CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 22 Monads & Design Patterns
Dec 2, 2014

AppsFlyer

Mobile Analytics Company

Based in San Francisco

2 Billion events per day

Traffic double in 3 months

Grew from 6 to 50 people past year

Technologies used

- Redis, Kafka, Couchbase, CouchDB, Neo4j
- ElasticSearch, RabbitMQ, Consul, Docker, Mesos
- MongpDB, Riemann, Hadoop, Secor, Cascalog, AWS

AppsFlyer - Python Based

Started code base in Python

After two years python could not handle the traffic

Problems caused by

- String manipulations
- Python memory management

Their options

Rewrite parts in C & wrap in Python

Rewrite in programming language more suitable for data proccessing

- Wanted to try Functional Programming

Scala vs. OCaml vs. Haskell vs. Clojure

Scala

- Functional & Object Oriented
- They wanted pure Functional

OCaml

- Smaller community
- Only one thread runs at a time even on multicore

Haskell

- Monads made us cringe in fear

Clojure

- Runs on JVM
- Access to mutable state if needed
- Now have 10 Clojure engineers

Monads

What are they?

Why do they make engineers cringe in fear?

Monoids & Monads

Monoid

Binary Function Two parameters	Integer +
Parameters and returned value have same type	2 + 1
Identity value	2 + 0
Associatively	(2+3) + 4 = 2 + (3 + 4)

Monoid

Binary Function

- Two parameters

Parameters and returned value - same type

Identity value

Associatively

Java String concat

```
“hi”.concat(“ Mom”);

“hi”.concat(“”)

“hi”.concat(“Mom”.concat(“!”))
“hi”.concat(“Mom”).concat(“!”)
```

Monoid

Binary Function

Two parameters

Parameters and returned value - same type

Identity value

Associatively

Sets union

```
“hi”.concat(“ Mom”);
```

```
“hi”.concat(“”)
```

```
“hi”.concat(“Mom”.concat(“!”))
“hi”.concat(“Mom”).concat(“!”)
```

Monoid

Associative binary function F: X*X -> X
that has an identity

Haskell

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

Monad - Some Motivation

Exceptions

Interrupt program flow

```
(filter foo [a b c d e f g h])
```

Swift - optionals

```
let possibleNumber = "123"
let convertedNumber = possibleNumber.toInt()

if (convertedNumber)
      println( convertedNumber! )
```

Pyramid Of Doom

```
let b = foo(a)
if b
    let c = bar(b)
    if c
        let d = fooBar(c)
        if d
            let e = barFoo(e)
            if e
                return e!
            return “No e”
        return “No d”
    return “No c”
return “No b”
```

Clojure-like example

```
(-> some-collection
    foo
    bar
    fooBar
    barFoo)
```

What if one of the functions (foo, etc) returns an optional?

All the rest of the functions need handle them

Haskell Monad

Contains a context & four functions

return

- return :: a -> m a
- Takes a value and wraps in a monad

bind

- (>>=) :: m a -> (a -> m b) -> m b
- Take a
  - monad
  - function that requires a regular value and returns a monad
  - Applies the function to the monad

Haskell Monad

Contains a context & four functions

>>

```
(>>) :: m a -> m b -> m b
```

First argument is ignored

Error

Monad Laws

What are Monads used for?

