OOP and FP

Richard Warburton

What on earth are you talking about?

SOLID Principles

Design Patterns

Anthropology

In Quotes ...

"OOP is to writing a program, what going through airport security is to flying"

- Richard Mansfield

"TDD replaces a type checker in Ruby in the same way that a strong drink replaces sorrows."

- byorgey

In Quotes ...

"Brain explosion is like a traditional pasttime in #haskell"

"Some people claim everything is lisp. One time I was eating some spaghetti and someone came by and said: 'Hey, nice lisp dialect you're hacking in there'"


HD

Caveat: some unorthodox definitions may be provided

What on earth are you talking about?

SOLID Principles

Design Patterns

Anthropology

SOLID Principles

- Basic Object Oriented Programming Principles
- Make programs easier to maintain
- Guidelines to remove code smells

Single Responsibility Principle

- Each class/method should have single responsibility
- Responsibility means “reason to change”
- The responsibility should be encapsulated

```
int countPrimes(int upTo) {
  int tally = 0;
  for (int i = 1; i < upTo; i++) {
    boolean isPrime = true;
    for (int j = 2; j < i; j++) {
      if (i % j == 0) {
        isPrime = false;
      }
    }
    if (isPrime) {
      tally++;
    }
  }
  return tally;
}
```

```
int countPrimes(int upTo) {
  int tally = 0;
  for (int i = 1; i < upTo; i++) {
    if (isPrime(i)) {
      tally++;
    }
  }
  return tally;
}
boolean isPrime(int number) {
  for (int i = 2; i < number; i++) {
    if (number % i == 0) {
      return false;
    }
  }
  return true;
}
```

```
long countPrimes(int upTo) {
  return IntStream.range(1, upTo)
                  .filter(this::isPrime)
                  .count();
}

boolean isPrime(int number) {
  return IntStream.range(2, number)
                  .allMatch(x -> (number % x) != 0);
}
```

Higher Order Functions

- Hard to write single responsibility code in `Java` before 8

- Single responsibility requires ability to pass around behaviour

- Not just functions, Higher Order Functions

Open Closed Principle

"*software entities should be open for extension, but closed for modification*"

- Bertrand Meyer

Example: Graphing Metric Data

OCP as Polymorphism

- Graphing Metric Data
  - `CpuUsage`
  - `ProcessDiskWrite`
  - `MachineIO`

- `GraphDisplay` depends upon a `TimeSeries` rather than each individually

- No need to change `GraphDisplay` to add `SwapTime`

OCP as High Order Function

```
// Example creation
ThreadLocal<DateFormat> formatter =
  withInitial(() -> new SimpleDateFormat());

// Usage
DateFormat formatter = formatter.get();

// Or ...
AtomicInteger threadId = new AtomicInteger();
ThreadLocal<Integer> formatter =
  withInitial(() -> threadId.getAndIncrement());
```

OCP as Immutability

- Immutable Object cannot be modified after creation
- Safe to add additional behaviour
- New pure functions can’t break existing functionality because it can’t change state

Liskov Substitution Principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

* Excuse the informality

A subclass behaves like its parent.

* This is a conscious simplification

1. Where the parent worked the child should.
2. Where the parent caused an effect then the child should.
3. Where parent always stuck by something then the child should.
4. Don’t change things your parent didn’t.

Functional Perspective

- Inheritance isn’t key to FP
- Lesson: don’t inherit implementation and LSP isn’t an issue!
- Composite Reuse Principle already commonly accepted OOP principle

Interface Segregation Principle

"The dependency of one class to another one should depend on the smallest possible interface"

- Robert Martin

Factory Example

```
interface Worker {
  public void goHome();
  public void work();
}
```

`AssemblyLine` requires instances of `Worker: AssemblyWorker` and `Manager`

The factories start using robots...

… but a `Robot` doesn't `goHome()`

Nominal Subtyping

- For `Foo` to extend `Bar` you need to see `Foo extends Bar` in your code.

