Selfish Purity

THE SCIENCE OF

LAZINESS

Outline

- What is Functional Programming?
- Reasonability
- Testability
- Concurrent…ability

Insanity: doing the same thing over and over again and expecting different results.

Albert Einstein

IOP

```
launchMissiles();          // => destroys North Korea
```

IOP

```
launchMissiles();          // => destroys North Korea
launchMissiles();          // => throws OutOfMissilesError
```

IOP

```
LaunchStatus status = launchMissiles();

status;
status;
```

```
public interface LaunchDoer {
  public LaunchStatus unsafeLaunch();
}


LaunchDoer doer = launchMissiles();

doer;
doer;
doer;

// no missiles have been launched!
```

FP

- Referential Transparency

FP

- Referential Transparency
- Equational Reasoning

FP

- Referential Transparency
- Equational Reasoning

$$t_1 = t_1' \implies t_2 = t_2[t_1 \mapsto t_1']$$

FP

- Referential Transparency
- Equational Reasoning

$$t_1 = t_1' \implies t_2 = t_2[t_1 \mapsto t_1']$$

- Rearrange your code without fear!

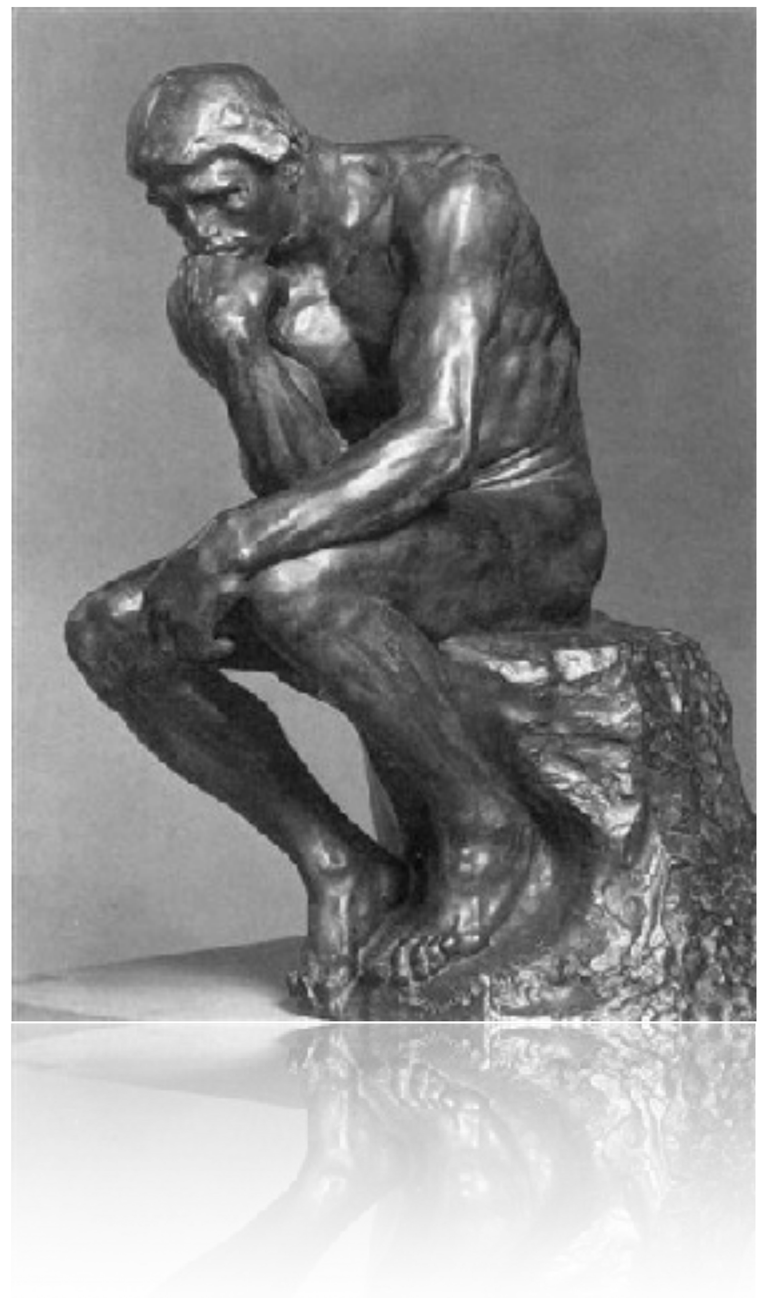
FP

- Referential Transparency
- Equational Reasoning

$$t_1 = t_1' \implies t_2 = t_2[t_1 \mapsto t_1']$$

- Rearrange your code without fear!
    - Less "magic" to remember

Reasonability



```java
public class ConcreteCementService
    extends AbstractCementService {

  @Springy
  public void pour(int amount) {
    dao.startTransaction();
    checkInventory(amount);
    setInventory(getInventory() - amount);

    if (!dao.commit() && !dao.checkTimeout()) {
      dao.rollback();
      pour(amount);
    }
  }
}
```

```java
public class ConcreteCementService
    extends AbstractCementService {

  @Springy
  public void pour(int amount) {
    dao.transact(new Transaction() {
      public void run() {
        checkInventory(amount);
        setInventory(getInventory() - amount);
      }
    }, TIMEOUT);
  }
}
```

```java
public class ConcreteCementService
    extends AbstractCementService {

  public void pour(int amount) {
    dao.transact(new Transaction() {
      public void run() {
        if (checkInventory(amount)) {
