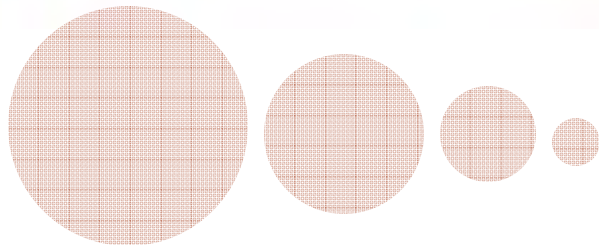


Complementing Unit Test with Dependency Injection and Mock Objects



IndicThreads.com Conference On Java Technology
26th & 27 Oct. 2007 Pune, India



Paulo Caroli & Siddharth Dawara
ThoughtWorks

About Paulo Caroli

The ThoughtWorks logo is displayed in white text on a dark purple rectangular background.

- A technologist at ThoughtWorks US
- Master's Degree in Software Engineering
- Sun Certified J2EE Architect
- More than 13 years in SW Development.
- Agile Techniques, such as XP, TDD, Continuous Integration
- Object Oriented development practices
- More at www.caroli.org



About Siddharth Dawara

The logo for ThoughtWorks, featuring the company name in white text on a dark purple background with a subtle pattern.

- A geek at ThoughtWorks India
- Web technologies fan
- Agile Techniques, such as XP, TDD, Continuous Integration
- Object Oriented development practices
- More at sdawara@thoughtworks.com



Agenda

- Unit Test
- Unit Test and TDD
- Testing Dependent Components
- Dependency Injection
- Mock Objects
- Case Study
- Conclusion
- Q & A



Unit Test



Unit Test

“Unit Test is a procedure used to validate that individual units of functional code are working properly. “



Unit Test

- Is usually done by developers
- Improves quality
- Facilitates changes (refactoring)
- Simplifies integration
- Enables automation
- Provides effective system documentation.
- Makes your life simpler



JUnit

- A open source regression testing framework
- Written by Erich Gamma and Kent Beck
- Used by developers to implement unit tests in Java
- Part of XUnit family



JUnit

- Provides tools for:
 - assertions
 - running tests
 - aggregating tests (suites)
 - reporting results



Sample Functional Code

```
public class Foo {  
    public Foo() {}  
  
    public String helloWorld() {  
        return "Hello World";  
    }  
  
    public boolean truth() {  
        return true;  
    }  
}
```



How to Test it?

- Debugger
 - Needs human intervention, slow, boring
- Print statements
 - Needs human intervention, slow, polluted code, "Scroll Blindness"
- Use JUnit
 - Automatic, quick, no code pollution, fun



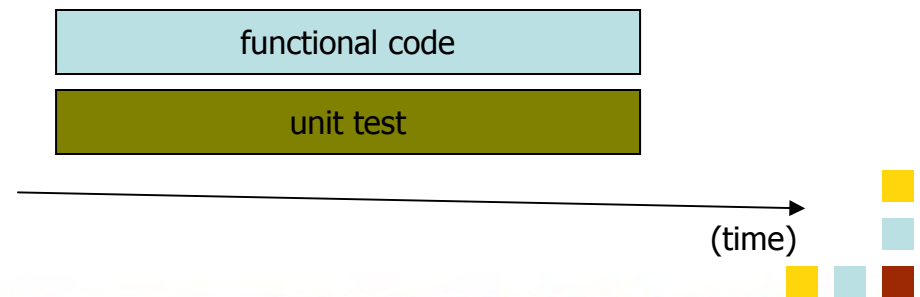
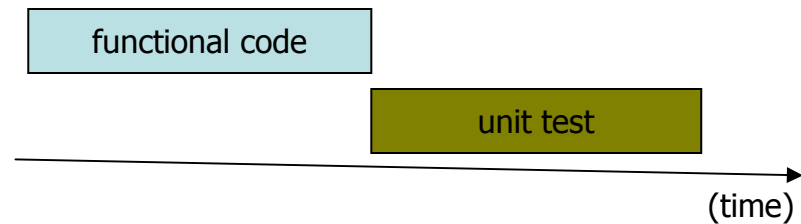
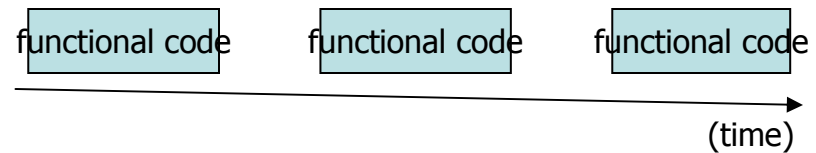
Sample JUnit Code

```
public class FooTest extends TestCase {  
    private Foo foo = new Foo();  
    public void testHelloWorld() {  
        assertEquals("Hello World",  
                    foo.helloWorld());  
    }  
    public void testTruth() {  
        assertTrue(foo.truth());  
    }  
}
```



When to Unit Test?

- No Unit Test
- Test Last development (TLD)
 - You'll spend less time overall
- Test First Development (TFD)
 - Time savings on debugging



TDD



Test Driven Development

“Test-Driven Development (TDD) is an evolutionary approach to development which instructs you to have test-first development intent. Basically, you start by writing a test and then you code to elegantly fulfill the test requirements.”

