Test-Driven Development is a Paradox

A Talk for Programmers

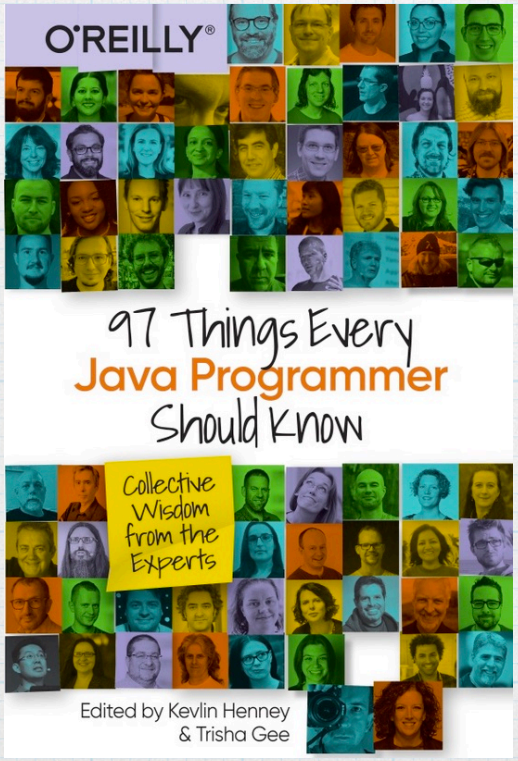
Burk Hufnagel - Solution Architect
Daugherty Business Solutions

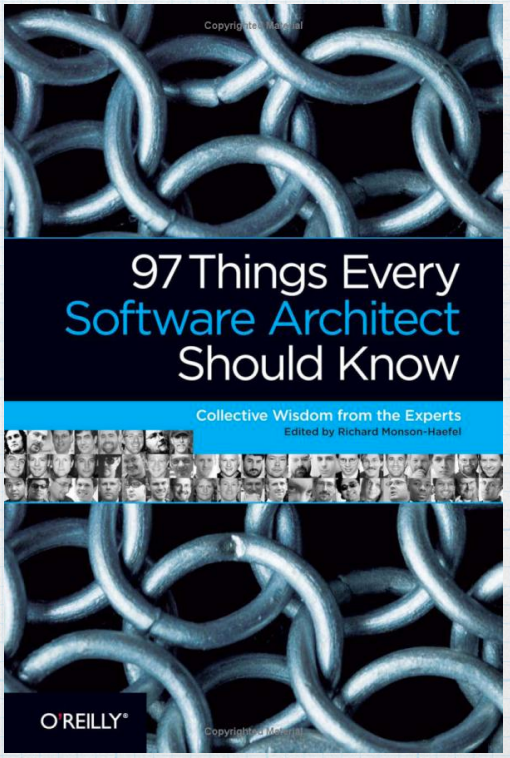
Who is Burk Hufnagel?

Programmer\Solution Architect with

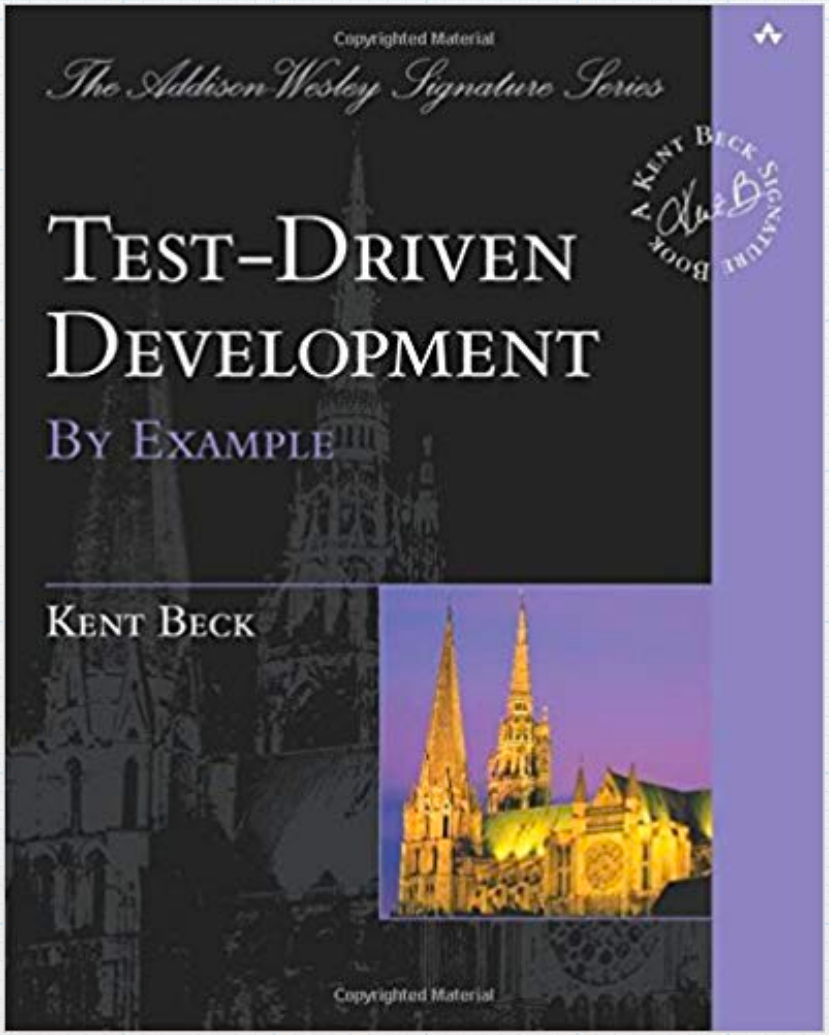
Daugherty Business Solutions

Java™ Champions



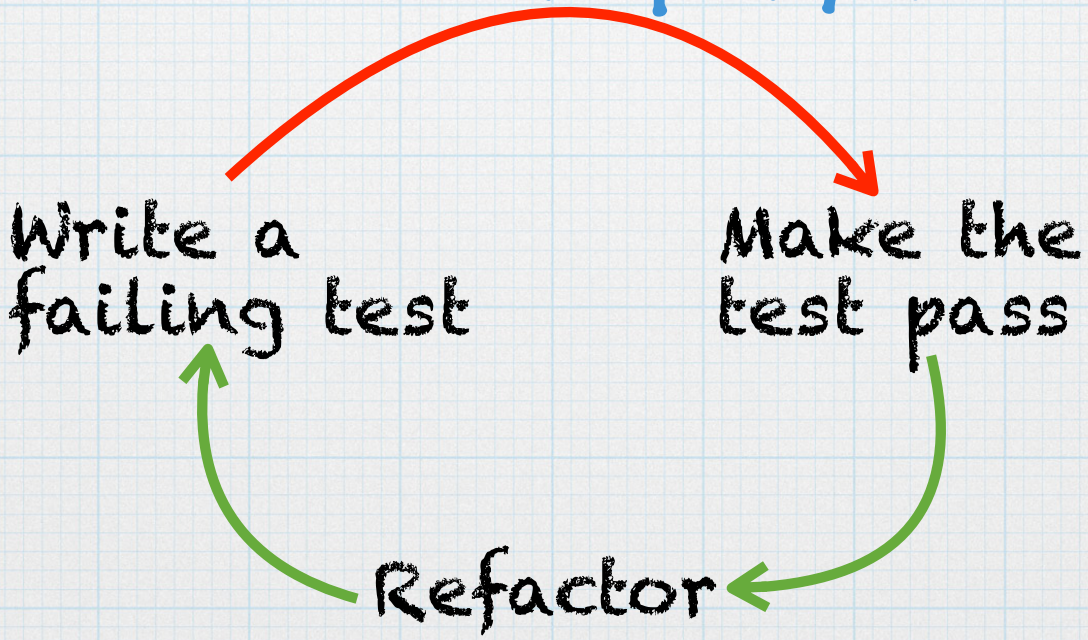


What is TDD?


Copyrighted Material
The Addison Wesley Signature Series
A KENT BECK SIGNATURE BOOK
TEST-DRIVEN DEVELOPMENT
BY EXAMPLE
KENT BECK
Copyrighted Material

What is TDD?

* TDD is a simple, three step process
  * Write a failing automated test
  * Write production code to make it pass
  * Refactor to clean up any mess



What Is A Paradox?

A seemingly self-contradictory statement that, when investigated, may prove to be true.

— Oxford Languages, publishers of the OED

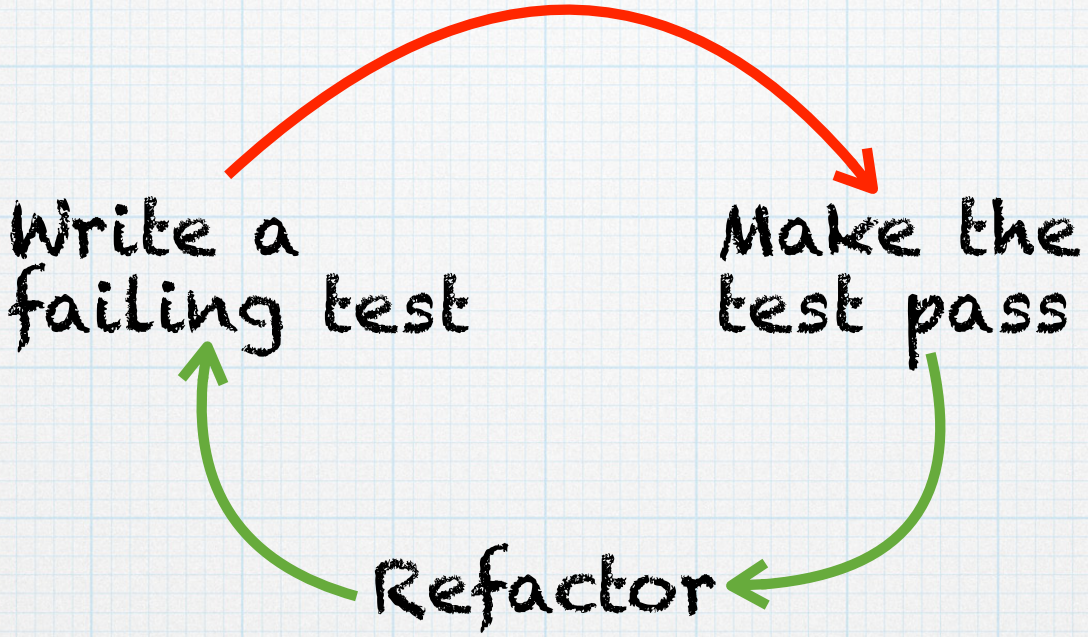
"less is more"

What makes TDD a Paradox?

