Buy a Feature: an Adventure in Immutability and Actors

David Pollak
BayFP August 22, 2008
David Pollak

- Not strict, but pretty lazy
- Lead developer for *Lift* web framework
- Scala since November 2006, Ruby/Rails, Java/J2EE
- Spreadsheet junky (writing more than using)
- Paying work (all *Lift* based):
  - Enthiosys' Buy a Feature
  - SAP's ESME project
  - Gump-it: stuff worth missing
About Buy a Feature (online)

- The first of Enthiosys' online Innovation Games
- Serious Gaming for Agile Product Management
- Game Play:
  - Create a list of product features with estimated costs
  - 4-8 player buy features that they want
  - Motivate negotiations between players
  - Learn how players sell each other on features
Buy a Feature
About Scala & *Lift*

- **Scala**
  - Hybrid OO & Functional Language
  - Compiles to Java Byte-Code and runs fast on JVM
  - Compatible with Java libraries
  - FP concepts including Actors and Immutability

- **Lift**
  - Concise, powerful web framework
  - Leverages Scala's functional features
  - Awesome Comet and AJAX support
Buy a Feature Architecture

- *Lift* based Comet front-end
- UI state managed in *Lift* CometActors
- All user interaction via JSON messages/events
- Events sent to GameActor
- GameActor updates GameBoard and writes events
- GameActor sends GameBoard, etc. to CometActors
Actors – Why?

- Excellent concurrency management
- Event oriented
- Asynchronous

```scala
  case EndGame =>
    recordGameEnding()
    this ! ChatMessage(Empty, timeNow,
                     "Game Ended", Empty, Empty)
    eachListener(_ ! EndGame)
```
Actors – Where?

- **UI**
  - Pushes UI state changes out to browser
  - Listen for incoming events/messages

- **Cross-session Game managers**
  - Incoming events serialized
  - Incoming events → New State
  - New State → Listners (other Actors)
Events – Why?

- Anything that can change state is an Event
- Events are timestamped and written to RDBMS
- Events can be replayed through the system for TiVo style game replay and pausing
- Complementary to Actors
Events – Where?

- Browser → Server (CometActor)
- CometActor → GameActor
- GameActor → RDBMS
- GameActor → Listners (mostly UI CometActor)
- CometActor → Browser
Post-Processing

- Game Events are recalled, in order from RDBMS
- Game Events are send through the GameBoard
- GameBoard is queried for results
- GameBoard is immutable, so a separate copy can be associated with each Event
- Thus, there's a freeze-frame at each event
Defects

- *Lift* session bugs
  - Lots of stupid problems working around J2EE sessions
  - Why? I'm a moron
- Parsing
  - Users entering free text → lots of unexpected input
  - Most of our tests are here
- Post-processing
  - Didn't use GameBoard, but rolled my own – bad results
  - Too many GameBoards in memory
Team Integration

- Disbelief over code size
- Attempts to dive below the abstractions
- Java-like coding on the road to functional
- Eventual adoption of map, fold, and filter
- NPE: Thing of the past
- Lack of tool support and examples in the wild are speed bumps, especially with existing code
- Need a team mentor to help with transition
Conclusion

- Amazing productivity for people once up the FP curve
- Very low defect rate
- None of the defects were concurrency related!!
- None of the defects were concurrency related!!
- Very flexible system (added Flash front end in a day)
End

- Questions?
Scala: Functions are Objects

- Objects can be passed as parameters
- Functions are syntactically easy to create
  
  ```scala
  var name = ""
  SHtml.text(name, name = _)  
  ```
- They bind to variables/values (e.g. name)
Partial Functions

- PartialFunction[A,B] extends Function1[A,B]
- isDefinedAt(x: A)
- Better known as pattern matching:

```scala
{
  case Foo(bar) => bar
  case Baz(dog) => dog
}
```
Composing Partial Function

```scala
{ case Foo(bar) => bar
    case Baz(dog) => dog
} orElse { // compose
    case Moo(cow) => cow
    case Meow(cat) => cat
}
```
Extractors and Guards

- Extract data while matching other parts in a pattern:
  \{ case "Foo" :: id :: Nil => doIt(id) \}

- Guards:
  \{ case "Foo" :: id :: Nil
     if isValid(id) && loggedIn_? =>
     doIt(id) \}
Remembering Functions

- Functions are Objects
- Map[String, String => XML]
- Map[String, PartialFunction[String, XML]]
- GET /ajax?OPAQUE_ID=someValue
- Map[OPAQUE_ID](someValue)
XML literals and manipulation

- In Scala, XML is like String: supported at the language level and immutable
  `<foo>{(1 to 10).
    map(i => <val>{i}</val>)}</foo>`
- `(xml \ "val").map(_.text.toInt).
  .foldLeft(0)(_ + _) == 55`
Actors and Partial Functions

- Threadless, stackless units of execution
- React to events and otherwise consume nothing but memory

```
react(PartialFunction[Any, Any]) → react {case Foo(bar) => doSomething(bar)
  case Baz(dog) => doElse(dog) }

react(reactorHndlr orElse defaultHndler)
```
Lift REST APIs

- LiftRules.addDispatchBefore {
case RequestMatcher(
  RequestState(
    "showstates":: xs, _,_)) =>

  XmlServer.showStates(xs) }

- def showStates(...) = XmlResponse(
  <states renderedAt={timeNow.toString}>
...  </states>)}
Lift and HTML forms

- `var name = ""`  
- `text(name, name = _)`  
- `def setLocale(loc: String) ...`  
- `select(Locale.getAvailableLocales.toList.map(lo => (lo.toString, lo.getDisplayName)), setLocale)"`
Lift & AJAX

- AJAX elements are bound to functions:
  - `a(() => {cnt = cnt + 1; SetHtml("cnt_id", Text(cnt.toString))}, "click me")`
  - `ajaxSelect(opts, v => DisplayMessage("You selected " + v))`
Lift CometActors

- *Lift* deals with all the plumbing:
  
  ```scala
  def render = bind("time" => timeSpan)
  override def lowPriority = {
    case Tick => reRender(false)
  }
  ```