- Relationship explicit between types based on the name of the type

- Common in Statically Typed, OO languages: Java, C++

```
class AssemblyWorker implements
Worker

class Manager implements Worker

class Robot implements Worker
```

```
public void addWorker(Worker worker) {
    workers.add(worker);
}

public static AssemblyLine newLine() {
    AssemblyLine line = new AssemblyLine();
    line.addWorker(new Manager());
    line.addWorker(new AssemblyWorker());
    line.addWorker(new Robot());
    return line;
}
```

Structural Subtyping

- Relationship implicit between types based on the shape/structure of the type

- If you call `obj.getFoo()` then `obj` needs a `getFoo` method

- Common in wacky language: Ocaml, Go, C++ Templates, Ruby (quack quack)

```
class StructuralWorker {

  def work(step:ProductionStep) {
    println(
      "I'm working on: "
      + step.getName)
  }

}
```

```
def addWorker(worker: {def work(step:ProductionStep)}) {
  workers += worker
}

def newLine() = {
  val line = new AssemblyLine
  line.addWorker(new Manager())
  line.addWorker(new StructuralWorker())
  line.addWorker(new Robot())
  line
}
```

Hypothetically …

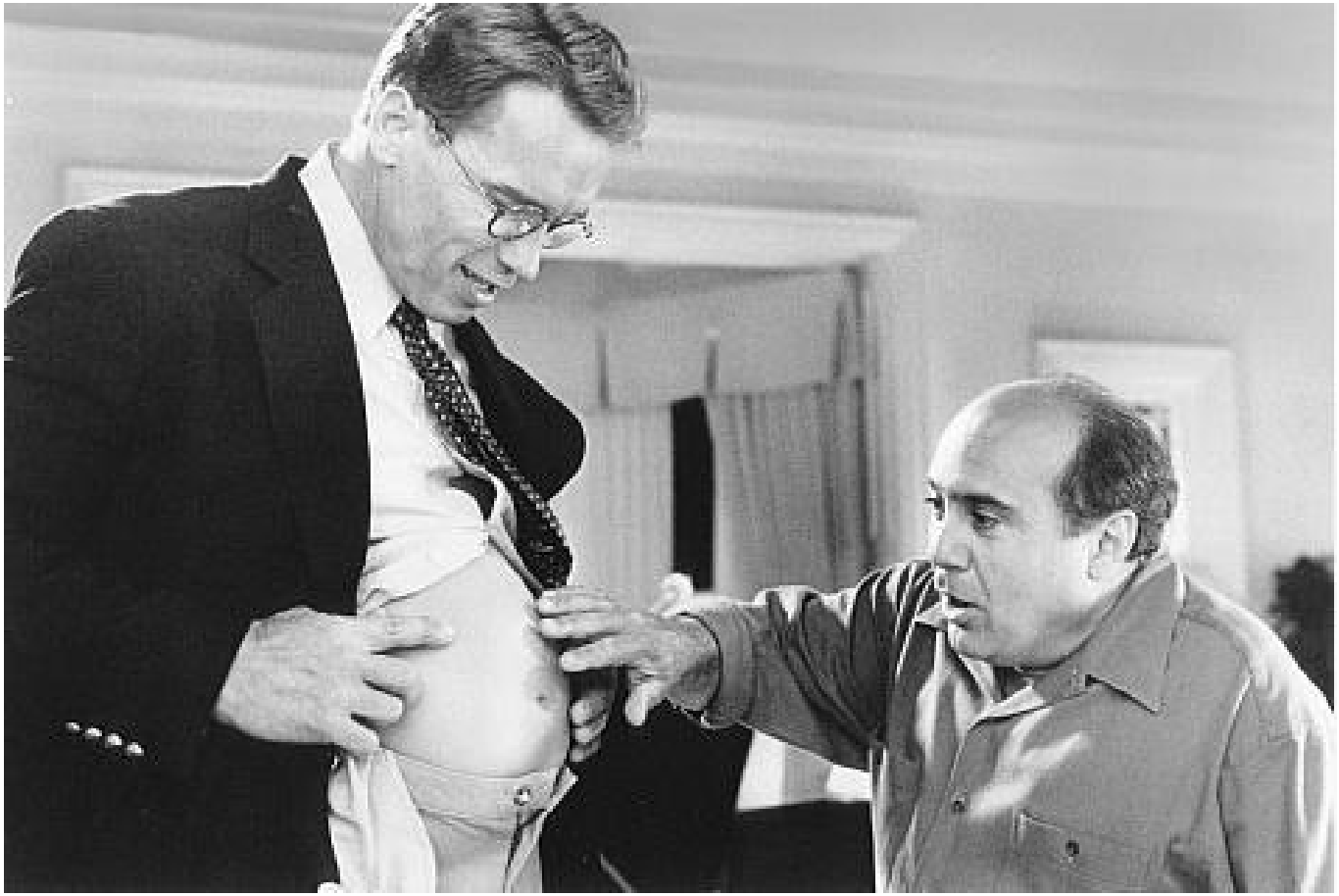
```
def addWorker(worker) {
  workers += worker
}

def newLine() = {
  val line = new AssemblyLine
  line.addWorker(new Manager())
  line.addWorker(new StructuralWorker())
  line.addWorker(new Robot())
  line
}
```

