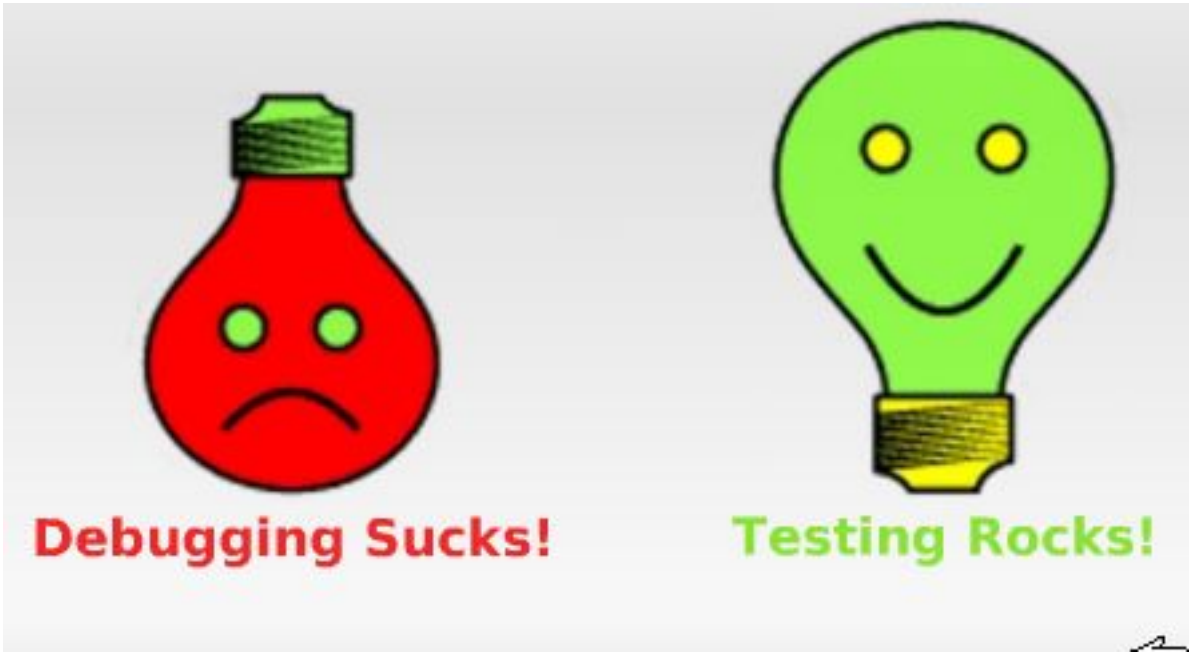


Unit Testing

Ran Tavory
outbrain
2009

The secret to Happiness

The secret to writing good tests:

Write TESTABLE CODE

We are so lucky

Testable == Readable == Reusable ==
Modular == Robust

Code

Decouple object construction from application logic

Wrong:

```
class House {
  private final Kitchen kitchen = new Kitchen();
  private boolean isLocked;
  private boolean isLocked() {
    return isLocked;
  }
  private boolean lock() {
    kitchen.lock();
    isLocked = true;
  }
}
```

Decouple object construction from application logic (2)

Right:

```
class House {
  private final Kitchen kitchen;
  private boolean isLocked;
  public House(Kitchen k) {
    kitchen = k;
  }
  private boolean isLocked() {
    return isLocked;
  }
  private boolean lock() {
    kitchen.lock();
    isLocked = true;
  }
}
```

Dependency Injection and The Law of Diameter

Wrong:

```
class Mechanic {
 Engine engine;
 Mechanic(Context context) {
  engine = context.getEngine();
 }
}
```

Right:

```
class Mechanic {
 Engine engine;
 Mechanic(Engine eng) {
  engine = eng;
 }
}
```

Say NO to Globals;
Beware of the Singleton

- Globals are bad, we all know that
- They are particularly bad in tests
- Tests fail together but problems can not be reproduced in isolation.
- Order of the tests matters.
- The APIs are not clear about the order of initialization and object instantiation
- more...

- **Singletons are Globals in Disguise.**

Favor **Composition** over *Inheritance*

- **Inheritance != Code Reuse**

- Use inheritance only where polymorphism is required
- At run-time you can not chose a **different inheritance**, but you can chose a **different composition**
  - Inheriting from *AuthenticatedServlet* will make your sub-class very hard to test since every test will have to mock out the authentication

Favor polymorphism over conditionals

- Switch statement => Polymorphism
- Repeated conditions => Polymorphism

Key point:

Many simple classes are much easier to test then one complex class

Service object and **Value objects**

- Value objects are:
  - Data beans
  - Very easy to construct
  - Dumb setters/getters
  - Never mocked
  - Don't need an interface
  - Leaf objects
- Service objects:
  - Do the interesting work
  - Their constructors ask for lots of other objects for collaboration
  - Good candidates for mocking
  - Tend to have an interface and tend to have multiple implementations

Do not mix them!