In Haskell all functions are pure

Monad contexts can have side effects

All I/O in Haskell is done in monads

Monads allow you to compose computational steps together

Monads in Clojure

let

for

->

->>

Monads Tutorial For Clojure Programmers

http://onclojure.com/2009/03/05/a-monad-tutorial-for-clojure-programmers-part-1/

Design Patterns

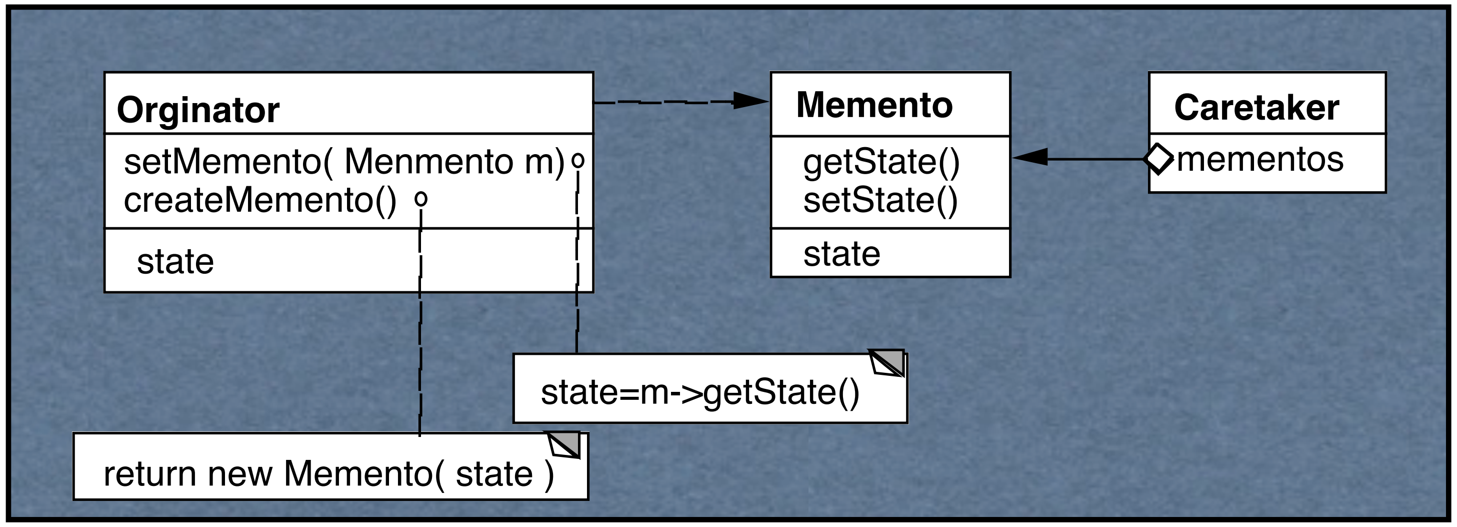
The Functional Pattern Joke

OO Pattern	Functional Equivalent
Adapter	Functions
Bridge	Functions
Chain of responsibility	Functions
Command	Functions
Composite	Functions
Decorator	Just Functions
Facade	Functions
Flyweight	Functions
Mediator	Functions
Observer	Functions
Strategy	Functions
Template method	Still Just Functions

Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

undo, rollbacks



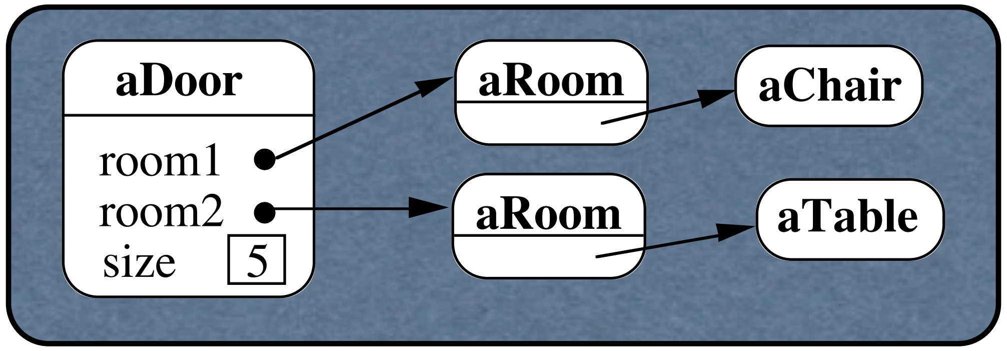
Only originator:

- Can access Memento’s get/set state methods
- Create Memento

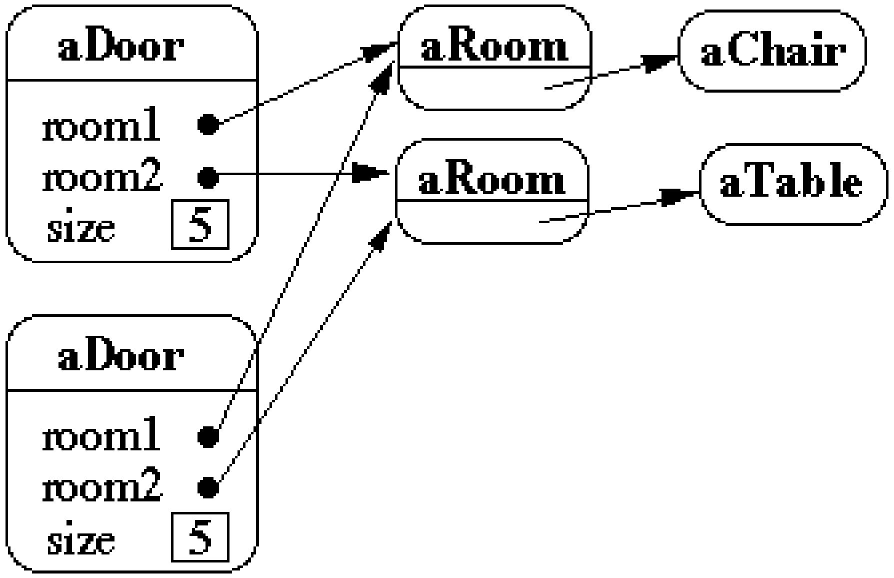
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects



Shallow Copy



Memento Pattern & Functional Programming

Immutable data

No need to copy the data

Just save current data

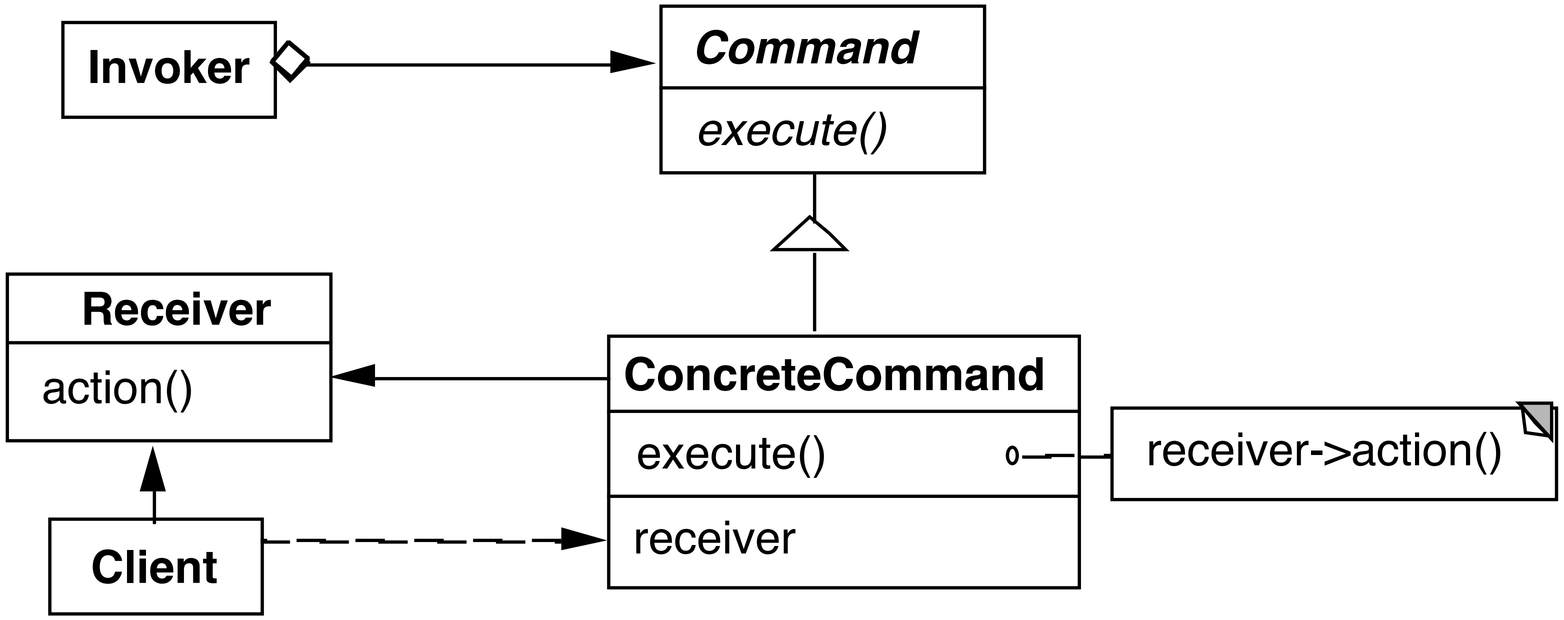
```
(def state-history (atom []))

(defn add-state
  [state]
  (swap! state-history conj state))

(defn previous-state
  []
  (let [last-state (last @state-history)]
    (swap! state-history pop)
    last-state))
```

Command Pattern

Encapsulates a request as an object



Example

Button in a GUI

When press button remove the current selected row of table

Command Class

```
public class RemoveRowCommand extends Command {
    private Table target;

    public RemoveRowCommand(Table target) {
        this.target = target;
    }

    public execute() {
        int selection = target.getSelection();
        target.removeRow(selection);
    }
}
```

Using the Command

```
Button removeSelection = new Button();
Command removeRow = new RemoveRowCommand(ourTable);
removeSelection.action(removeRow);
```