Functional Interfaces

- An `interface` with a single abstract method
- By definition the minimal interface!
- Used as the inferred types for lambda expressions in Java 8

Thoughts on ISP

- Structural Subtyping removes the need for Interface Segregation Principle
- Functional Interfaces provide a nominal-structural bridge
- ISP != implementing 500 interfaces



Dependency Inversion Principle

Dependency Inversion Principle

- Abstractions should **not** depend on details, details **should** depend on abstractions

- Decouple glue code from business logic

- Inversion of Control/Dependency Injection is an implementation of DIP

Streams Library

```
album.getMusicians()
     .filter(artist -> artist.name().contains(“The”))
     .map(artist -> artist.getNationality())
     .collect(toList());
```

Resource Handling & Logic

```
List<String> findHeadings() {
  try (BufferedReader reader
       = new BufferedReader(new FileReader(file))) {

    return reader.lines()
                 .filter(isHeading)
                 .collect(toList());
  } catch (IOException e) {
    throw new HeadingLookupException(e);
  }
}
```

Business Logic

```
private List<String> findHeadings() {
  return withLinesOf(file,
                     lines -> lines.filter(isHeading)
                                   .collect(toList()),
                     HeadingLookupException::new);
}
```

Resource Handling

```
<T> T withLinesOf(String file,
                  Function<Stream<String>, T> handler,
                  Function<IOException,
                           RuntimeException> error) {

  try (BufferedReader reader =
    new BufferedReader(new FileReader(file))) {

    return handler.apply(reader.lines());
  } catch (IOException e) {
    throw error.apply(e);
  }
}
```

DIP Summary

- Higher Order Functions also provide Inversion of Control
- Abstraction != `interface`
- Functional resource handling, eg `withFile` in haskell

All the solid patterns have a functional equivalent

The same idea expressed in different ways

What on earth are you talking about?

SOLID Principles

Design Patterns

Anthropology

Command Pattern

- Receiver - performs the actual work.
- Command - encapsulates all the information required to call the receiver.
- Invoker - controls the sequencing and execution of one or more commands.
- Client - creates concrete command instances

Macro: take something that’s long and make it short

```
public interface Editor {

   public void save();

   public void open();

   public void close();

}
```

```
public interface Action {

   public void perform();

}
```

```
public class Open implements Action {

    private final Editor editor;

    public Open(Editor editor) {

        this.editor = editor;

    }

    public void perform() {

        editor.open();

    }

}
```

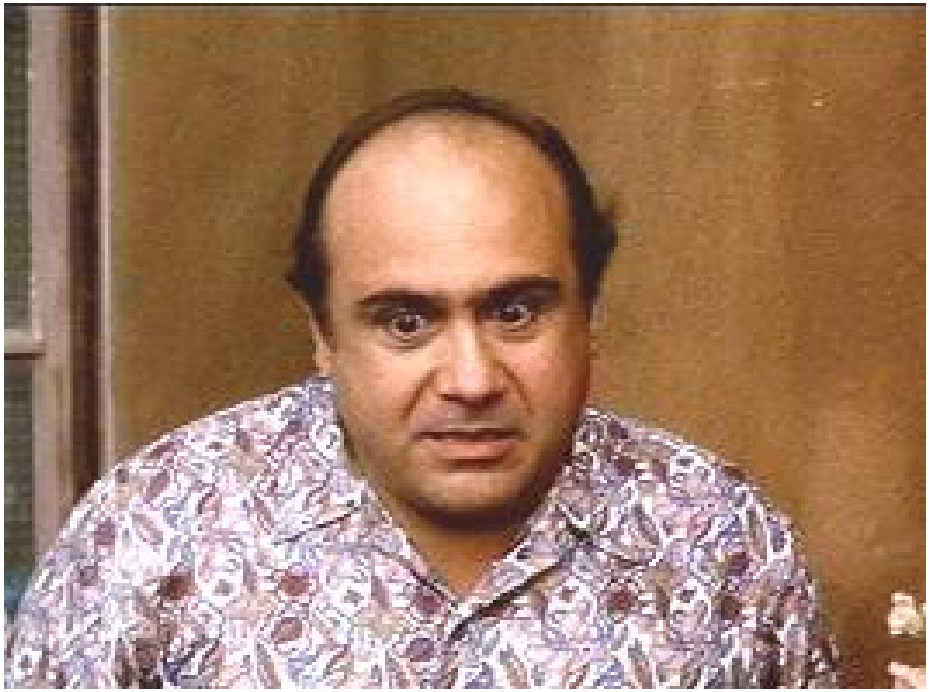
```
public class Macro {

    private final List<Action> actions;

    …

    public void record(Action action) {

        actions.add(action);

    }

    public void run() {

    actions.forEach(Action::perform);

    }

}
```