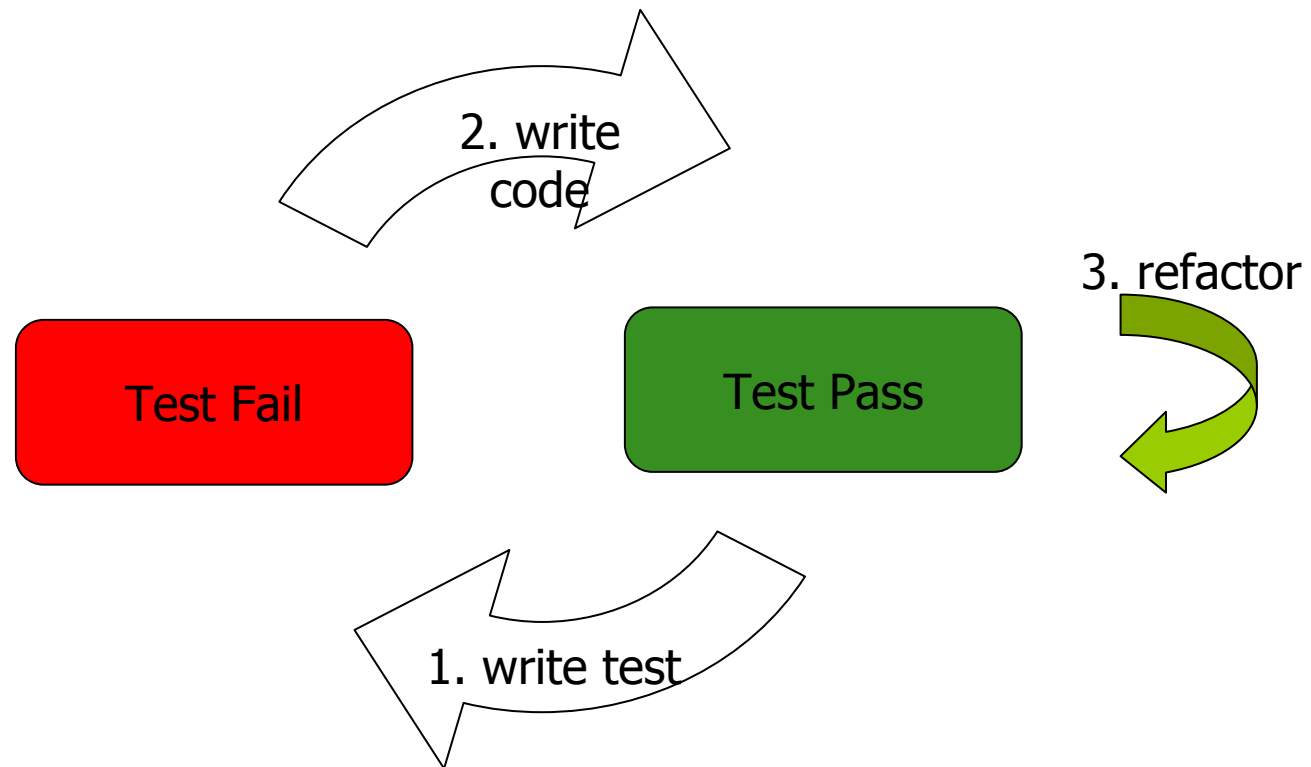


Test Driven Development

- Small successful, tested steps.
- Do the simplest thing that could possibly work.
- The goal is to produce working clean code that fulfills requirements



TDD Cycle



Step 1 – Write Test Code

- Guarantees that every functional code is testable
- Provides a specification for the functional code
- Helps to think about design
- Ensure the functional code is tangible



Step 2 – Write Functional Code

- Fulfill the requirement (test code)
- Write the simplest solution that works
- Leave Improvements for a later step
- The code written is only designed to pass the test
 - no further (and therefore untested code is not created)



Step 3 – Refactor

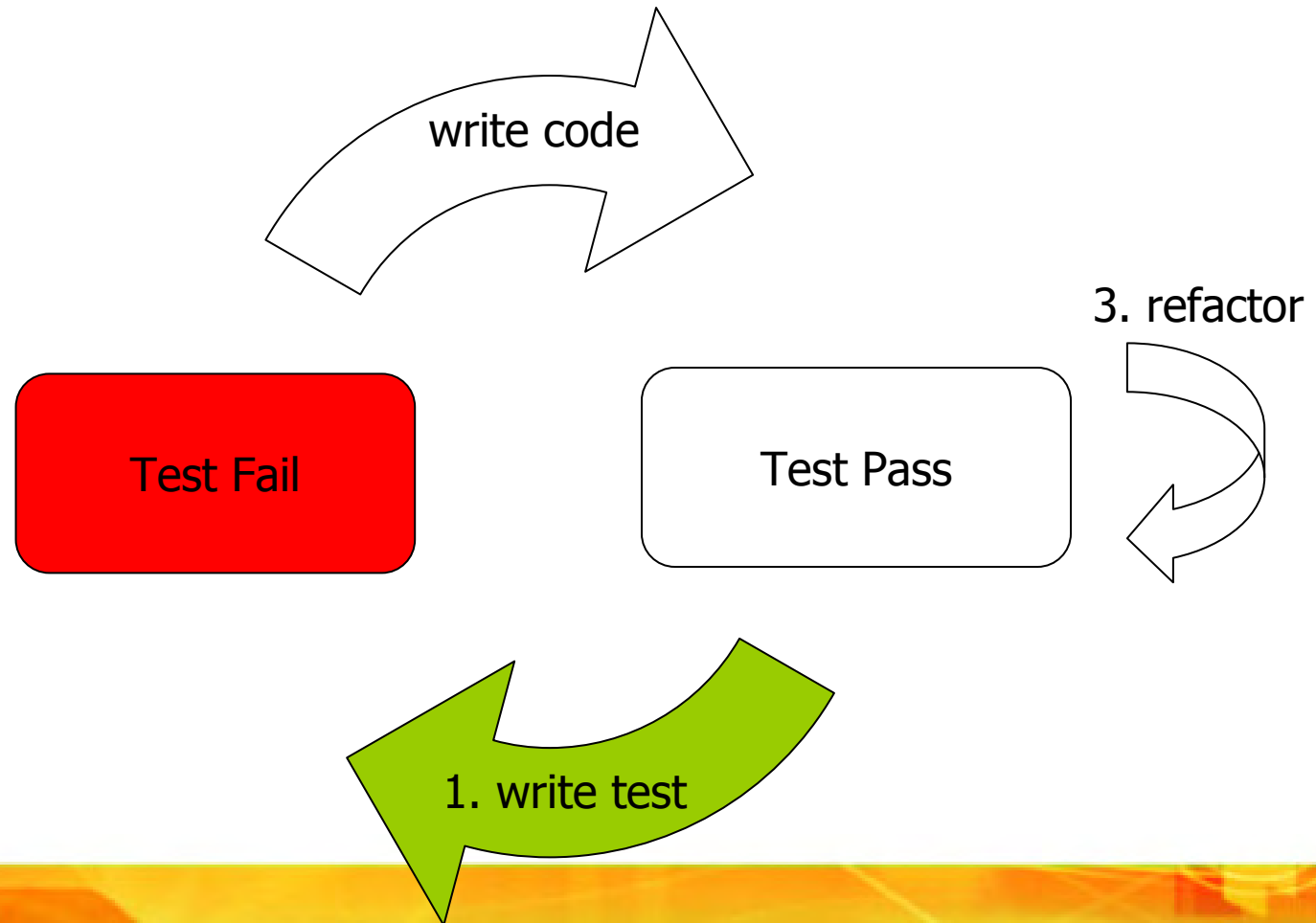
- Clean-up the code (test and functional)
- Make sure the code expresses intent
- Remove code smells
- Re-think the design
- Delete unnecessary code



