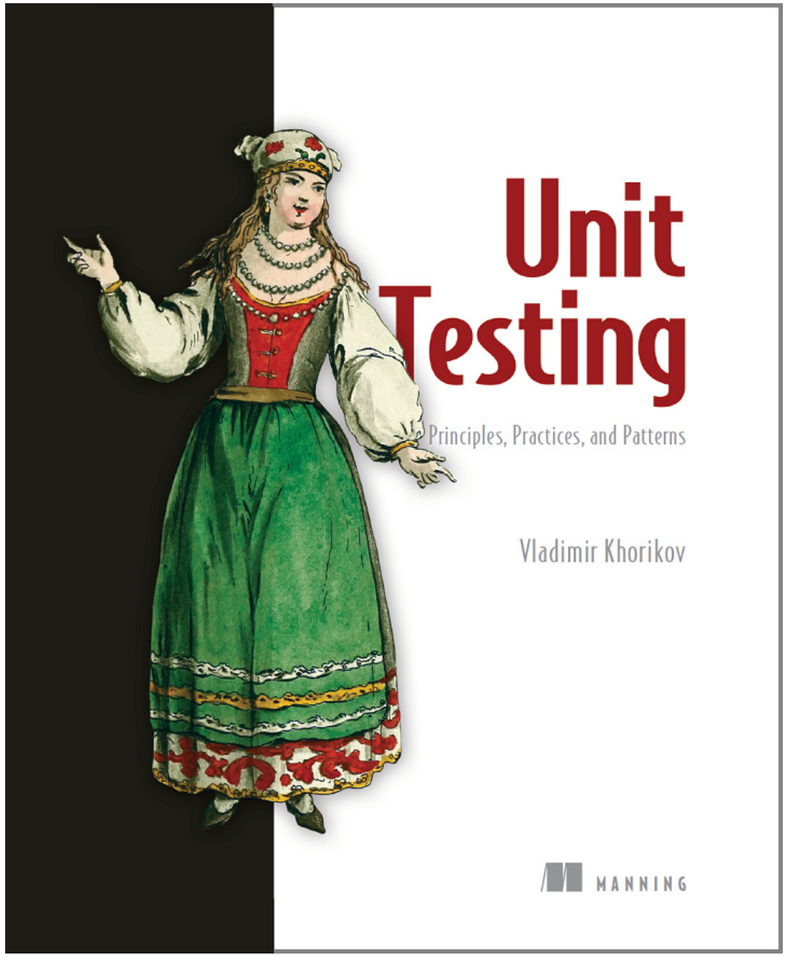
How valuable are your Tests?

Workshop


YouTube DE
ferdinand ade
DOMAIN
DRIVEN
DESIGN
EUROPE
0:00 / 51:38
Functional Domain Modelling - Marco Emrich and Ferdinand Ade - DDD Europe 2023
Domain-Driven Design Europe
30.600 Abonnenten
Abonniert
57
Teilen
Clip
Speichern

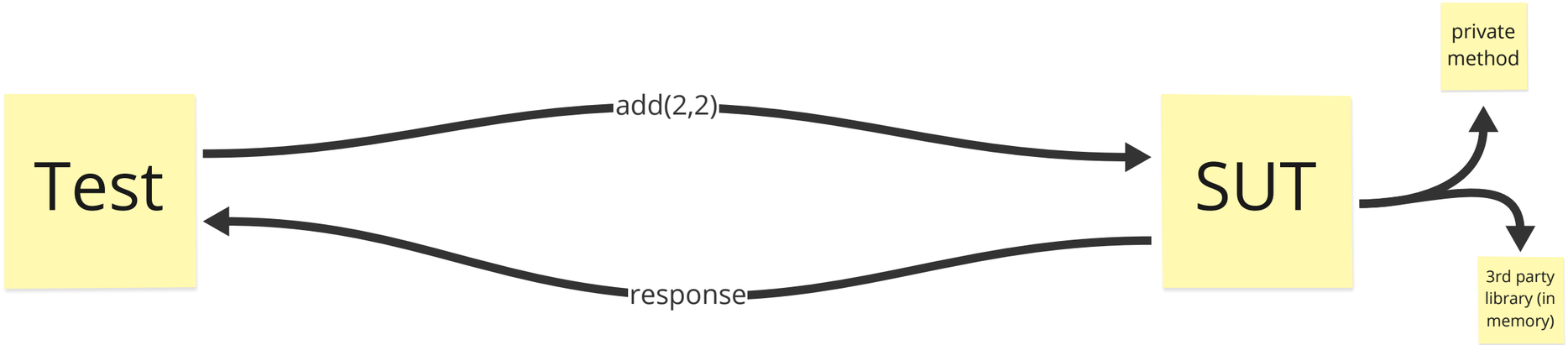
What makes a test 'good' for you?


