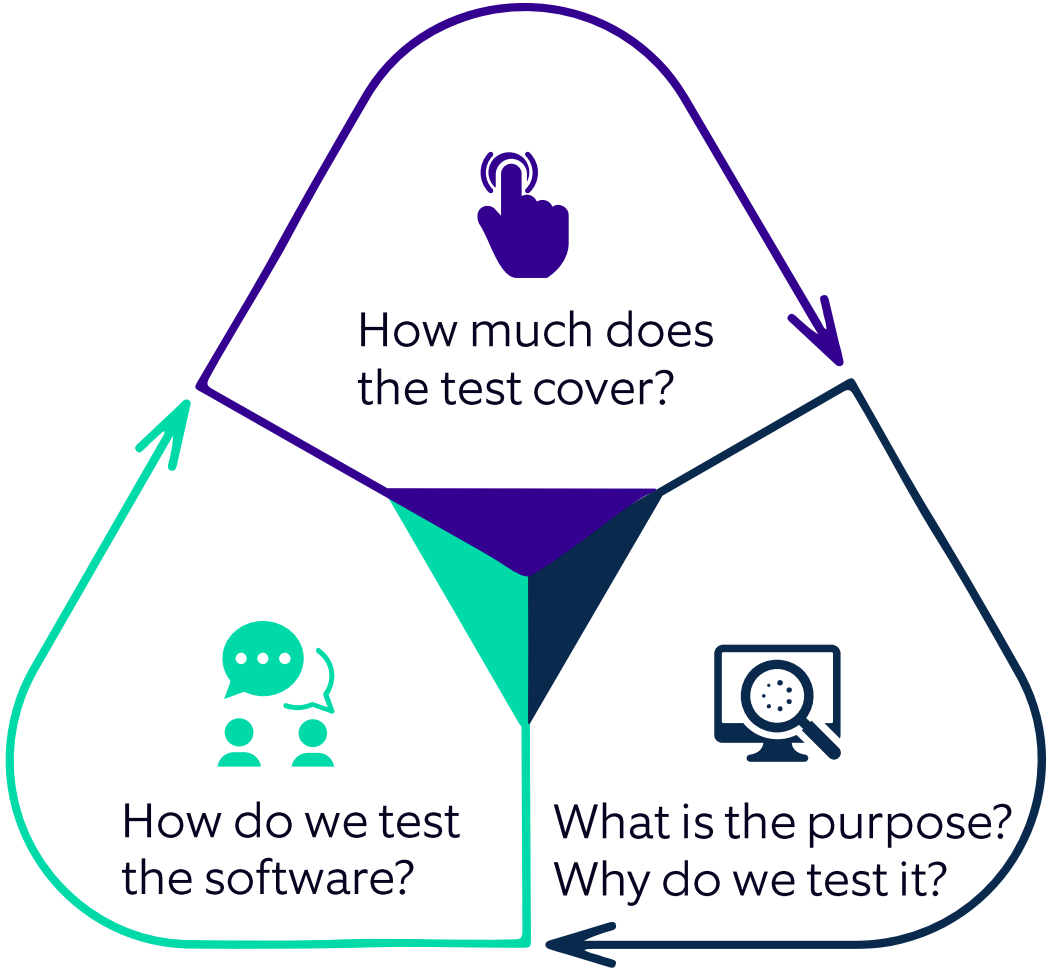
nagarro

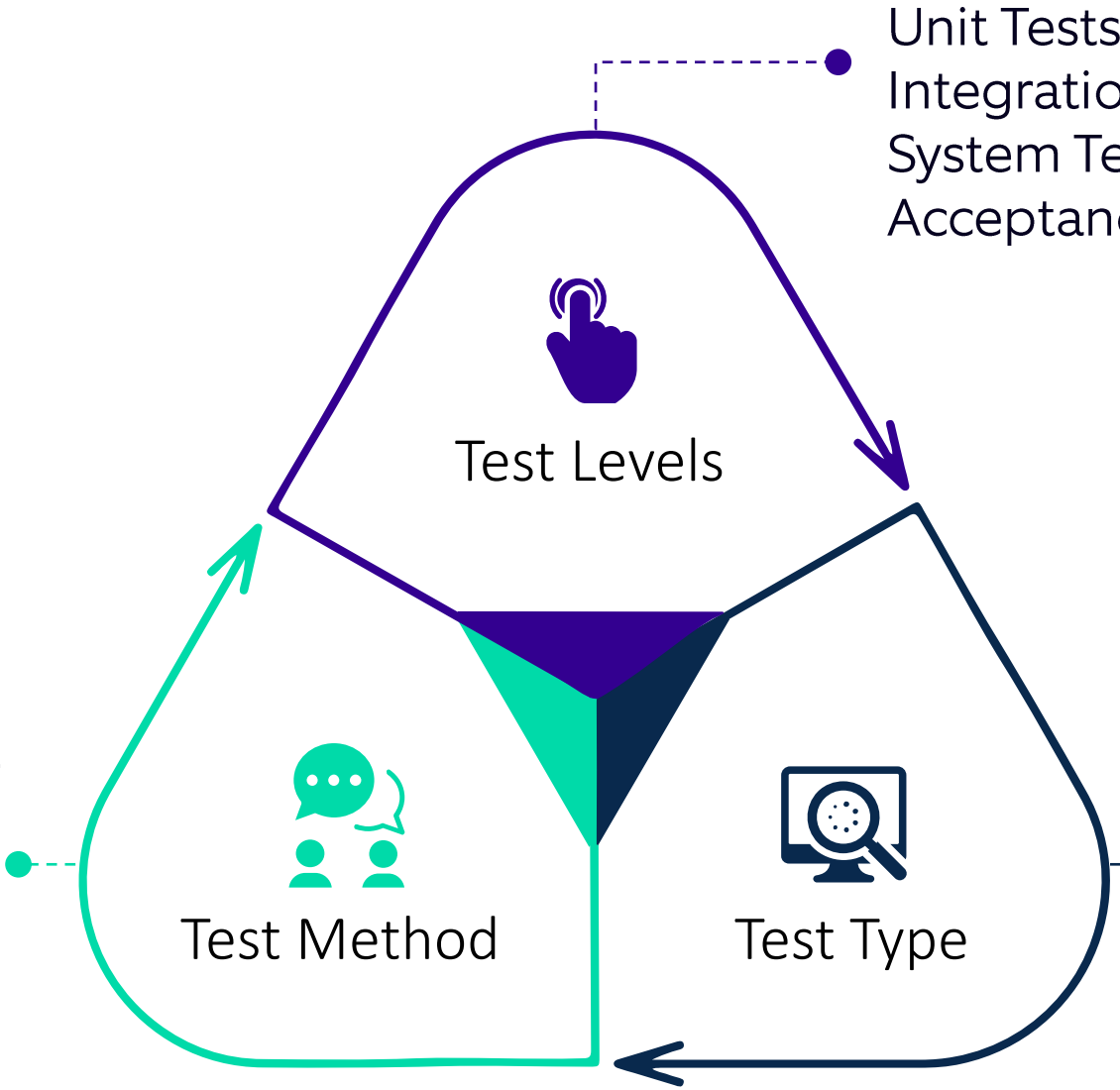
03

Testing Theory, Practice and Automation

Classifications of Tests



Classifications of Tests

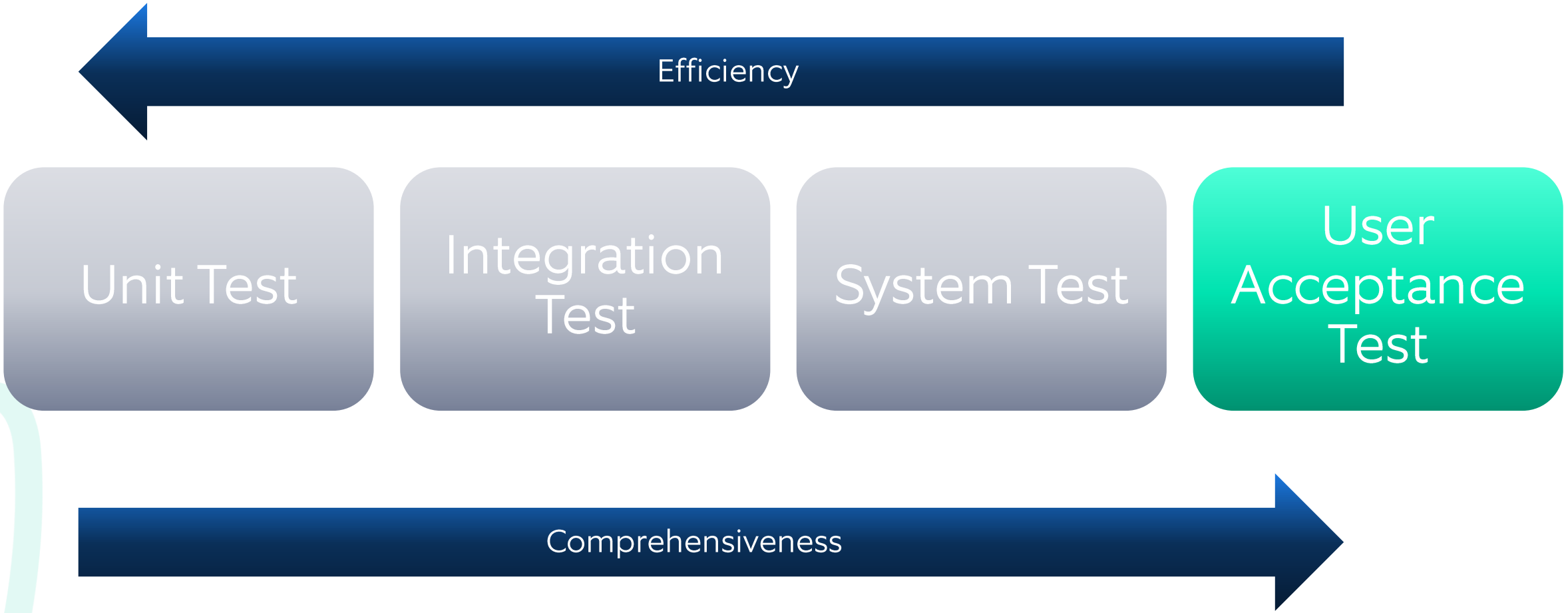


Unit Tests,
Integration Test,
System Test,
Acceptance Tests

black-box vs. white-box,
manual vs. automated

Functional Test,
Non-functional Test,
Smoke Tests,
Regression Test

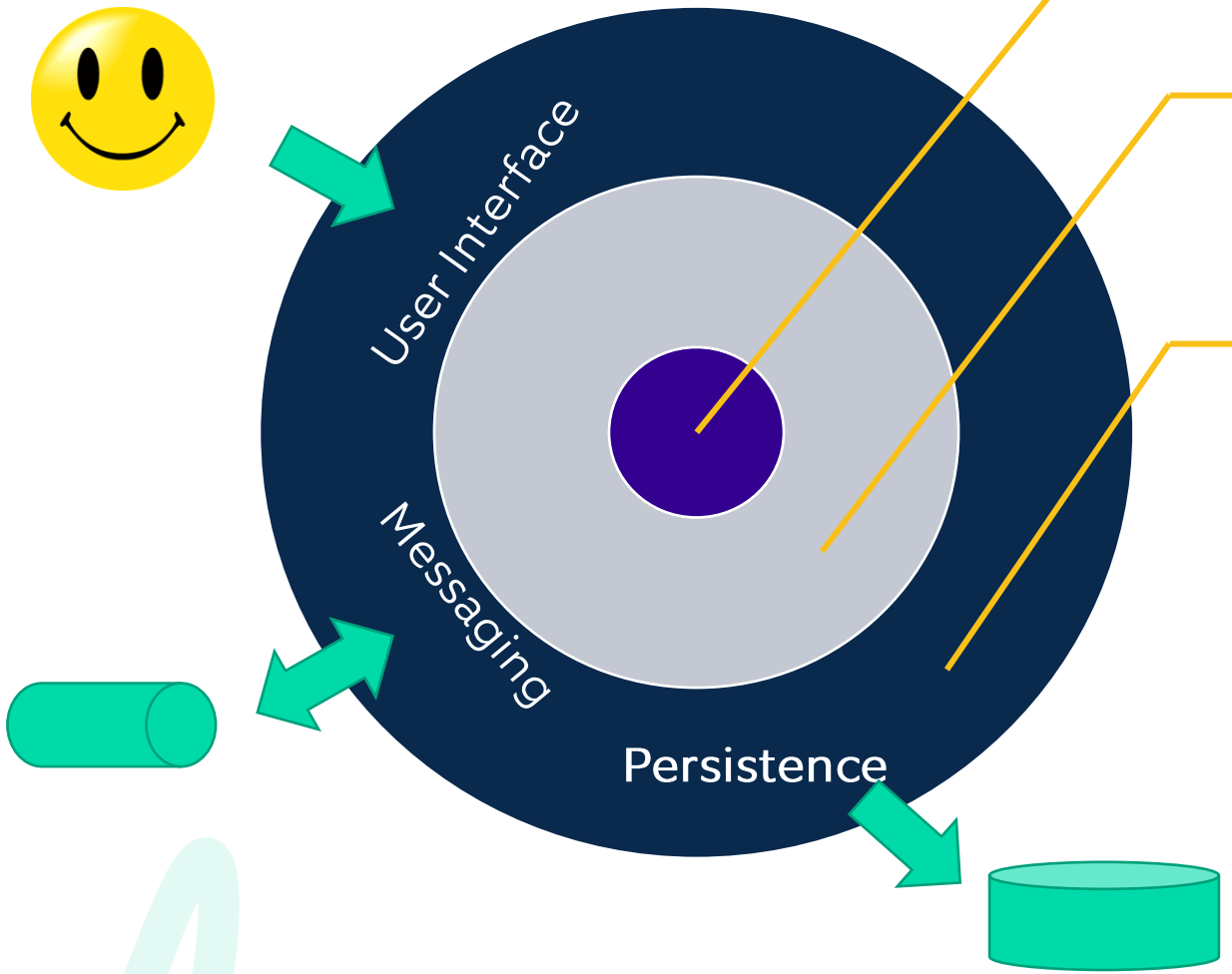
Levels of Testing



Comparing Different Levels of Tests

Test Type	What	Who	Tools	Execution
Unit Test	Methods and Classes	Developer	xUnit	Development, Continuous Integration
Integration Test	Interfaces, Services, Modules	Developer, Tester	xUnit, Scripts, DSL	Test Team, Continuous Integration
System Test	GUI, Database, Webservice	Tester	(xUnit), Mercury, Rational Functional Tester	Test Team, Continuous Integration
User Acceptance Test	System Business Requirements	User, Customer		Running System