TDD helps you produce
clean working code that fulfills requirements



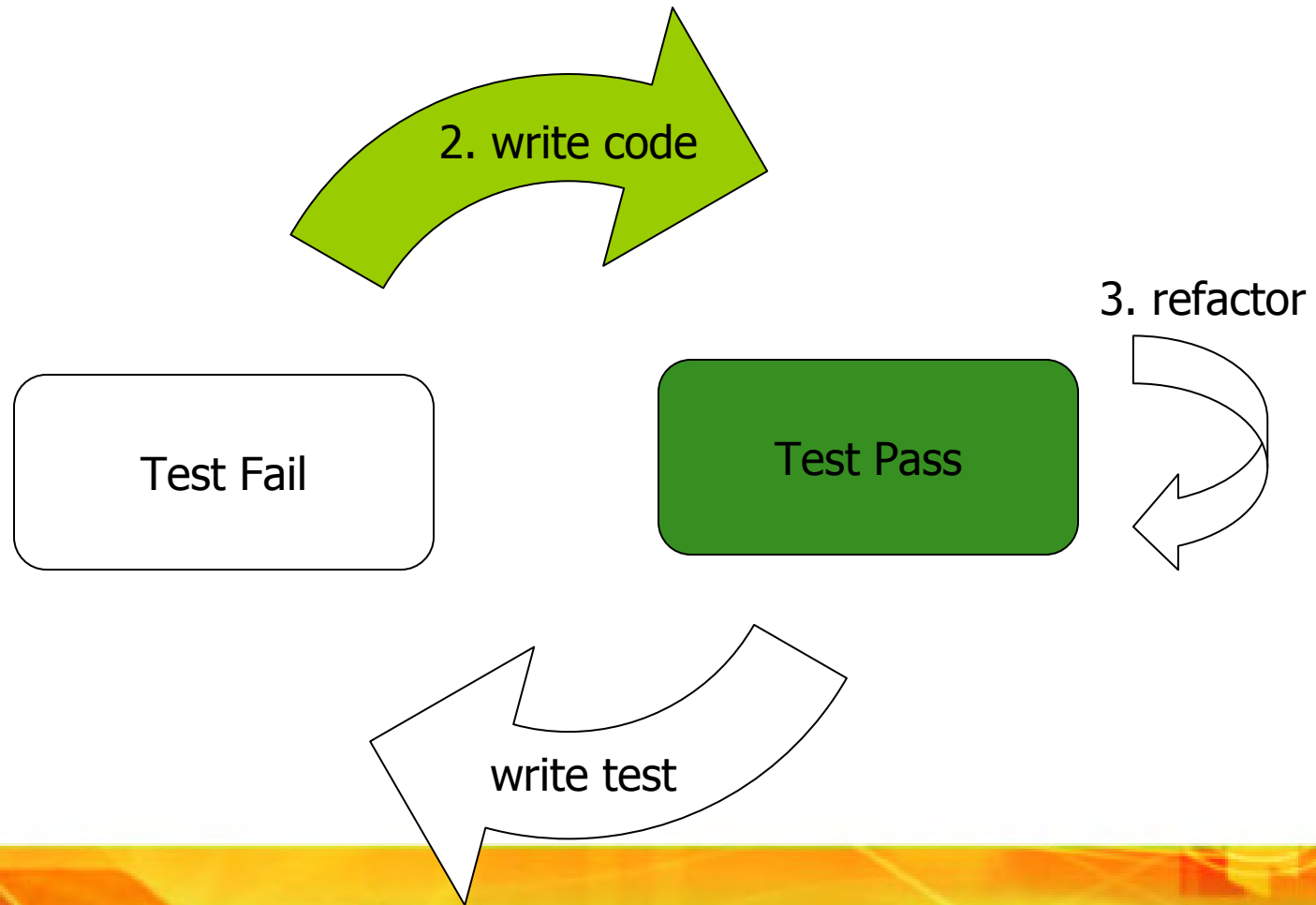
TDD Cycle – Step 1



code that fulfills requirements



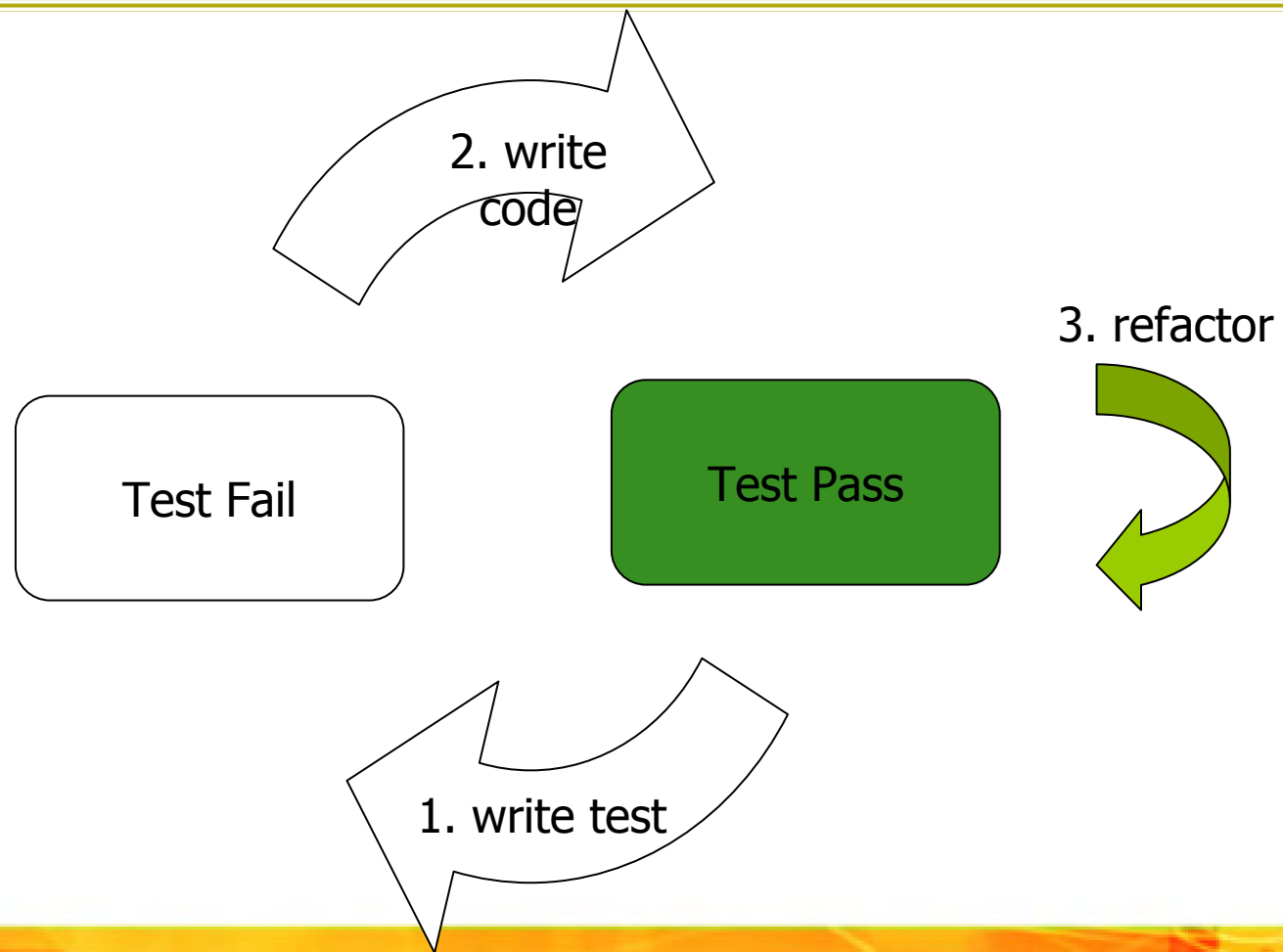
TDD Cycle – Step 2



working code that fulfills requirements



TDD Cycle – Step 3



clean working code that fulfills requirements