Button class is written to call execute when button is pressed

Clojure Example

```
(def button
  (seesaw/button
   :text "Remove Selection"
   :listen [:action (fn [event](
                   (let [selectedRow (seesaw/selection ourTable)]
                        (seesaw/remove-at! ourTable selectedRow))]))
```

More General

```
(defn removeRow!
    [table event]
    (let [selectedRow (seesaw/selection table)]
        (seesaw/remove-at! table selectedRow)))

(def button
  (seesaw/button
   :text "Remove Selection"
   :listen [:action (partial removeRow ourTable)]))
```

Command Pattern Supports Undo

- Modify class
  - Add undo method

- Keep stack of past commands

- Undo
  - Pop the stack
  - Call undo on element removed from stack

```
public class RemoveRowCommand extends Command {
    private Table target;
    private int rowIndex;
    private Row removedRow;

    public RemoveRowCommand(Table target) {
        this.target = target;
    }

    public void execute() {
        rowIndex = target.getSelection();
        removedRow = target.getRow(rowIndex);
        target.removeRow(rowIndex);
    }

    public void undo() {
        if (removedRow == nil) return;
        target.addRow(removedRow, rowIndex);
        removedRow = nil;
    }
}
```

```
Button removeSelection = new Button(“Remove Selection);
Command removeRow = new RemoveRowCommand(ourTable);
removeSelection.action(removeRow);
Button undoRemove = new Button(“Undo”); // needs work here
undo.action(removeRow)
```

Converting Objects to Clojure data

Class	Map
Field name	keyword as key in map
new Person("Sachin", "Tendulkar", 40);	{:first-name "Sachin" :last-name "Tendulkar" :age 40 :phone-numbers {}}

Undo - Using maps & multimethods

Store the data needed for undo in a map

Use multimethod to perform undo

Undo - Add Subtract Example

Data needed to undo addition

- Current value
- Value added

```
{:command :add :value 10 :amount 2}
```

Data needed to undo subtractiom

- Current value
- Value subtracted

```
{:command :subtraction :value 10 :amount 2}
```

The Multimethod

```
(defmulti undo :command)

(defmethod undo :add
  [{:keys [value amount]}]
  (- value amount))

(defmethod undo :subtract
  [{:keys [value amount]}]
  (+ value amount))
```

```
(def example
      {:command :add :value 10 :amount 2})

(undo example)
```

Adding the Table

```
(defmulti undo :command)

(defmethod undo :add
  [{:keys [value amount]}]
  (- value amount))

(defmethod undo :subtract
  [{:keys [value amount]}]
  (+ value amount))

(defmethod undo :remove-row
  [{:keys [table row-index row]}]
  (seesaw/insert-at! table row row-index))
```

Updated Row

```
(defn removeRow!
   [table event]
   (let [selected-index (seesaw/selection table)
          selected-row (seesaw/value-at selected-index)]
         (seesaw/remove-at! table selectedRow)
         (save-command {:command :remove-row
                                    :row selected-row
                                    :row-index selected-index)))

(def button
 (seesaw/button
  :text "Remove Selection"
  :listen [:action (partial removeRow ourTable)]))
```

Command History

```
(def command-history (atom []))

(defn save-command
  [command]
  (swap! command-history conj command))

(defn previous-command
  []
  (let [last-command (last @command-history)]
    (swap! command-history pop)
    last-command))
```

Memento Pattern

Idea - save current state

OO implementation

Copy objects

Deal with information hiding

Functional implementation

Just save current state

Command Pattern

Idea: Save data needed to perform an operation

OO Implementation

Separate class for data

Interface for executing method

Functional implementation

Use map for the data

What is the Pattern?

The idea?

The implementation?

What is important?

Iterator Pattern

Provide a way to access the elements of a collection sequentially without exposing its underlying representation

```
LinkedList<Strings> strings =  new LinkedList<Strings>();

for (String element : strings) {
        if (element.size % 2 == 0)
                System.out.println(element);
}

Iterator<String> list = strings.iterator();
while (list.hasNext()){
        String element = list.next();
        if (element.size % 2 == 0)
                System.out.println(element);
        }
}
```

Iterator Pattern - Clojure

sequences

Strategy Pattern

defines a family of algorithms,
encapsulates each algorithm, and
makes the algorithms interchangeable within that family.

Java Example

```
class OrderableList {
        private Object[ ] elements;
        private Algorithm orderer;

        public OrderableList(Algorithm x) {
                orderer = x;
        }

    public void add(Object element) {
       elements = orderer.add(elements,element);
    }
```

Clojure Example

```
(sort-by last {:b 1 :c 3 :a 2})
```

Just pass in a function