Levels of Tests in the Architecture



Domain, Business Logic
→ Unit Tests

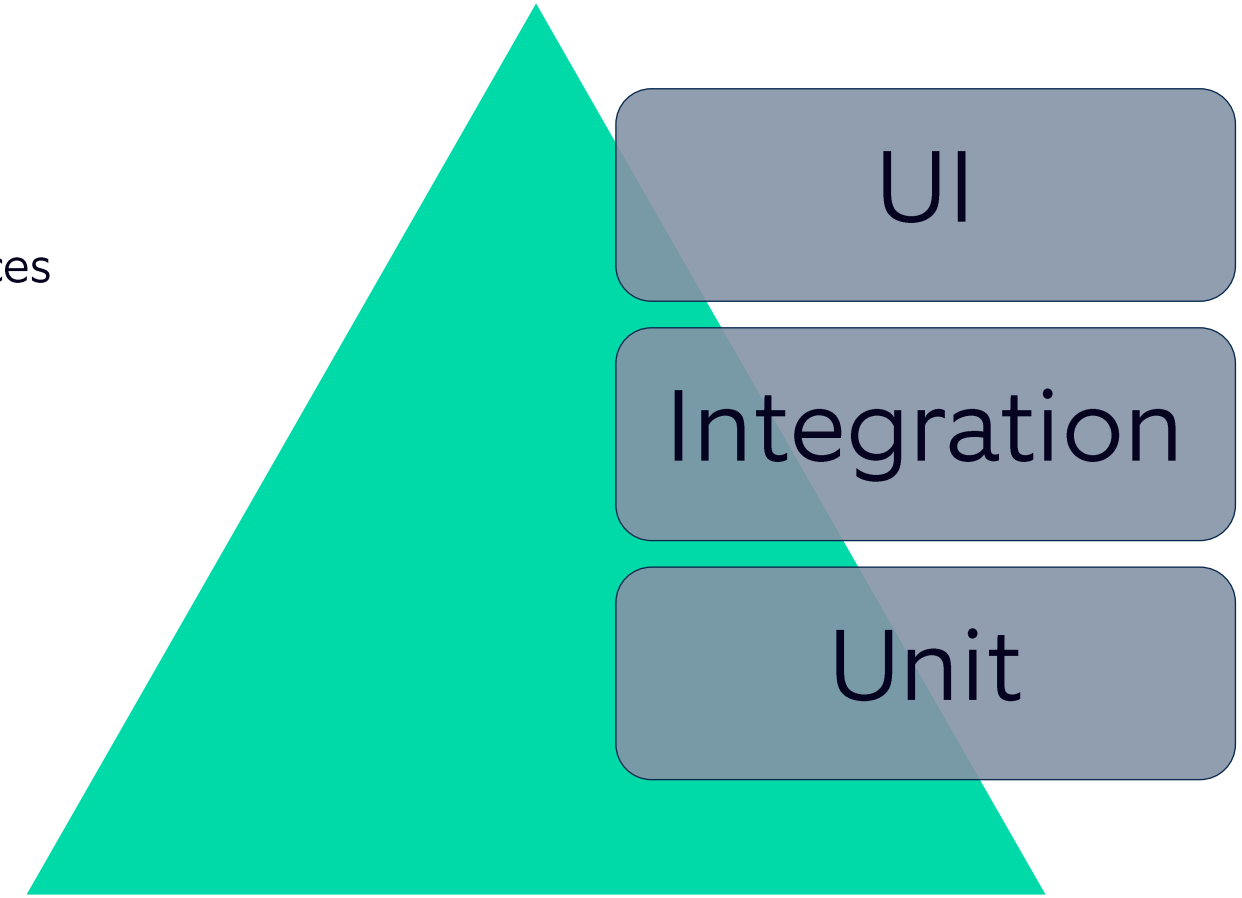
Application Services
→ Integration Tests

Infrastructure, User Interfaces
→ System Tests

The Test Automation Pyramid

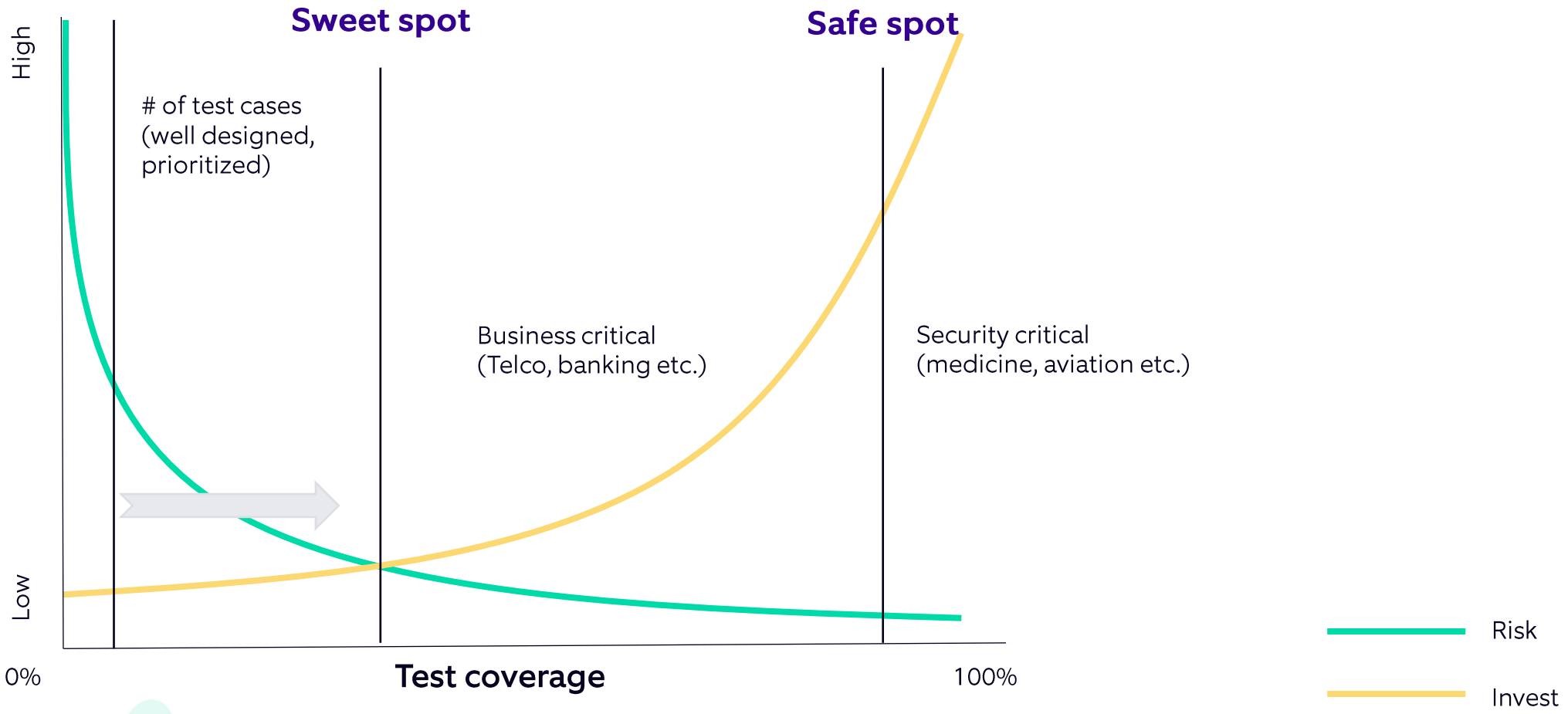
Looks like test levels, but …

- UI or End-To-End testing (E2E)
  - Use-Case level
  - Via User Interface or other interfaces
- Integration testing
  - Service interfaces
- Unit testing
  - Components
  - Lowest level

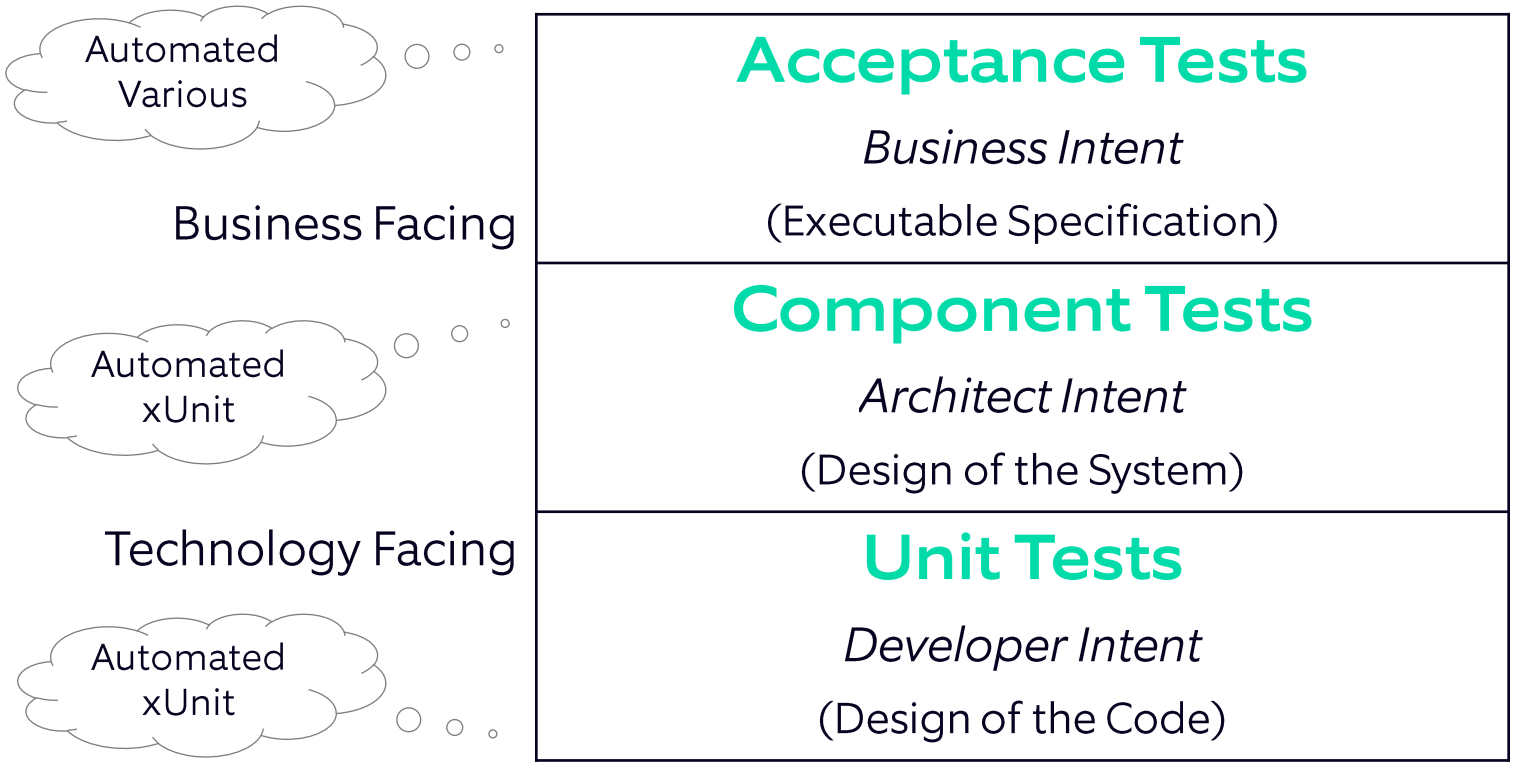


Quality Assurance

How much test automation is enough?



Test Automation Strategy – per functionality



Based on: http://xunitpatterns.com/TestStrategy.html

nagarro

Unit Test Definition

Unit Test Definition

*A **unit** is considered the smallest testable part of an application.*

*A **unit test** is an automated test to determine whether a small piece of code behaves as expected.*

What are Unit Tests?

