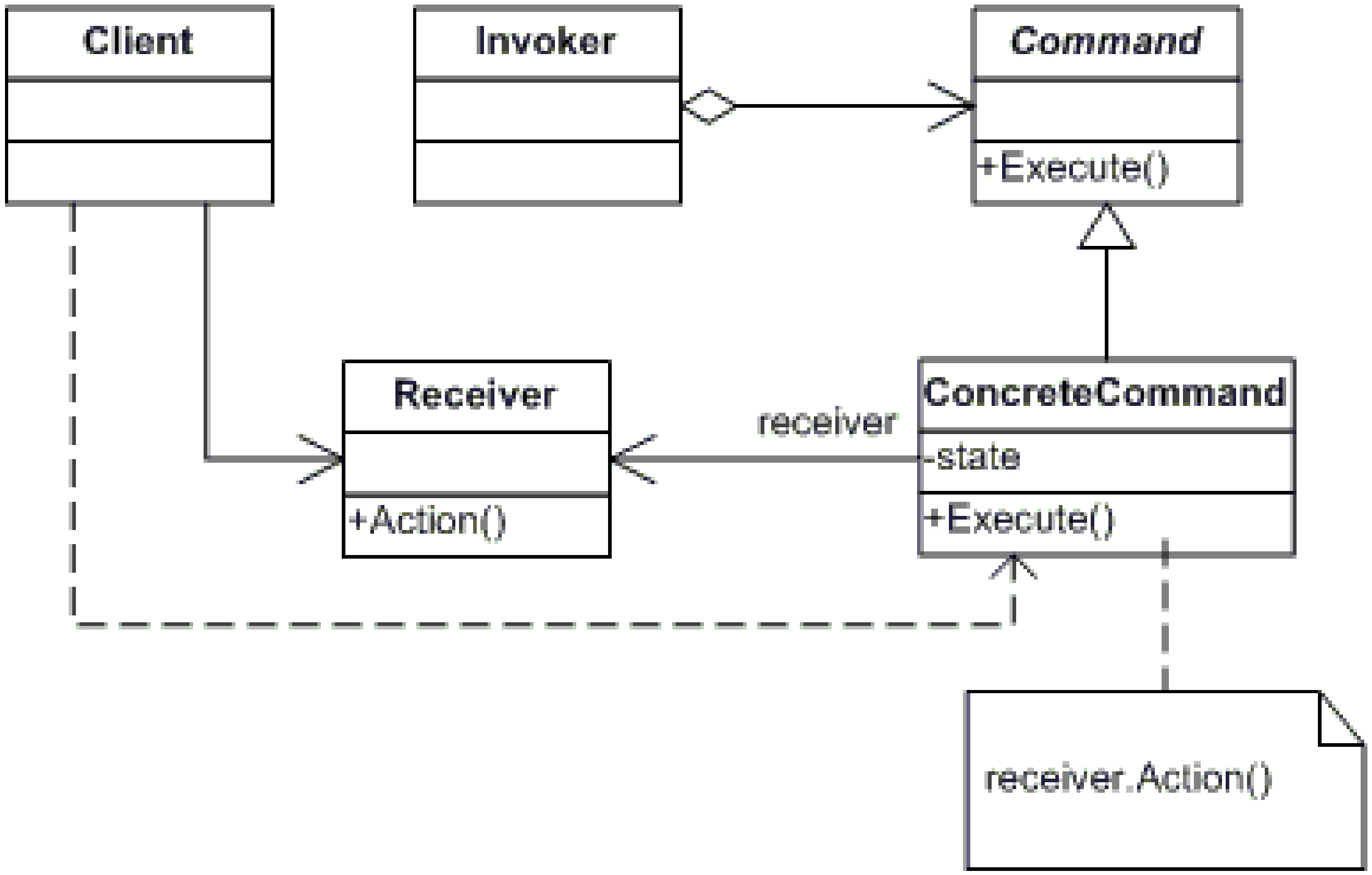
Concurrency Patterns: Active Object
Monitor
Thread Pool

Command pattern

The command interface is like a *button* on a remote control

- It does *one thing*: `execute`, such as delete a character (or turn off the TV)
  - High cohesion
- It may have no internal logic and simply invoke a *remote method,*
  e.g. `editor.deleteOperation()`
  - Loose coupling (separation of concerns)

Command pattern



Command pattern

- To add a command to any system we need three things:
  - A method in the **`Receiver`** class that carries out the effect of the command;
  - A **`ConcreteCommand`** class that implements the `Command` interface and carries out the command by calling the appropriate method in the `Receiver` (via its `execute` method);
  - An entry in the **`Client`** class that provides some way to access this concrete command and pass it to the **`Invoker`**

Issues

- This pattern seems to require a lot of classes (e.g. one for each command)

Question: Is this a good feature or a bad feature?

Commands in Ewoks

- To add a command to our editor we need three things:
  - A method in the **`Editor`** class that carries out the effect of the command;
  - A *function* of type **`Command`** that carries out the command by calling the appropriate method in the `Editor` (via its `apply` method);
  - An entry in the **`Keymap`** that links a keypress to this function so it can be carried out in the main command loop of the **`Editor`**.

Commands as Functions

```
type Command = (Editor => Boolean)
```

- Scala has shortcuts for writing functions so these are equivalent:

```
(editor: Editor) => editor.insertCommand(’x’)

editor => editor.insertCommand(’x’)

(_.insertCommand(’x’))
```

```
 for (ch <- Display.printable)
        keymap += ch -> (_.insertCommand(ch.toChar))
```

Ewoks: the whole story (so far)

- 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.

Summary

- Observer pattern
- MVC architecture
  - clean separation of roles
  - standard for GUI programs
  - combines several patterns
- Command pattern
  - encapsulates requests as objects
  - each command has an invoker, and a receiver

See also *Head First Design Patterns*: Chapters 2 & 6

Next lecture: undoing commands