TDD says writing and running automated tests results in getting done sooner, and increases the quality of the code you produce.

But, you **must** write the tests before writing the production code.

What makes TDD a Paradox?



But, you **must** write the tests before writing the production code.

Session goals

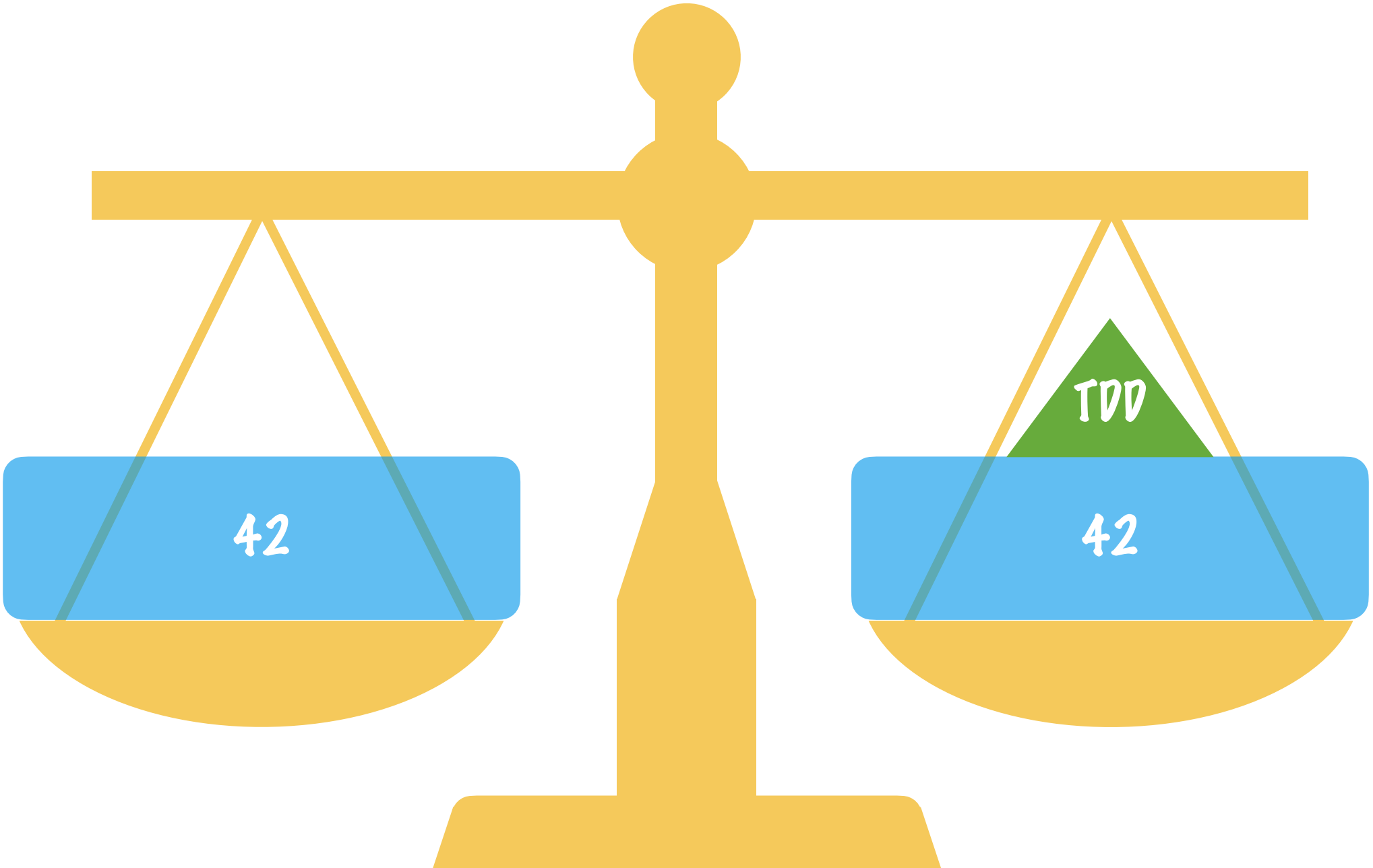
* Resolve the paradox of TDD
* Show how to start TDD today and do it on your current project

Resolve the Paradox

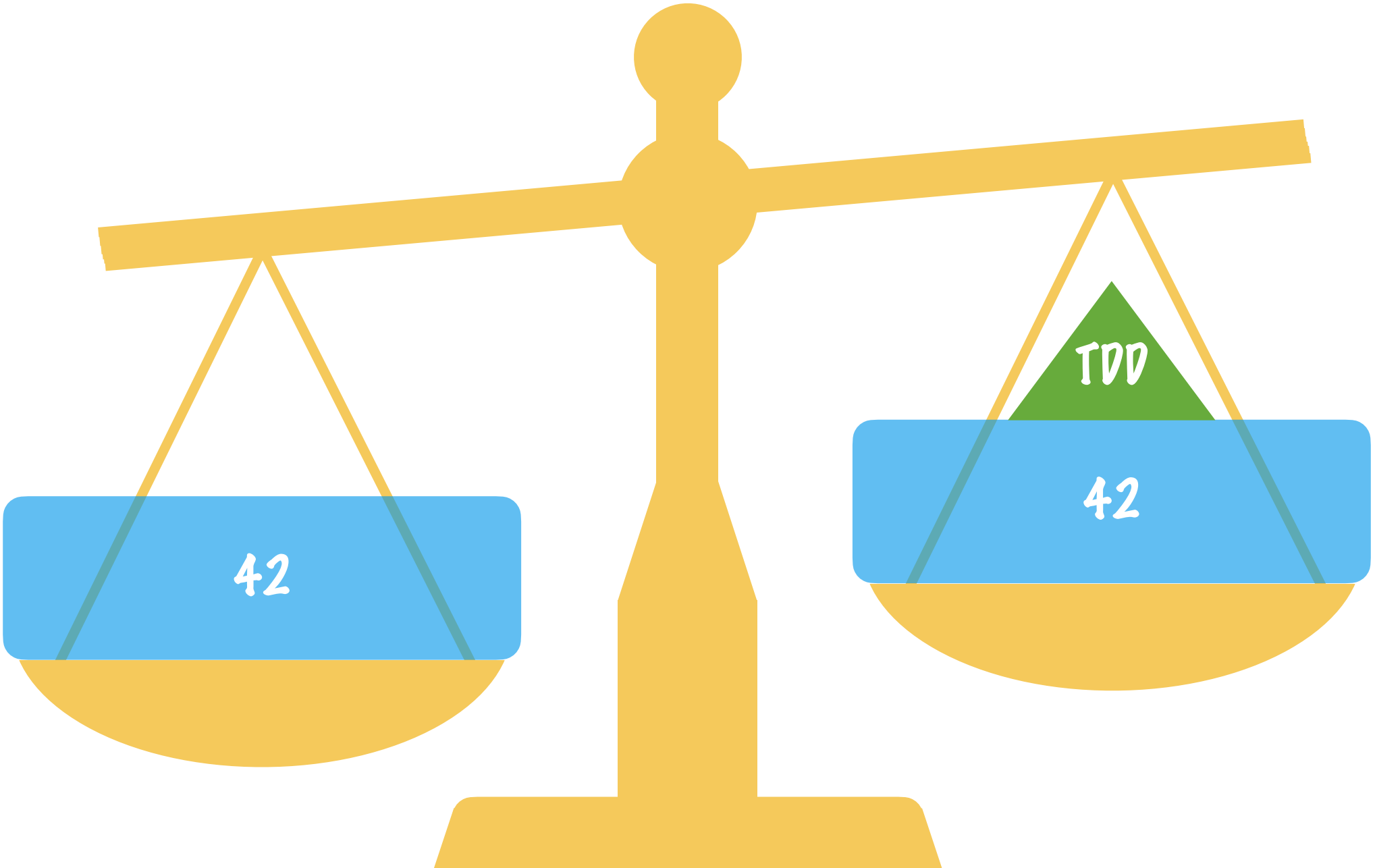
1. How can adopting TDD mean getting done sooner?
2. How can writing tests first increase code quality?
3. How can you test code that doesn't exist?