Unit Tests

- Tests made by Developers
- Test is a small functional piece of code
- Preferably created during the implementation, but can be created after implementation

xUnit

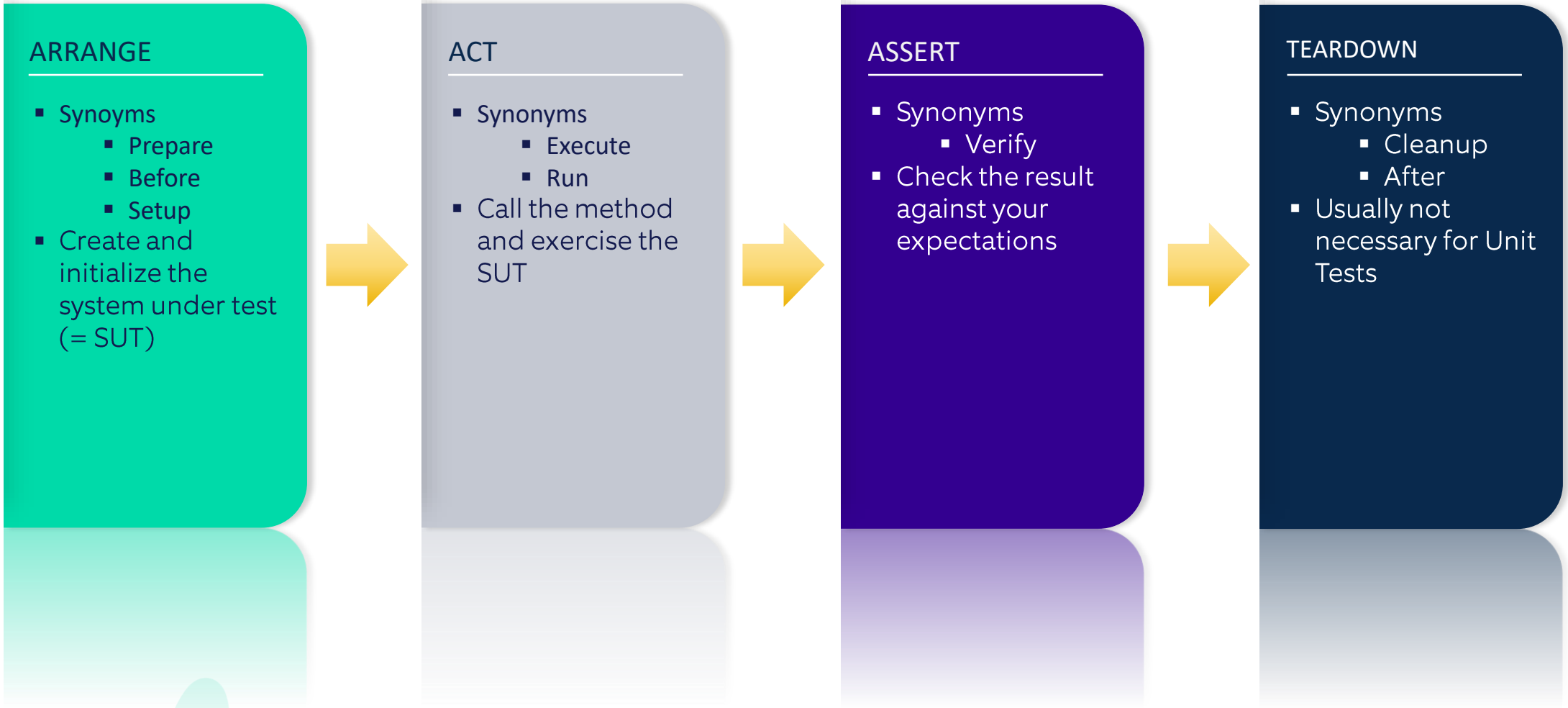
- Developer Frameworks for testing
- Primary used in Unit tests, but can be used to execute other test types

Benefits of Unit Tests

- Less bugs
- Bugs can be localized more easily
- Bugs are fixed more easily
- New features are easier implemented
- Comprehensibility of the code
- Documentation of the code

Unit tests provide the programmer with continuous feedback on the correctness of the code!

Unit Test Phases



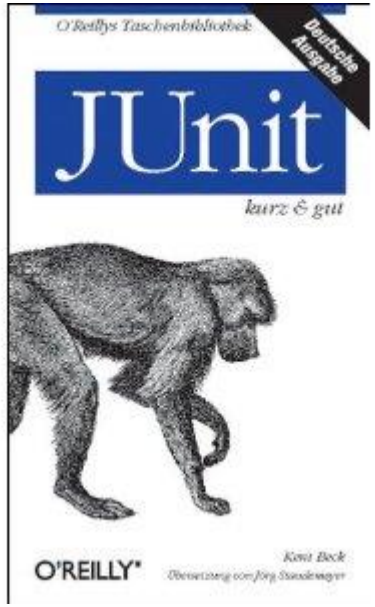
http://xunitpatterns.com/Four%20Phase%20Test.html

nagarro

xUnit Basics

What is xUnit?

Smalltalk powered by



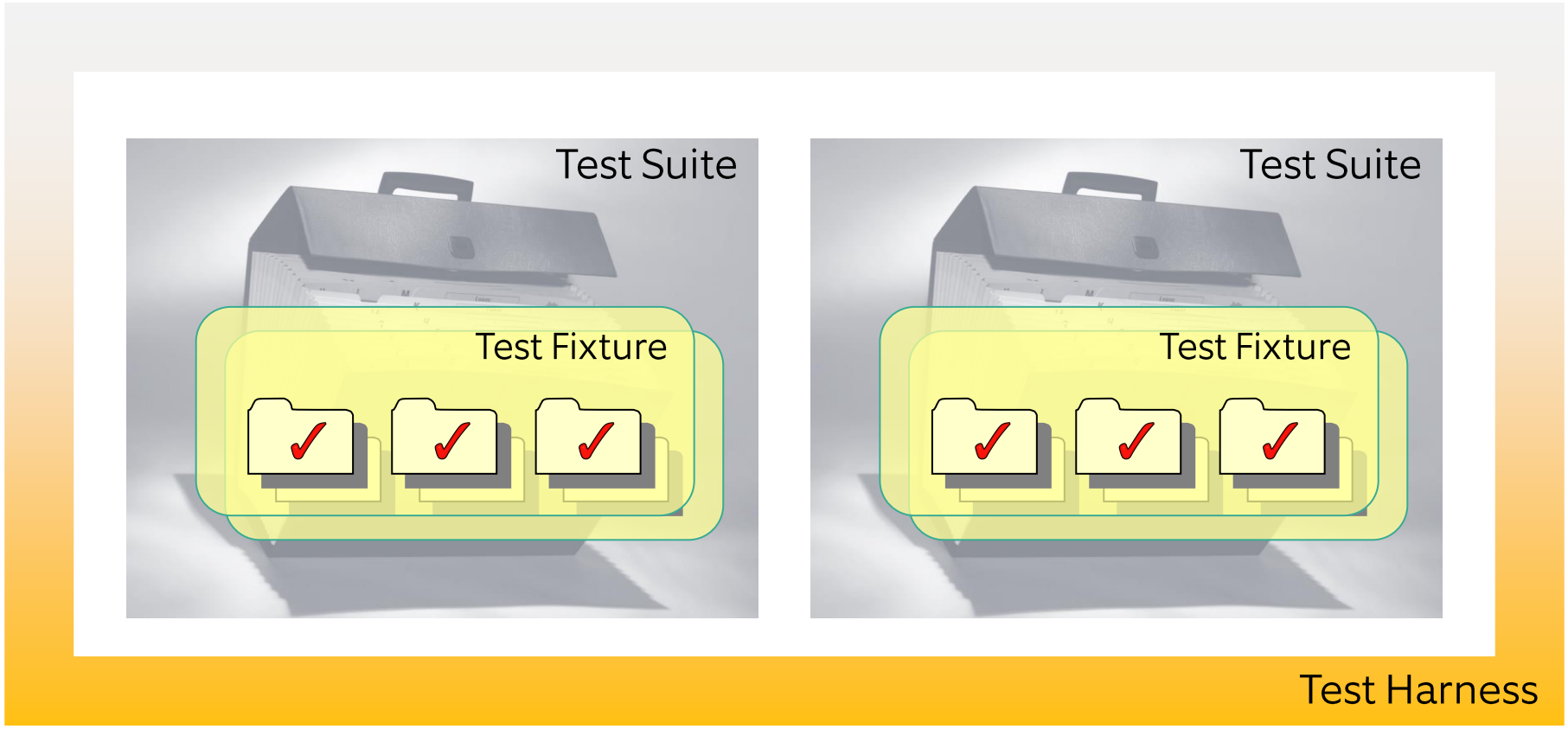
xUnit.net

What is xUnit?

- Developer frameworks for testing
- Primary used for Unit tests, but can be used to execute other types of tests
- Collective name for several unit testing frameworks
- Structure and functionality is derived from SUnit (Smalltalk), designed by Kent Beck in 1998
- Examples:
  - JUnit (ported to Java by Kent Beck and Erich Gamma)
  - NUnit
  - xUnit.net

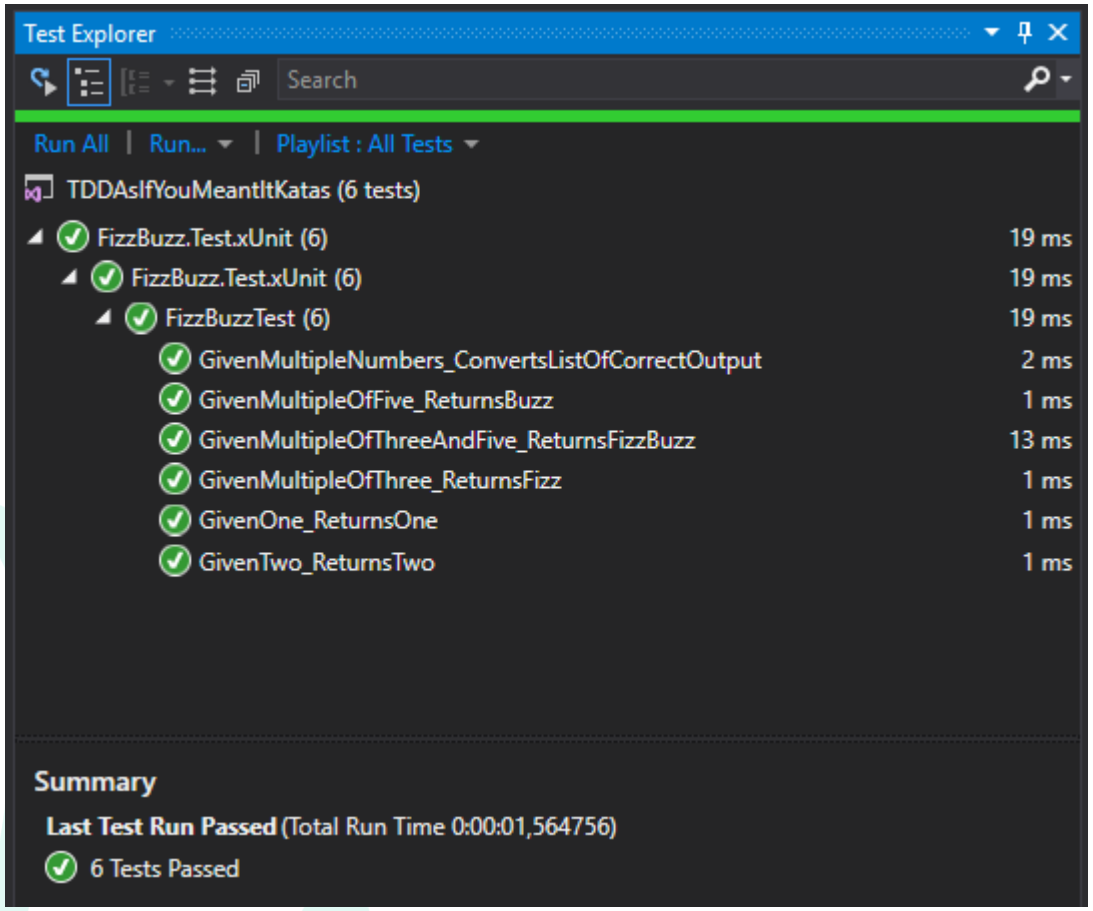
https://en.wikipedia.org/wiki/XUnit

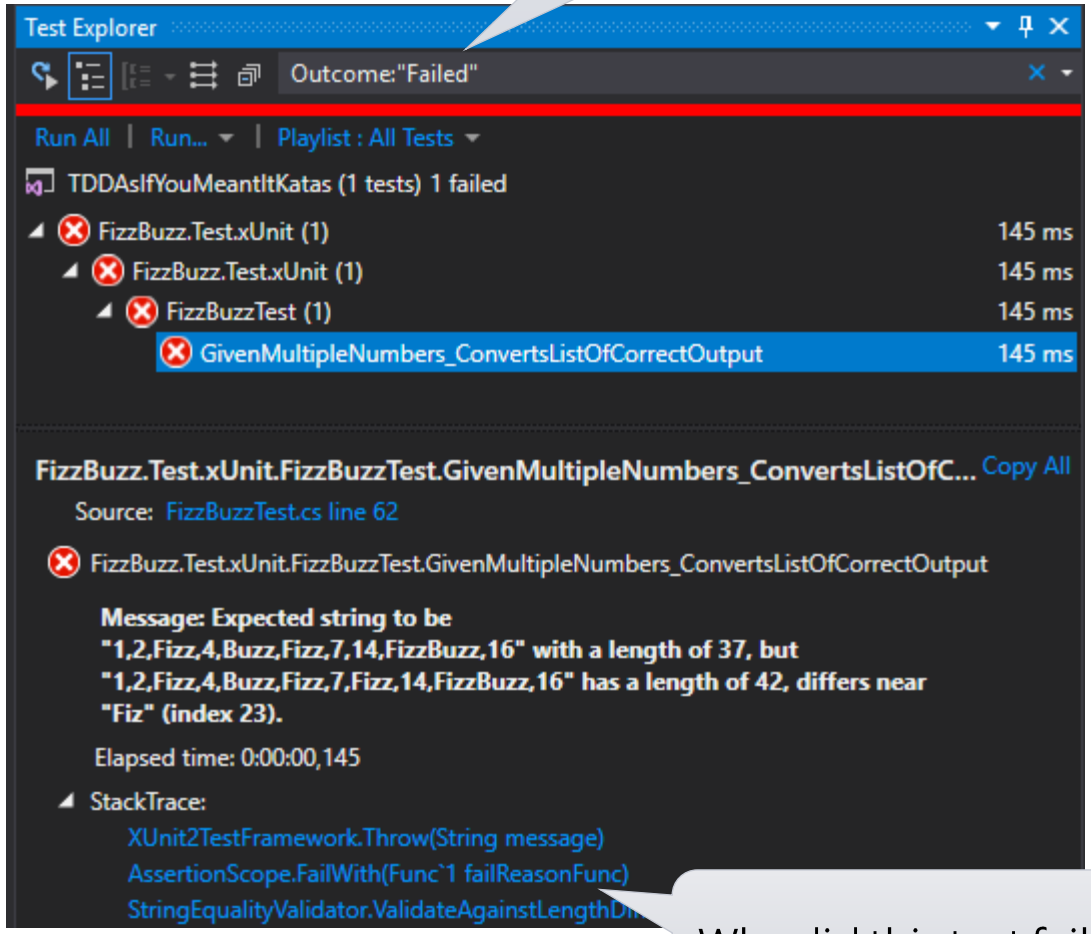
Unit Test Components (2)