Unit
Testing
Principles, Practices, and Patterns
Vladimir Khorikov
MANNING

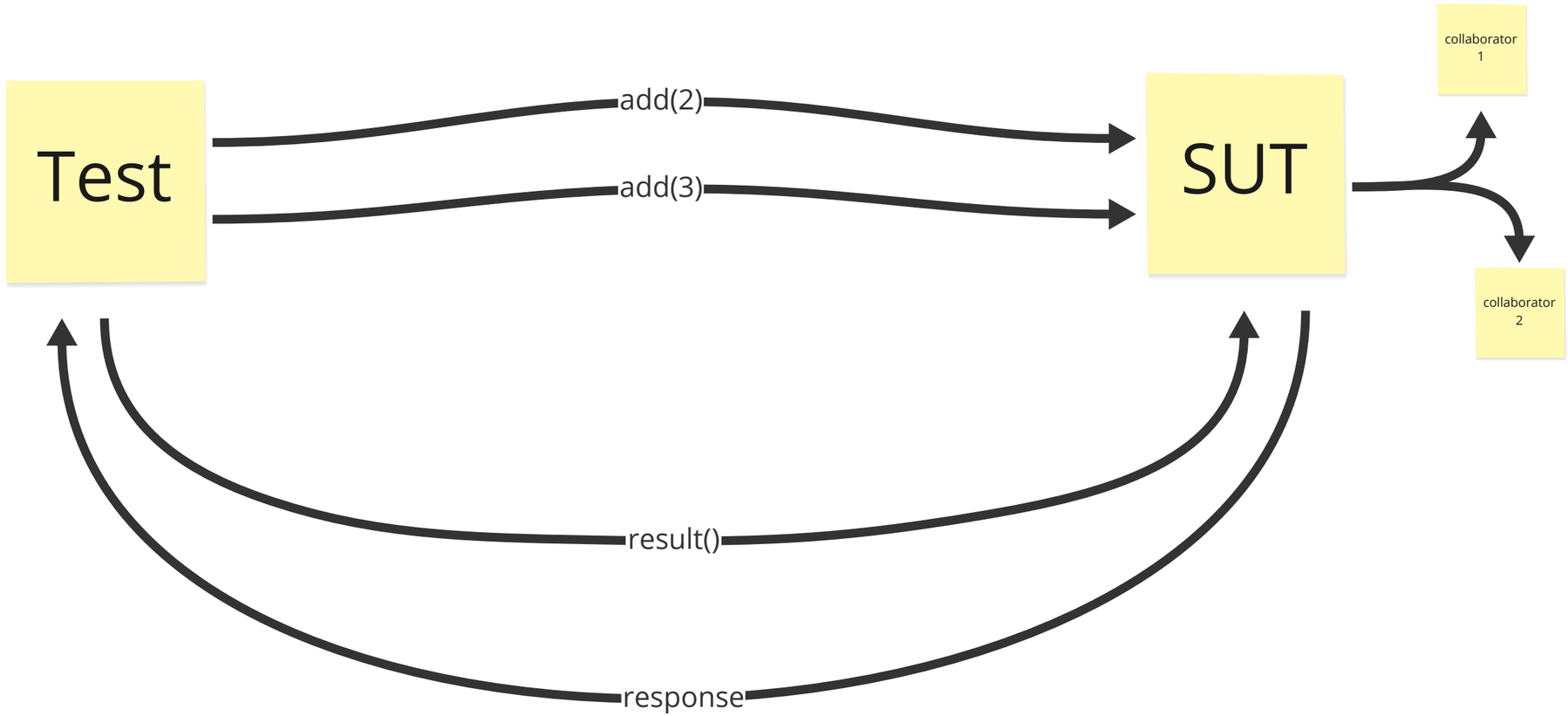
3 types of tests

- Output Based
- State Based
- Behaviour Based

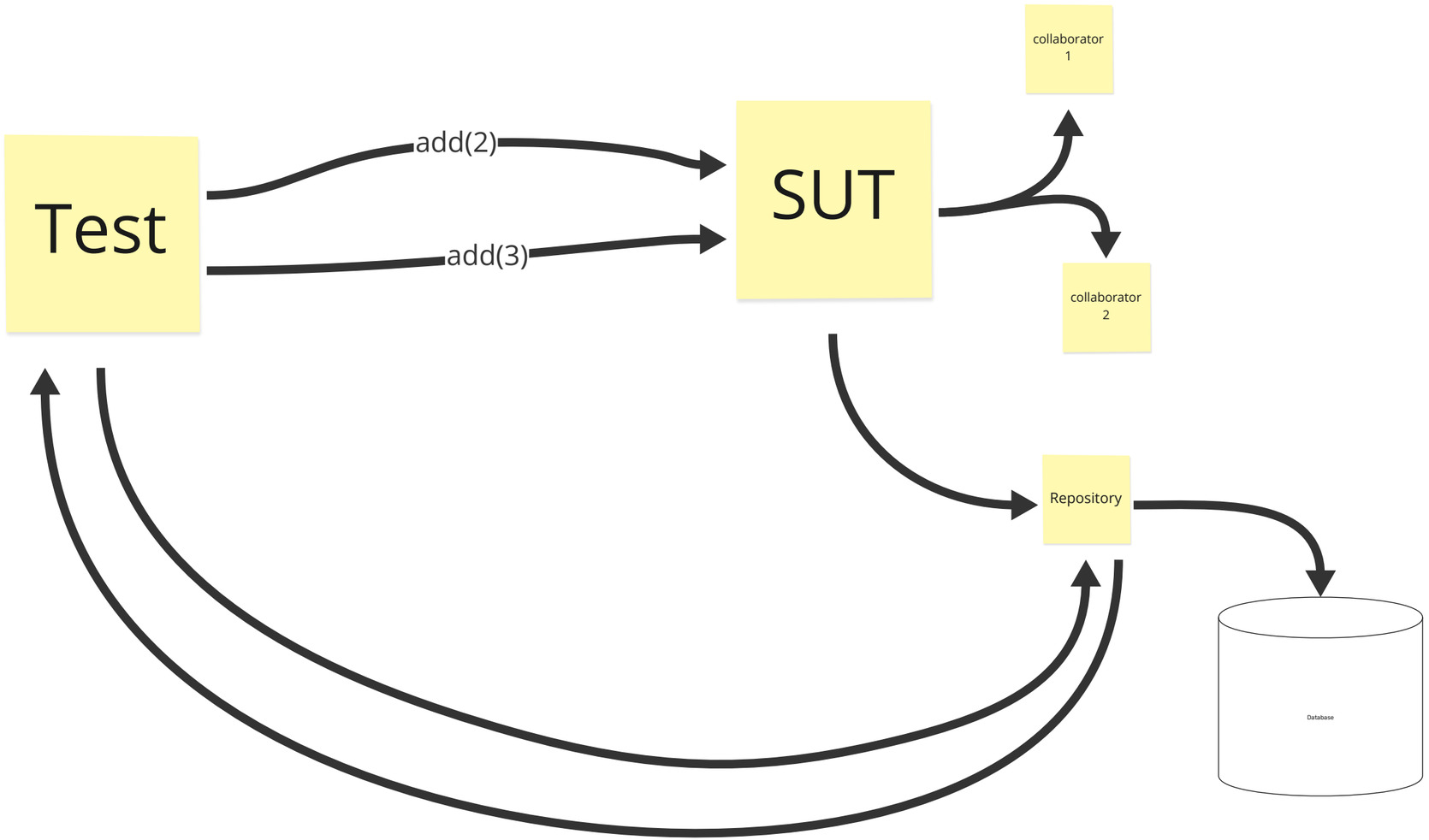
Output Based



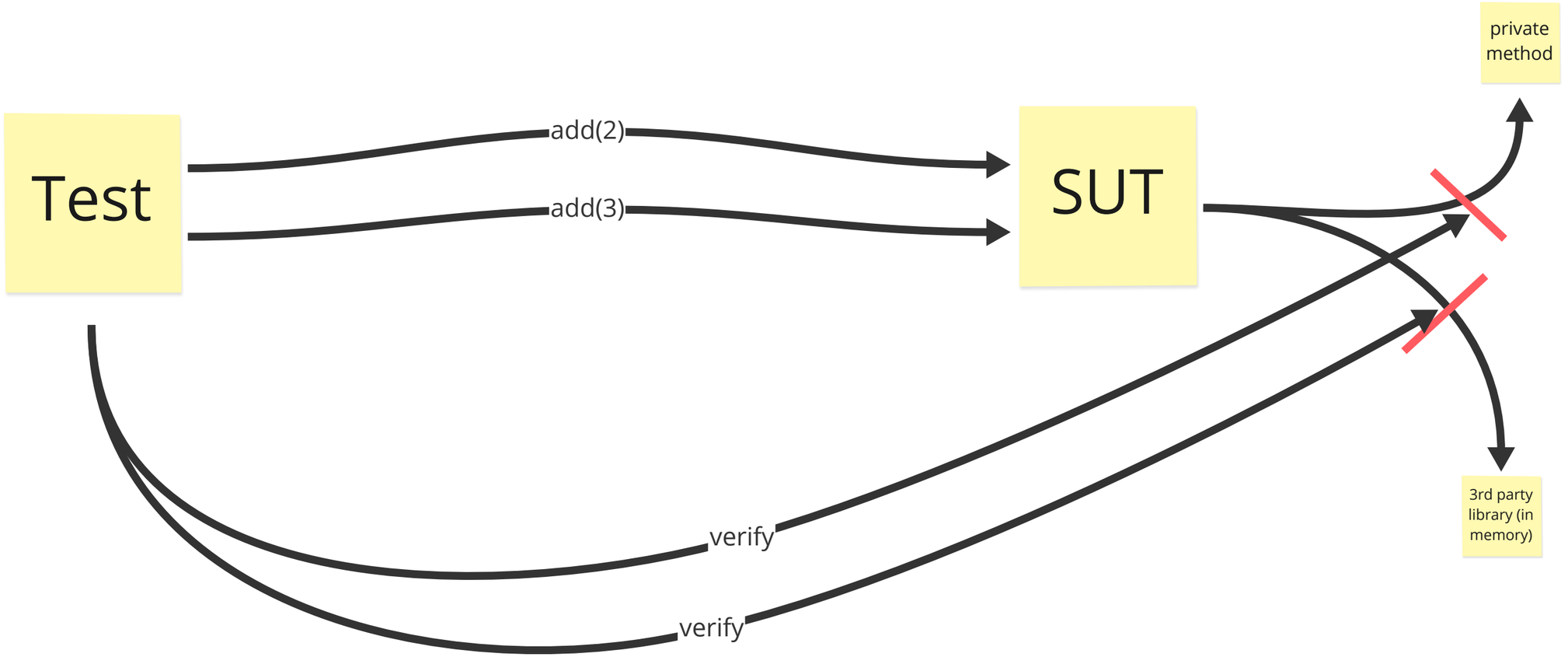
State Based


Test
add(2)
add(3)
SUT
collaborator 1
collaborator 2
result()
response

