Test Driven Development Training

Agenda

- Which tests?
- Test doubles
- Test coverage
- The TDD principles
- Building Maintainable tests
- SOLID principles in the context of TDD
- Using TDD for Refactoring & code legacy
- Going further
- Annexes

Logistics

- Course hours
- Lunch and breaks
- Other questions ?

Which tests?

Agenda

- *Which tests?*
- Test doubles
- Test coverage
- The TDD principles
- Building Maintainable tests
- SOLID principles in the context of TDD
- Using TDD for Refactoring & code legacy
- Going further
- Annexes

Different type of tests

- There are different types of tests depending on the needs:
  - Test a unit (methods, classes, ...) isolated from its system: ***Unit Test***.
  - Test collaboration between units (partial or real interactions): ***Integration Test***.
  - Test an application as a user: ***Functional Test***.
  - Test performance of a stressed application: ***Performance Test***.

Unit Tests - Principles

- « Unit Test » = « Test » + « Unit ».
- What does « Unit » mean ?
- Each « Unit » interacts with « Actors »
  - Application « Actors »
    - Services,
    - Repositories, ...
  - External « Actors »
    - Database.
    - File System.
    - Web Service.
    - Other Application, ...

Unit Tests - Principles

- Typically, a « Unit » is a method or a class.
- **Goal**: Control inputs and check outputs
  - Absolute control over external environment.
  - Help to test boundaries.

Unit Tests - Isolation Principle

- Unit Tests must be reliable:
  - All Unit Tests must be run independently from each other.
  - No order or dependencies.
  - Consistent over time.
  - Test Fixture helps to control state of Unit Tests.
- Test execution must to be fast to provide immediate feedback.
- Use pattern **given-when-then** or **AAA** (Arrange / Act / Assert).
- Help low coupling between components.

Unit Tests - Process

- ***Setup***: Configure component to test and environment.
- ***Given***: Configure inputs.
- ***When***: Run.
- ***Then***: Check outputs.
- Feedback: OK / KO

Unit Tests - Benefits (1/2)

- Safer
  - Independent validation of each component.
  - Help find malfunctions faster.
- Faster
  - No need to have the entire environment: Testbed.
- Earlier
  - Ability to test without depending on an entire environment.

Unit Tests - Benefits (1/2)

- More concise
  - Avoid multiple test on same functionalities of a component.
- More accurate
  - Help to test boundaries: absolute control on external environment.
- More robust
  - Help to improve code design.
  - Help code refactoring.
  - Protected from regressions.

Unit Tests - Drawbacks

- More code to write and to maintain
  - Depends on the type of component to test.
  - Tests must be maintained according to the changes made into to main codebase.
- Need for good tests organisation
  - Avoid dependencies between tests.

Unit Tests - Summary

- More benefits than drawbacks.
- Mandatory in certain fields :
  - Automotive industry
  - Aeronautics
  - Electronics
  - ...

Integration Tests - Principles

- **Goal**: check that units interact correctly.
- If Unit Tests have been written for the component:
  - Only test nominal case.
  - No need to test abnormal cases.
- Different levels of Integration Testing
  - With some units (2 or more).
  - With all the application units (like in production).

Integration Tests - Principles

- May have side effects:
  - Calling other applications
    - Development or installation of test instances.
  - Sending mail
    - Using fake mail address.
  - Updating database
    - Reset database after tests.
  - ...

Integration Tests - Benefits

- More global
  - Help to check connections between components.
  - Help to check interactions between components.
- More concise
  - No need to test boundaries and abnormal cases.
  - Most of the time, one test is enough to check components well interaction.

Integration Tests - Drawbacks

- More unstable
  - Any component can produce errors (network, database, etc.).
  - Sensitive to code modifications.
- Takes longer
  - More components needed for initialization.
  - Slow-downs due to database loading, network latency, etc..

Integration Tests vs Unit Tests

- Unit Tests are more precise
  - Setting inputs manually.
  - Test boundaries and abnormal cases.
  - Test each functionalities in isolation
- Integration Tests help to test interaction between components.
- They are complementary
  - Need to think about ROI.

Integration Tests - Summary

- Help to test interactions between components.
- May involve few components or all components (like in production).
- Most of the time, only testing the nominal case is needed.

Functional Test - Principles

- Tests in real conditions, faking a user.
- **Goal**: Tests from a user's perspective
  - Most of the time, done manually by navigating into the application.
- Testing tools depend on the graphic interface.