Visual Studio Test Explorer

Green or Red





xUnit.net – Our First Test

```
using Xunit;
using XUnitExamples;

namespace XUnitExamplesTest
{
    public class PersonTest
    {
        [Fact]
        public void FullAge_Persons()
        {
            // setup
            Person mrSmith = new Person(name: "John Smith", age: 37);

            // exercise
            bool result = mrSmith.IsFullAge();

            // verify
            Assert.True(result);
        }
    }
}
```

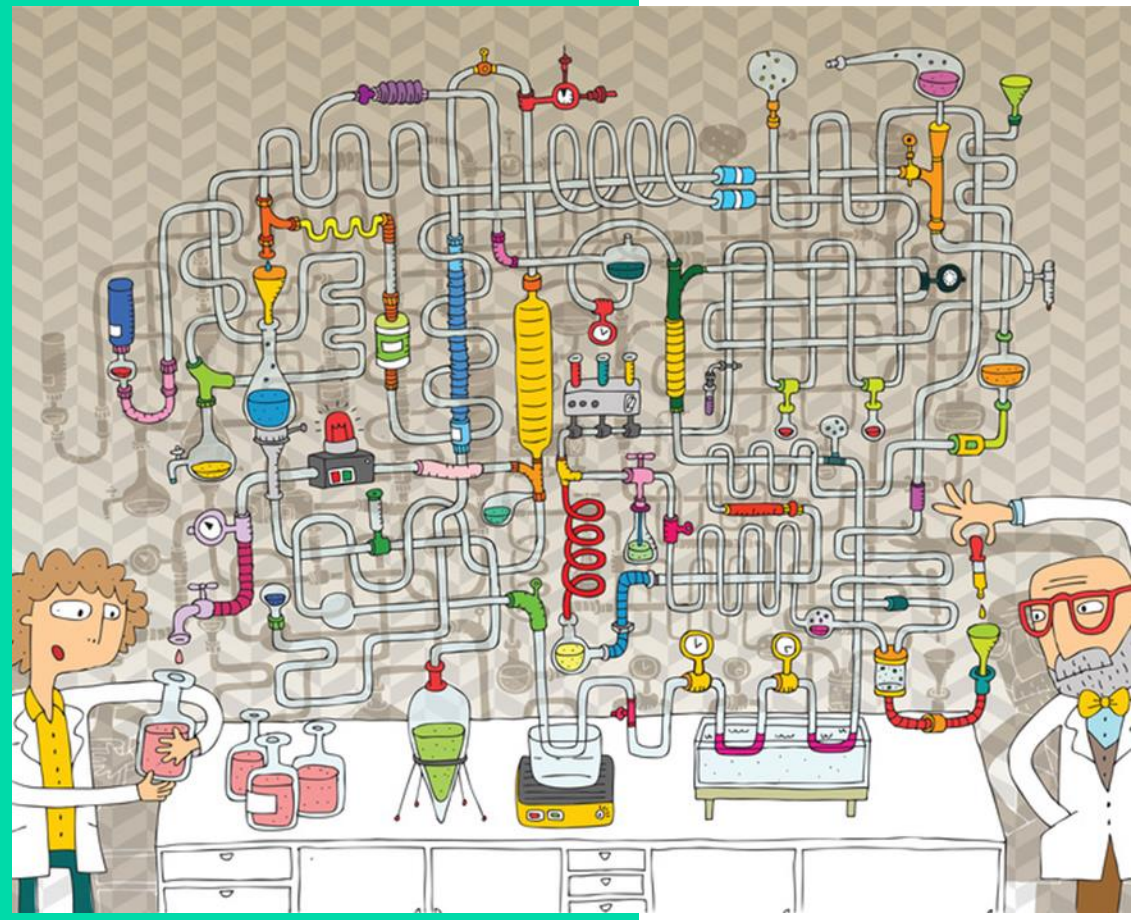
xUnit.net – Assertions

Built-In Assertions

```
[Fact]
public void BuiltIn()
{
    Assert.Equal("xUnit training@Nagarro", "xUnit training@Nagarro");
    Assert.True("xUnit training@Nagarro".Contains("xUnit"));
    Assert.NotEqual("xUnit", "NUnit");

    List<string> collection = new List<string>() { "xUnit", "JUnit", "NUnit" };
    Assert.Contains("xUnit", collection);
}

[Fact]
public void FluentAssertions()
{
    string testingFramework = "xUnit";
    testingFramework.Should().StartWith("x");

    testingFramework.Should().NotBeNullOrEmpty();

    List<string> collection = new List<string>() { "xUnit", "JUnit", "NUnit" };
    collection.Should().Contain("xUnit");
}
```

Fluent Assertions

Further Concepts



- Data-Driven-Tests
- Exception-Handling
- Category Annotation
- Test-Suites
- Custom Assertions
- Test-Builder
- Build Process Integration

Hands-On: „xUnit.net Basics“

Implement unit tests to solve the following tasks:

- Implement a new class Calculator with the method double Divide(doule, double)
- Create a corresponding test project and test class for Calculator.Divide() and create at least the 3 test cases (see table)
- Note: throw not finite number exception if divisor is NaN
- Implemement 3 different types of assertions for the first case
- Learning objectives
  - xUnit.net Basics
  - Comparison of assertion types

Dividend	Divisor	Ergebnis
Double	Double	Double
Double	0.0	PositiveInfinity
Double	PositiveInfinity	0.0
Double	Nan	NotFiniteNumberException

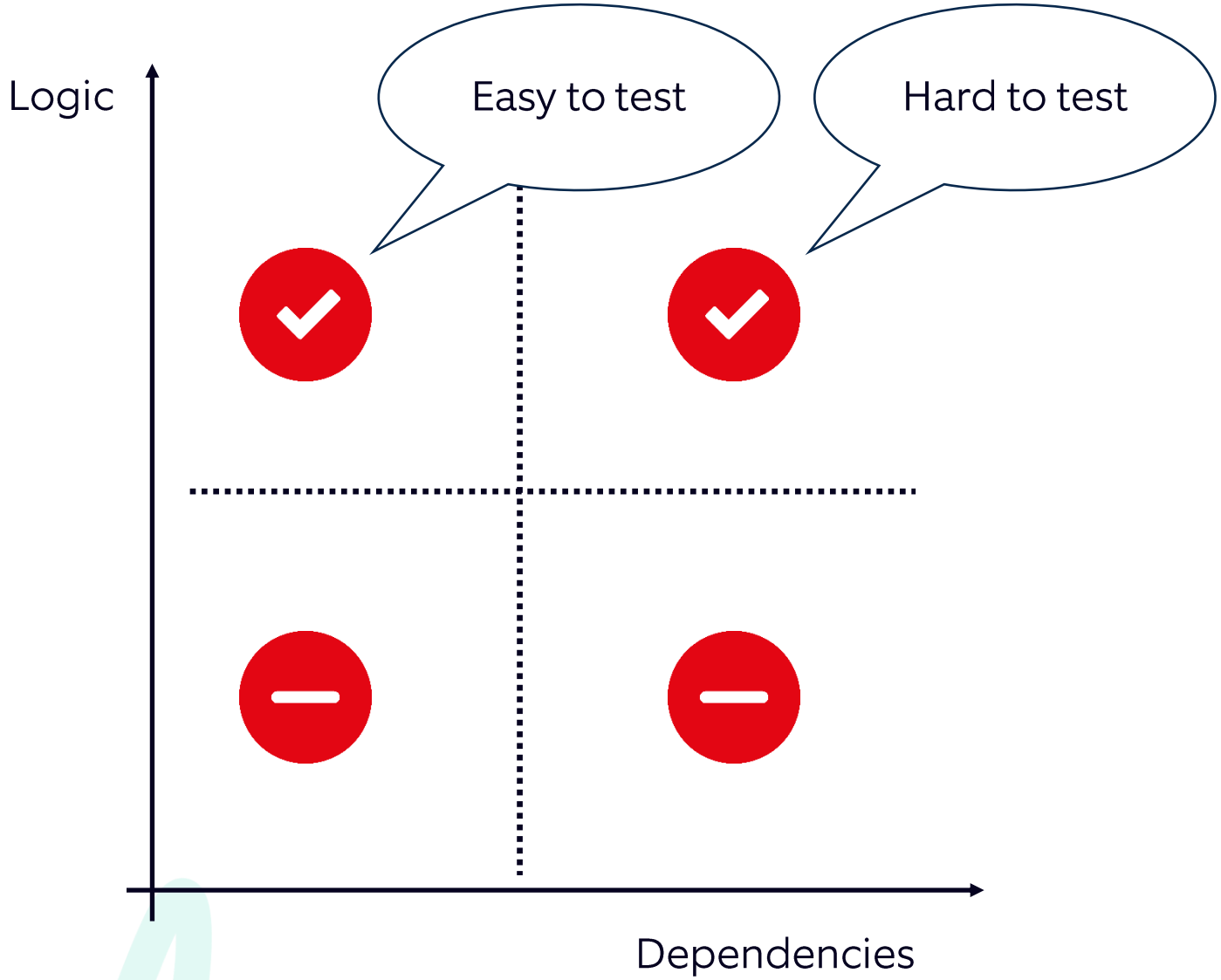
nagarro

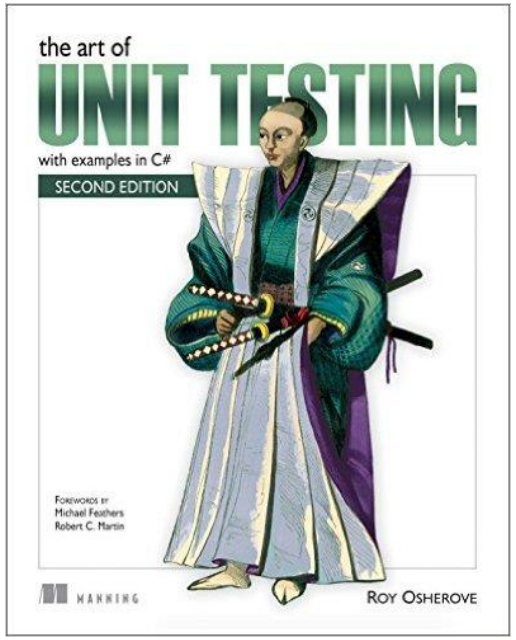
What to test?

Exercise – Which code should (not) be tested with Unit Tests?

- Think about which parts of your code you would not unit test. Can you explain why it would not make sense to test this specific kind of code?
- How would you prioritize what to test and what to dismiss?
- Gather in groups
- 3 minutes time for brainstorming
- Present and explain your insights to the audience

Priority Factors





What Is Not Tested with Unit Tests?

- User-Interface
  - Mocking and recording of GUI elements
  - Tests are redundant with UI code
- Valueobject classes
  - Beans that only have get/set methods and implement no further logic
- Complex Frameworks
  - eg. ASP .Net Core, Entity Framework
- Delegates and Facades
- Generated Code

1. No unit tests for trivial classes without logic
2. Focus on unit test, avoid other types of tests

Examples for Unit Tests

- Low level functions
  - APIs with defined in- and output
- Small isolated calculation modules
  - Mathematical functions / algorithms
- Internally closed classes
  - Parsers, generators, validators, formatters
  - Decision trees/tables
- Common
  - Units with low dependencies, no side-effects and without complex initialization

$$K_t = K_0 \cdot \left\{1 + \frac{p}{100}\right\}^t$$

	Entscheidungstabelle	Regeln							
		R1	R2	R3	R4	R5	R6	R7	R8
Bedingungen	Antrag vollständig	j	j	j	j	n	n	n	n
	telefonische Klärung ausreichend	j	j	n	n	j	j	n	n
	Antrag rechnerisch richtig	j	n	j	n	j	n	j	n
Aktionen	telefonische Klärung	-	-	-	-	x	x	-	-
	Abrechnung ergänzen	-	-	-	-	x	x	-	-
	Rückfrage schreiben und absenden	-	-	-	-	-	-	x	x
	Antragsformular korrigieren	-	x	-	x	-	x	-	-
	Antragsformular unterschreiben und weiterleiten	x	x	x	x	x	x	-	-

```xml
<?xml version="1.0"?>
<quiz>
 <frage>
  Wer war der fünfte
  deutsche Bundespräsident?
 </frage>
 <antwort>
  Karl Carstens
 </antwort>
 <!-- Anmerkung: Wir
  brauchen mehr Fragen -->
</quiz>
```

XML

Choose small and simple units and not the most complex!

nagarro

Hands-On: Unit Testing

The Guilded Rose

The Gilded Rose – The System

Hi and welcome to team Gilded Rose.