Why TDD

- Non TDD way:
lots of code

- TDD way:
clean working code that fulfills requirements



TDD Demo



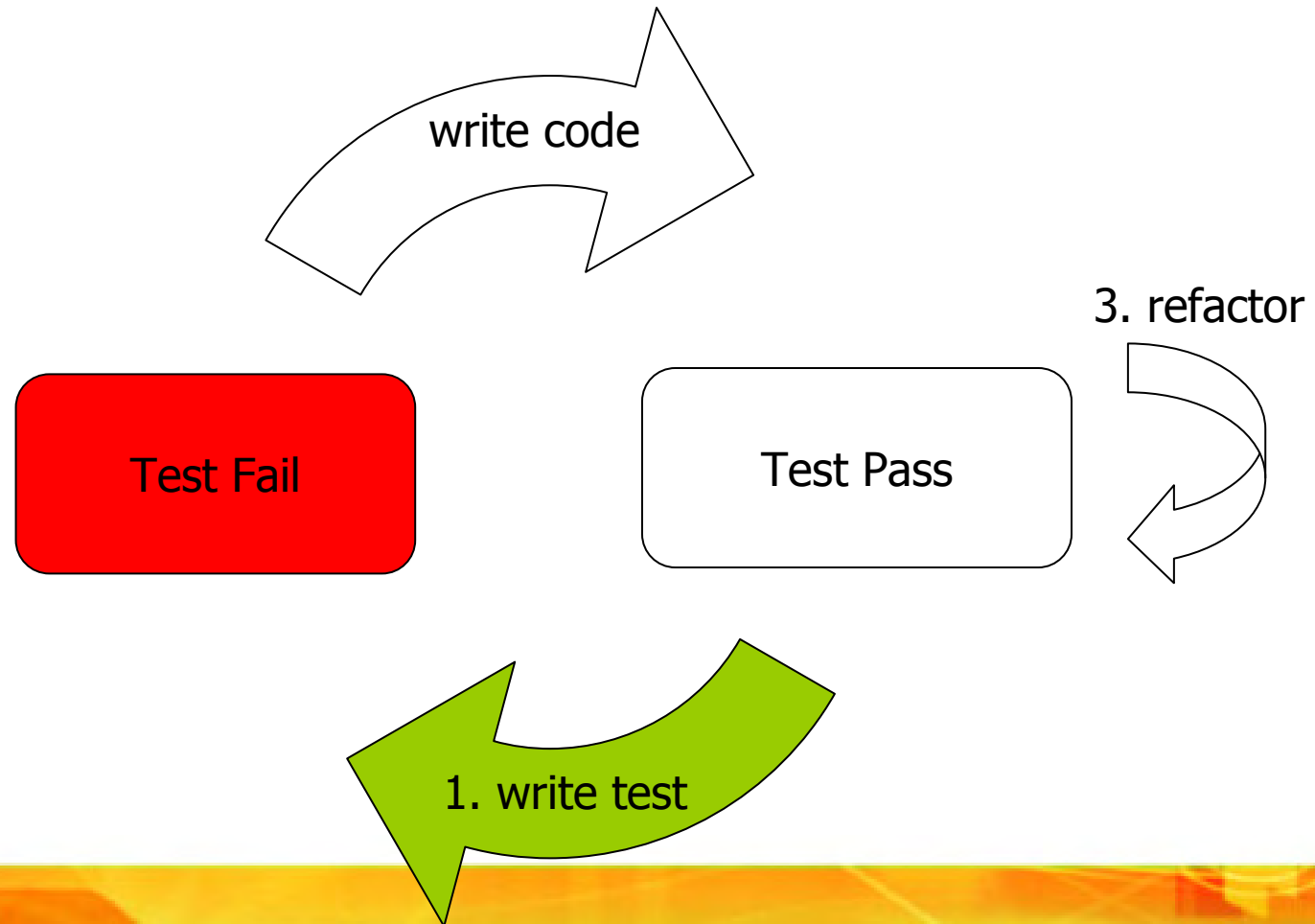
Task: build a factorial calculator

- Factorial

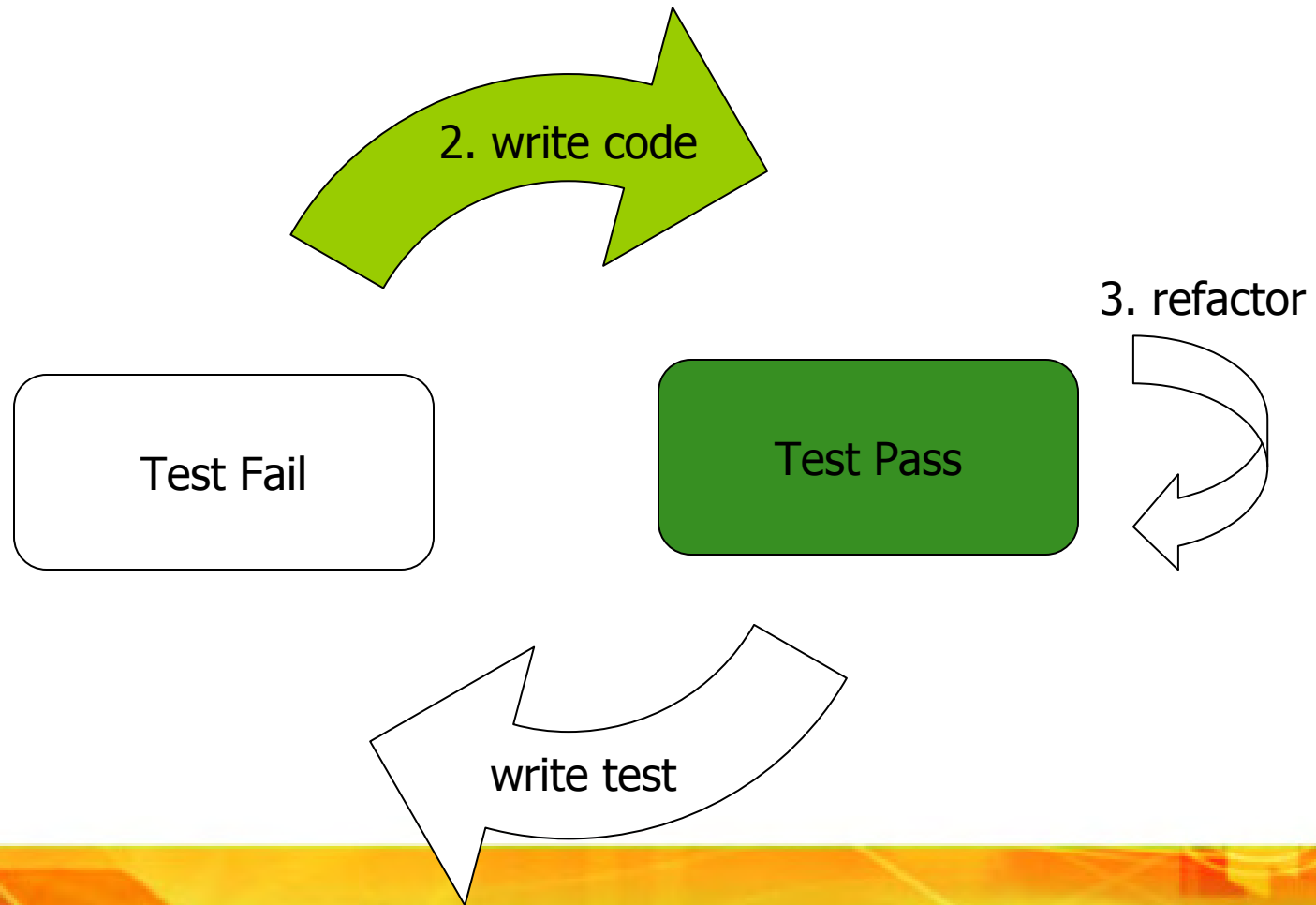
- $0! = 1$
- $1! = 1$
- $2! = 2 = 2*1$
- $3! = 6 = 3*2*1$
- $4! = 24 = 4*3*2*1$
- $5! = 120 = 5*4*3*2*1$
- ...