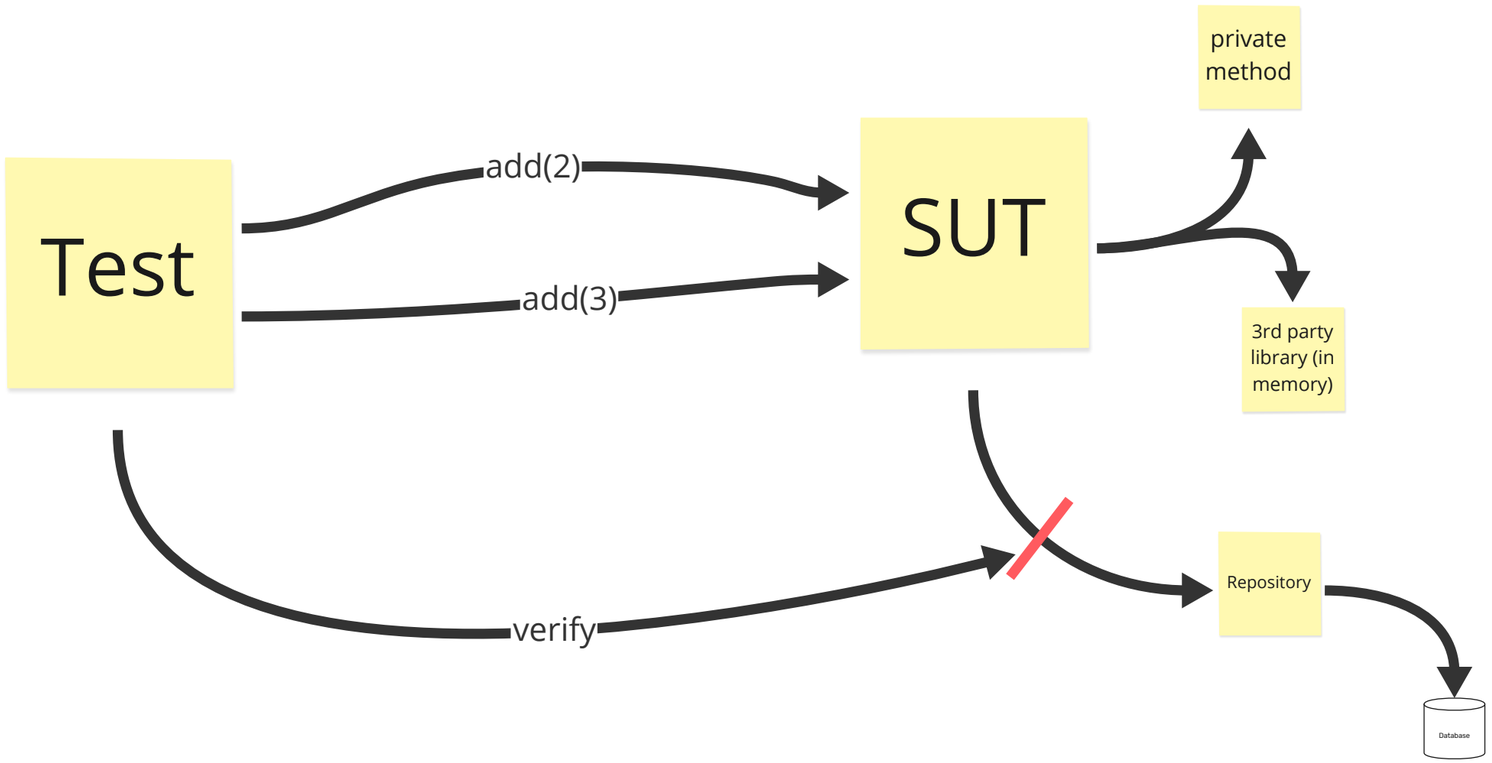
State Based 2



Verification Based



Verification Based 2



A Good (Unit) Test
(according to Vladimir Khorikov)

- Protection against regressions

- Resistance to refactoring

- Fast feedback

- Maintainability

Protection against regressions (more -> better)

- The amount of code that is executed (also: libs, frameworks etc)
- The complexity of that code