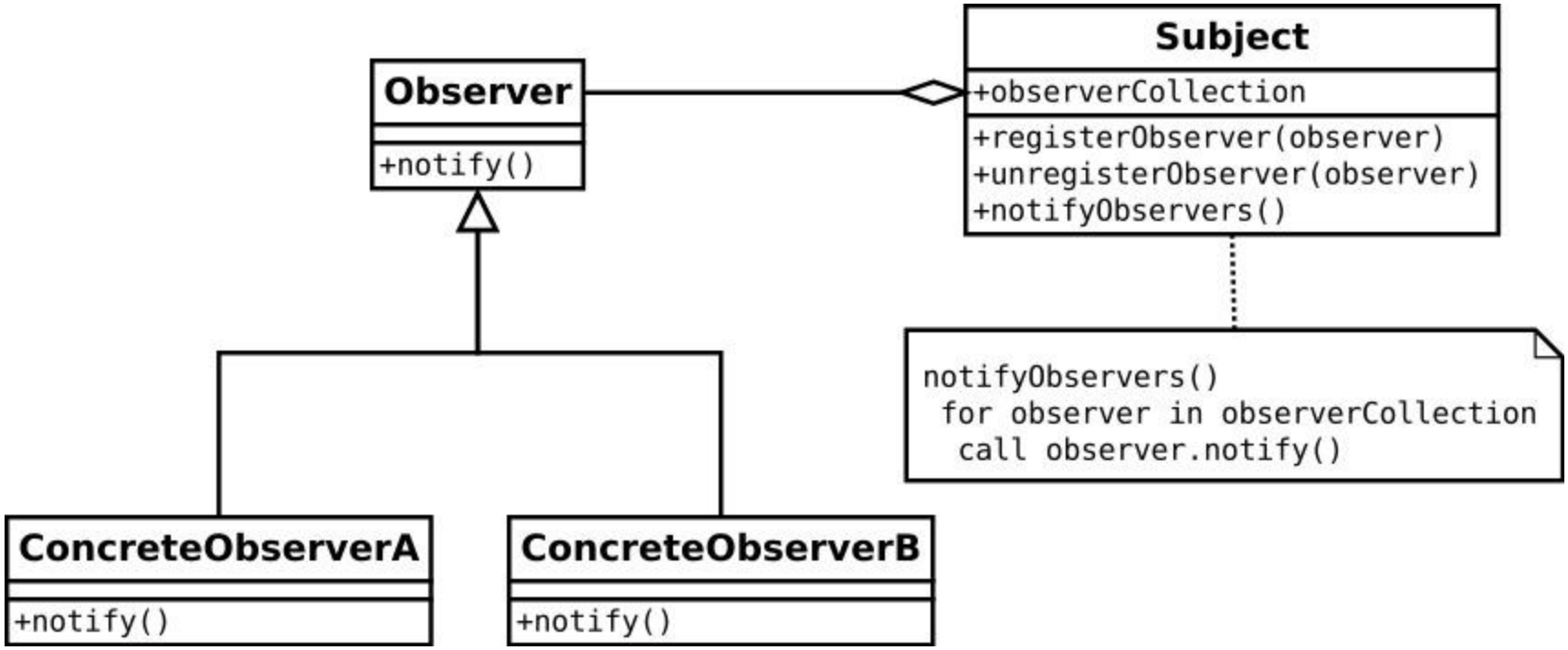
```
Macro macro = new Macro();
macro.record(new Open(editor));
macro.record(new Save(editor));
macro.record(new Close(editor));
macro.run();
```