Adopt TDD == Done sooner

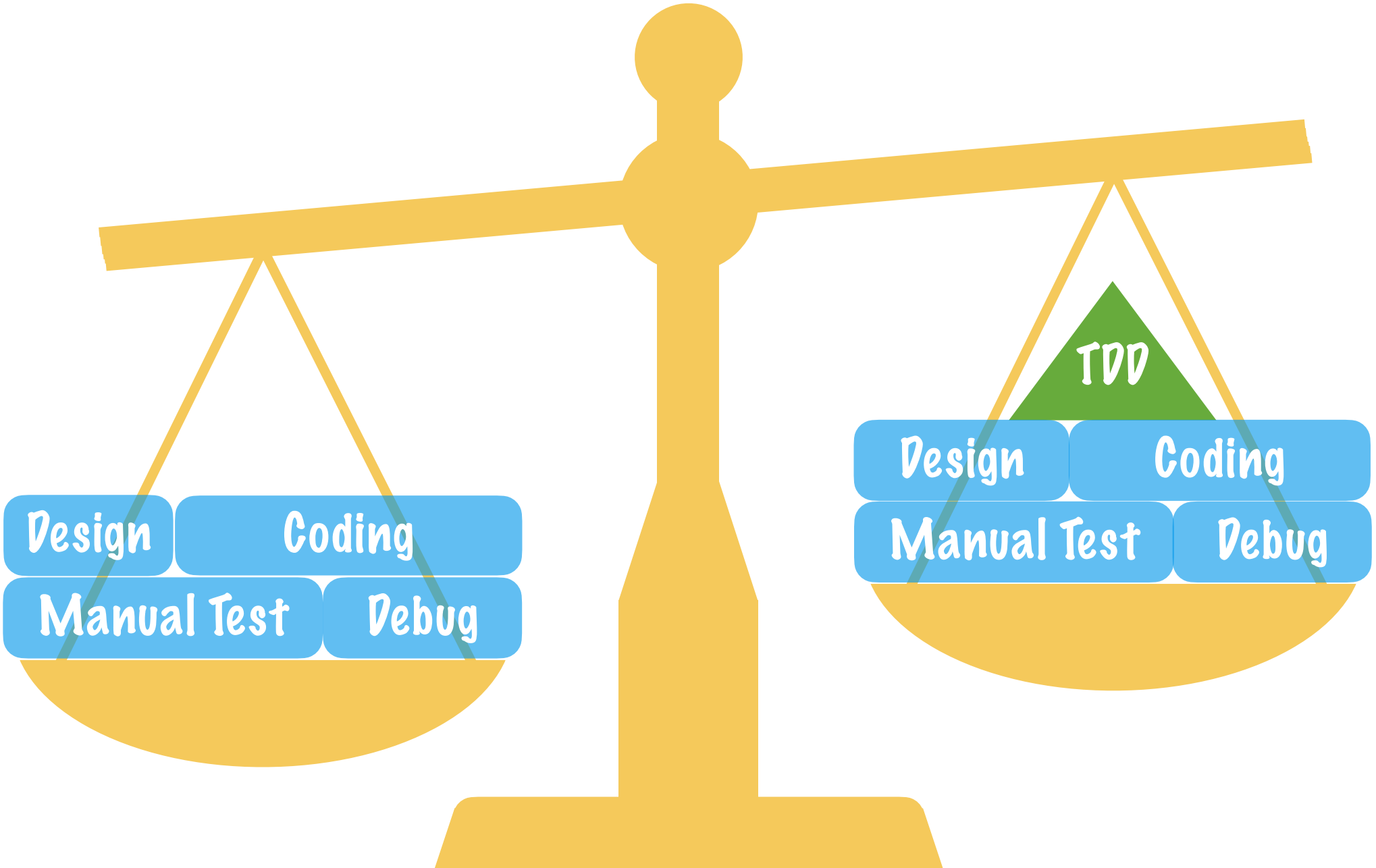
Adopt TDD == Done sooner



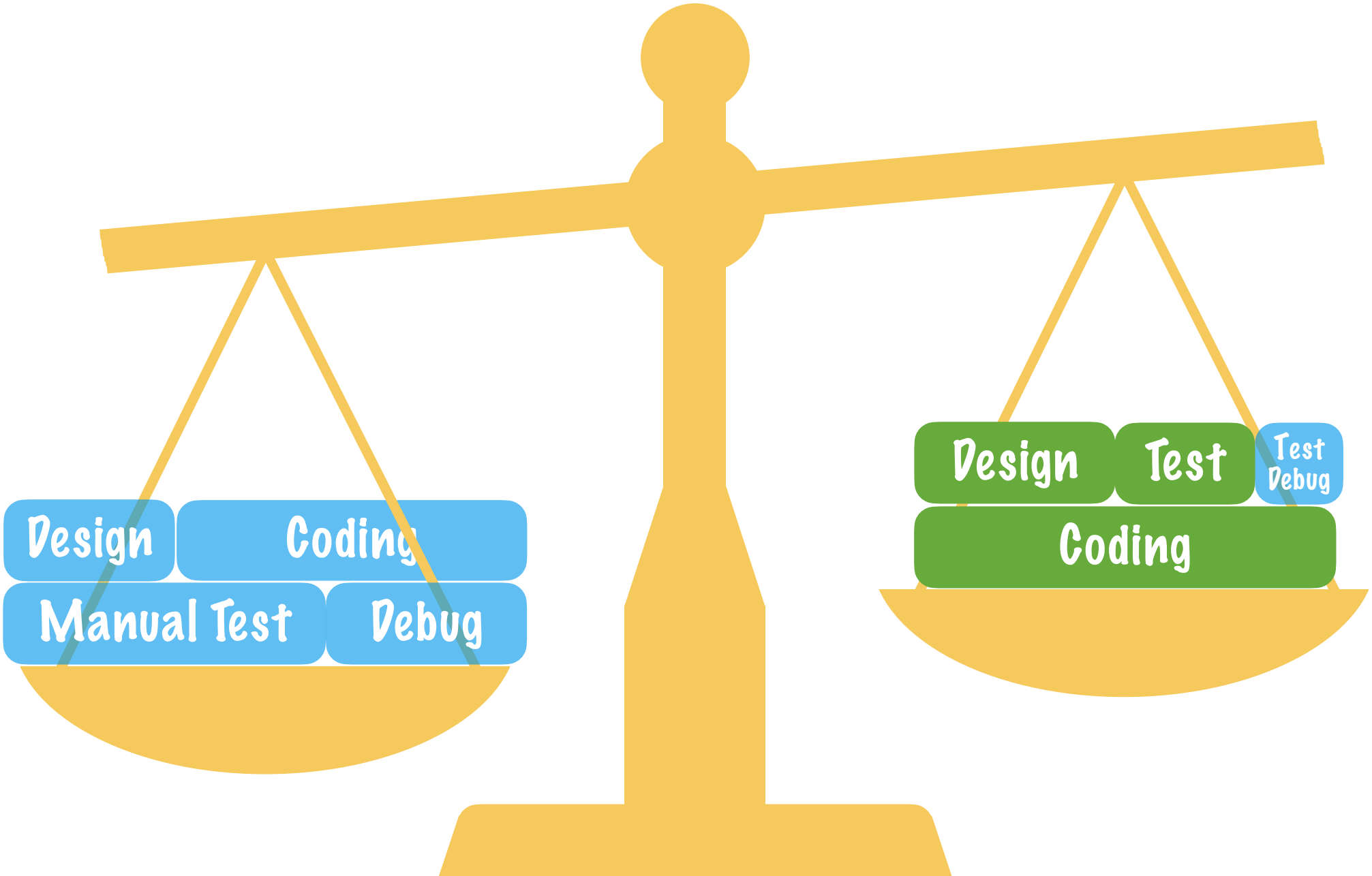
Adopt TDD == Done sooner



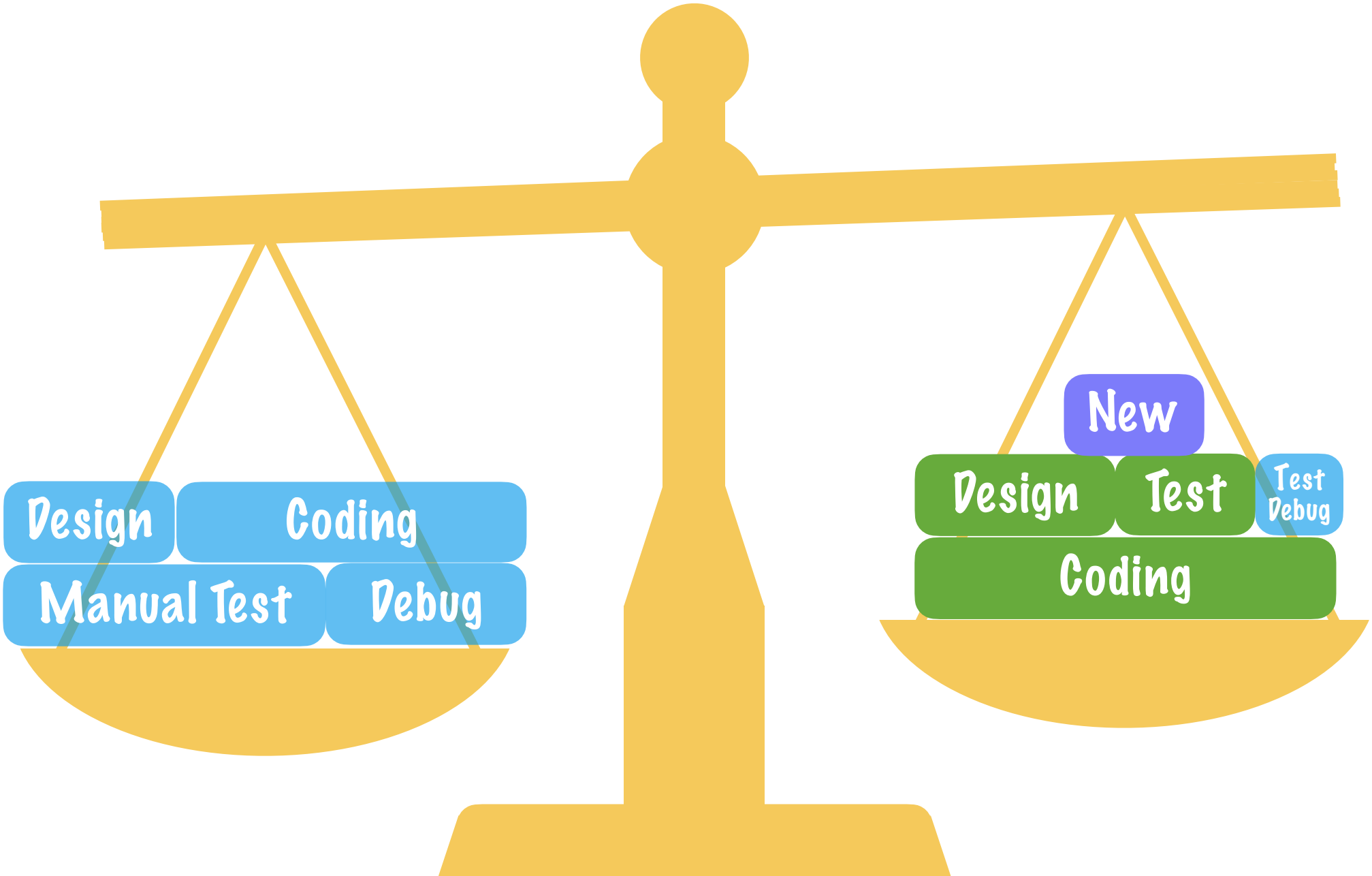
Adopt TDD == Done sooner



Adopt TDD == Done sooner



Adopt TDD == Done sooner



Resolve the Paradox

1. ~~How can adopting TDD mean getting done sooner?~~
2. How can writing tests first increase code quality?
3. How can you test code that doesn't exist?