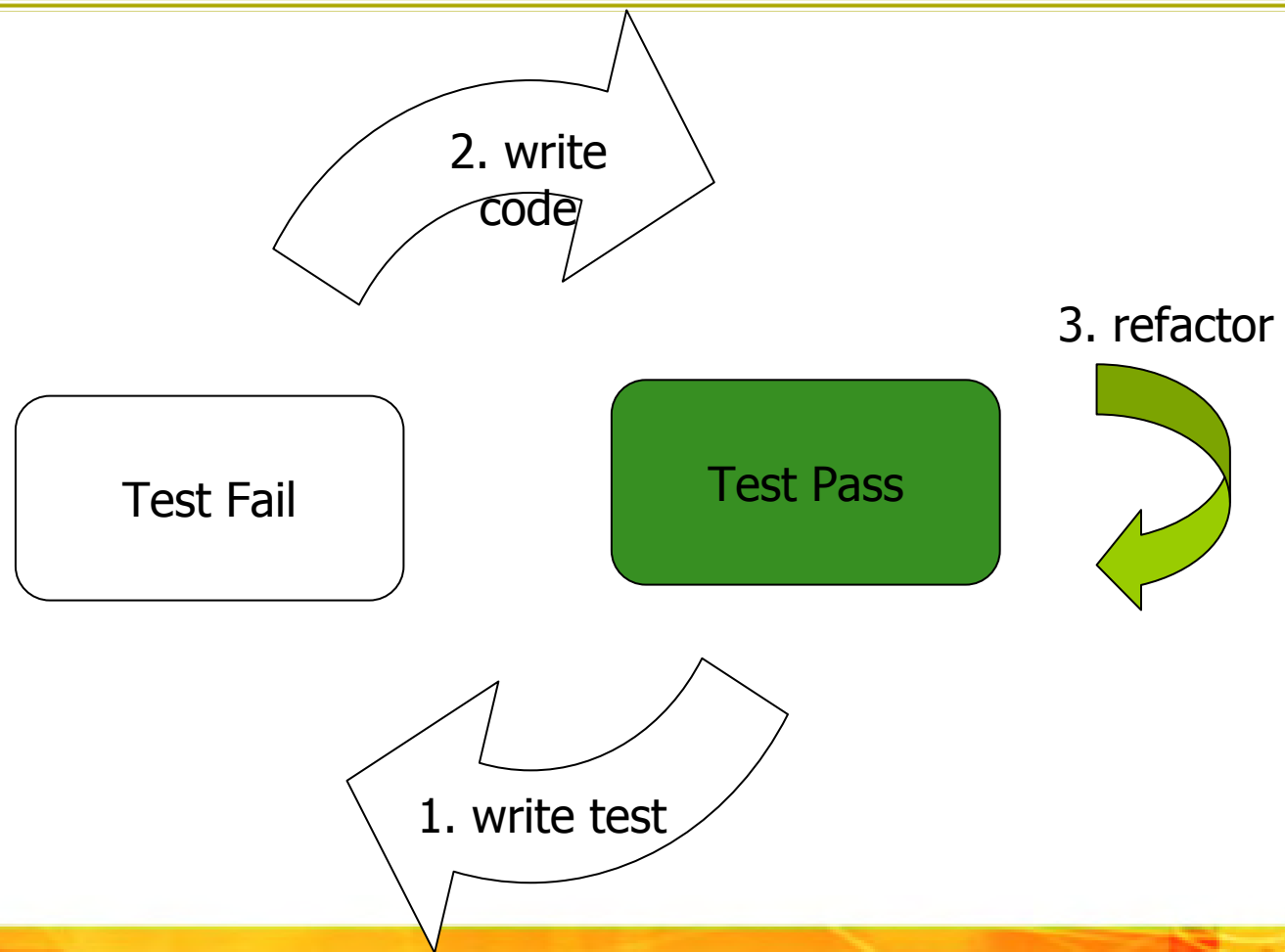
TDD Cycle – Step 1



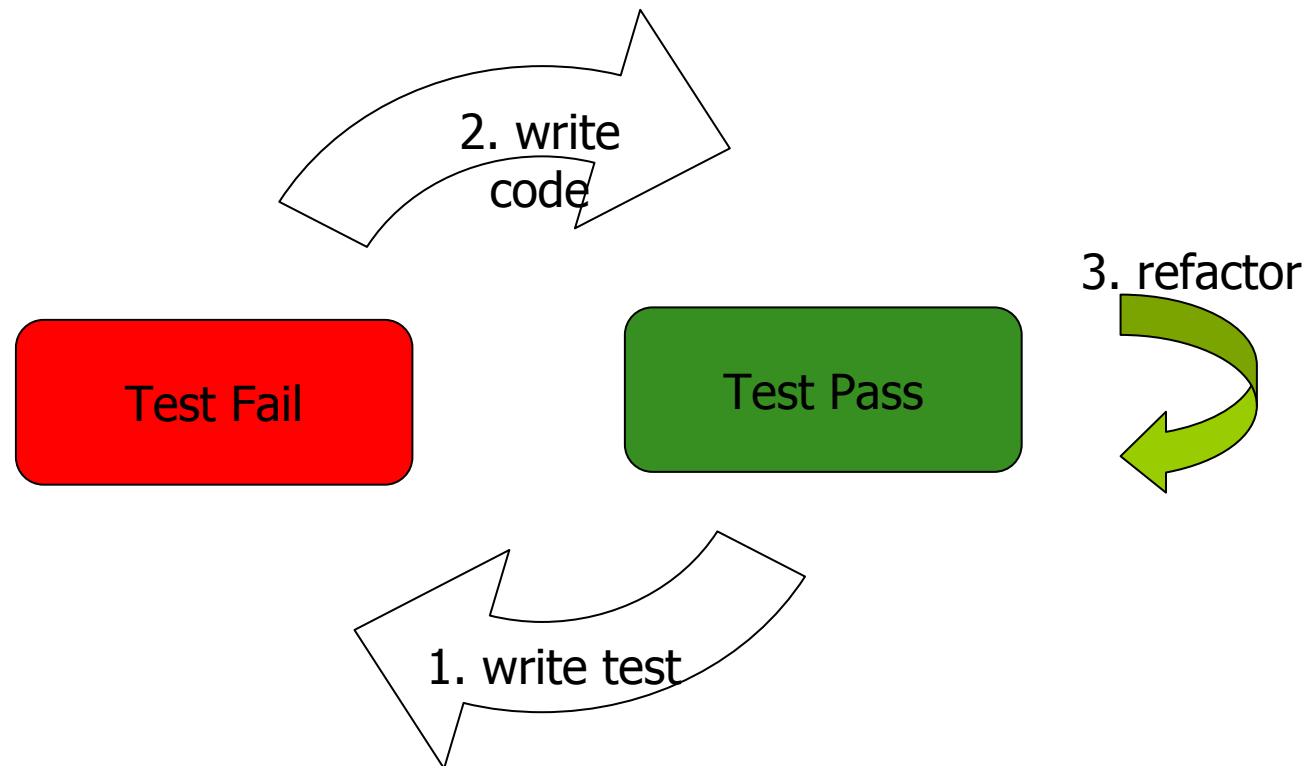
TDD Cycle – Step 2



TDD Cycle – Step 3

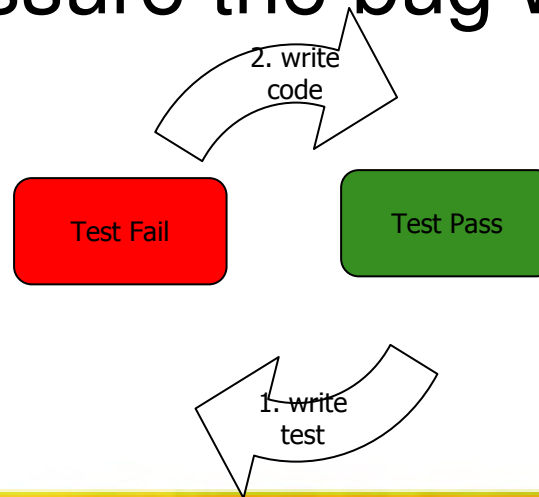


TDD Cycle Repeated



TDD - Bug Fixing

- Create new test to show bug
- Now, fix the bug and watch the bar go green
- Your tests assure the bug won't reappear



Testing Dependent Components

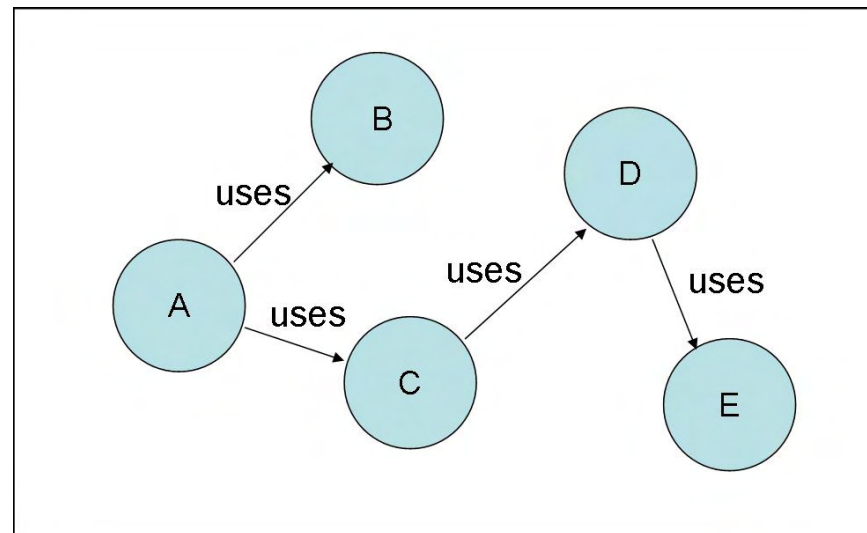


But how do I Unit
Test Object