- The code’s domain significance

Protection against regressions why?

- we want to avoid false negatives: our tests are green, but the software is actually broken
- if that happens a few times, we loose trust in our test suite and need to start testing everything manually again

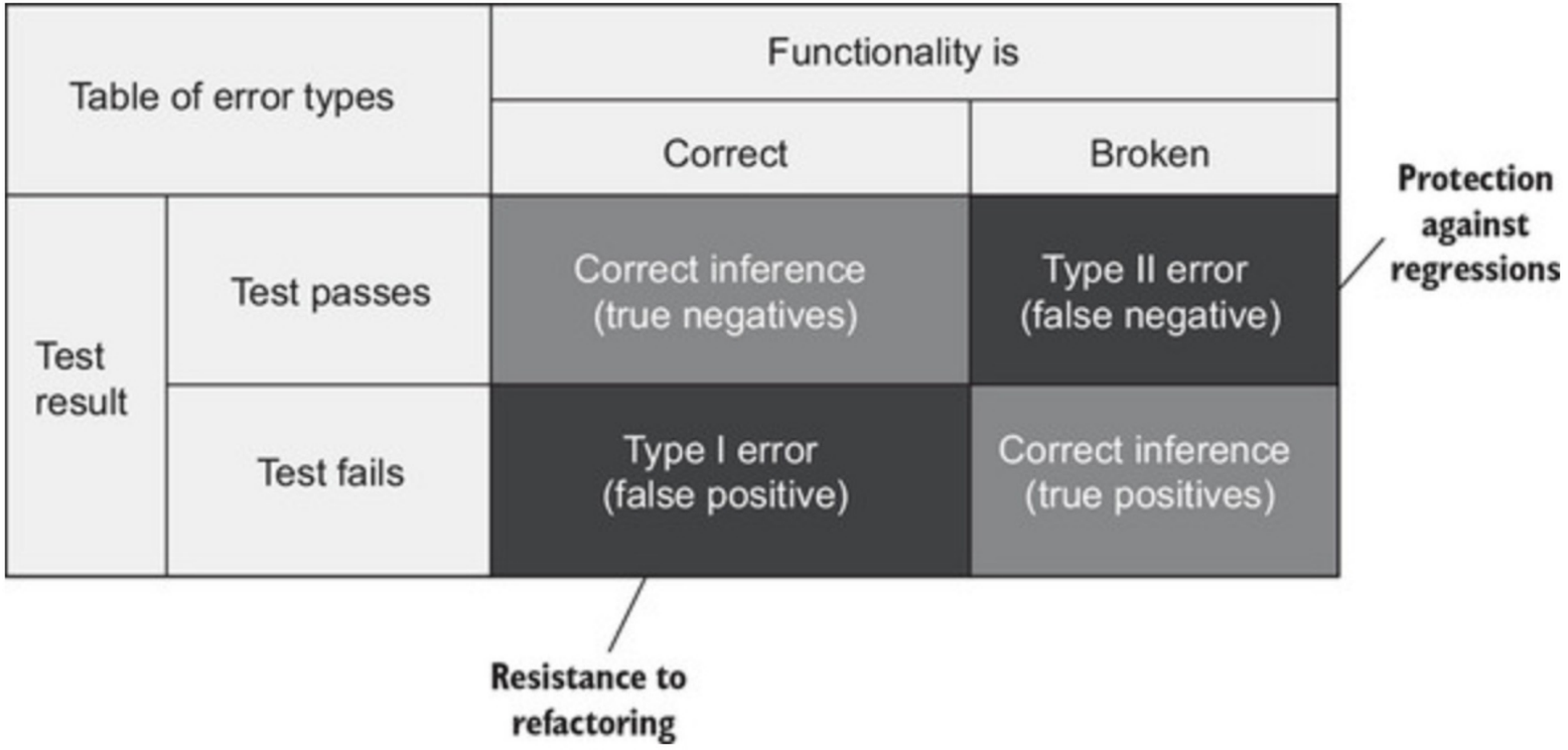
Resistance to refactoring (change structure w/o behaviour)