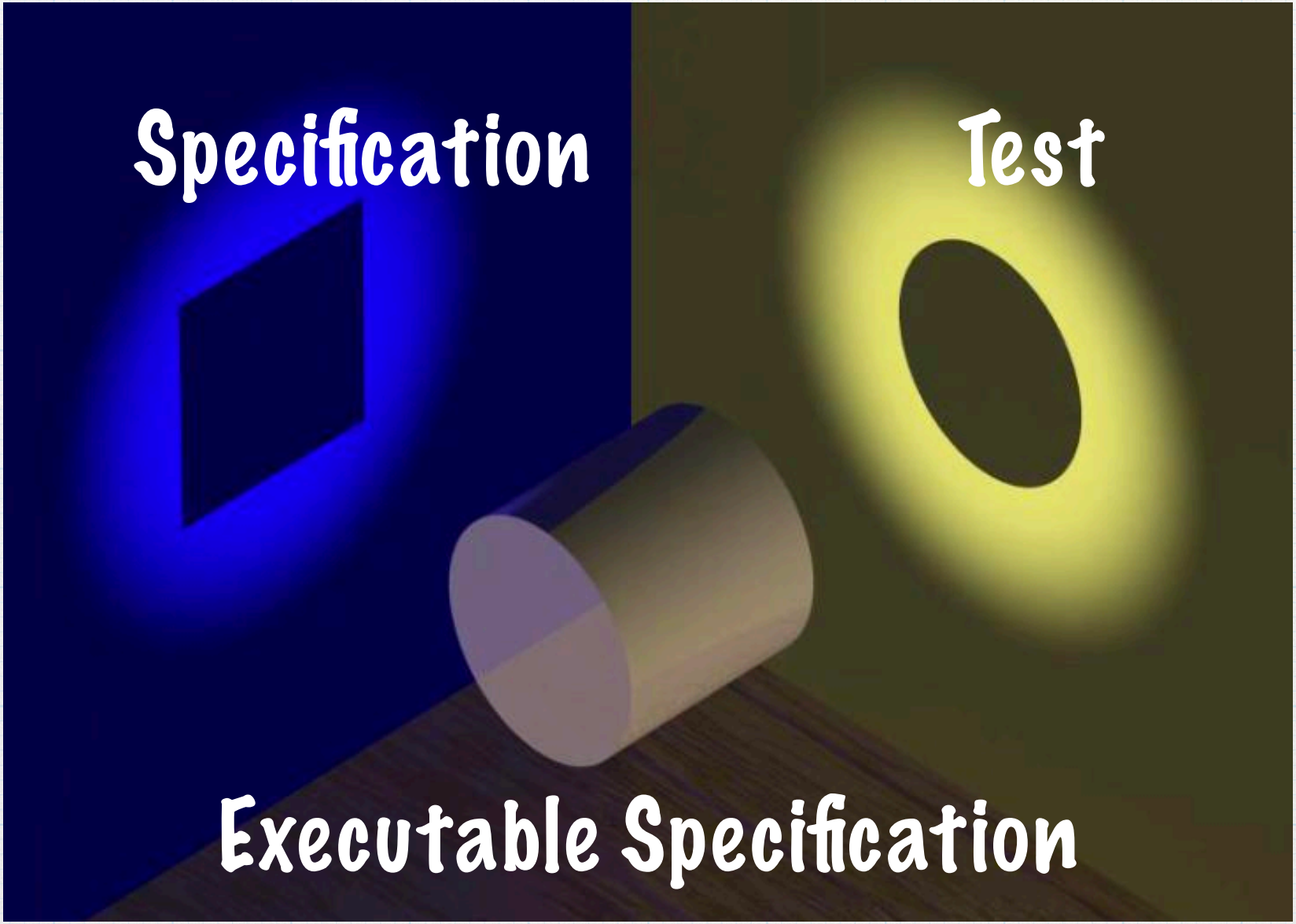
Resolve the Paradox

1. ~~How can adopting TDD mean getting done sooner?~~
2. ~~How can writing tests first increase code quality?~~
3. How can you test code that doesn't exist?

Uncle Bob's 3 Laws of TDD

1. Only write production code when there is a failing unit test.
2. Only write enough of a test for it to fail, and not compiling counts.
3. Only write enough production code to make the failing test pass.

Executable Specification



Write Specifications

* Based on requirements
* Describe what should happen
* You must understand the details: no guessing, ask business folks
* Lets you separate design from implementation

But don't stop here...

Executable Specifications

* Adds code to the spec so it can be run, as a test, against production code
* Can generate living documentation for business & technical folks
* Bridges the gap between designing and implementing solutions

Burk's Rules for TDD

0) Only write a spec when there is an unmet requirement.

1) Write just enough code for spec to fail. Not running == failing.

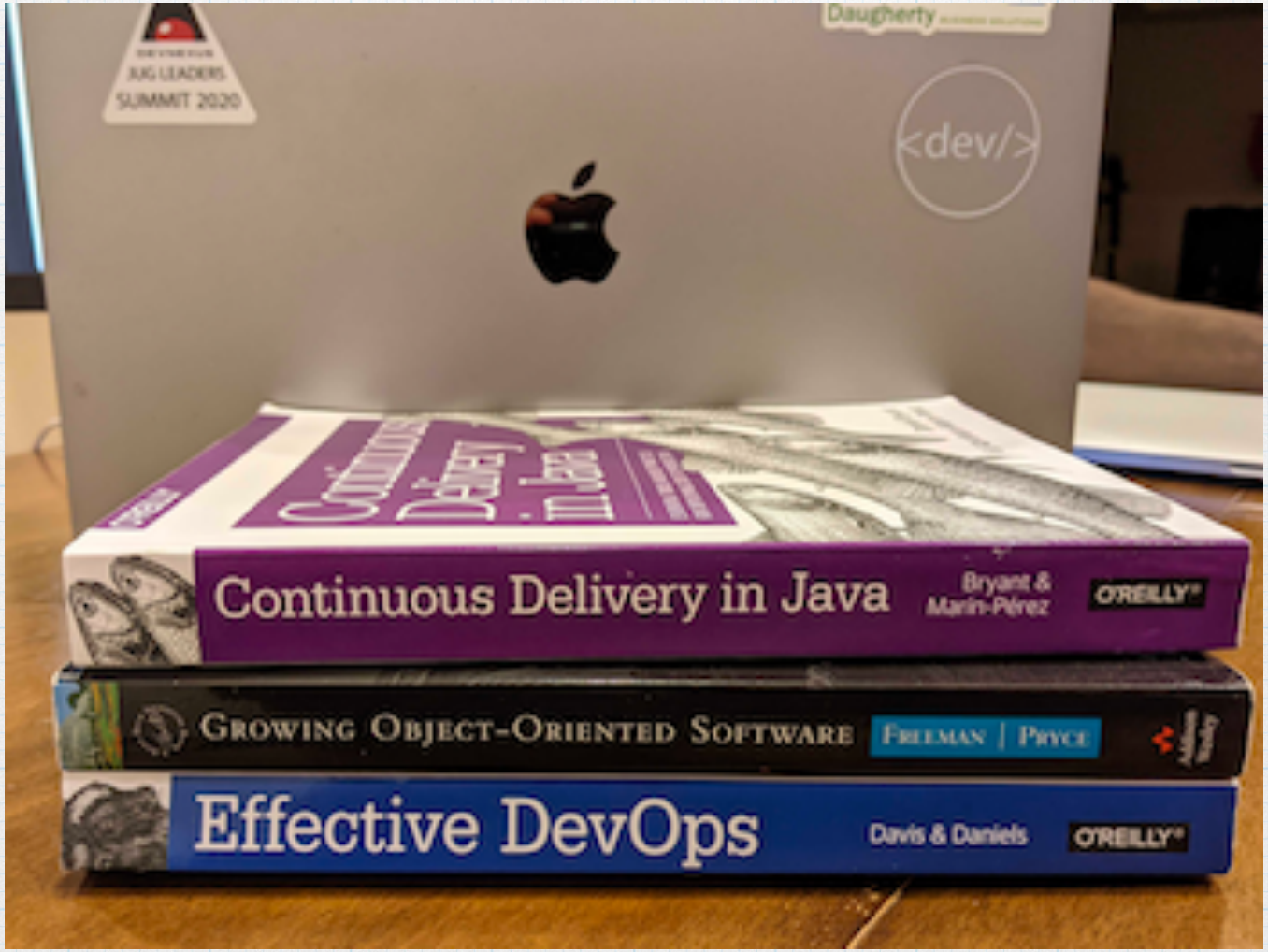
2) Write just enough production code to make the test pass.

3) Refactor the code to clean up any mess

4) Check code in when all tests pass

Let's TDD a Stack

A stack stores and returns data in a Last-in-First-out manner; like a stack of books or plates on a table.



Stack Requirements - Biz

A stack stores and returns data in a Last-in-First-out manner; like a stack of books or plates on a table.

* push() — Adds an item to the top of the stack.
* pop() — Removes & returns top item. If it's empty, throws EmptyStackException.
* isEmpty() — Returns true when no items are stored and false when there are.
* peek() — Returns a reference to the top item. If it's empty, throws EmptyStackException.

Note: Prioritized by the business, based on expected value

Dev Team Analysis

A stack stores and returns data in a Last-in-First-out manner; like a stack of books or plates on a table.

* No maximum stack size mentioned. Is this an error, or intentional?
* No external way to test push() without pop() or peek() already working
* Agree push() and pop() are critical, but believe peek() and isEmpty() can be added later

Stack Requirements - Dev

A stack stores and returns data in a Last-in-First-out manner; like a stack of books or plates on a table.