Graphs, Circular
Dependency...



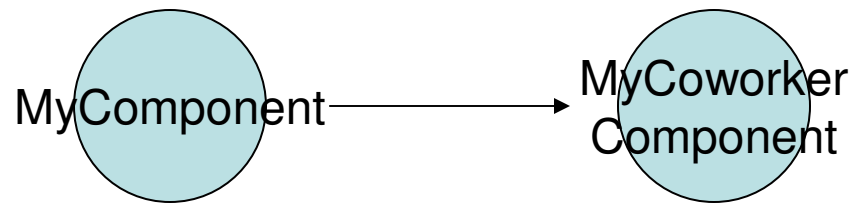
Object Graph

- How to test component A?
- TestA Unit Test, but what about B, C, D...



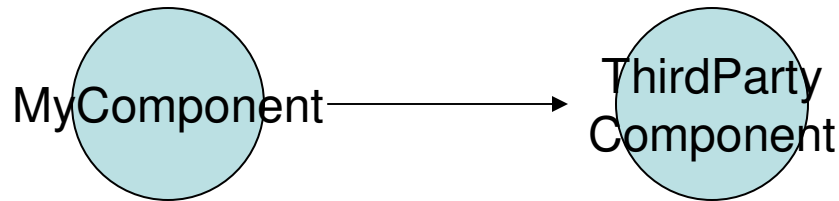
My Coworker Component

- Does MyComponentTest have to wait on MyCoworkerComponent to be tested?



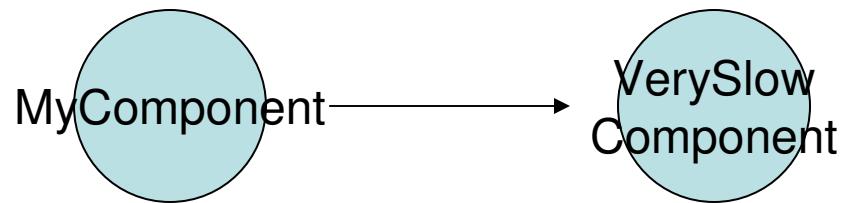
Third Party Component

- Does MyComponentTest have to wait on ThirdPartyComponent to be tested?