Functional Test - Benefits

- More concrete
  - Check the application behaviour from the user's perspective.
- More reassuring
  - Check the application works as expected.

Functional Test - Drawbacks

- Very unstable
  - Almost any modification (even minor like HTML, CSS, etc.), may break tests.
- Harder to maintain
  - Must be updated frequently (after each modification).
  - Hard to understand (lots ot inputs, settings, etc. to consider).

Functional Test - Summary

- Help to test the real application as it will be used.
- Very unstable and hard to maintain.

Performance Tests - Principles

- Test application behaviour under heavy load (dozens, hundreds, thousands,... of users)
- Help to identify :
  - Performance bottlenecks.
  - Configuration errors.
  - Critical resources.
- The goal is not to detect any functionnal issues
  - Use Unit, Integration and Functional Tests for this purpose.

Performance Tests - Principles

- Must not be written before the end of development
  - Unlike other kinds of tests.
- Before development
  - Can be used to validate a POC's performances.
- Tools depends of the technology used
  - Some can be used for a different purpose

Performance Tests - Flow

- Tweak the configuration to « break the application ».
- Find the cause.
- Improve the application, and check for better results.

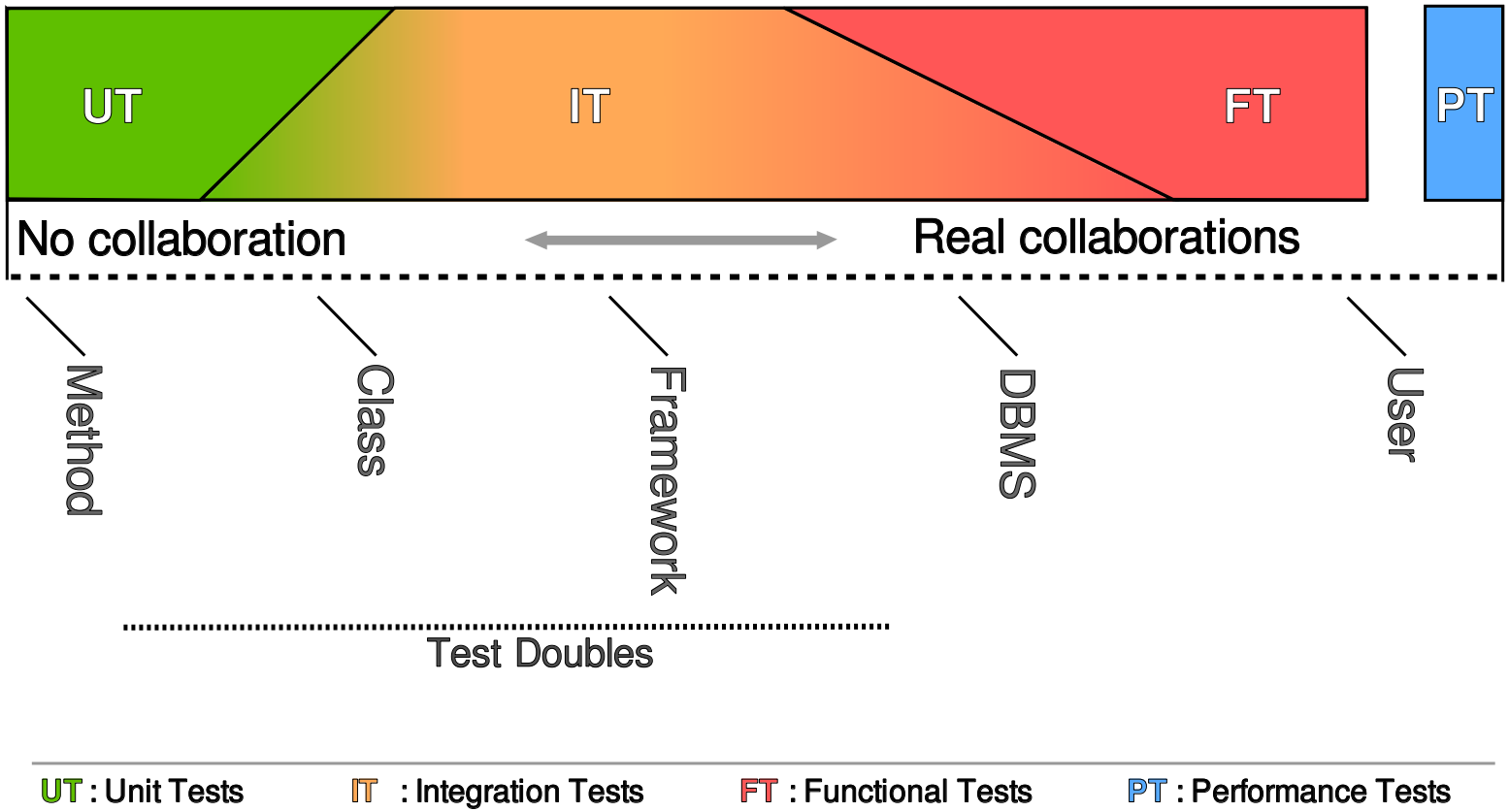
Performance Tests - Summary

- Help finetune an application's configuration.
- It is needed to know the specificities of each components.
- Never-ending cycle.

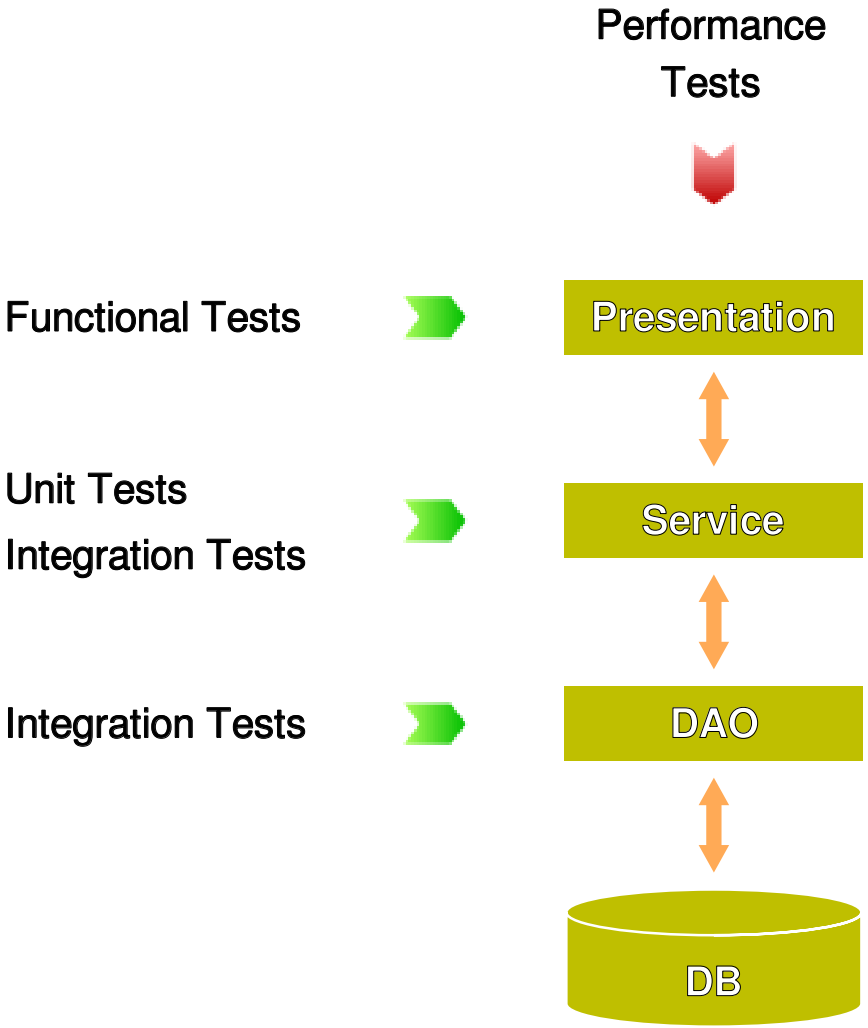
Summary

- Many different test types
  - Each adapted to a specific need.
  - Each coming with a set of specific tools.
- Impossible to test everything!
  - Choose depending on ROI.
- Understand the difference between « More » and « Better »
  - « More »: more tests, sometimes redundant.
  - « Better »: smarter, precise.
- « What am I testing ? »