          setInventory(getInventory() - amount);
        } else {
          dao.retryTransaction();
        }
      }
    }, TIMEOUT);
  }
}
```

Small vs Large

- **Imperative**

Small vs Large

- **Imperative**
    - Instructions are stateless and predictable

Small vs Large

- **Imperative**
    - Instructions are stateless and predictable
    - Services are stateful and require care

Small vs Large

- **Imperative**
  - Instructions are stateless and predictable
  - Services are stateful and require care
- **Functional**

Small vs Large

- **Imperative**
  - Instructions are stateless and predictable
  - Services are stateful and require care
- **Functional**
  - Expressions are stateless and predictable

Small vs Large

- **Imperative**
    - Instructions are stateless and predictable
    - Services are stateful and require care
- **Functional**
    - Expressions are stateless and predictable
    - Expressions are stateless and predictable

Testability

```java
public interface IMessageService {
  public void init();
  public void enqueue(Message m);

  public void flush();      // <--- test this!
}
```

```java
public interface IMessageService {
  public void init();
  public void enqueue(Message m);

  public MessageSink flush();
}
```

```java
public interface MessageSink {
  public MessageSink write(Message m);
}


public class InMemoryMessageSink {
  public InMemoryMessageSink write(Message m) {
    // ...
  }

  public Message[] getMessages() {
    // ...
  }
}
```

Testability

- Side effects are hard to observe!

Testability

- Side effects are hard to observe!
    - Requires mocks and massive setup

Testability

- Side effects are hard to observe!
  - Requires mocks and massive setup
  - Still can miss things

Testability

- Side effects are hard to observe!
  - Requires mocks and massive setup
  - Still can miss things
- General pattern

Testability

- Side effects are hard to observe!
  - Requires mocks and massive setup
  - Still can miss things
- General pattern
  - Lift your uncontrolled side effects into data