- false positives: red tests, although the application as a whole still works
- happens, when test focuses on implementation details & internals (how) rather than observable behaviour (what)
- takes a while to become a problem (until first refactoring)

Resistance to refactoring (change structure w/o behaviour)

- hinders refactoring if low
- people loose trust in their test suite and/or are not 'alarmed' when tests turn red anymore

Figure 4.3. The relationship between protection against regressions and resistance to refactoring. Protection against regressions guards against false negatives (type II errors). Resistance to refactoring minimizes the number of false positives (type I errors).



Fast feedback

- when tests are fast, we run them often
- when we run them often, the feedback loop becomes faster
- when we notice an error earlier, it's easier and thus cheaper to localize and fix

Maintainability

- how hard is it to understand (and change) the test?
- how hard is it to run the test? (database server, network issues etc, they all need to be maintained)

Calculation of value

- multiply:
- *protection against regressions[0..1]*
- *resistance to refactoring[0..1]*
- *fast feedback[0..1]*
- maintainability*[0..1]*

Perfect Test?

- *protection against regressions*, *resistance to refactoring*, and *fast feedback* are mutually exclusive

Figure 4.6. End-to-end tests provide great protection against both regression errors and false positives, but they fail at the metric of fast feedback.

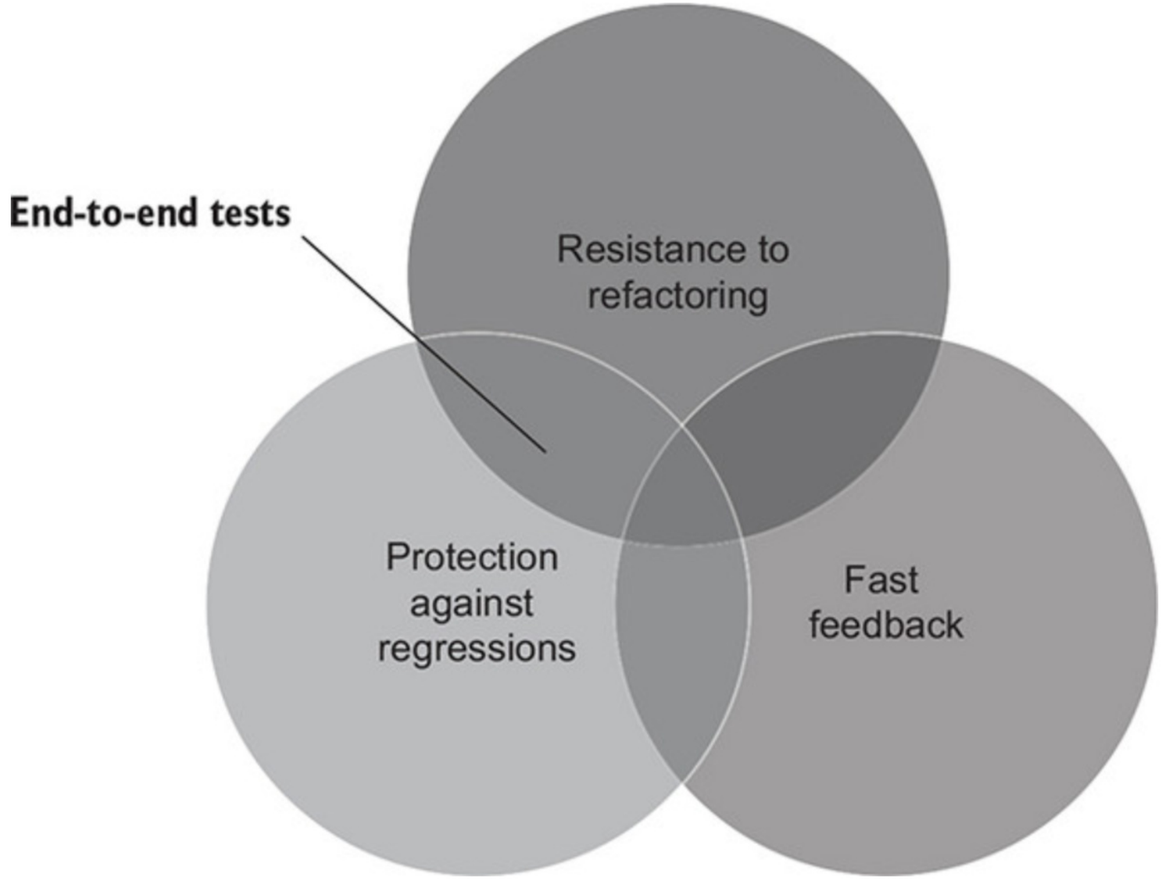


Figure 4.7. Trivial tests have good resistance to refactoring, and they provide fast feedback, but such tests don't protect you from regressions.