Summary



Default case

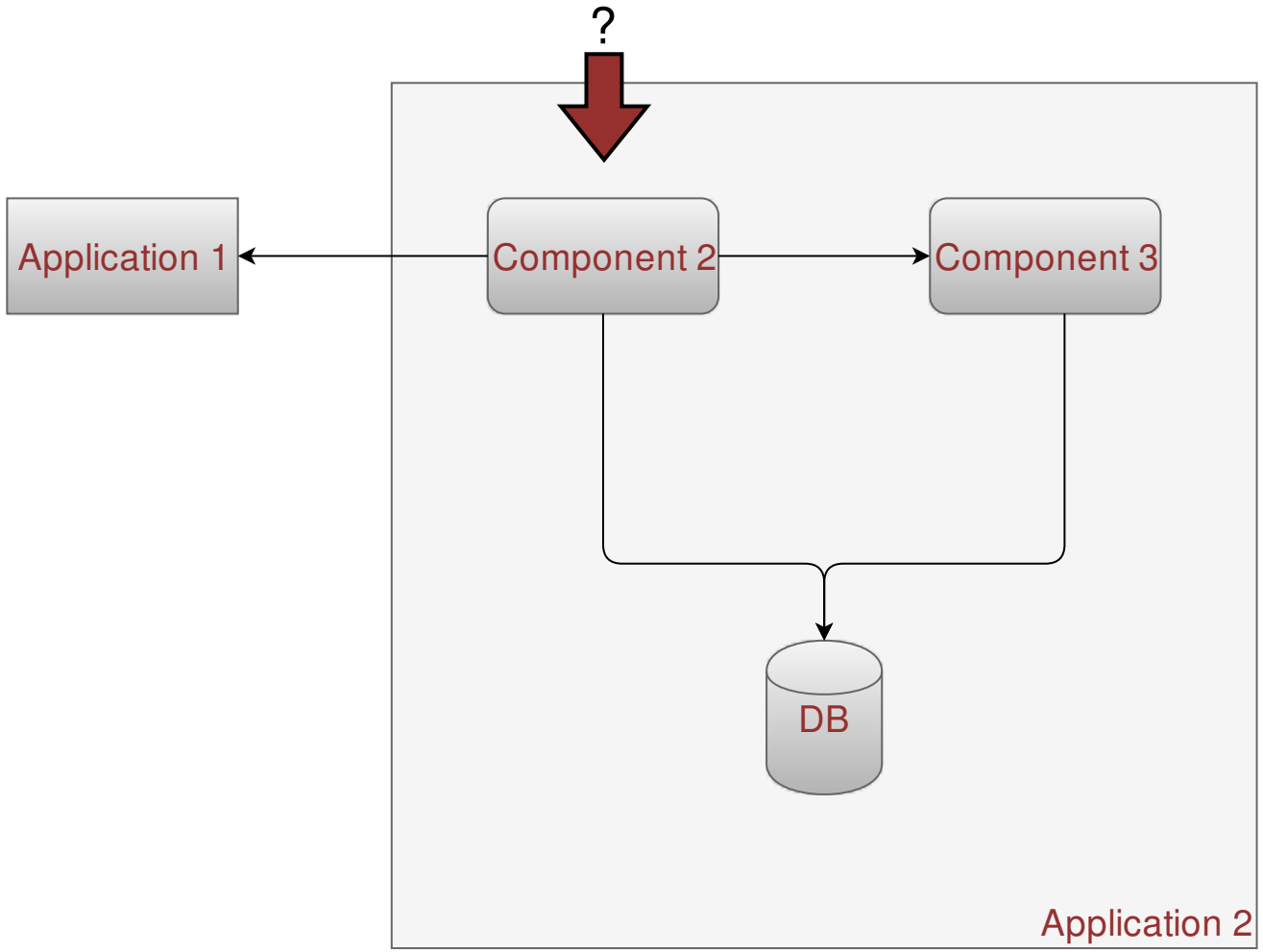


Test doubles

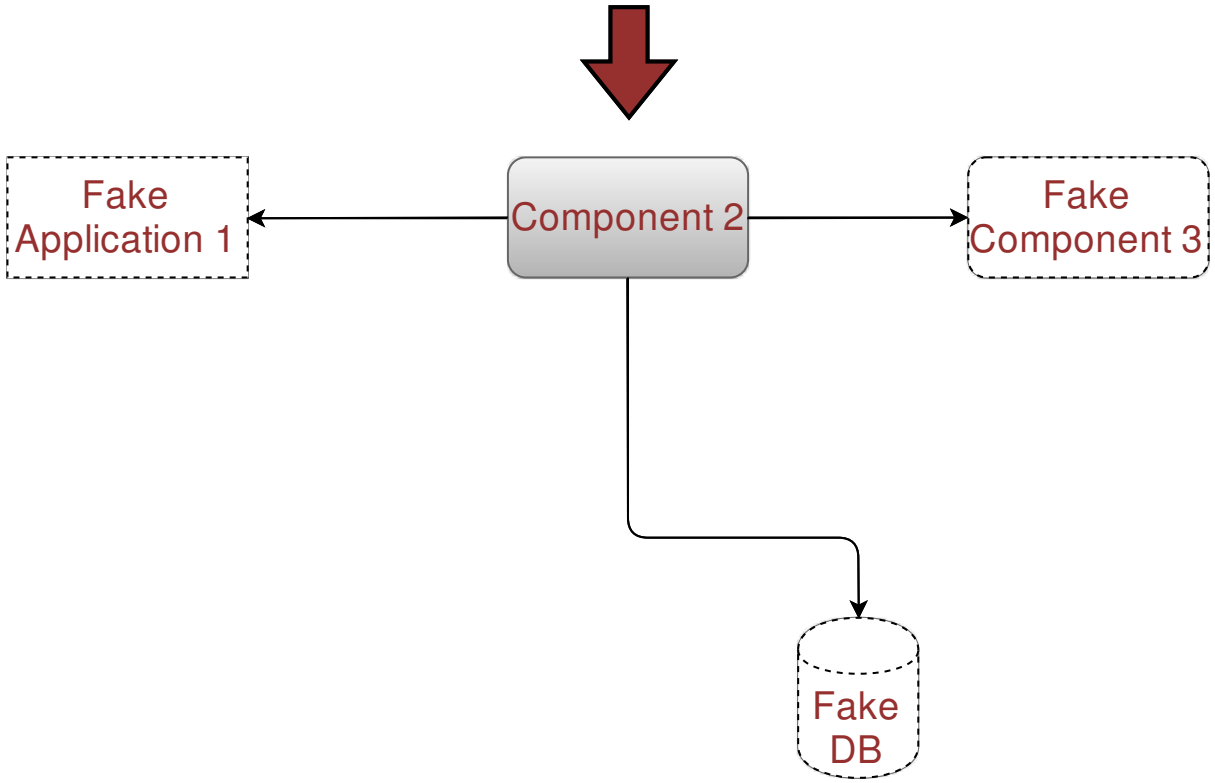
Agenda

- Which tests?
- *Test doubles*
- Test coverage
- The TDD principles
- Building Maintainable tests
- SOLID principles in the context of TDD
- Using TDD for Refactoring & code legacy
- Going further
- Annexes

How to create Unit Tests ?



White box / black box testing



Test Doubles - Principles

- Help isolate the tested unit from its internal dependencies.
  - Using fake implementation of real interfaces (cf. Liskov substitution principle).
- Necessary to fake:
  - An unavailable component.
  - An unavailable application while developing.
  - An element that's too complex or too time-consuming to be practical (database, ...).

Dummy

- A **Dummy** component fulfills the original component's contract (ie. implements its interface).
- All method must throw an exception.
- The object is never called directly, only its presence is needed
  - Passing it as a parameter to another method.
  - As an unused service.

Stub

- A **Stub** component fulfills the original component's contract (ie. implements its interface).
- Some of its methods always return the same values and ignore the inputs.

Spy

- A **Spy** component fulfills the original component's contract (ie. implements its interface).
- Help to check that some methods have been called (or not) as expected.
- Implements methods whose goal is to:
  - Count methods calling.
  - Record inputs parameter.

Mock

- A **Mock** fulfills the original component's contract (ie. implements its interface).
- **Mocks** are elementary implementations
  - Methods like **Stubs** with conditional treatments.
  - Handle some boundaries.