* pop() — Removes & returns top item. If it's empty, throws EmptyStackException
* push() — Adds an item to the top of the stack
* isEmpty() — Returns true when no items are stored and false when there are
* peek() — Returns a reference to the top item. If it's empty, throws EmptyStackException

Stack Scenarios

Given a new Stack:

* Calling pop should throw an EmptyStackException.
* Pushing a value and calling pop() should return it. Calling pop() again should throw an EmptyStackException.
* Pushing two values, then calling pop() should return the second value pushed.

```
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"


        when: "pop() is called"


        then: "it should throw an EmptyStackException"

    }
}
```

Generated documentation

Report for com.grokspock.devnexus2021.StackSpecification

<< Back

Summary:

Created on Sun Jan 31 22:04:11 EST 2021 by bth0624

Executed features	Passed	Failures	Errors	Skipped	Success rate	Time
1	1	0	0	0	100.0%	0.039 seconds

Features:

- Calling pop() on a new Stack should throw an EmptyStackException

Calling pop() on a new Stack should throw an EmptyStackException

Return

Given: a new instance

When: pop() is called

Then: it should throw an EmptyStackException

Generated by Athaydes Spock Reports

```
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"


        when: "pop() is called"


        then: "it should throw an EmptyStackException"

    }
}
```

```
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"


        then: "it should throw an EmptyStackException"

    }
}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"


        then: "it should throw an EmptyStackException"

    }
}
```

```groovy
public class Stack {


}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"


        then: "it should throw an EmptyStackException"

    }
}
```

```groovy
public class Stack {


}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"

    }
}
```

```groovy
public class Stack {


}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"

    }
}
```

```groovy
public class Stack {
    public void pop() {

    }
}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"

    }
}
```

```groovy
public class Stack {
    public void pop() {

    }
}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"
        thrown EmptyStackException
    }
}
```

```groovy
public class Stack {
    public void pop() {

    }
}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"
        thrown EmptyStackException
    }
```

Run: StackSpecification

Tests failed: 1 of 1 test – 25 ms

Test Results 25 ms
StackSpecification 25 ms
Calling pop() on a new Stack 25 ms

```
Expected exception of type 'java.util.EmptyStackException', but no exception was thrown

    at org.spockframework.lang.SpecInternals.checkExceptionThrown(SpecInternals.java:81)
    at org.spockframework.lang.SpecInternals.thrownImpl(SpecInternals.java:68)
    at com.grokspock.devnexus2021.StackSpecification.Calling pop() on a new Stack should throw an EmptyStackException(StackSpecification.groovy:39)
```

```groovy
    public void pop() {

    }
}
```

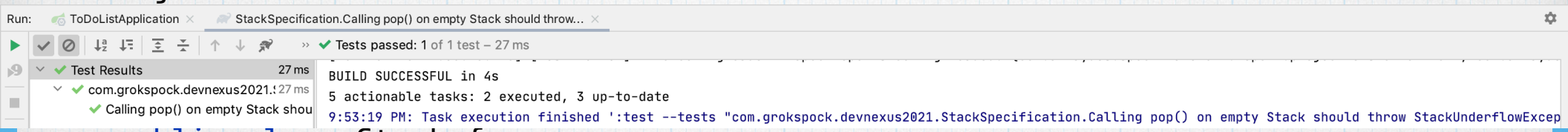
```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"
        thrown EmptyStackException
    }
}
```

```groovy
public class Stack {
    public void pop() {

    }
}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"
        thrown EmptyStackException
    }
}
```

```groovy
public class Stack {
    public void pop() {
        throw new EmptyStackException();
    }
}
```

```groovy
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"
        thrown EmptyStackException
    }
```




```groovy
public class Stack {
    public void pop() {
        throw new EmptyStackException();
    }
}
```

```
class StackSpec extends Specification {
    def "Calling pop() on a new Stack should throw an EmptyStackException"() {
        given: "a new instance"
        def stack = new Stack()

        when: "pop() is called"
        stack.pop()

        then: "it should throw an EmptyStackException"
        thrown EmptyStackException
    }
}
```

Refactor!

```
public class Stack {
    public void pop() {
        throw new EmptyStackException();
    }
}
```

Stack Scenarios

Given a new Stack:

* Calling pop should throw an EmptyStackException.
* Pushing a value and calling pop() should return it. Calling pop() again should throw an EmptyStackException.
* Pushing two values, then calling pop() should return the second value pushed.

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"


    expect: "when pop() is called, it should return 5"

    when: "pop() is called again"

    then: "it should throw an EmptyStackException"

}
```

```
public class Stack {
    public void pop() {
        throw new EmptyStackException();
    }
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"

    when: "pop() is called again"

    then: "it should throw an EmptyStackException"

}
```

```
public class Stack {
    public void pop() {
        throw new EmptyStackException();
    }
}
```

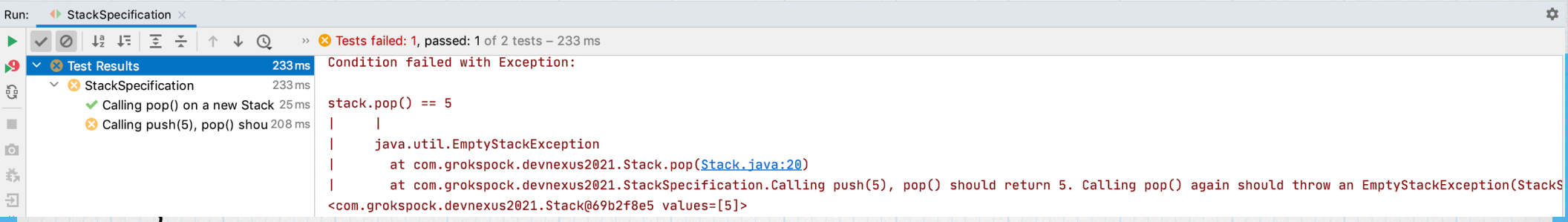
```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"

    when: "pop() is called again"

    then: "it should throw an EmptyStackException"

}
```

```
public class Stack {
    public void pop() {
        throw new EmptyStackException();
    }

    public void push(int value) {

    }
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

```
public class Stack {
    public void pop() {
        throw new EmptyStackException();
    }

    public void push(int value) {

    }
}
```

```groovy
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```




```groovy
    }

    public void push(int value) {

    }
}
```

Report for com.grokspock.devnexus2021.StackSpecification

[<< Back]

Summary:

Created on Sun Jan 31 21:35:09 EST 2021 by bth0624

Executed features	Passed	Failures	Errors	Skipped	Success rate	Time
2	1	1	0	0	50.0%	0.270 seconds

Features:

- Calling pop() on a new Stack should throw an EmptyStackException
- Calling push(5), pop() should return 5. Calling pop() again should throw an EmptyStackException

Calling pop() on a new Stack should throw an EmptyStackException

Return

Given: a new instance

When: pop() is called

Then: it should throw an EmptyStackException

Calling push(5), pop() should return 5. Calling pop() again should throw an EmptyStackException

Return

Given: we call push(5) on a new instance

Expect: when pop() is called, it should return 5

When: when calling pop() again

Then: it should throw an EmptyStackException

The following problems occurred:

- Condition failed with Exception:

```
stack.pop() == 5
|     |
|     java.util.EmptyStackException
|
at com.grokspock.devnexus2021.Stack.pop(Stack.java:16)
|
at com.grokspock.devnexus2021.StackSpecification.Calling push(5), pop() should return 5.
Calling pop() again should throw an EmptyStackException(StackSpecification.groovy:28)
<com.grokspock.devnexus2021.Stack@466d49f0 value=[5]>
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

```
public class Stack {

    public int pop() {
        throw new EmptyStackException();
    }

    public void push(int value) {

    }
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

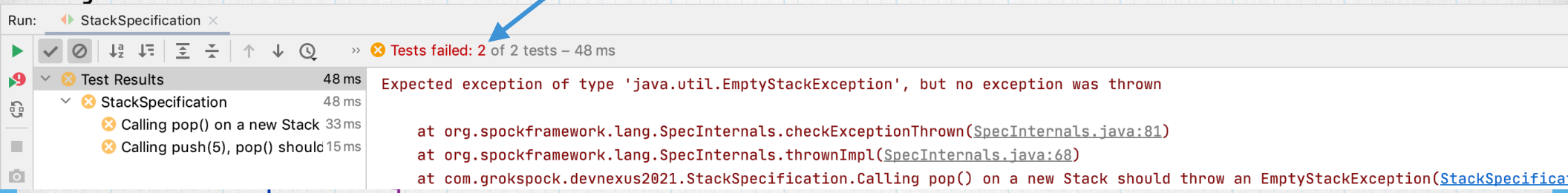
```
public class Stack {

    public int pop() {
        return value;
    }

    public void push(int value) {
        this.value = value;
    }
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

```
public class Stack {
    private Integer value = null;

    public int pop() {
        return value;
    }

    public void push(int value) {
        this.value = value;
    }
}
```

```groovy
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

Two tests failed




```groovy
        return value;
    }

    public void push(int value) {
        this.value = value;
    }
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

```
public class Stack {
    private Integer value = null;

    public int pop() {
        return value;
    }

    public void push(int value) {
        this.value = value;
    }
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

```
public class Stack {
    private Integer value = null;

    public int pop() {
        if (value == null) {
            throw new EmptyStackException();
        }

        return value;
    }

    public void push(int value) {
        this.value = value;
    }
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
public class Stack {
    private Integer value = null;

    public int pop() {
        if (value == null) {
            throw new EmptyStackException();
        }

        int topValue = value;
        value = null;
        return topValue;
    }

    public void push(int value) {
        this.value = value;
    }
```

```groovy
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```


```java
public int pop() {
    if (value == null) {
        throw new EmptyStackException();
    }

    int topValue = value;
    value = null;
    return topValue;
}

public void push(int value) {
    this.value = value;
}
```

```
def "Pushing a value and calling pop() should return it.
     Calling pop() again should throw an EmptyStackException."() {
    given: "we call push(5) on a new instance"
        def stack = new Stack()
        stack.push(5)
    expect: "when pop() is called, it should return 5"
        stack.pop() == 5
    when: "pop() is called again"
        stack.pop()
    then: "it should throw an EmptyStackException"
        thrown EmptyStackException
}
```

Refactor!

```
public class Stack {
    private Integer value = null;

    public int pop() {
        if (value == null) {
            throw new EmptyStackException();
        }

        int topValue = value;
        value = null;
        return topValue;
    }

    public void push(int value) {
        this.value = value;
    }
```

Report for com.grokspock.devnexus2021.StackSpecification

[<< Back]

Summary:

Created on Sun Jan 24 13:37:17 EST 2021 by bth0624

Executed features	Passed	Failures	Errors	Skipped	Success rate	Time
2	2	0	0	0	100.0%	0.025 seconds

Features:

Calling pop() on a new Stack should throw an EmptyStackException

Return

Given: a new instance

When: pop() is called

Then: it should throw an EmptyStackException

Calling pop() after push(5) should return 5. Calling pop() again should throw an EmptyStackException

Return

Given: a new Stack instance

Expect: calling pop should return 5

When: pop() is called

Then: ----

Stack Scenarios

Given a new Stack:

* Calling pop should throw an EmptyStackException.
* Pushing a value and calling pop() should return it. Calling pop() again should throw an EmptyStackException.
* Pushing two values, then calling pop() should return the second value pushed.