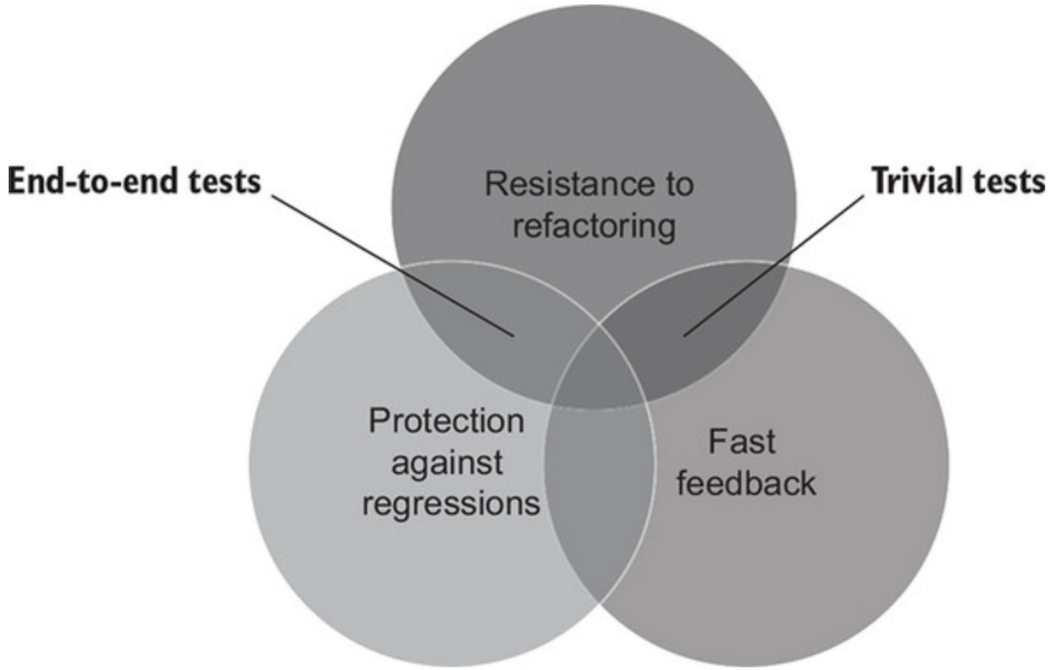
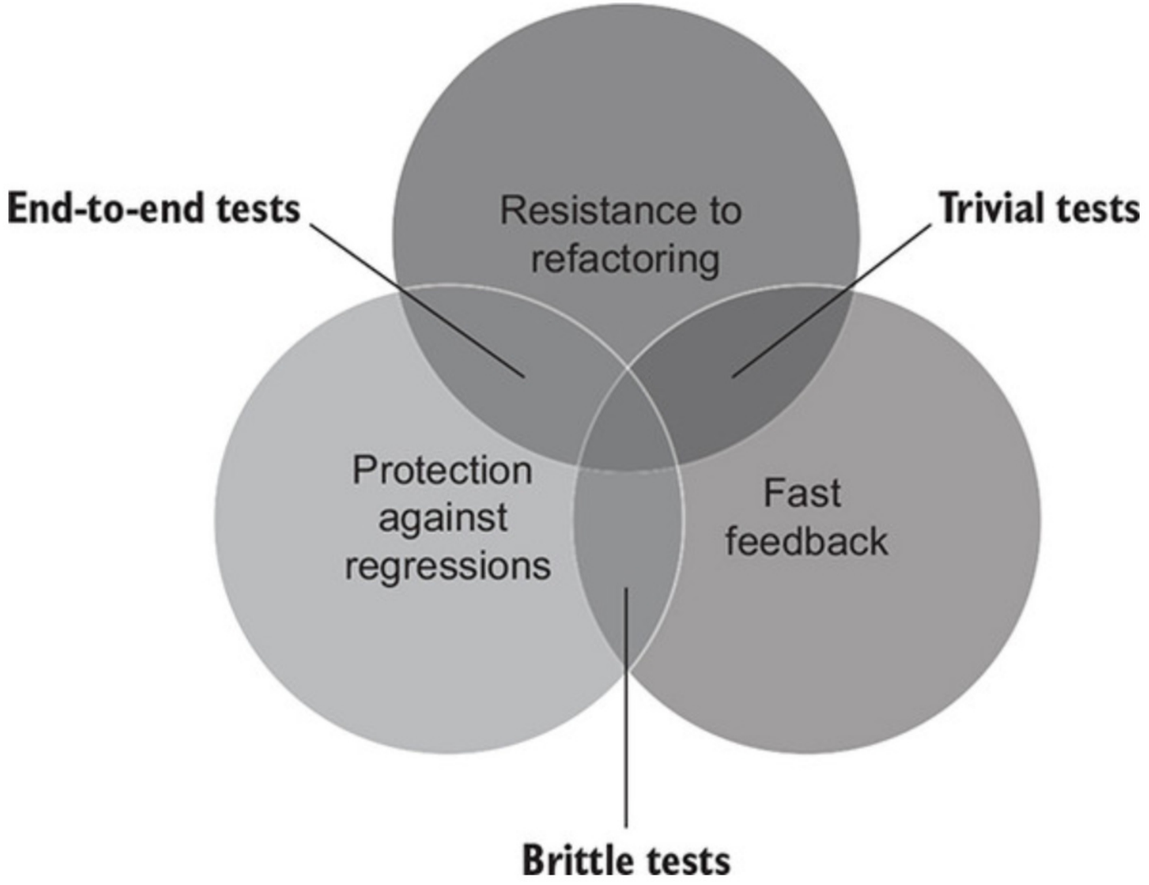