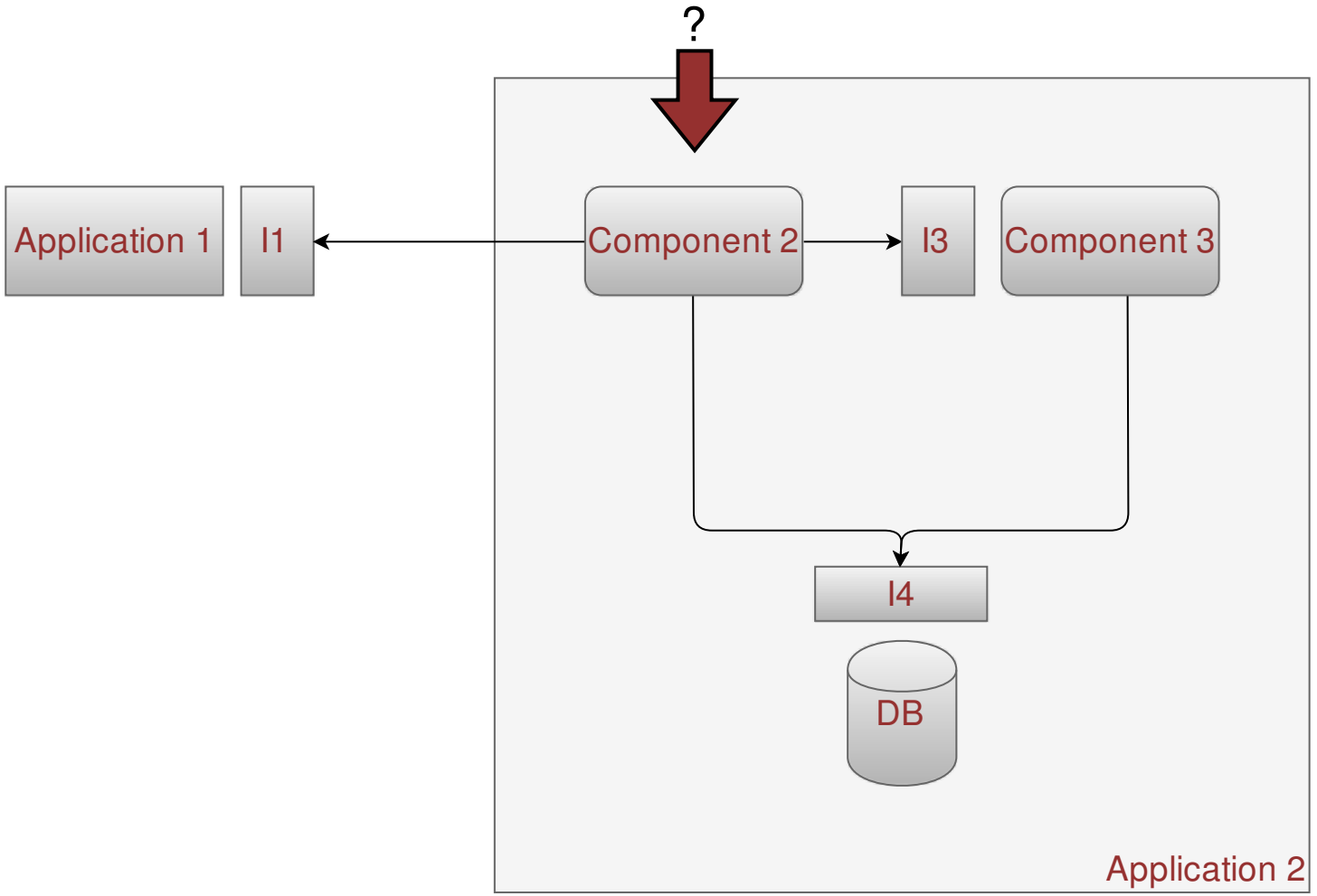
Fake Object

- A **Fake Object** fulfills the original component's contract (ie. implements its interface).
- Almost like a real implementation.
- Don't call any external systems.