As you know, we are a small inn with a prime location in a prominent city ran by a friendly innkeeper named Allison. We also buy and sell only the finest goods. Unfortunately, our goods are constantly degrading in quality as they approach their sell by date. We have a system in place that updates our inventory for us. It was developed by a no-nonsense type named Leeroy, who has moved on to new adventures.

First an introduction to our system:

- All items have a SellIn value which denotes the number of days we have to sell the item
- All items have a Quality value which denotes how valuable the item is
- At the end of each day our system lowers both values for every item

Pretty simple, right? Well this is where it gets interesting:

- Once the sell by date has passed, Quality degrades twice as fast
- The Quality of an item is never negative
- "Aged Brie" actually increases in Quality the older it gets
- The Quality of an item is never more than 50
- "Sulfuras", being a legendary item, never has to be sold or decreases in Quality
- "Backstage passes", like aged brie, increases in Quality as it's SellIn value approaches: Quality increases by 2 when there are 10 days or less and by 3 when there are 5 days or less but Quality drops to 0 after the concert

Just for clarification, an item can never have its Quality increase above 50, however "Sulfuras" is a legendary item and as such its Quality is 80 and it never alters.

The Gilded Rose – Your Task

- Implement as many Unit Tests as necessary for the class GildedRose
- Clone the repository “01-CI_and_Unit_Testing”
- Implement your tests in the provided template GildedRoseTest
- Measure the achieved coverage with Ecl-Emma for Eclipse or another the code coverage tool

- Commit your changes and execute the tests in your Jenkins job.
- Add a JUnit report to your Jenkins job.

- Constraint: ***Do not alter the code of the classes*** *GildedRose or Item!*

nagarro

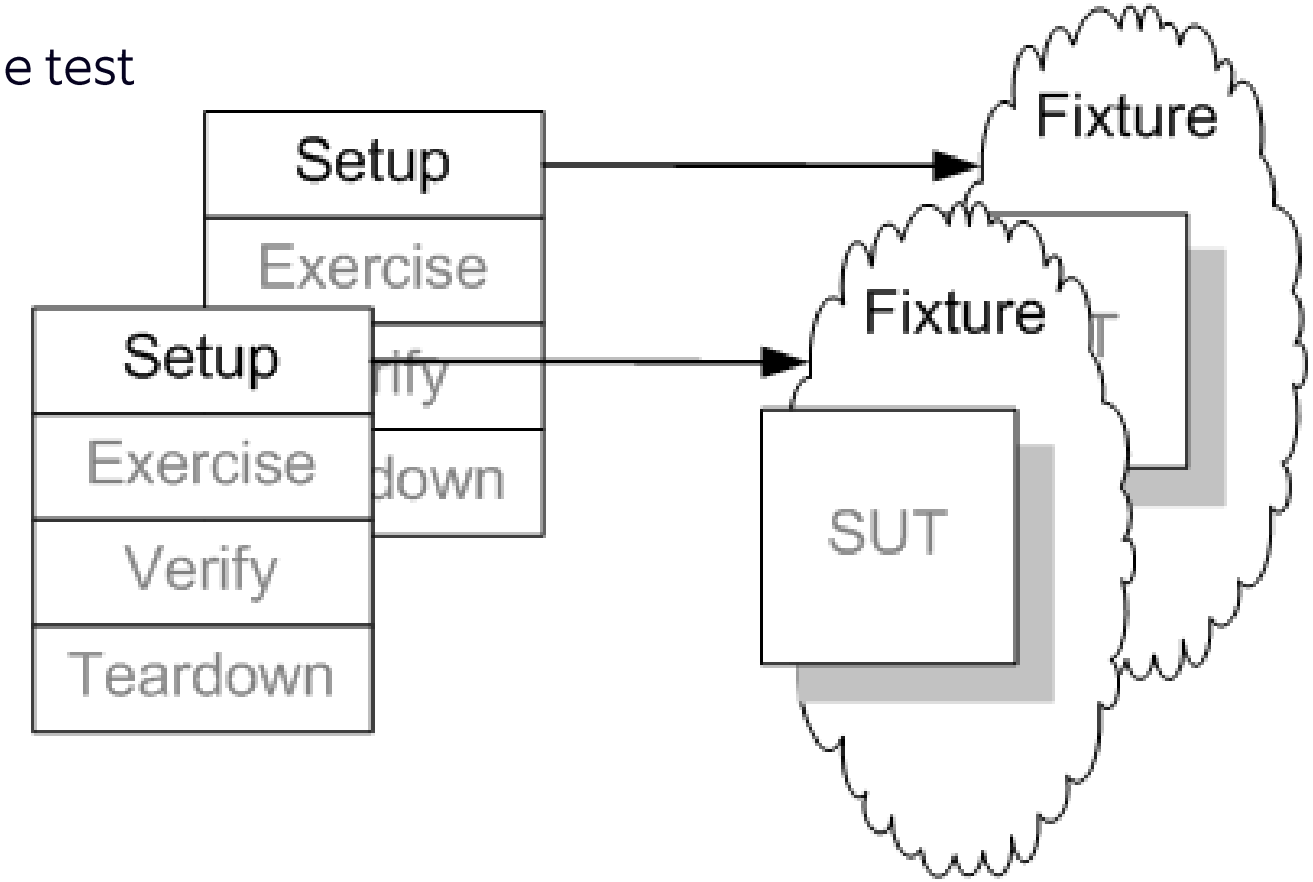
How to structure my Unit Tests?

Test Fixture

- Everything needed to exercise SUT
- State of the test environment before the test

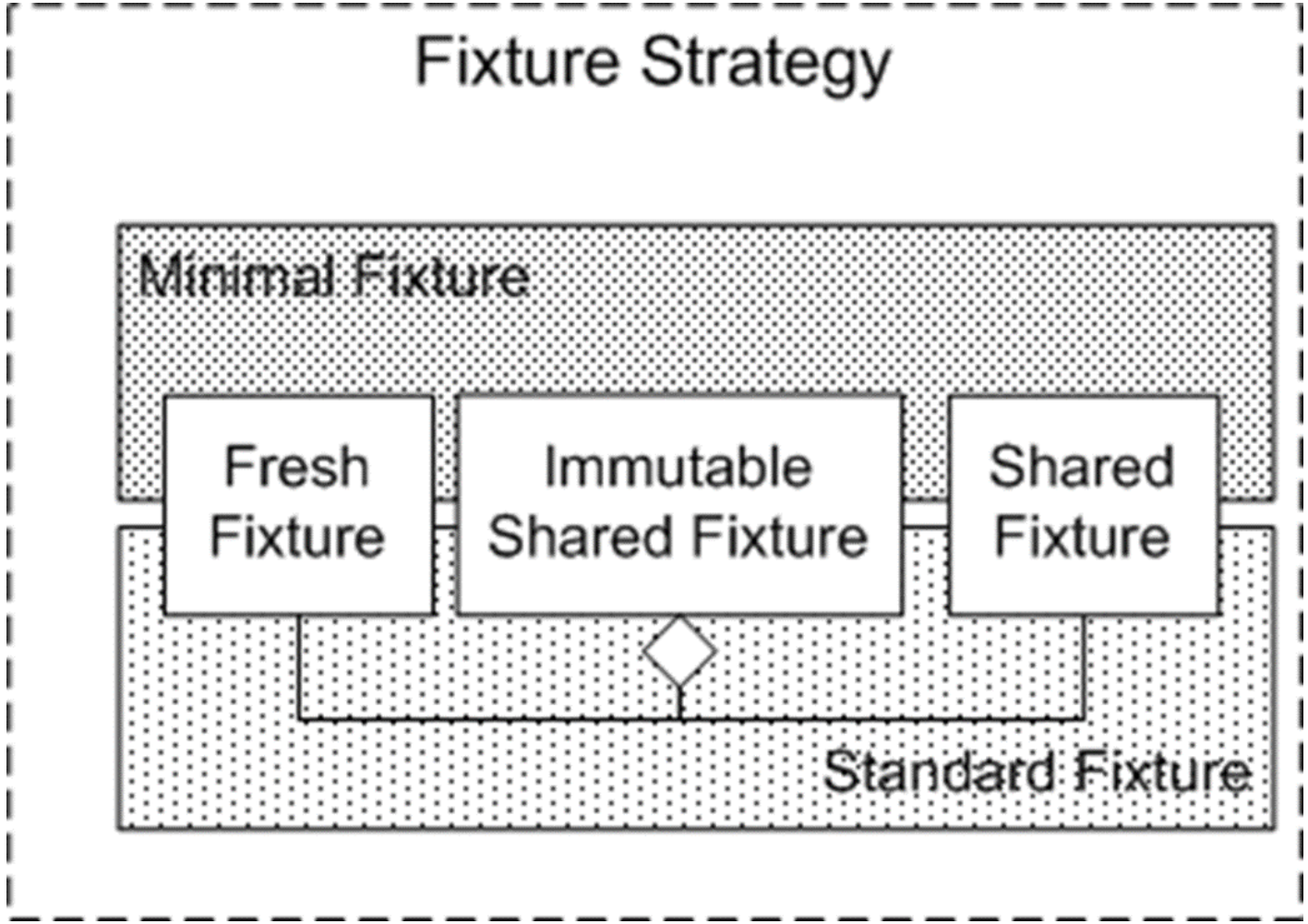
Examples

- XML-Marshaller: immutable shared
- Enumerations: immutable shared
- In-Memory-DB: shared (rollback and teardown)
- Spring Context: shared

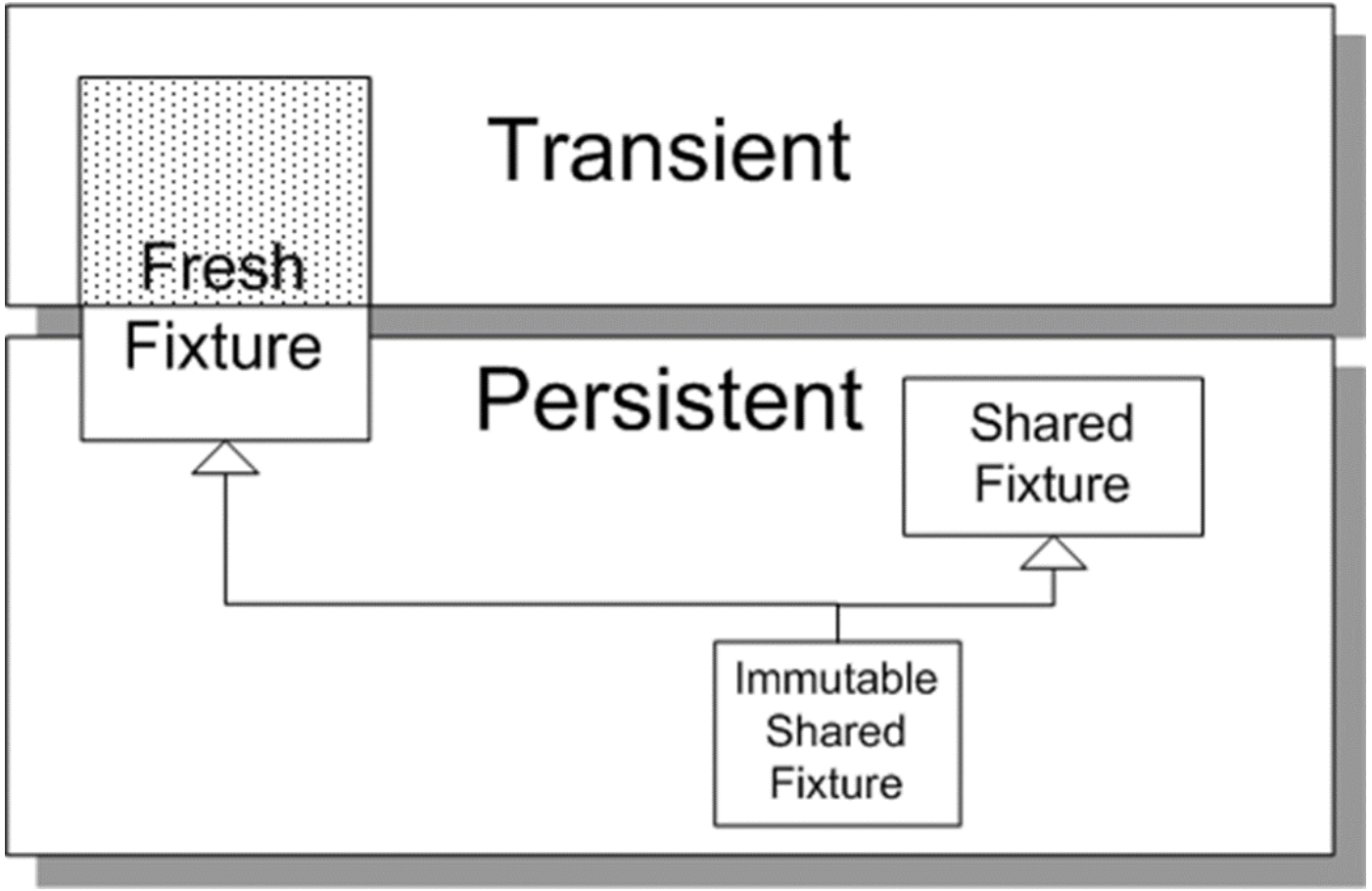


http://xunitpatterns.com/Fresh%20Fixture.html

Text Fixture Strategy



Mechanics of implementing fixture strategies



Organizing Tests

- Testcase
  - Map each testcase to a method
- Organize collection of testcases
  - per class
  - per feature / requirements
  - per fixture / start state