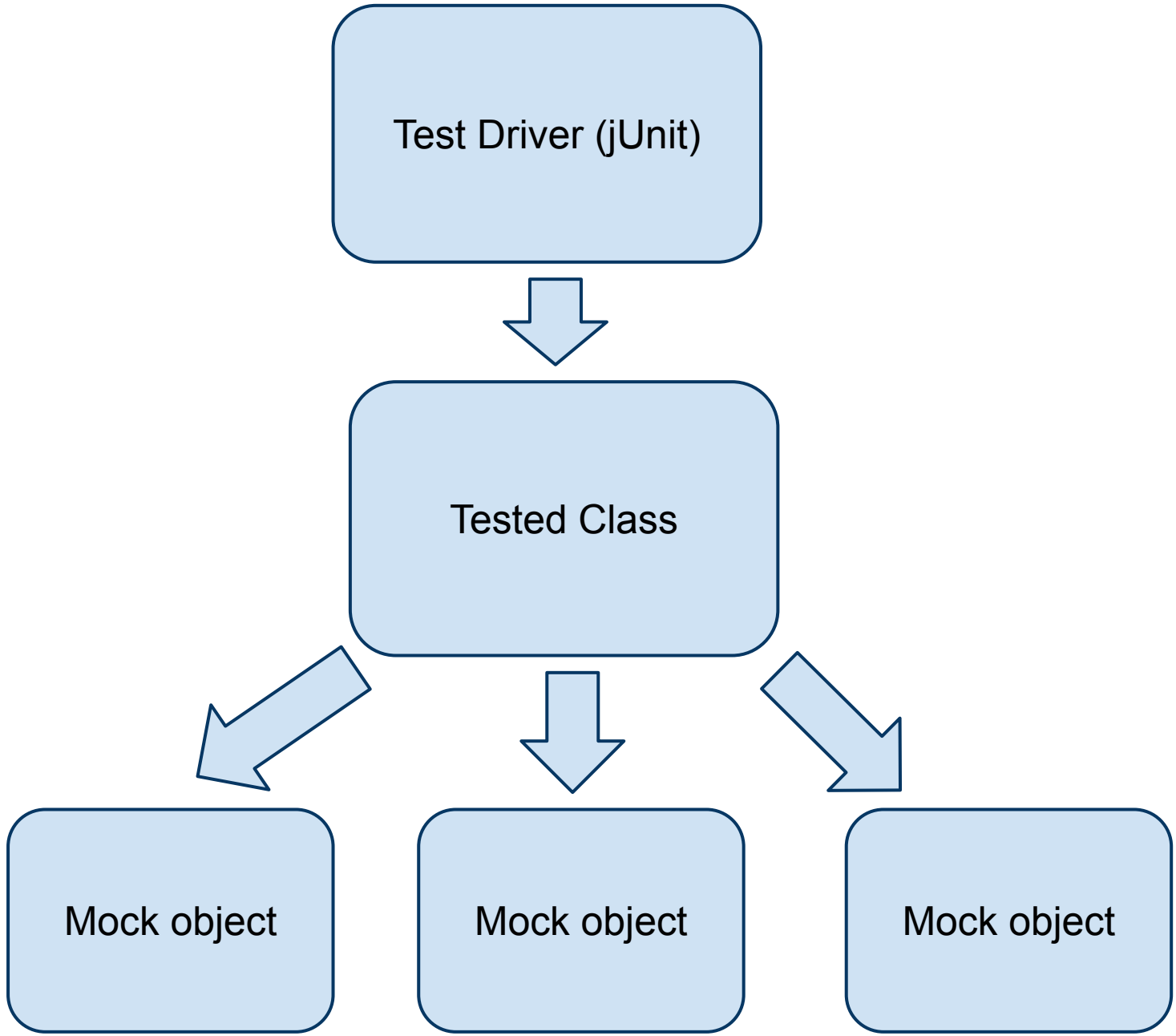
Service object and **Value objects** (2)

- **Value objects** :
  - Never take a service object in the constructor.
  - Easily constructed using `new`
  - Testing is very easy.
- **Service objects**:
  - Constructed using a factory or a DI framework
  - Testing is hard. Use a mocking system to assist

Do not make **Shatnez**

- A class should do one and only one coherent thing
- Signs of poor design:
  - A class has **And** in it's name (ReaderAndWriter)
  - Reading public method names is not enough to understand
  - Class has data members that are only used in some methods, or some scenarios
  - Class has static methods that only operate on parameters.
- Shatnez classes are harder to test.