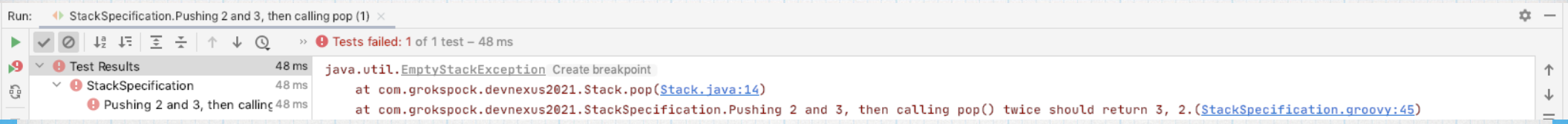
```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"

    when: "42 and 23 are pushed"


    and: "pop() is called"

    then: "it should return 23"

}
```

```
public class Stack {
    private Integer value = null;

    public int pop() {
        if (value == null) {
            throw new EmptyStackException();
        }

        int topValue = value;
        value = null;
        return topValue;
    }

    public void push(int value) {
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

```
public class Stack {
    private Integer value = null;

    public int pop() {
        if (value == null) {
            throw new EmptyStackException();
        }

        int topValue = value;
        value = null;
        return topValue;
    }

    public void push(int value) {
```

```groovy
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23”
        returnedValue == 23
}
```




```java
public int pop() {
    if (value == null) {
        throw new EmptyStackException();
    }

    int topValue = value;
    value = null;
    return topValue;
}

public void push(int value) {
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

```
public class Stack {
    private Integer value = null;

    public int pop() {
        if (value == null) {
            throw new EmptyStackException();
        }

        int topValue = value;
        value = null;
        return topValue;
    }

    public void push(int value) {
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

```
public class Stack {
    private List<Integer> value = new ArrayList<>();

    public int pop() {
        if (value == null) {
            throw new EmptyStackException();
        }

        int topValue = value;
        value = null;
        return topValue;
    }

    public void push(int value) {
```

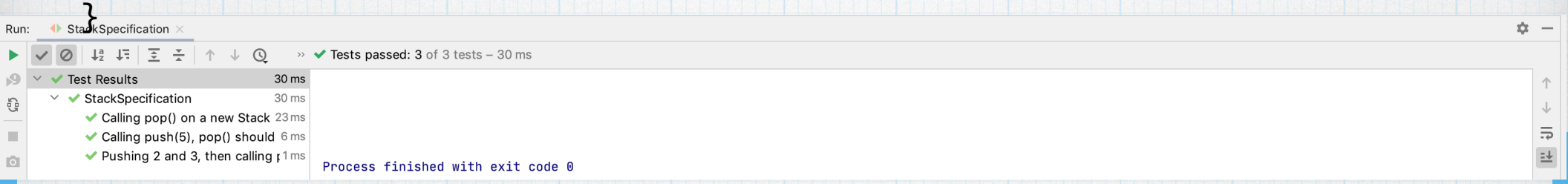
```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

```
public class Stack {
    private List<Integer> value = new ArrayList<>();

    public int pop() {
        if (value.isEmpty()) {
            throw new EmptyStackException();
        }

        return value.remove(value.size()-1);
    }

    public void push(int value) {
        this.value = value;
    }
}
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

```
public class Stack {
    private List<Integer> value = new ArrayList<>();

    public int pop() {
        if (value.isEmpty()) {
            throw new EmptyStackException();
        }

        return value.remove(value.size()-1);
    }

    public void push(int value) {
        this.value.add(value);
    }
}
```

```groovy
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```




```java
public int pop() {
    if (value.isEmpty()) {
        throw new EmptyStackException();
    }

    return value.remove(value.size()-1);
}

public void push(int value) {
    this.value.add(value);
}
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

Refactor!

```
public class Stack {
    private List<Integer> value = new ArrayList<>();

    public int pop() {
        if (value.isEmpty()) {
            throw new EmptyStackException();
        }

        return value.remove(value.size()-1);
    }

    public void push(int value) {
        this.value.add(value);
    }
}
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

Refactor!

```
public class Stack {
    private List<Integer> values = new ArrayList<>();

    public int pop() {
        if (value.isEmpty()) {
            throw new EmptyStackException();
        }

        return value.remove(value.size()-1);
    }

    public void push(int value) {
        this.value.add(value);
    }
}
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

Refactor!

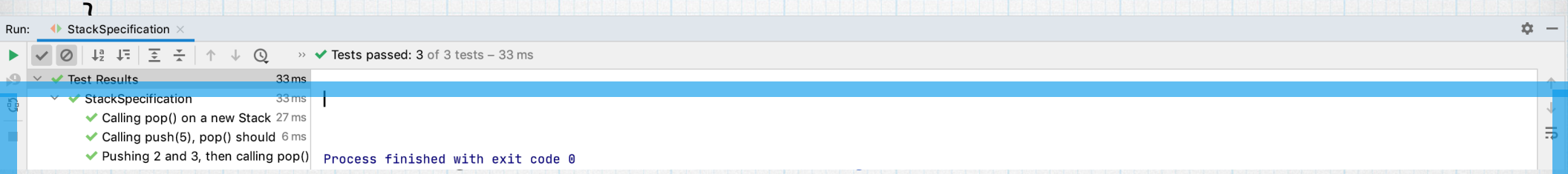
```
public class Stack {
    private List<Integer> values = new ArrayList<>();

    public int pop() {
        if (values.isEmpty()) {
            throw new EmptyStackException();
        }

        return values.remove(values.size()-1);
    }

    public void push(int value) {
        this.values.add(value);
    }
}
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```

Refactor!

```
public class Stack {
    private List<Integer> values = new ArrayList<>();

    public int pop() {
        if (values.isEmpty()) {
            throw new EmptyStackException();
        }

        return values.remove(value.size()-1);
    }

    public void push(int value) {
        values.add(value);
    }
}
```

```
def "Pushing two values, then calling pop() should return
     the second value pushed."() {
    given: "a new instance"
        def stack = new Stack()
    when: "42 and 23 are pushed"
        stack.push(42)
        stack.push(23)
    and: "pop() is called"
        def returnedValue = stack.pop()
    then: "it should return 23"
        returnedValue == 23
}
```




```
public int pop() {
    if (values.isEmpty()) {
        throw new EmptyStackException();
    }

    return values.remove(value.size()-1);
}

public void push(int value) {
    values.add(value);
}
```

Stack Scenarios

Given a new Stack:

* Calling pop should throw an EmptyStackException.
* Calling pop after pushing 5, should return 5. Calling pop again should throw an EmptyStackException.
* Pushing 2 and 3, then calling pop() twice should return 3, 2.

```
@Narrative("""
Like a stack of books on a table, a Stack is a data structure
that stores and returns data in a Last-in-First-out manner.

Methods:
  push() – Adds an item to the top of the stack.
  pop()  – Removes & returns top item. If it's empty, throws EmptyStackException.
""")


public class Stack {
    private List<Integer> values = new ArrayList<>();

    public int pop() {
        if (values.isEmpty()) {
            throw new EmptyStackException();
        }

        return values.remove(values.size()-1);
    }

    public void push(int value) {
        values.add(value);
    }
}
```

Report for com.grokspock.devnexus2021.StackSpecification

<< Back

Summary:

Created on Wed Feb 03 22:00:11 EST 2021 by bth0624

Executed features	Passed	Failures	Errors	Skipped	Success rate	Time
3	3	0	0	0	100.0%	0.037 seconds

Like a stack of books on a table, a Stack is a data structure
that stores and returns data in a Last-in-First-out manner.

Methods:
push() — Adds an item to the top of the stack.
pop() — Removes & returns top item. If it's empty, throws EmptyStackException.

Narrative Text

Features:

- Calling pop() on a new Stack should throw an EmptyStackException
- Calling push(5), pop() should return 5. Calling pop() again should throw an EmptyStackException
- Pushing 2 and 3, then calling pop() twice should return 3, 2.

Calling pop() on a new Stack should throw an EmptyStackException

Return

Given: a new instance
When: pop() is called
Then: it should throw an EmptyStackException

Calling push(5), pop() should return 5. Calling pop() again should throw an EmptyStackException

Return

Given: we call push(5) on a new instance
Expect: when pop() is called, it should return 5
When: when calling pop() again
Then: it should throw an EmptyStackException

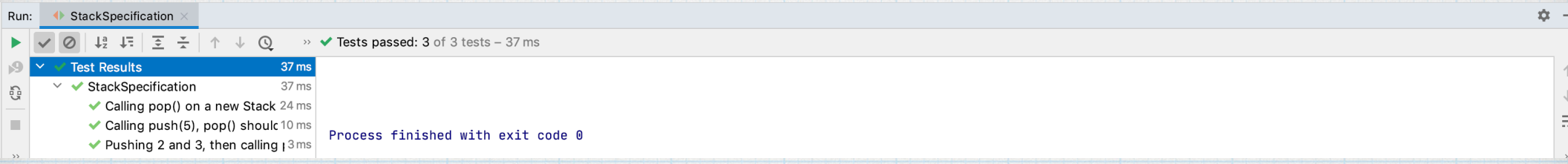
Return

Common Concerns

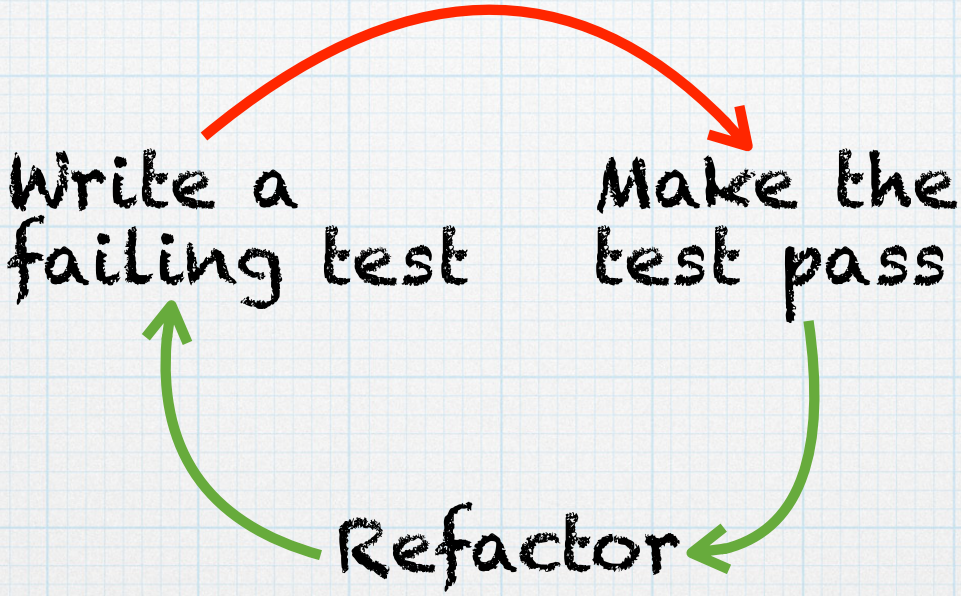
* Rewriting tests if we change the implementation

```
public class Stack {
    private List<Integer> values = new ArrayList<>();


    public int pop() {
        if (values.isEmpty()) {
            throw new EmptyStackException();
        }

        return values.remove(values.size()-1);
    }

    public void push(int value) {
        values.add(value);
    }
}
```