In software engineering, a test case is a specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Wikipedia

Test Method Naming Strategies

1. MethodName_stateUnderTest_expectedBehaviour
   - `Schedule_unscheduledFlight_flightIsInScheduledState`
2. MethodName_expectedBehaviour_stateUnderTest
   - `Schedule_flightIsInScheduledState_unscheduledFlight`
3. FeatureToBeTested
   - `SchedulingAnUnsheduledFlight`
4. Should_expectedBehaviour_when_stateUnderTest
   - `Should_beInScheduledState_when_Schedule`
5. When_stateUnderTest_then_expectedBehaviour
   - `When_unscheduledFlight_then_FlightIsInScheduledState`
6. Given_stateUnderTest_when_action_then_expectedBehaviour
   - `Given_unscheduledFlight_when_Schedule_then_FlightIsInScheduledState`

Don‘t start the method names with test

given / state fixture | when / action feature | then / expected

Testcase Class per Class

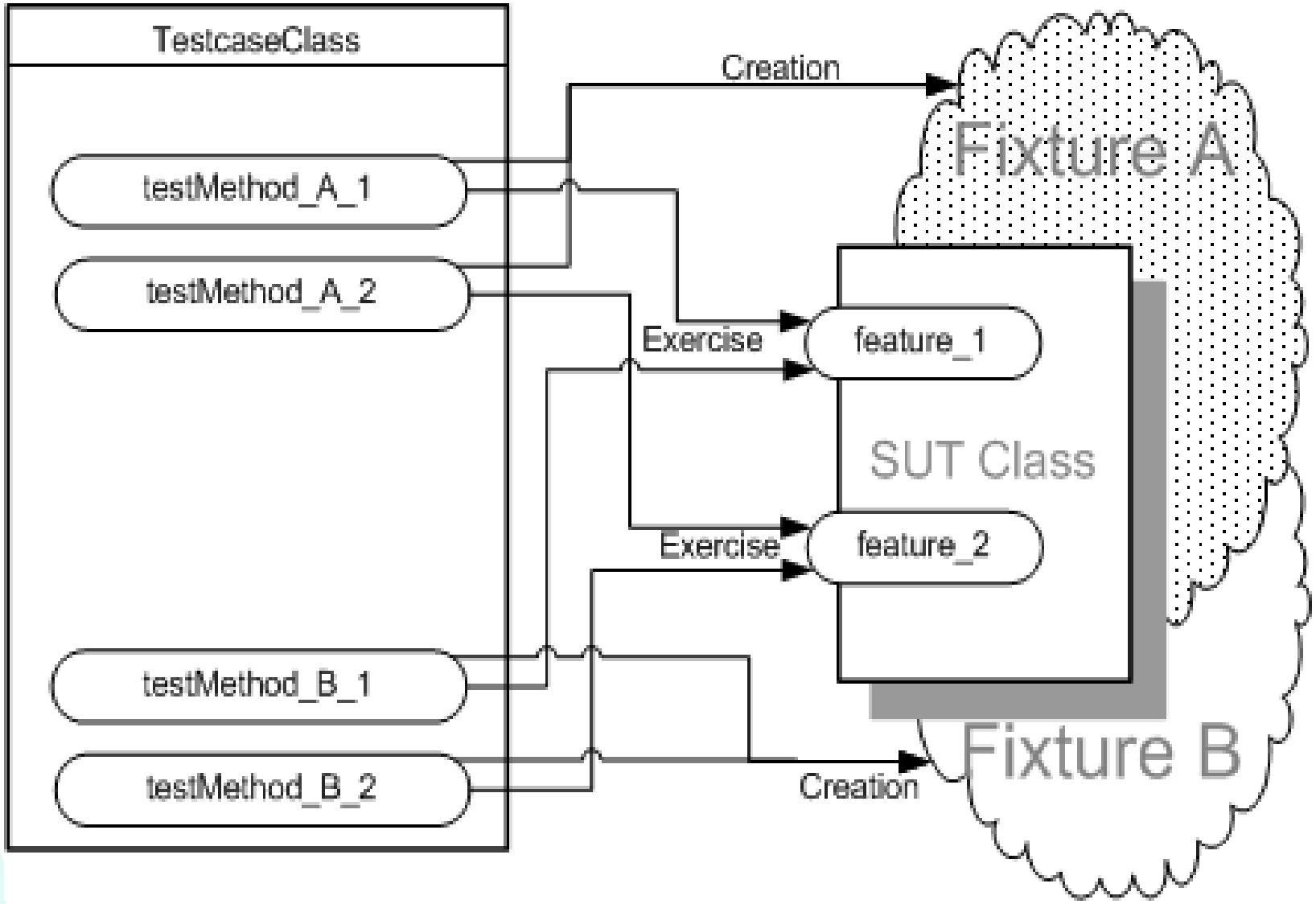
```
public class FlightStateTest {
    public void Schedule_fromUnscheduledState () {
        ...
    }

    public void Schedule_fromScheduledState () {
        ...
    }
}
```

given / state fixture

when / action feature

then / expected

Test Case Class per Class

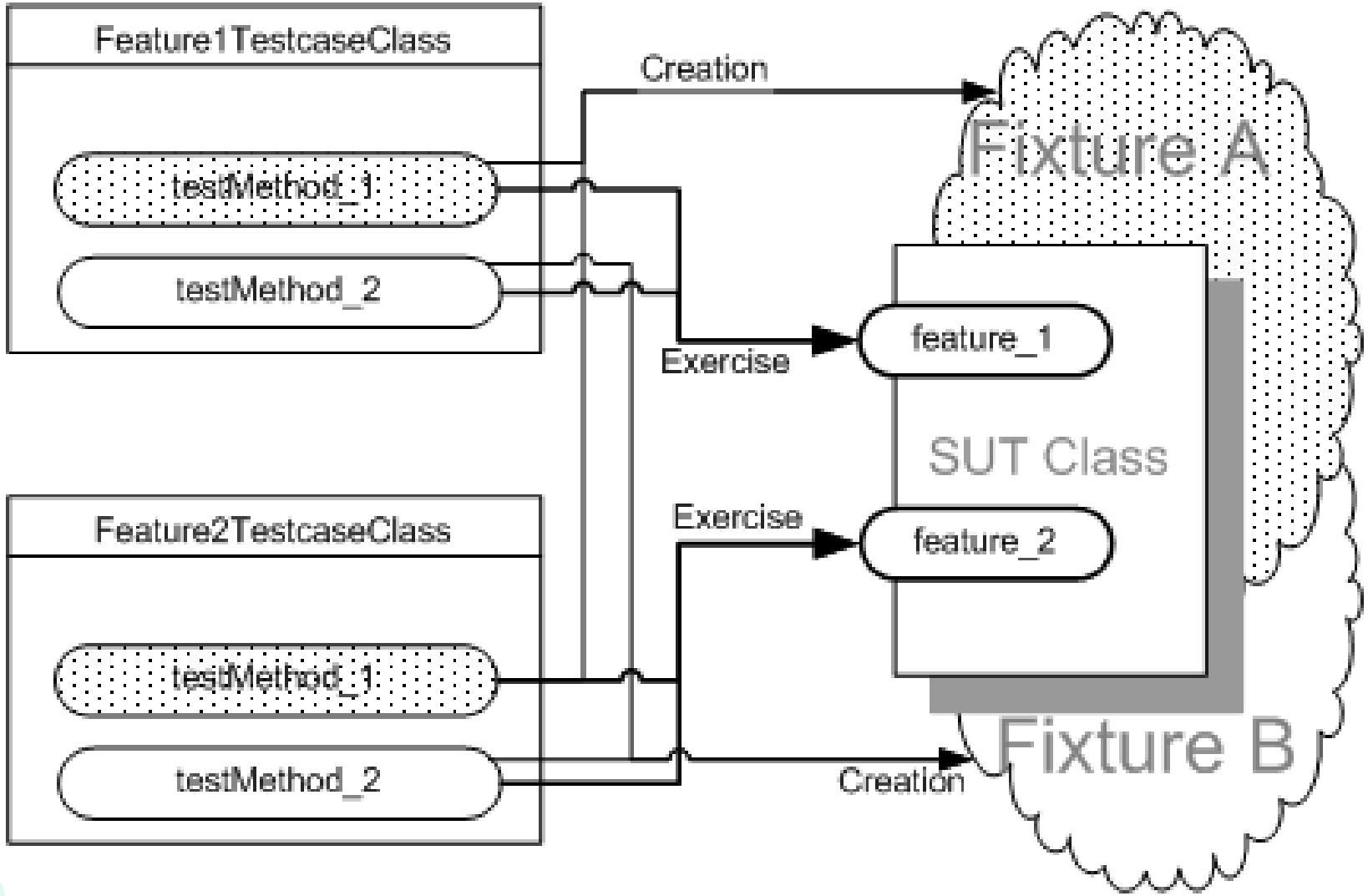


Testcase Class per Feature

```
public class ScheduleFlightTest {
    public void ScheduledState_shouldThrowInvalidRequestEx() {
        ...
    }

    public void UnscheduledState_shouldEndUpInScheduled() {
        ...
    }
}
```

given / state fixture | when / action feature | then / expected

Test Case Class per Feature



Testcase Class per Fixture

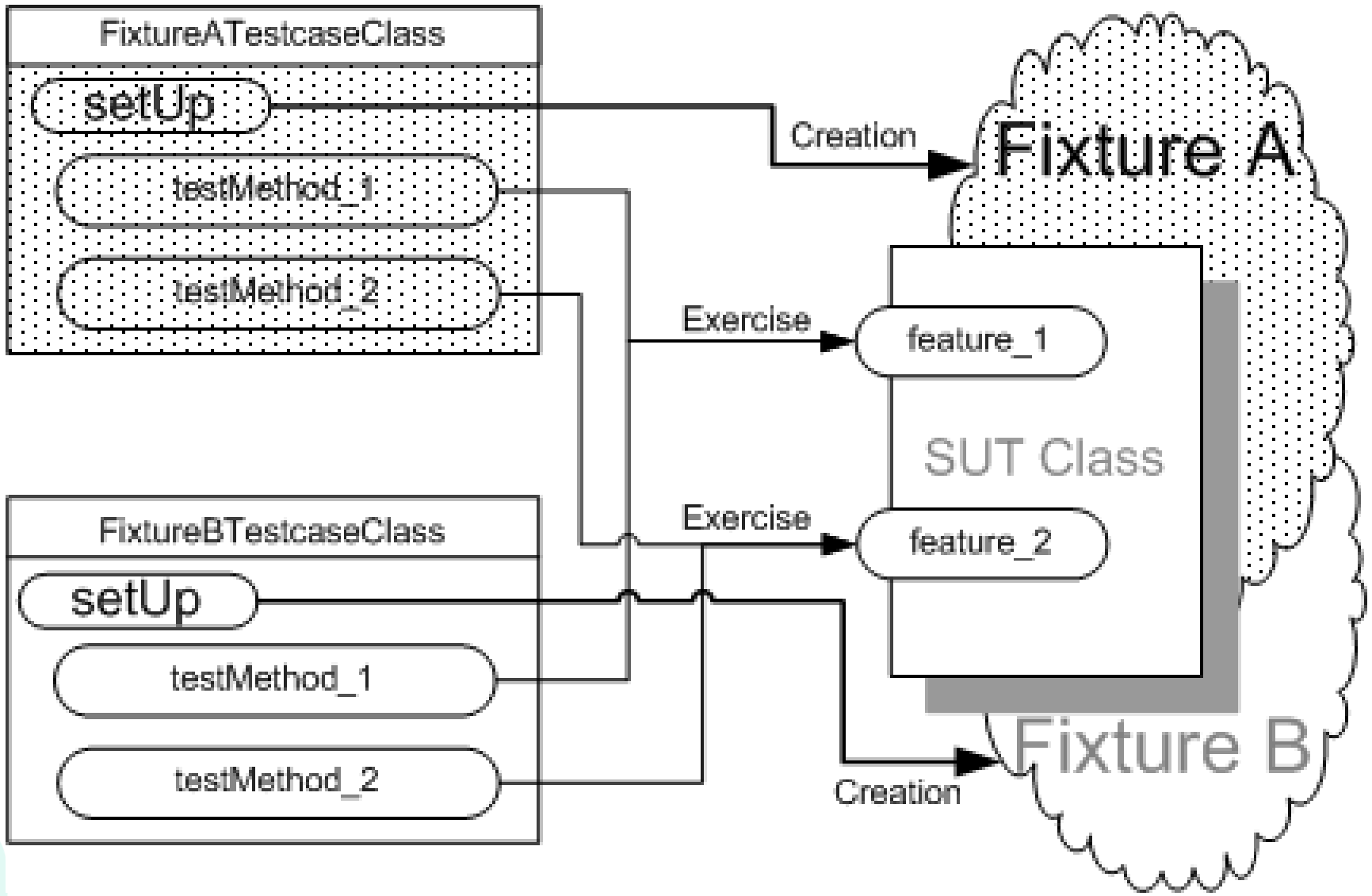
```
public class ScheduledFlightTest {
    public void Schedule_shouldThrowException () {
        ...
    }
    public void Schedule_shouldEndInUnscheduledState () {
        ...
    }
}

public class UnscheduledFlightTest {
    public void Schedule_shouldEndInScheduledState () {
        ...
    }
    public void Deschedule_shouldThrowException () {
        ...
    }
}
```

given / state fixture

when / action feature

then / expected

Test Case Class per Fixture



nagarro

Test Coverage

Dealing with Coverage Metrics

```
public static bool IsFullAgedAustria(int ageInYears)
{
    if (ageInYears >= 18)
        return true;

    return false;
}
```

```
[Fact]
public void TestIsFullAge()
{
    LegalUtils.IsFullAgedAustrian(17).Should().BeFalse();
}
```

What does 66,7% really tell you?

$$\text{Code Coverage (Test Coverage)} = \frac{\text{Lines Covered}}{\text{Overall Lines of Code}} = \frac{2}{3} = 66{,}7\%$$

$$\text{Branch Coverage} = \frac{\text{Branches Traversed}}{\text{Overall Branches}} = \frac{1}{2} = 50\%$$

Dealing with Coverage Metrics

```
public static bool IsFullAgedAustria(int ageInYears)
{
    return ageInYears >= 18;
}
```

```
[Fact]
public void TestIsFullAge()
{
    LegalUtils.IsFullAgedAustrian(17).Should().BeFalse();
}
```

Does our test really cover more now?

$$\text{Code Coverage (Test Coverage)} = \frac{\text{Lines Covered}}{\text{Overall Lines of Code}} = \frac{1}{1} = 100\%$$

$$\text{Branch Coverage} = \frac{\text{Branches Traversed}}{\text{Overall Branches}} = \frac{1}{2} = 50\%$$

Dealing with Coverage Metrics

```
public static bool IsFullAgedAustria(int ageInYears)
{
    return ageInYears >= 18;
}
```

```
[Fact]
public void TestIsFullAge()
{
    bool isFullAged = LegalUtils.IsFullAgedAustrian(17);
    isFullAged = LegalUtils.IsFullAgedAustrian(18);
}
```

$$\text{Code Coverage (Test Coverage)} = \frac{\text{Lines Covered}}{\text{Overall Lines of Code}} = \frac{1}{1} = 100\%$$

$$\text{Branch Coverage} = \frac{\text{Branches Traversed}}{\text{Overall Branches}} = \frac{2}{2} = 100\%$$

100% coverage …
but the test is
assertion free!!!