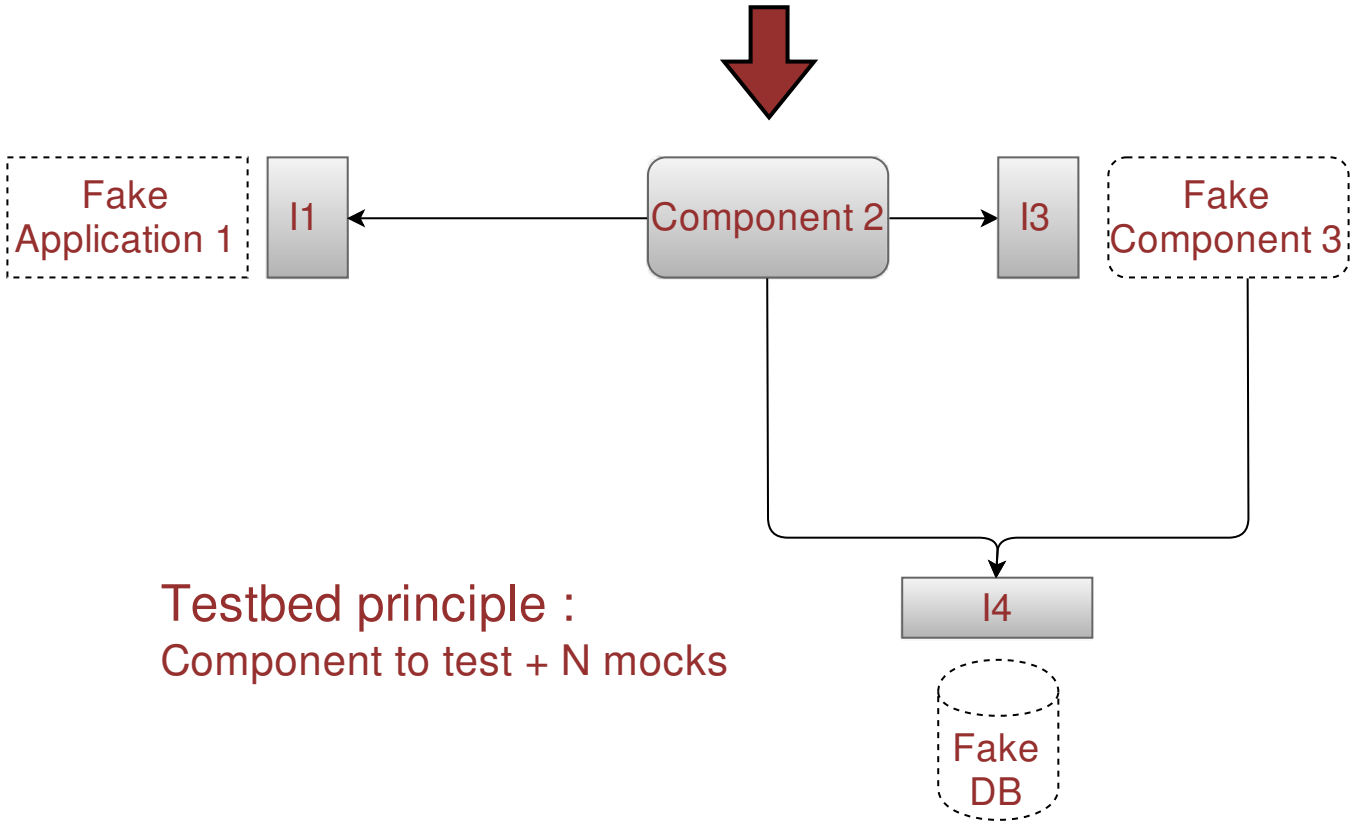
Using them

- Terminology is not really important.
- Keep in mind how to simply fake components / services / etc.
  - **Spy** can combine all behaviours.
  - A Test Double may be **Dummy** on some methods and **Fake** on others.
- Do what you really need.

Using interfaces (1/2)



Using interfaces (2/2)



Testbed principle :
Component to test + N mocks

Test Doubles caracteristics

- Look like the real object.
- Never contain business code.
- Are simple
  - No need to test them.
  - A default implementation is a good start.
- Are allowed to break encapsulation principle.
- Only used for Unit Tests or Integration Tests purposes.

When to use Test Doubles ?

- When the real object contain business code
- When objects interacts with other systems
  - Other components.
  - Other applications.
  - A database.
  - ...

Test Coverage

Agenda

- Which tests?
- Test doubles
- *Test coverage*
- The TDD principles
- Building Maintainable tests
- SOLID principles in the context of TDD
- Using TDD for Refactoring & code legacy
- Going further
- Annexes

Principles

- Metrics to identify which parts of your program are covered by tests
  - Percentage of code accessed by tests.
  - Metrics are computed for each method, function, classe, file,...
- Tools are needed.

Results analysis

- Percentage of code accessed by tests don't indicate the quality of the tests themselves
  - A code that is 100% covered may still be lacking tests.
  - On the opposite, a low code coverage definitely shows the code is not tested enough.
  - Coverage only indicate the percentage of code accessed by tests.

Results analysis

- Setting a percentage target may be counter productive
  - Tests of getters and setters just grow code coverage percentage without any added value.
  - Tests of impossible cases instead of actual business cases.
- The metric helps to improve code ***quality***
  - Helps find parts of code that are unsufficiently tested (or not at all).
  - Helps find tests that do nothing (aka. Liar tests)

Summary

- Useful metric but needs careful analysis
  - Helps findind parts of code that are unsufficiently tested (or not at all).
  - An excessively low code coverage is a warning.
  - 100% of code coverage doesn't mean 100% of tested code.
- Don't aim for a specific percentage
  - Tests quality is more important than this metric.