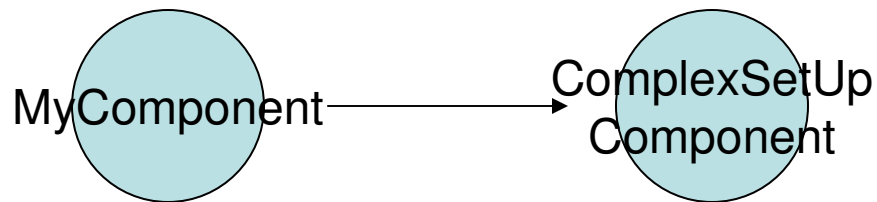
Very Slow Component

- Does MyComponentTest have to wait on VerySlowComponent to be tested?



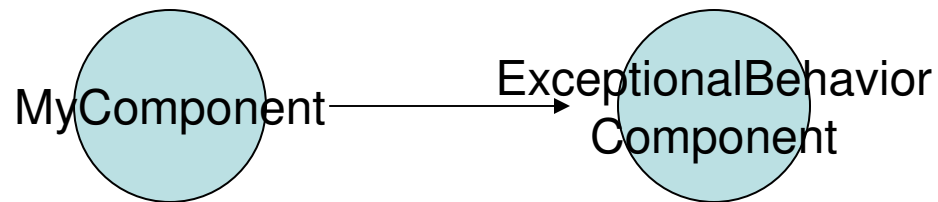
Component with Complex Set up

- Does MyComponentTest really have to setup with ComplexSetUpComponent?



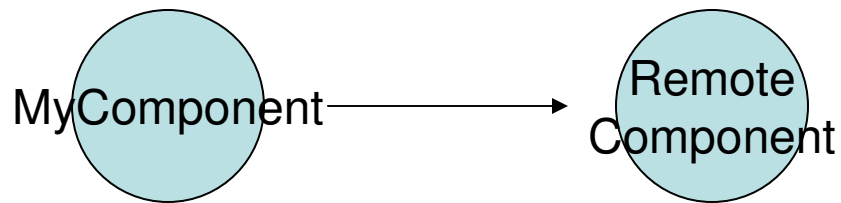
Component with Exceptional Behavior

- How to test MyComponent for other component exceptional behavior?



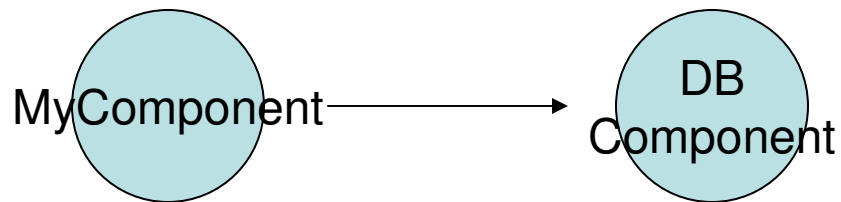
Remote Component

- Does MyComponentTest have to wait on RemoteComponent to be tested?



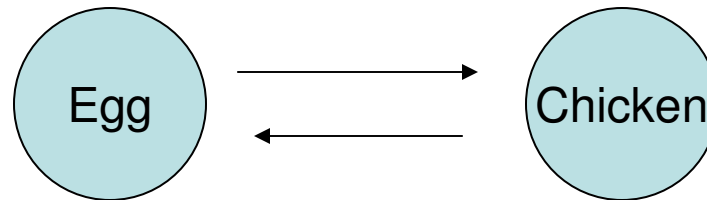
DB Component

- Does MyComponentTest have to wait on DBComponent to be tested?



Circular Dependency

- How to test circular dependency?



Dependency
Injection and
Mock Objects



Dependency Injection



Dependency Injection

“Dependency injection avoids dependencies on the implementations of collaborating classes by depending only on interfaces that those classes adhere to.”



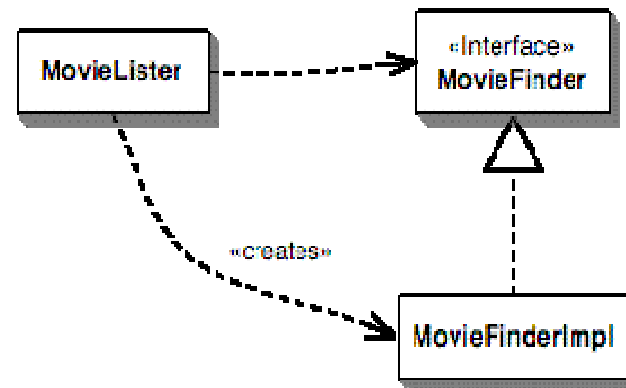
Dependency Injection

- Helps design loosely coupled components
- Improves testability
- Simplifies unit testing
- Increases flexibility and maintainability
- Minimizes repetition
- Supplies a plug architecture
- Relies on interfaces



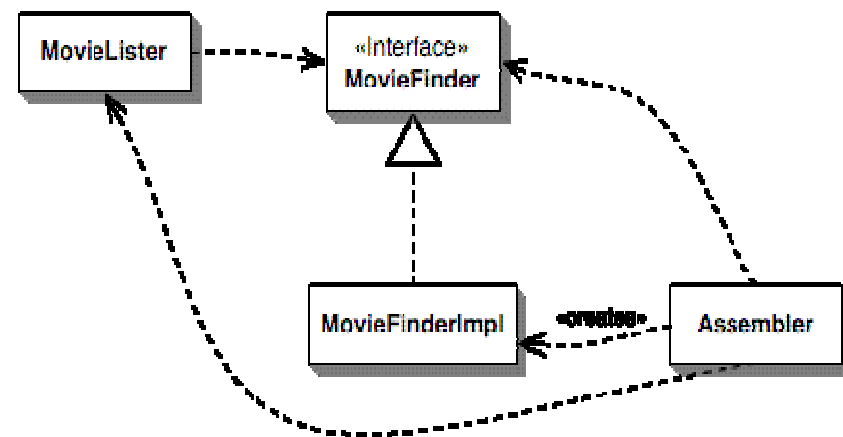
Without Dependency Injection

- MovieLister creates MovieFinderImpl
- MovieLister knows where to find MovieFinderImpl
- MovieLister chooses MovieFinderImpl
- Hard to test MovieLister



With Dependency Injection

- Assembler creates MovieFinderImpl
- Assembler injects MovieFinderImpl into MovieLister
- MovieLister does not choose the concrete implementation of MovieFinder
- Easier to test MovieLister



Constructor Dependency Injection

```
public interface Service {  
    public void doSomething();  
}  
  
public class Client {  
    private Service service;  
    public Client(Service service) {  
        this.service = service;  
    }  
  
    public void doIt() {  
        service.doSomething();  
    }  
}
```



Setter Dependency Injection

```
public interface Service {  
    public void doSomething();  
}  
  
public class Client {  
    private Service service;  
    public Client() { }  
  
    public setService (Service service) {  
        this.service = service;  
    }  
  
    public void doIt() {  
        service.doSomething();  
    }  
}
```



Dependency Injection Containers

- Pico Container
 - <http://www.picocontainer.org/>
- Spring Frameworks
 - <http://www.springframework.org/>
- Guice
 - <http://code.google.com/p/google-guice/>