The big picture



Unit vs. Integration

Unit Tests:

- Keeps testing close to the relevant code
- Relatively easy to test all code paths
- Easy to see if someone inadvertently changes the behavior of a method
- Great as documentation for your classes and methods
- Speed Speed Speed
- Unit testing is a **development** tool
- Much harder to write for UI components than for non-GUI

Integration Tests:

- It's nice to have nuts and bolts in a project, but integration testing makes sure they fit each other
- Harder to localize source of errors
- Harder to tests all (or even all critical) code paths
- Harder to maintain
- Harder to run (requires a full fledged environment, DB, web server

Tools - Mockito


```
import static org.mockito.Mockito.*;

//mock creation:
List mockedList = mock(List.class);

//using mock object - doesn't throw any "unexpected interaction" exception:
mockedList.add("one");
mockedList.clear();

//selective & explicit verification:
verify(mockedList).add("one");
verify(mockedList).clear();
```

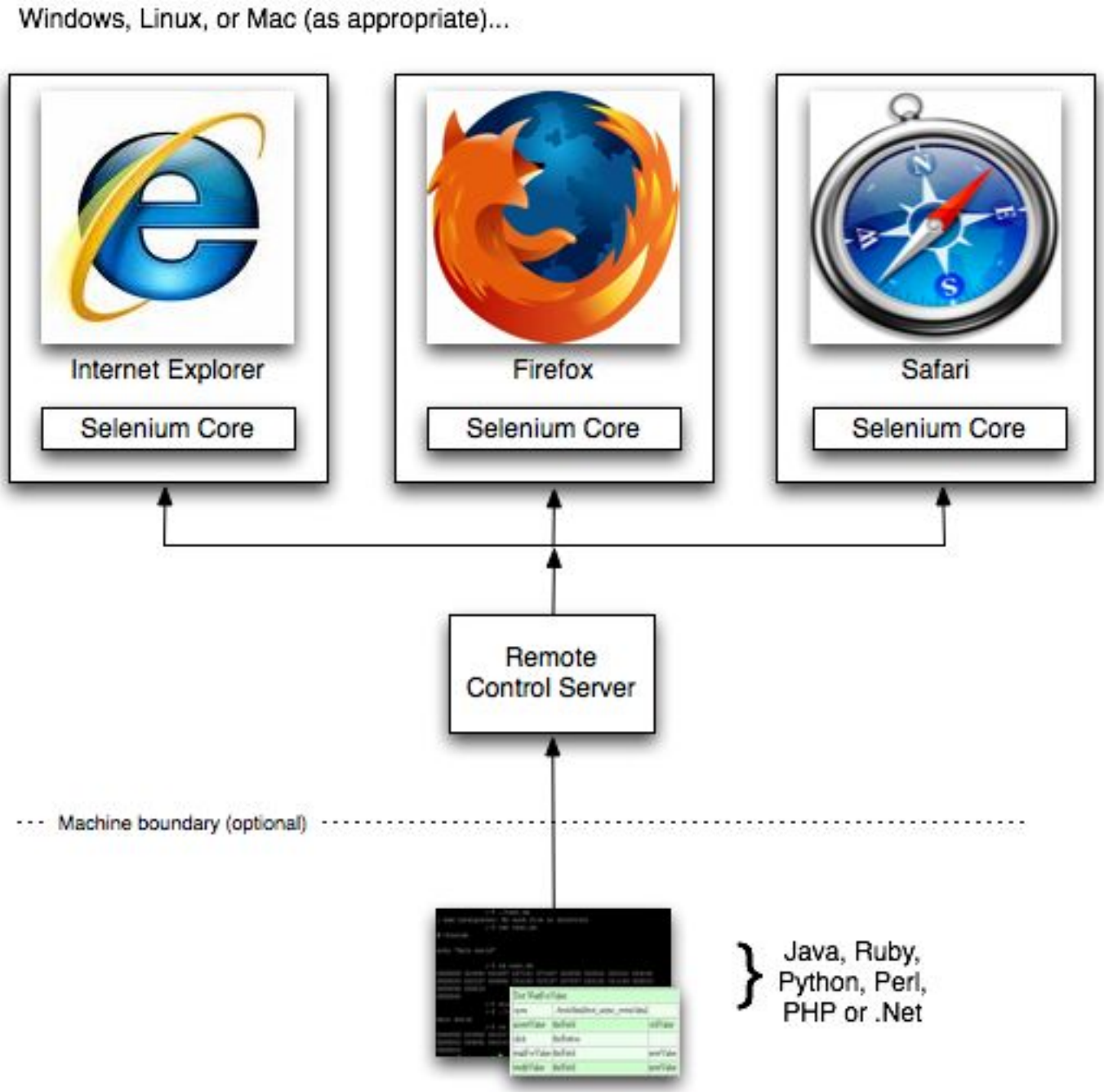
```
//You can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

//stubbing - before execution
when(mockedList.get(0)).thenReturn("first");

//following prints "first"
System.out.println(mockedList.get(0));

//following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

Tools - Selenium



References

- Writing Testable Code
- How to Think About the "new" Operator with Respect to Unit Testing
- Program to an interface, not an implementation
- Law of Demeter
- Breaking the Law of Demeter is Like Looking for a Needle in the Haystack
- Beware of the Singleton
- Mockito