The Command Object is a Function

```
Macro macro = new Macro();
macro.record(() -> editor.open());
macro.record(() -> editor.save());
macro.record(() -> editor.close());
macro.run();
```

Observer Pattern




Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
Observer
+notify()
notifyObservers()
for observer in observerCollection
call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

Concrete Example: Profiler

```
public interface ProfileListener {

    public void accept(Profile profile);

}
```

```
private final List<ProfileListener> listeners;

public void addListener(ProfileListener listener) {
    listeners.add(listener);
}

private void accept(Profile profile) {
    for (ProfileListener listener : listeners) {
        listener.accept(profile)
    }
}
```

Previously you needed to write this **EVERY** time.

```
Consumer<T>       === T → ()
ProfileListener === Profile → ()
ActionListener  === Action → ()
```

```java
public class Listeners<T> implements Consumer<T> {

    private final List<Consumer<T>> consumers;

    public Listeners<T> add(Consumer<T> consumer) {
        consumers.add(consumer);
        return this;
    }

    @Override
    public void accept(T value) {
        consumers.forEach(consumer -> consumer.accept(value));
    }
```

```
public ProfileListener provide(
    FlatViewModel flatModel,
    TreeViewModel treeModel) {

    Listeners<Profile> listener = new
Listeners<Profile>()
            .of(flatModel::accept)
            .of(treeModel::accept);

    return listener::accept;
}
```

Existing Design Patterns don’t need to be thrown away.

Existing Design Patterns can be improved.

What on earth are you talking about?

SOLID Principles

Design Patterns

Anthropology

Popular programming language evolution follows Arnie’s career.

The 1980s were great!

Programming 80s style

- Strongly multiparadigm languages
  - Smalltalk 80 had lambda expressions
  - Common Lisp Object System
- Polyglot Programmers
- Fertile Language Research
- Implementation Progress - GC, JITs, etc.

The 1990s ruined everything

90s and 2000s Market Convergence

- Huge Java popularity ramp
  - Javaone in 2001 - 28,000 attendees
  - Servlets, J2EE then Spring
- Virtual death of Smalltalk, LISP then Perl
- Object Oriented Dominance

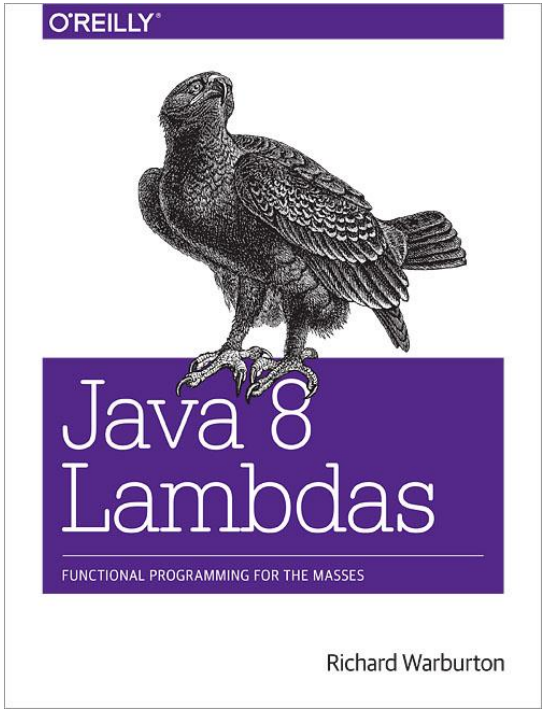
Now everyone is friends

Increasingly Multiparadigm

- Established languages going multiparadigm
  - Java 8 - Generics + Lambdas
  - C++ - Templates, Lambdas

- Newer Languages are multi paradigm
  - F#
  - Ruby/Python/Groovy can be functional
  - New JVM languages:
    - Scala
    - Ceylon
    - Kotlin

http://java8training.com



http://is.gd/javalambdas

@richardwarburto

http://insightfullogic.com