Coverage - What & How Much to Test?

As much as it makes sense!

- Trade-off between effort vs. value
- Think about potential exception cases

- Consider different metrics
  - Branch, line coverage, etc.
  - Use tools to identify uncovered code

- Interprete metrics correctly
  - Low coverage numbers = you’re not testing enough
  - High coverage numbers != good test quality

The Right-BICEP

Right: Are the results **Right**?

- Make sure you validate the expected results and that you have at least understood the happy path

B: Are all the **Boundary** conditions **CORRECT**?

- Consider the edges of the input domain (out of bounds values, numeric overflows, badly formatted data, duplicates in lists, incorrectly ordered sequences, etc.)

I: Checking **Inverse** Relationships

- Check behaviour with its logical inverse (e.g. multiply number by itself to verify square-root implementation)

C: **Cross-Checking** using other means

- Verify that a different library produces the same result (e.g. verify custom square-root implementation with built in Java implementation) or that the overall result contained within your SUT still fits (e.g. number of checkout out books plus the number of books in the shelves has to equal the overall number of books in the collection)

E: Forcing **Error** Conditions

- Also consider the error path (e.g. network outage, out-of-memory, etc.)

P: **Performance** Characteristics

- Measure performance of potentially slow code before optimizing it and run such tests separate from fast unit tests

Pragmatic Unit Testing in Java 8 with JUnit, Jeff Langr et. al.

Remember Boundary Conditions with CORRECT

C: Conformance of the value (to an expected format)

O: Appropriate **Ordering** (i.e. ordered or unordered) of the set of values

R: The value is **Reasonable** within minimum and maximum values

R: The code does NOT **Reference** anything external that is not under it‘s own control

E: The value **Exists** (is non-zero, non-null, present in a collection, etc.)

C: The **Cardinality** of the values is guaranteed (i.e. there are exactly enough values)

T: Everything is happening in order – at the right **Time** and in **Time** (absolute and relative timing)

Pragmatic Unit Testing in Java 8 with JUnit, Jeff Langr et. al.

Exercise – How many Unit Tests would you implement for this method?

Bungee jumping online weight check

- Input to the software can be any positive integer
- Has to be **at least 40 kg and at max 140 kg**

- How many test cases and different input values do we need?
- Can you explain the chosen value?
- Discuss the results with all participants

```
public static bool CheckBungeeJumperWeight(int weight)
{
    if (weight < 40)
    {
        return false;
    }
    if (weight > 140)
    {
        return false;
    }
    return true;
}
```

Equivalence Class Partitioning - Example

- Bungee jumping online weight check
  - Input to the software can be any positive integer
  - Has to be **at least 40 kg and at max 140 kg**
- *Valid inputs*: 40 – 140, *Invalid inputs*: <= 39 or >= 141
- **Valid class:** 40 – 140 e.g. 45 kg
- **Invalid class 1:** 39 or any value below e.g. 30 kg
- **Invalid class 2:** 141 or any value above e.g. 167 kg
- Only **one valid** and **two invalid conditions** are tested via representatives
- Edge cases are not considered

Invalid	Valid	Invalid
30 kg	45 kg	167 kg
Paritition 1	Paritition 2	Paritition 3

Equivalence Class Partitioning

- Blackbox testing technique to select less and effective test cases for input values spanning a range
- **Equivalence** Class **Partitioning (ECP)** helps to divide input data into different data classes considered equivalent
- Partitions are derived from requirements into different **valid** and **invalid partitions**
- The system is assumed to behave the same for all values within a certain partition
  - If one condition works in a partition, others will work too
  - If one condition does not work in a partition, none of the conditions in the same partition will work
- Significant reduction of representative partitions considering several input parameters

Boundary Value Analysis - Example

- **Example**: Bungee jumping online weight check
  - Input to the software can be any positive integer
  - Has to be **at least 40 kg and at max 140 kg**

- **Solution:** using equivalence class partinioning and boundary value analysis
  - Edge cases are selected and considered
  - Edge case values also serve as representatives for different equivalence classes

Invalid	Valid		Invalid
39 kg	40 kg	140 kg	141 kg
Paritition 1	Paritition 2		Paritition 3

Boundary Value Analysis

- Blackbox testing techiques to effectively select representatives of boundary values in a range

- Boundary value testing tests at the „**extreme ends**“ between specific paritions of input values
  - Equivalence Class Partitioning can be used beforehand to select corresponding partitions

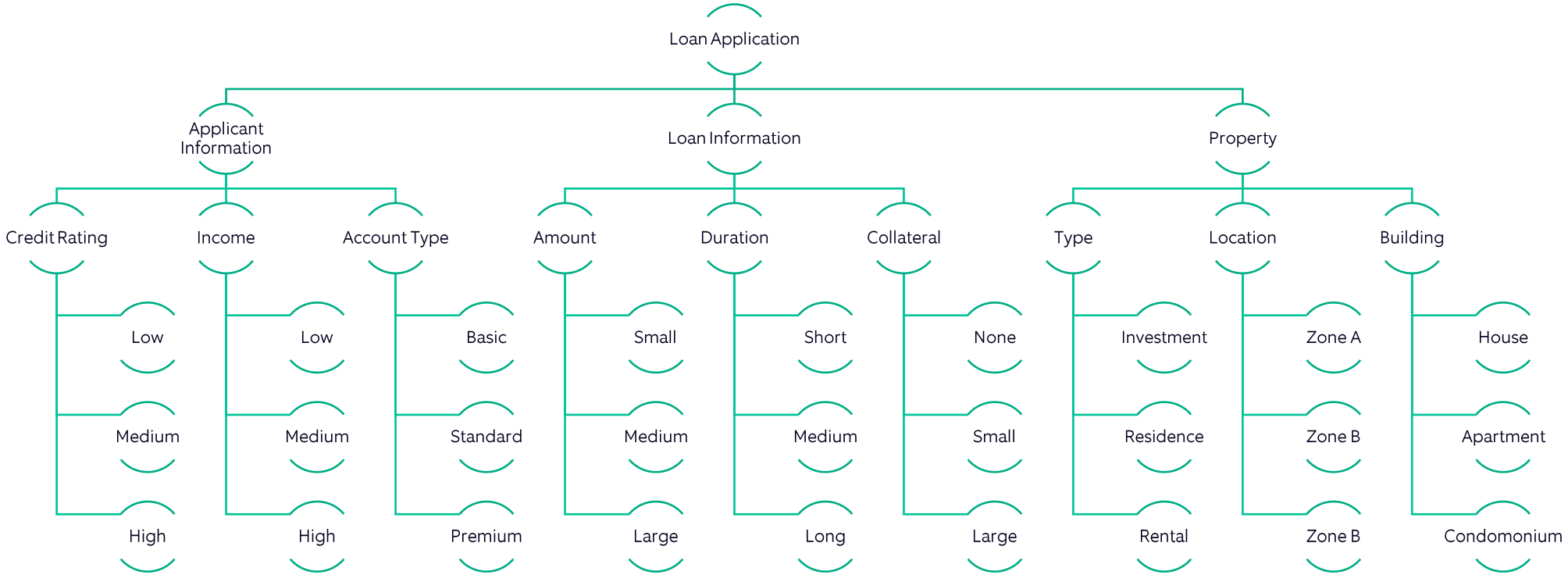
- Only test at the edges between the equivalence class partitions (aka two-value boundary value testing)

- Edge cases are more likely to reveal programming errors!

Pair-wise Testing

- Rather than combining all possible permutations of input parameters and the possible values
  - Find the minimum set of test cases that covers all possible pairs combinations
- Assumption: pairs will already most likely cover the majority of possible defects
  - Additional permutations increase the number of test cases extensively
  - More permutations not very likely to catch more defects that the pairs already would
  - Increases maintenance cost by adding little value
- Possible pairs combinations can be determined using tool support
- Requires knowledge of the input domain and thus knowledge of all the possible parameter values that are possible
- Using other techniques such as boundary analysis and equivalence partitioning to reduce the set of possible input parameters into categories can further reduce the possible pairs combinations

Pair-wise Testing - Example



9 input parameters, 3 different possible values each → Yields in 3 ^ 9 ~ 20K possible permutations

Pair-wise Testing - Example

Test Nr.	Credit Rating	Income	Account Type	Loan Amount	Loan Duration	Loan Collateral	Property Type	Property Location	Property Type
1	Medium	Medium	Basic	Small	Long	None	Investment	Zone C	House
2	High	Low	Standard	Small	Short	Small	Residence	Zone B	Apartment
3	Medium	High	Standard	Large	Medium	Large	Rental	Zone A	Condomonium
4	Low	Low	Premium	Medium	Long	None	Rental	Zone B	Condomonium
5	High	Low	Basic	Medium	Medium	Large	Residence	Zone C	House
6	Low	High	Premium	Large	Short	None	Investment	Zone A	Apartment
7	Low	Medium	Premium	Large	Medium	Small	Residence	Zone B	House
8	Low	High	Standard	Small	Long	Large	Investment	Zone B	House
9	Medium	Medium	Basic	Medium	Short	Small	Rental	Zone C	Apartment
10	High	Medium	Standard	Medium	Short	None	Residence	Zone A	House
11	High	Low	Standard	Large	Long	Small	Investment	Zone C	Condomonium
12	Medium	Low	Premium	Small	Medium	Large	Investment	Zone A	Apartment
13	High	High	Premium	Small	Short	Large	Residence	Zone C	Condomonium
14	Low	High	Basic	Medium	Short	Small	Investment	Zone C	Condomonium
15	High	Medium	Basic	Small	Long	Small	Rental	Zone A	House
16	Medium	Medium	Basic	Large	Medium	None	Residence	Zone B	Condomonium
17	High	Medium	Standard	Small	Long	Large	Residence	Zone A	Apartment

Pairwise testing reduces the number of test cases to **17** instead of ~ **20K** to cover the majority of possible defects!