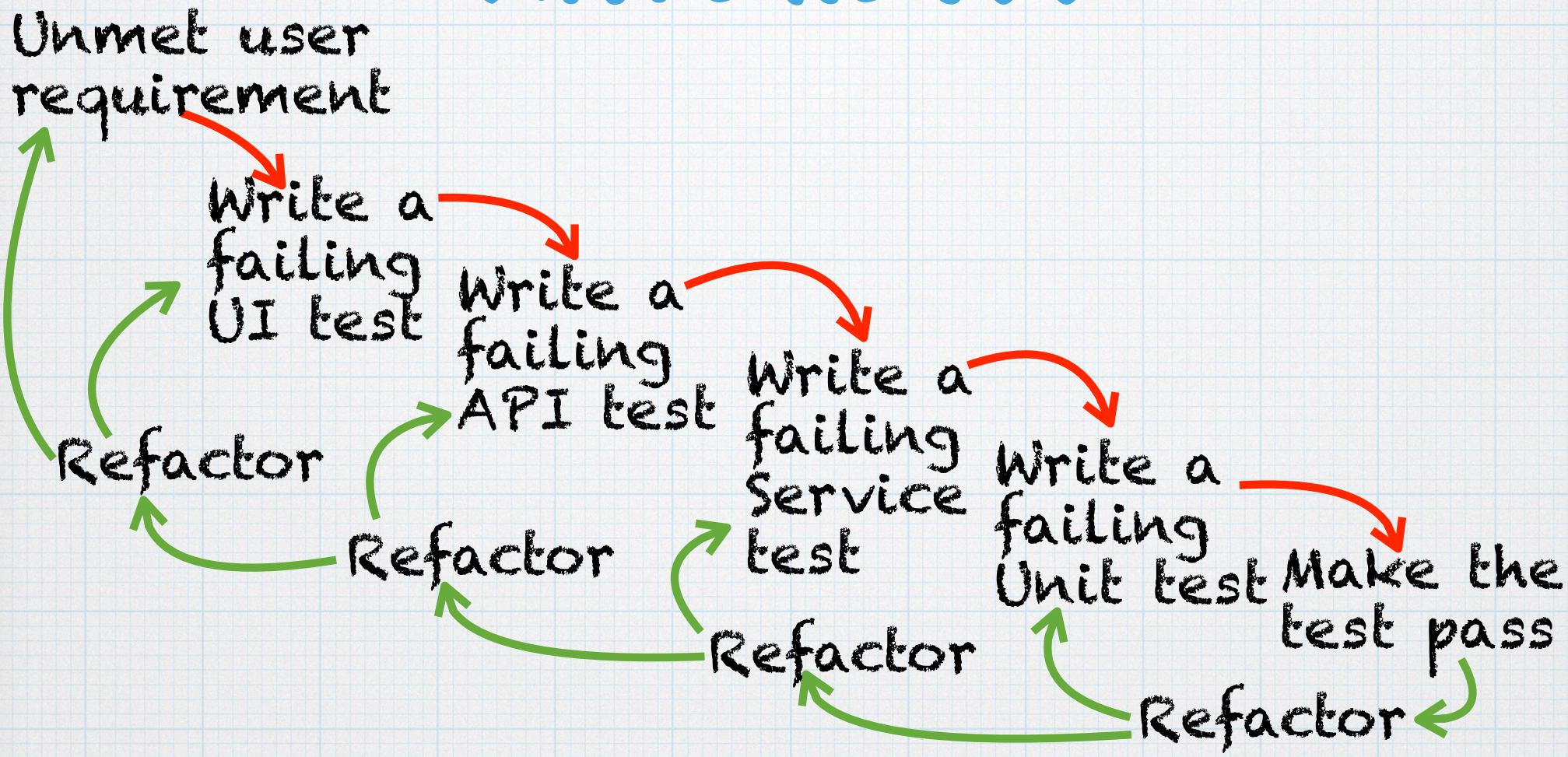
```java
public class Stack {
    private int[] values = new int[1000]; // Max items pushed on stack
    private int valuesCount = 0;          // Track count of items pushed

    public int pop() {
        if (valuesCount == 0) {           // Was values.isEmpty()
            throw new EmptyStackException();
        }

        return values[-valuesCount];      // Was values.remove(values.size()-1)
    }

    public void push(int value) {
        values[valuesCount++] = value;    // Was values.add(value);
    }
}
```



Extreme TDD



Extreme TDD


Unmet user requirement
Write a failing UI test
Write a failing API test
Write a failing Service test
Write a failing Unit test
Make the test pass
Refactor
Refactor
Refactor
Refactor

Resolve the Paradox

1. ~~How can adopting TDD mean getting done sooner?~~
2. ~~How can writing tests first increase code quality?~~
3. ~~How can you test code that doesn't exist?~~

Session goals

* ~~Resolve the paradox of TDD~~
* Show how to start TDD today and do it on your current project

Start with TDD Today

Start with TDD Today

Start with TDD Today

Start with TDD Today

create a new practice

choose your language & test-framework

- Go, testing
- Groovy, JUnit
- Groovy, Spock
- Haskell, hunit
- Java 17, Approval
- Java 18, JUnit
- Java, Cucumber
- Java, Cucumber-Spring
- Java, JMock
- Java, JUnit-Sqlite
- Java, Mockito
- Java, PowerMockito
- JavaScript, Cucumber
- JavaScript, Mocha+chai+sinon
- JavaScript, assert
- JavaScript, assert+jQuery

```
import spock.lang.*

class HikerSpec extends Specification {

    def "life the universe and everything" () {
        def douglas = new Hiker()
        expect:
            douglas.answer() == 42
    }
}
```

next

Start with TDD Today

create a new practice

choose your language & test-framework

- Go, testing
- Groovy, JUnit
- Groovy, Spock
- Haskell, hunit
- Java 17, Approval
- Java 18, JUnit
- Java, Cucumber
- Java, Cucumber-Spring
- Java, JMock
- Java, JUnit-Sqlite
- Java, Mockito
- Java, PowerMockito
- JavaScript, Cucumber
- JavaScript, Mocha+chai+sinon
- JavaScript, assert
- JavaScript, assert+jQuery

next

```
// A simple example to get you started
// JUnit assertion - the default Java assertion library
// https://junit.org/junit5/

import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
public class HikerTest {

    @Test
    void life_the_universe_and_everything() {
        int expected = 42;
        int actual = Hiker.answer();
        assertEquals(expected, actual);
    }
}
```

TDD on a Current Project

* For a new project, pick a requirement, identify a scenario, and write a spec...
* For an existing project, try Defect-Driven Testing.

Session goals

* ~~Resolve the paradox of TDD~~
* ~~Show how to start TDD today and do it on your current project~~

Thank you!