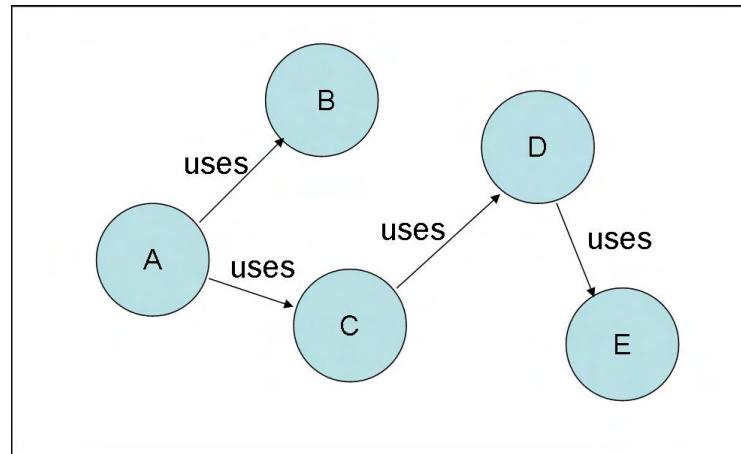


Mock Objects



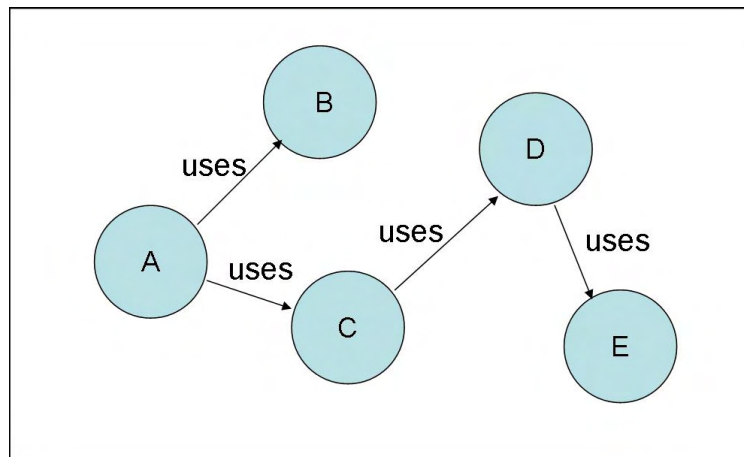
Mock Objects

- How to test component A?
- TestA Unit Test, but what about B, C, D...

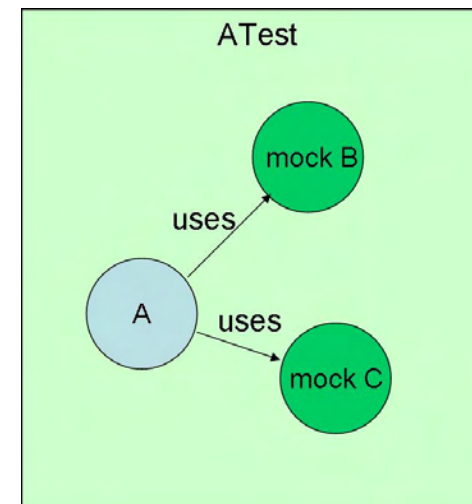


Mock Objects

- Use a Unit Test Framework – JUnit - for A unit test
- Use a Mock Object Framework for mocking B and C.



unit test



Mock Objects

“Mock Objects are stubs that mimic the behavior of real objects in controlled ways. “



Mock Objects

Mock Objects will enable your unit test to mimic behavior and verify expectations on dependent components.



Mock Objects

Use Mock Objects when testing dependent components, for example:

Object Graph, Circular Dependency, your Coworker Component, a Third Party Component, a Very Slow Component, a Component with Complex Set up, a Component with Exceptional Behavior, a Remote Component, a DB Component



Greetings Sample



Multi-language Greeting System

- `greetings.sayHello ("English", "John");`
 - "Hello John"
- `greetings.sayHello ("Italian", "John");`
 - "Ciao John"
- `greetings.sayHello ("Hindi", "John");`
 - "Namaste John"



Greeting Class

```
public class Greeting {  
  
    private ITranslator translator;  
  
    public Greeting(ITranslator translator) {  
        this.translator = translator;  
    }  
  
    public String sayHello(String language, String name) {  
        return translator.translate(  
            "English", language, "Hello") + " " + name;  
    }  
}
```



ITranslator Interface

```
public interface ITranslator {  
  
    String translate(  
        String fromLanguage ,  
        String toLanguage,  
        String word);  
  
}
```



How to test Greeting?



Complement your Unit Test with **Dependency Injection** and **Mock Objects**



Common Mock Objects Frameworks Functionality

1. Set-up
2. Instantiate the mock object
3. Inject the mock object
4. Set expectations and behavior for the mock object
5. Verify expectations for the mock object



JMock 2.2.0



1. Set-up

```
import junit.framework.TestCase;
import org.jmock.Mockery;
import org.jmock.Expectations;

public class GreetingTest extends TestCase {
    public void testGreetingInAnyLanguage() throws Exception {
        // set up
        Mockery context = new Mockery();
        ...
    }
}
```



2. Instantiate the mock object

```
import junit.framework.TestCase;
import org.jmock.Mockery;
import org.jmock.Expectations;

public class GreetingTest extends TestCase {
    public void testGreetingInAnyLanguage() throws Exception {
        // set up
        Mockery context = new Mockery();
        final ITranslator mockTranslator =
        context.mock(
```



3. Inject the mock object

```
import junit.framework.TestCase;
import org.jmock.Mockery;
import org.jmock.Expectations;

public class GreetingTest extends TestCase {
public void testGreetingInAnyLanguage() throws Exception {
    // set up
    Mockery context = new Mockery();
    final ITranslator mockTranslator =
    context.mock(
```



4. Set expectations and behavior for the mock object

```
public class GreetingTest extends TestCase {
public void testGreetingInAnyLanguage() throws Exception {
    // set up
    Mockery context = new Mockery();
    final ITranslator mockTranslator =
    context.mock(
```



5. Verify expectations for the mock object

```
...  
Mockery context = new Mockery();  
final ITranslator mockTranslator =  
context.mock(  

```

```
ITranslator.class);
```

```
Greeting greeting = new Greeting(mockTranslator);
```

Pune, India



**Mock Objects Sample at:
www.mocksamples.org**



Dependency Injection Containers Demo



Dependency Injection Containers Sample at: www.helloopensource.org



Session Learning's



Session Learning's

- Understand the need for Mock Object frameworks
- Peek into the basics of JMock - a Mock Object Open Source framework
- Understand Dependency Injection and how it improves testability



Session Learning's

- Distinguish Constructor and Setter Dependency Injection
- Understand and experience TDD as applied to more complicated unit tests.
- Glimpse of the leading Open Source Dependency Injection Frameworks Spring, PicoContainer and Guice



Questions?

THANK YOU!