Testability

- Side effects are hard to observe!
  - Requires mocks and massive setup
  - Still can miss things
- General pattern
  - Lift your uncontrolled side effects into data
  - Evaluate data in prod, *observe* data in tests

```java
public interface IFileSystem {
  public void createFile(String name);
  public void write(String name, byte[] data);
  public byte[] read(String name);
}
```

```java
public interface IFileSystem {
  public FSLogic createFile(String name);
  public FSLogic write(String name, byte[] data);
  public FSLogic read(String name, Consumer<byte[]> c);
}
```

```java
public interface FSLogic {
  // returns null if last instruction
  public FSLogic next();
}

public class TouchFile extends FSLogic {

  public TouchFile(String name, FSLogic next) { ... }

  public String getName() { ... }
}

public class WriteData extends FSLogic { ... }
public class ReadData extends FSLogic { ... }
public class DeleteData extends FSLogic { ... }
```

```java
public class ConcreteFileSystem extends IFileSystem {
  public FSLogic createFile(String name) {
    return new TouchFile(name, null);
  }

  public FSLogic write(String name, byte[] data) {
    return new WriteFile(name, data, null);
  }

  public FSLogic read(String name, Consumer<byte[]> c) {
    return new ReadFile(name, c, null);
  }
}
```

```
// writes things out to disk!
public void evaluate(FSLogic logic, File basePath) {
  ...
}
```

Testability

- Define an algebra of operations

Testability

- Define an algebra of operations
  - The simpler, the better!

Testability

- Define an algebra of operations
    - The simpler, the better!
- Write two interpreters for that algebra

Testability

- Define an algebra of operations
  - The simpler, the better!
- Write two interpreters for that algebra
  - "Real" interpreter performs effects

Testability

- Define an algebra of operations
  - The simpler, the better!
- Write two interpreters for that algebra
  - "Real" interpreter performs effects
  - "Fake" interpreter allows you to inspect

Testability

- Define an algebra of operations
  - The simpler, the better!
- Write two interpreters for that algebra
  - "Real" interpreter performs effects
  - "Fake" interpreter allows you to inspect
- Free monads make this *very* easy!

Concurrent…ability

Concurrency is hard.

Daniel Spiewak

```java
final Result[] results = {null, null};

Thread t1 = new Thread() {
  public void run() {
    results[0] = computeLeft(input);
  }
};

Thread t2 = new Thread() {
  public void run() {
    results[1] = computeRight(input);
  }
};

t1.join();
t2.join();

mergeResults(results[0], results[1]);
```

```
Future<Result> f1 = Future.future(new Callable<Result>() {
  public Result call() {
    return computeLeft(input);
  }
});


Future<Result> f2 = Future.future(new Callable<Result>() {
  public Result call() {
    return computeRight(input);
  }
});

f1.zip(f2).map(new Mapper<Tuple2<Result, Result>, Result>() {
  public Result apply(Tuple2<Result, Result> pair) {
    return mergeResults(pair._1(), pair._2());
  }
});
```

```scala
val f1 = future {
  computeLeft(input)
}

val f2 = future {
  computeRight(input)
}

f1 zip f2 map {
  case (left, right) => mergeResults(left, right)
}
```

```scala
val f1 = future {
  computeLeft(input)
}

val f2 = future {
  computeRight(input)
}


for {
  merged <- f1 zip f2 map {
    case (left, right) => mergeResults(left, right)
  }

  derived <- deriveResults(merged)
} yield derived
```

Control Flow

- Computation dependency

Control Flow

- Computation dependency
    - Sequential (`flatMap`/`for`-comprehensions)

Control Flow

- Computation dependency
  - Sequential (`flatMap`/`for`-comprehensions)
  - Parallel (`zip`)

Control Flow

- Computation dependency
  - Sequential (`flatMap`/`for`-comprehensions)
  - Parallel (`zip`)
- There's an abstraction for that™

Control Flow

- Computation dependency
  - Sequential (`flatMap`/`for`-comprehensions)
  - Parallel (`zip`)
- There's an abstraction for that™
  - Sequential = monads

Control Flow

- Computation dependency
  - Sequential (`flatMap`/`for`-comprehensions)
  - Parallel (`zip`)
- There's an abstraction for that™
  - Sequential = monads
  - Parallel = applicatives

Conclusion

- Behavior that differs situationally is confusing

Conclusion

- Behavior that differs situationally is confusing
- Data is easy to manipulate

Conclusion

- Behavior that differs situationally is confusing
- Data is easy to manipulate
    - Effects are not!

Conclusion

- Behavior that differs situationally is confusing
- Data is easy to manipulate
  - Effects are not!
- Think about control flow and concurrency follows