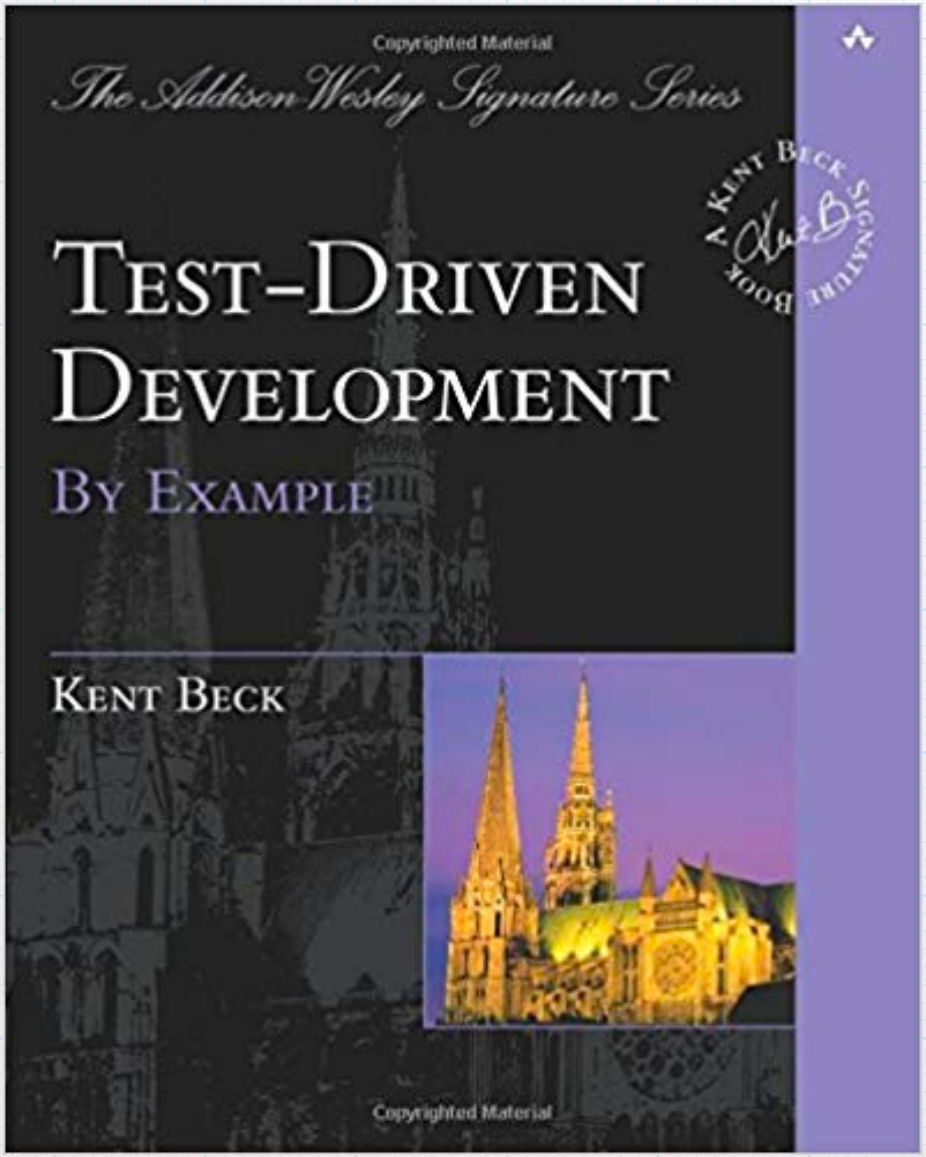
Spec well, and Prosper

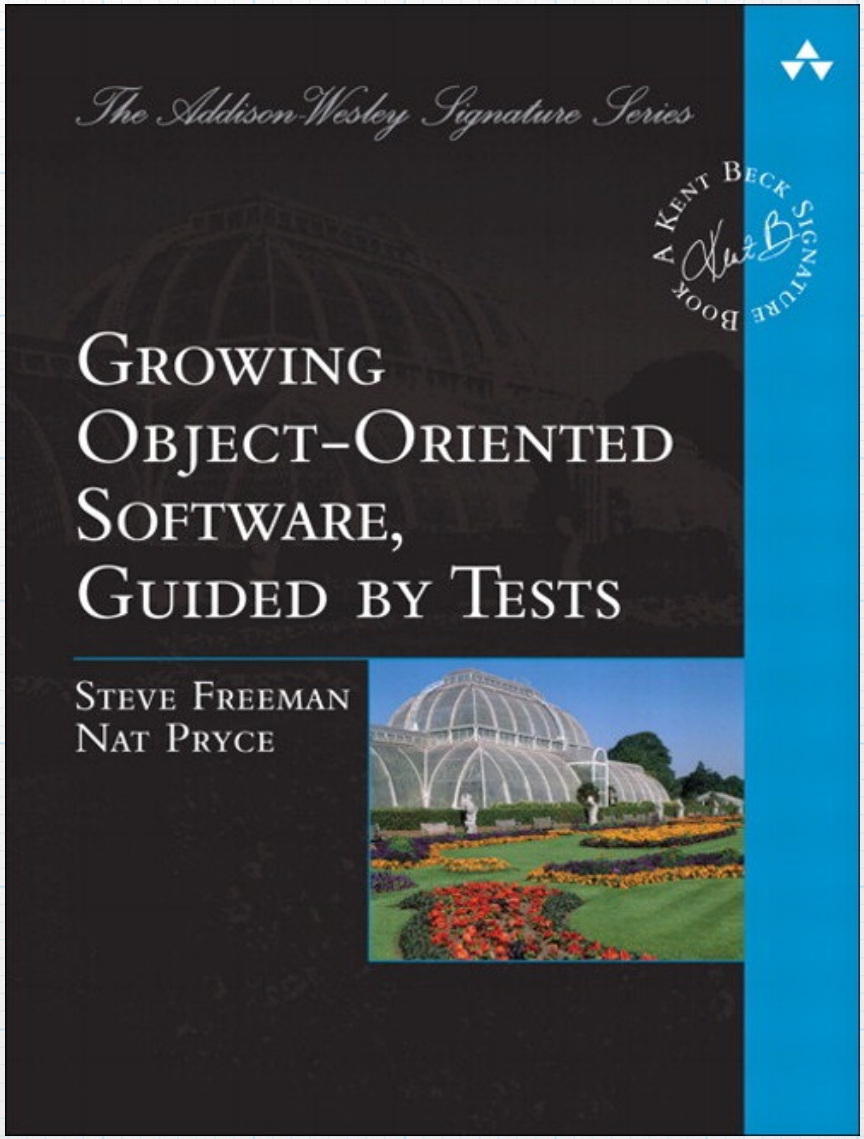
Questions & Contact Info

Feedback, questions, comments to:
burk.hufnagel@daugherty.com

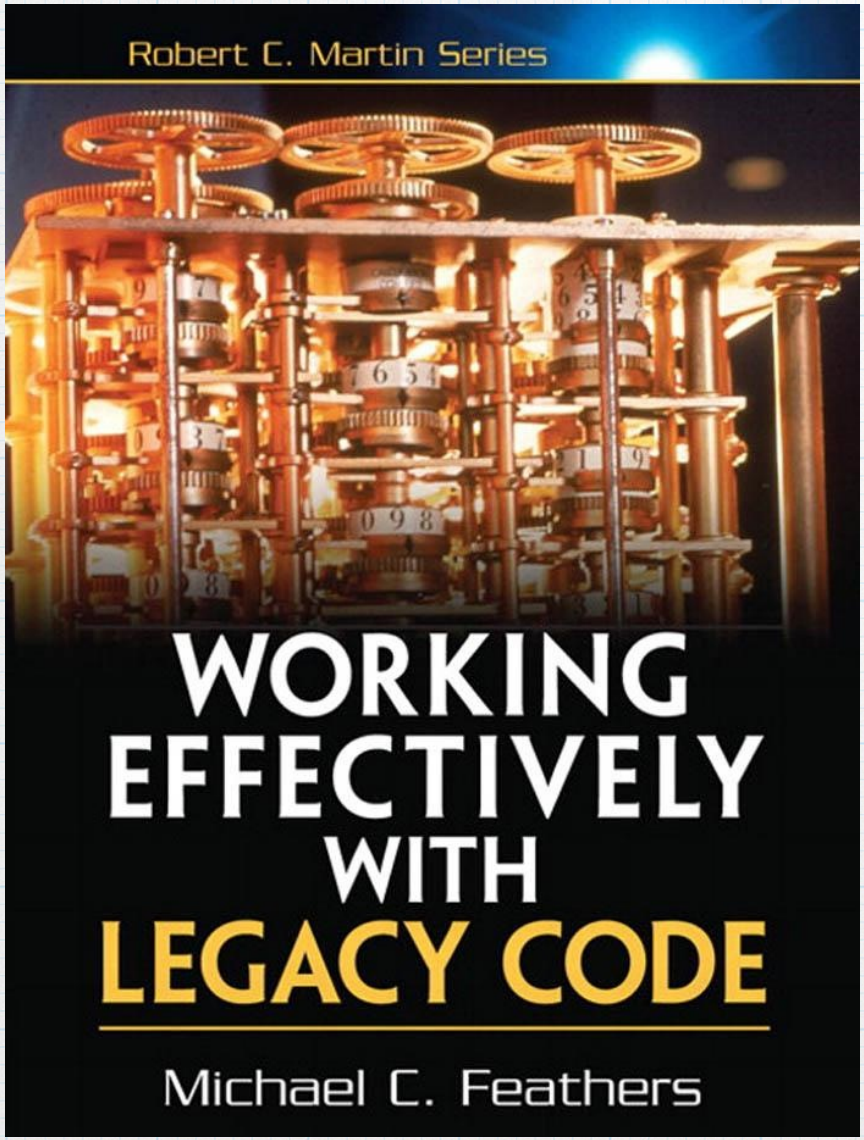
Twitter: @burkhufnagel

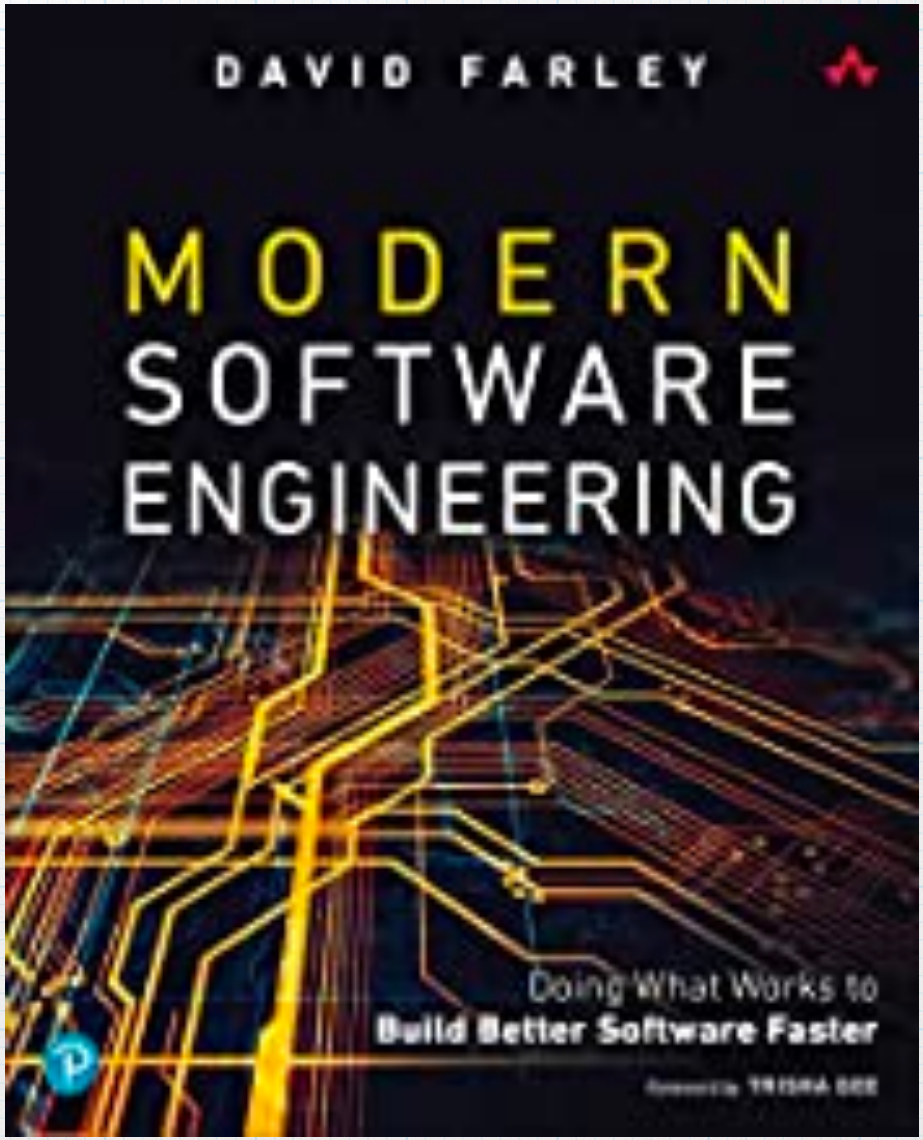
Recommended Reading





Recommended Reading





Recommended Reading