Table created with PICT, https://github.com/Microsoft/pict

Exercise – Let‘s identify the Testing-Pairs

- Our application creates sports recommendations

- Based on the following parameters there are 3 x 2 x 3 = 18 possible test-cases

Parameters	
Goal:	fun, strength, condition
Exhausting:	yes, no
Time invested:	low, medium, high

- Reduce the list by applying the pairwise testing method

Exercise – Let‘s identify the Testing-Pairs

All Combinations			
	Goal	Exhausting	Time invested
TC1	condition	yes	low
TC2	condition	no	low
TC3	condition	yes	medium
TC4	condition	no	medium
TC5	condition	yes	high
TC6	condition	no	high
TC7	fun	yes	low
TC8	fun	no	low
TC9	fun	yes	medium
TC10	fun	no	medium
TC11	fun	yes	high
TC12	fun	no	high
TC13	strength	yes	low
TC14	strength	no	low
TC15	strength	yes	medium
TC16	strength	no	medium
TC17	strength	yes	high
TC18	strength	no	high

2-Pair (Pairwise)			
	Goal	Exhausting	Time invested
TC1			
TC2			
TC3			
TC4			
TC5			
TC6			
TC7			
TC8			
TC9			
TC10			
TC11			
TC12			
TC13			
TC14			
TC15			
TC16			
TC17			
TC18			

nagarro

Testing Patterns- Best Practices and Smells

General Test Quality Problems

- **Test quality**
  - Wrong tests
  - Not understandable and complicated
  - Bad structured and not uniform

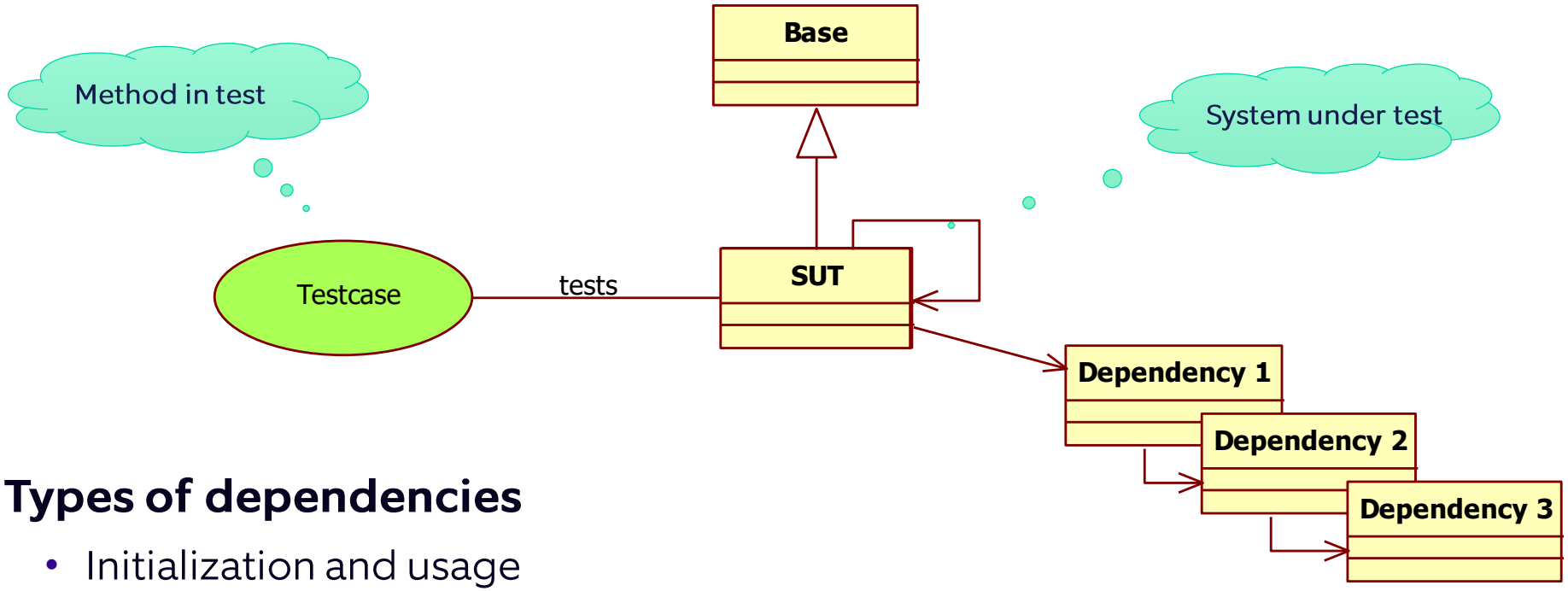
- **Execution is cumbersome**
  - Slow
  - Not repeatable
  - Not suitable for working environment

All this leads to not enough testing and demotivation!

Test Smells and Hints to Solve

- **Slow tests**
  - Often due to usage of Database or other services. → Use test double or fake database.
- **Hard-to-Test code**
  - Dependencies, missing constructors, asynchronous code → Use test doubles, humble objects and dependency breaking techniques.
- **Fragile tests**
  - Tests fail due to changes in the SUT that should not affect the tests. → check assumptions, break dependencies, „use the front door first“-principle (focus on public interfaces)
- **Test code duplications**
  - Same code is used several times in the tests. → refactor test code (e.g. „Exctract Method“)
- **Obscure tests**
  - Test purpose is hard to understand → refactor to single condition tests, improve naming

The Impact of Dependencies



- **Types of dependencies**
  - Initialization and usage of instance variables
  - Calling methods of base classes
  - Calling methods of same class
  - Instantiation of local object with new
  - Navigation via the object modell
  - Accessing static methods/members

Follow the F.I.R.S.T Principles

Fast

Independent (and Isolated)

Repeatable

Self-Validating

Timely (and Thorough)

Exercise - F.I.R.S.T Principles

- Build at least two groups
- Reflect on the meaning of the concepts behind the acronym and explain them to each other.
- Collect examples from your own work experience where one or more of the principles have been violated
- Present your insights

Follow the F.I.R.S.T Principles

- Fast
  - A test case shall be executed within a fraction of a second (milliseconds)
  - If tests run slow they usually won't be executed frequently
- Independent and Isolated
  - Each test case shall be able to run isolated from other tests
  - There shall be no assumption on the order in which tests are executed
- Repeatable
  - Each test shall be deterministic – i.e. it shall always produce the same results independent from the environment
  - Don't utilize non-deterministic components in your tests (e.g. clock time, random values, network connection etc.)
- Self-Validating
  - No manual inspection shall be required to determine if the test failed or passed
- Timely and Thorough
  - Write tests in parallel or even better before the production code is implemented (Test-driven development)
  - Tests shall be thorough and concentrate on the most important behaviour

Testing Patterns and Strategies

Different testing strategies for Unit Tests:

White Box

- A good starting point is the smallest or simplest „Unit“
- Different execution paths in a public method
- Requires knowledge of the inner workings of the SUT

Black Box

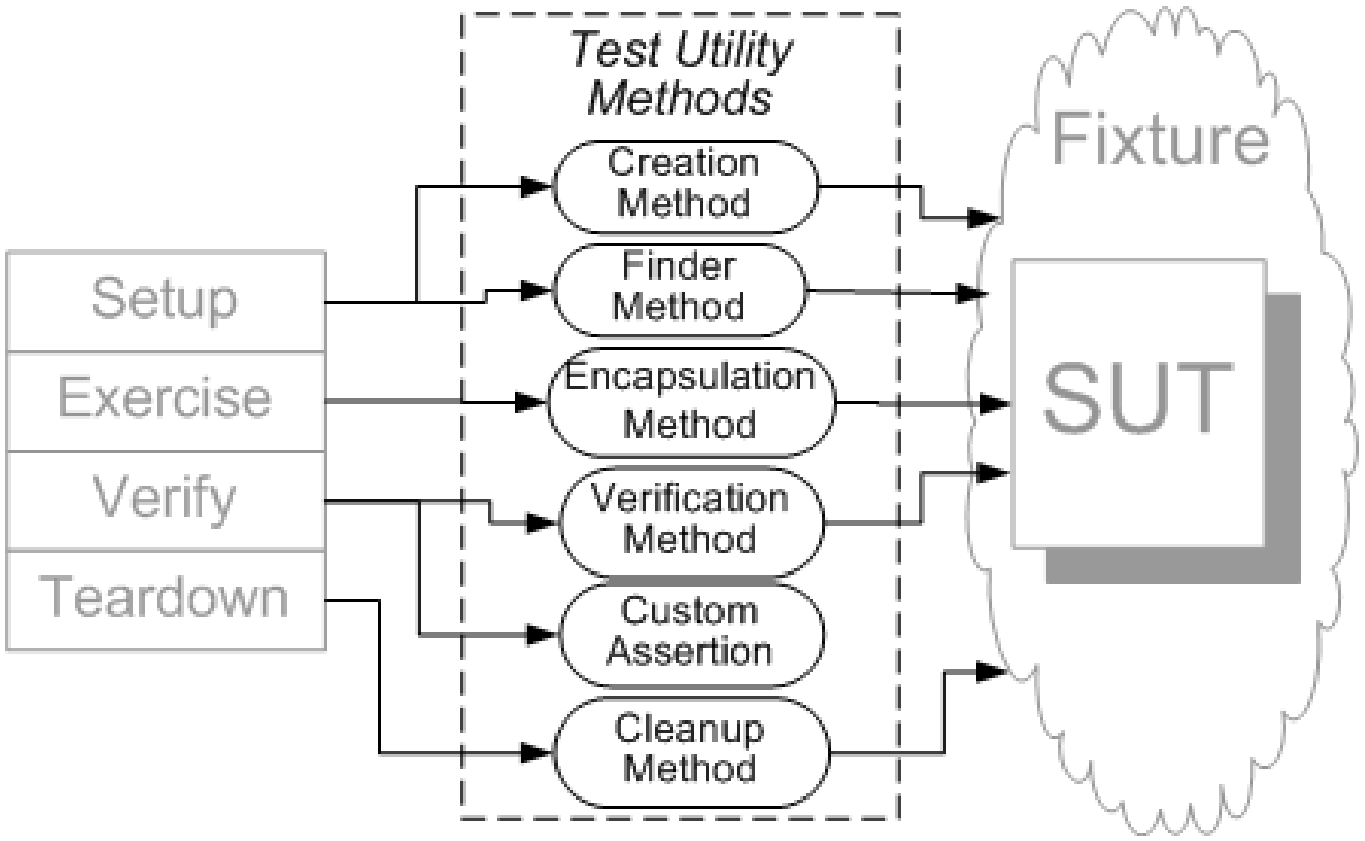
- Look for business specifications and boundaries
- Match requirements to your tests

Well-established testing patterns:

- Behavior Verification
- Configurable Test Double
- Dependency Injection
- Fake Object
- Humble Object
- Layer Test (Headless Application)
- Single Outcome Assertion
- Test Utility Method

Test Utility Method

- Reusable test logic
- Creation Method
  - Parameterized Method
  - Anonymous Method
  - Named State Reaching Method
- Finder Method
- Attachment Method



Guidelines for Good Tests (1)

- Consistent naming and structure
- Test the RIGHT things
- Avoid conditional logic in tests
- Avoid multiple assertions
- Test coverage
  - High test coverage
  - Using tools locally and in the build
- Test public methods only

Guidelines for Good Tests (2)

- Unit Test Isolation
  - *"In isolation" means separate and apart from the application, and all other units.*
  - No dependencies between tests
  - Dependent units are mocked
- Re-usability of the test code
- Easy to execute
  - Fast and repeatable
  - No dependencies on resources or external systems

Guidelines for Good Tests (3)

Implement tests that you can always rely on

Create maintainable and changeable test

Readability

› Readable ⇔ Maintainable

Refactoring Towards High Quality Tests

- Use consistent naming conventions and structure
  - Different source folder
  - Same package
  - Test method naming
  - Test class naming
- When a test fails you should understand what it is for
  - Make sure assertions are intention revealing
  - Use fluent assertion libraries like AssertJ
  - Avoid multiple asserts (unless they will fail the test for the same reason e.g. assert on the same object)
  - Implement custom assertion methods for logically related verifications

Refactoring Towards High Quality Tests

- Use “Extract Method” to make code more readable and to address duplications
- Use Builder and Factory patterns for instantiation and setup of complex objects which happens in several places
- Minimize fixture size by extracting common code
- Don’t use code in setup methods that is not common to all test cases!
  - Rather move it to helper methods or classes
- Stub out unstable and slow dependencies
  - Don’t access “real” databases, system time, networking, etc.

Verify the Tests

```
public class Discount {

    private final MarketingCampaign marketingCampaign;

    public Discount() {
        this.marketingCampaign = new MarketingCampaign();
    }

    public Money discountFor(Money netPrice) {
        if (marketingCampaign.isCrazySalesDay()) {
            return netPrice.reduceBy(15);
        }
        if (netPrice.moreThan(Money.ONE_THOUSAND)) {
            return netPrice.reduceBy(10);
        }
        if (netPrice.moreThan(Money.ONE_HOUNDRED) && marketingCampaign.isActive()) {
            return netPrice.reduceBy(5);
        }
        return netPrice;
    }
}
```

Example test run with eclEmma coverage

- Always execute existing tests before changing code or adding functionality!
- Use TDD to verify that the test is useful (Red, Green, Refactor)
- Break the production code to see if tests fail
  - Do it manually (e.g. replace conditional AND with OR)
  - Or use mutation testing tools (such as pitest)
- Do pairing when writing tests and test code reviews
- Write unit tests for test utilitiy classes
- Use tools to figure out which part of the production code is not exercised in the tests yet

Conclusion - What Makes a Valuable Test?

- Tests the **right things**
- **Focuses** primarily on business **logic**
- Has **low chance of producing false positives** (i.e. false alarms)
  - Due to high coupling on implementation details such (e.g. asserting on execution of a specific SQL statement – changing the query notation could happen frequently while still producing the same correct result)
- Has **high chance of catching regression** errors
  - The more core is getting executed, the higher the chance to catch a regression due to changes
- Provides **fast feedback**
- Has **low maintenance cost**
- Note: There is always a trade-off between Fast feedback, high chance of catching regression errors and low chance of catching false positives
  - Impossible to provide all three characteristics at the same time

nagarro

03e

Improving Test and Build Speed

Improving the Quality and Speed of Builds

- Running **thousands of unit tests within seconds** should be possible
- Choose different strategies if too slow
  - Multiple Threads
  - @Categories annotations for splitting tests into different groups that can ran separately
  - Remove tests
  - Make sure your tests don't have any timing conditions (i.e. waiting for a fixed amount of time after exercising the SUT before verifying the result)

- **Remove dependencies** from tests that **slow down the test execution speed**
  - Use **stable dependencies** that are **under your control**
  - **Stub or mock dependencies** such as databases that are **unstable and not under your control**
    - use doubles or in-memory databases instead

Improving the Quality and Speed of Builds

- **Break the build fast**
  - Execute the fastest tests first to **get early feedback**
- Use **separate build pipeline for tests that are slow** due to their nature (such as integration and performance tests)
  - Run the build detached from the common CI/CD process (e.g. nightly rather than executing them with every commit)
- **Avoid brittle tests** that tend to break the build occasionally or produce false positives
  - Due to high coupling to production code details
  - Due to non-deterministic dependencies (such as system time, random values, networking, etc.)
- Consider using **faster hardware** for build machines