Figure 4.8. Brittle tests run fast and they provide good protection against regressions, but they have little resistance to refactoring.



```java
public class MessageRenderer implements Renderer {
    private List<Renderer> subRenderers;

    public MessageRenderer() {
        this.subRenderers = List.of(
                new HeaderRenderer(),
                new BodyRenderer(),
                new FooterRenderer()
        );
    }

    public List<Renderer> getSubRenderers() {
        return subRenderers;
    }

    public String render(Message message) {
        return subRenderers.stream()
                .map(x -> x.render(message))
                .reduce("", (l, r) -> l + r);
    }
}
```

```java
class HeaderRenderer implements Renderer {

    @Override
    public String render(Message message) {
        return String.format("<h1>%s</h1>", message.header());
    }
}
```

```java
class BodyRenderer implements Renderer {

    @Override
    public String render(Message message) {
        return String.format("<b>%s</b>", message.body());
    }
}
```

```java
class FooterRenderer implements Renderer {

    @Override
    public String render(Message message) {
        return String.format("<i>%s</i>", message.footer());
    }
}
```

```java
public class MessageRendererTest1 {
    @Test
    void message_renderer() {
        var sut = new MessageRenderer();

        var renderers = sut.getSubRenderers();
        assertEquals( expected: 3, renderers.size());
        assertTrue(renderers.get(0) instanceof HeaderRenderer);
        assertTrue(renderers.get(1) instanceof BodyRenderer);
        assertTrue(renderers.get(2) instanceof FooterRenderer);
    }

}
```

New table

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

```java
public class MessageRendererTest2 {

    @Test
    public void message_renderer() throws IOException {

        var location = System.getProperty("user.dir");
        var sourceCode = new String(Files.readAllBytes(Path.of(location, ...more: "src", "main", "java", "codingdojo",  "MessageRenderer.java")));
        assertEquals(sourceCode,  actual: """
package codingdojo;

import java.util.List;

public class MessageRenderer implements Renderer {
    private List<Renderer> subRenderers;

    public MessageRenderer() {
        this.subRenderers = List.of(
                new HeaderRenderer(),
                new BodyRenderer(),
                new FooterRenderer()
        );
    }

    public List<Renderer> getSubRenderers() {
        return subRenderers;
    }

    public String render(Message message) {
        return subRenderers.stream()
                .map(x -> x.render(message))
                .reduce("", (l, r) -> l + r);
    }
}
""");
    }
}
```

New table

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

```java
public class MessageRendererTest3 {
    @Test
    public void message_renderer() {
        var sut = new MessageRenderer();
        var message = new Message( header: "h",  body: "b",  footer: "f");
        String html = sut.render(message);
        assertEquals( expected: "<h1>h</h1><b>b</b><i>f</i>", html);
    }
}
```

New table

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

```java
public class MessageRendererTest4 {
    @Test
    public void message_renderer() throws URISyntaxException, IOException {
        var header = new String(Files.readAllBytes(Paths.get(getClass().getResource( name: "header.txt").toURI())));
        var body = new String(Files.readAllBytes(Paths.get(getClass().getResource( name: "body.txt").toURI())));
        var footer = new String(Files.readAllBytes(Paths.get(getClass().getResource( name: "footer.txt").toURI())));
        var message = new Message(header, body, footer);
        var sut = new MessageRenderer();

        String html = sut.render(message);

        assertTrue(html.startsWith("<h1>Lorem ipsum dolor sit amet"));
        assertTrue(html.endsWith("doloribus asperiores repellat.</i>"));
    }
}
```

New table

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

```java
public class MessageRendererTest5 {

    @Test
    public void message_renderer() {
        var sut = new MessageRenderer();

        var renderers = sut.getSubRenderers();

        assertNotNull(sut);
        assertNotNull(renderers);
    }
}
```

New table

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

```
class Adder {
    fun add(number1: Int, number2: Int): Int {
        return number1 + number2
    }
}


class Calculator(private val adder: Adder) {
    private var sum: Int = 0

    fun add(valueToBeAdded: Int) {
        val newSum = adder.add(sum, valueToBeAdded)
        this.sum = newSum
    }

    fun result(): Int {
        return this.sum
    }
}
```

1

```
@Test
fun `adding 1 and 2 equals 3`() {
    val result: Int = Adder().add(number1, number2)

    assertThat(result).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

2

```
@Test
fun `adding 1 and 2 equals 3 (mutable test)`() {
    val adder = Adder()
    val sut = Calculator(adder)

    sut.add(1)
    sut.add(2)

    assertThat(sut.result()).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

3

```
@Test
fun `adding 1 and 2 equals 3 (isolated test with stub)`() {
    val adder = mockk<Adder>()
    every { adder.add(number1, number2) } returns 1
    every { adder.add(number1, number2) } returns 3
    val sut = Calculator(adder)

    sut.add(1)
    sut.add(2)

    assertThat(sut.result()).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

4

```
@Test
fun `adding 1 and 2 equals 3 (isolated test with mock, behaviour verification)`() {
    val adder: Adder = mock()
    every { adder.add(number1, number2) } returns 1
    every { adder.add(number1, number2) } returns 3
    val sut = Calculator(adder)

    sut.add(1)
    sut.add(2)

    verify(exactly = 1) { adder.add(number1, number2) }
    verify(exactly = 1) { adder.add(number1, number2) }
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

5

```
@Test
fun `adding 1 and 2 equals 3 (isolated test with mock, behaviour verification, output verification)`() {
    val adder: Adder = mock()
    every { adder.add(number1, number2) } returns 1
    every { adder.add(number1, number2) } returns 3
    val sut = Calculator(adder)

    sut.add(1)
    sut.add(2)

    verify(exactly = 1) { adder.add(any(), any()) }
    assertThat(sut.result()).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

6

```
@Test
fun `adding 1 and 2 equals result of adder`() {
    val adder = Adder()
    val sut = Calculator(adder)

    sut.add(1)
    sut.add(2)

    val result = adder.add(number1, number2)
    assertThat(sut.result()).isEqualTo(result)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

```
class Adder {
    fun add(number1: Int, number2: Int): Int {
        return number1 + number2
    }
}
```

```
class Calculator(private val adder: Adder) {
    private var sum: Int = 0

    fun add(valueToBeAdded: Int) {
        val newSum = adder.add(sum, valueToBeAdded)
        this.sum = newSum
    }

    fun result(): Int {
        return this.sum
    }
}
```

```
interface SumRepository {
    fun saveValue(sum: Int)
    fun fetchSum(): Int
}
```

```
class FileSumRepository(tempDir: Path) : SumRepository {
    private val filePath: String = "$tempDir/sum-persistence.txt"

    override fun saveValue(sum: Int) {
        File(this.filePath).printWriter().use { out ->
            out.print(sum.toString())
        }
    }

    override fun fetchSum(): Int {
        File(this.filePath).let { file ->
            if (file.exists()) {
                file
                    .readText()
                    .trim()
                    .toInt()
            } else {
                0
            }
        }
    }
}
```

```
class FakeSumRepository : SumRepository {
    private var privateSum = 0

    override fun saveValue(sum: Int) {
        this.privateSum = value
    }

    override fun fetchSum(): Int {
        return this.privateSum
    }
}
```

7

```
@Test
fun `sum is persisted by repository (test with mock and stub)`() {
    val adder = Adder()
    val repository = FakeSumRepository()
    val sut = Calculator4WithPersistentAdder(adder, repository)

    sut.add(1)
    sut.add(2)

    val sum: Int = repository.fetchSum()
    assertThat(sum).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

8

```
@Test
fun `sum is persisted by repository (test with mock and stub)`() {
    val adder = Adder()
    val repository = FakeSumRepository()
    val sut = Calculator4WithPersistentAdder(adder, repository)

    sut.add(1)
    sut.add(2)

    val sum: Int = repository.fetchSum()
    assertThat(sum).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

9

```
@Test
fun `sum is persisted by repository (test with mockk stub 1)`() {
    val adder = Adder()
    val sumRepository = mockk<SumRepository>()
    every { sumRepository.saveValue(any()) } just Runs
    every { sumRepository.fetchSum() } returns 0 andThen 1 andThen 2
    val sut = Calculator4WithPersistentAdder(adder, sumRepository)

    sut.add(1)
    sut.add(2)

    val sum: Int = sumRepository.fetchSum()
    assertThat(sum).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

10

```
@Test
fun `sum is persisted by repository (test with mockk stub 1)`() {
    val adder = Adder()
    val sumRepository = mockk<SumRepository>()
    every { sumRepository.saveValue(any()) } just Runs
    every { sumRepository.fetchSum() } returns 0 andThen 1 andThen 2
    val sut = Calculator4WithPersistentAdder(adder, sumRepository)

    sut.add(1)
    sut.add(2)

    val sum: Int = sumRepository.fetchSum()
    assertThat(sum).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

11

```
@Test
fun `sum is persisted by repository (test with mockk mock)`() {
    val adder = Adder()
    val sumRepository = mockk<SumRepository>()
    every { sumRepository.saveValue(any()) } just Runs
    every { sumRepository.fetchSum() } returns 3 andThen 1
    val sut = Calculator4WithPersistentAdder(adder, sumRepository)

    sut.add(1)
    sut.add(2)

    verify(exactly = 1) { sumRepository.saveSum(value:3) }
    verify(exactly = 1) { sumRepository.saveSum(value:1) }
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

12

```
@Test
fun `sum is persisted to file (test with real FileSumRepository)`(@TempDir tempDir: Path) {
    val adder = Adder()
    val repository = FileSumRepository(tempDir)
    val sut = Calculator4WithPersistentAdder(adder, repository)

    sut.add(1)
    sut.add(2)

    val result = File("$tempDir/sum-persistence.txt").let { file ->
        file
            .readText()
            .trim()
            .toInt()
    }
    assertThat(result).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

13

```
@Test
fun `sum is persisted by repository (test with real FileSumRepository)`(@TempDir tempDir: Path) {
    val adder = Adder()
    val repository = FileSumRepository(tempDir)
    val sut = Calculator4WithPersistentAdder(adder, repository)

    sut.add(1)
    sut.add(2)

    val sum: Int = repository.fetchSum()
    assertThat(sum).isEqualTo(3)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

14

```
@Test
fun `persists one`(@TempDir tempDir: Path) {
    val sut = FileSumRepository(tempDir)
    sut.saveSum(value:1)

    sut.saveSum(value:2)

    val result = sut.fetchSum()
    assertThat(result).isEqualTo(2)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

15

```
@Test
fun `can is zero when no sum has been saved yet`(@TempDir tempDir: Path) {
    val sut = FileSumRepository(tempDir)

    val result = sut.fetchSum()

    assertThat(result).isEqualTo(0)
}
```

	Protection against regressions	Resistance to Refactoring	Fast Feedback	Maintainability
1				
0.75				
0.5				
0.25				
0				

What is your main takeaway from this Session?

In what way(s) can the 4 pillars help you in your daily work?

codecentric

Senior IT Consultant

ferdinand.ade@codecentric.de

/in/ferdinand-ade/