Mutation Testing

- **Mutation Testing** consists of intentional code modifications to make sure tests coverage is done correctly.
- Kinds of mutations:
  - Replacing an operator
  - Removing an instruction
  - ...
- If no test is failing, you'll need to reconsider your code coverage.

Mutation Testing (Mutant Generation)

- Try applying all possible mutations to generate mutants.
- If the mutation compiles, then a mutant is generated.

Mutation Testing (Mutant Elimination)

- Tests are running over mutants.
- For each mutant:
  - If all tests are failing
    - Tests cover the mutant code modification, the mutant is eliminated.
  - If some tests are still passing
    - Tests don't cover the mutation, they don't cover the code properly.

Mutation Testing - Summary

- Help to reinforce code coverage metric.
- Consume a lot of resources.
- May be very slow.
- Need a tool.
- Only apply to Unit Tests.

The TDD principles

Agenda

- Which tests?
- Test doubles
- Test coverage
- *The TDD principles*
- Building Maintainable tests
- SOLID principles in the context of TDD
- Using TDD for Refactoring & code legacy
- Going further
- Annexes

TDD: Test Driven Development (1/2)

- Started in early 2000s.
- Principle: ***Write a test then just write the most elementary code to have it pass***.

TDD: Test Driven Development (2/2)

- TDD is not only a test method:
  - ***TDD is a design method***.
- Applicable on:
  - [project] New or existing project.
  - [developments] New developments or bug fixes.

An "Extreme Programming" practice

Fine scale feedback	Continuous process
• Pair Programming • Planning Game • ***Test Driven Development*** • Whole team	• Continuous Integration • Design Improvement • Small Releases
Shared understanding	**Programmer welfare**
• Coding Standards • Collective Code Ownership • Simple Design • System Metaphor	• Sustainable Pace

Why TDD ? (1/4)

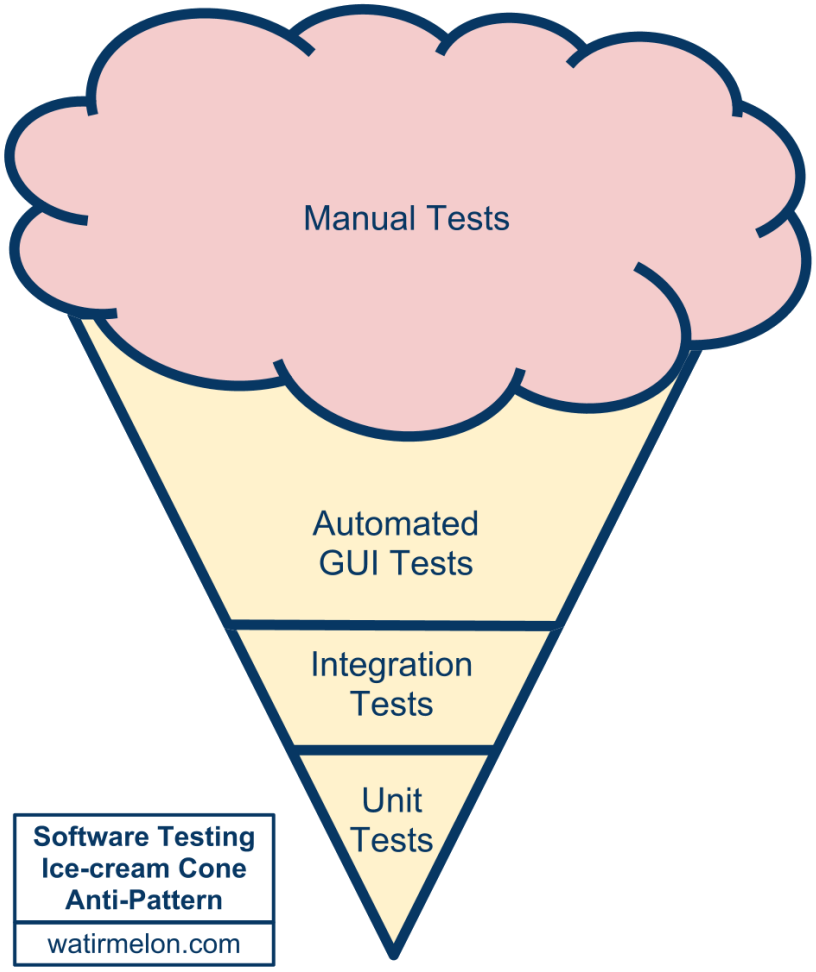
- Testing is mandatory to make code safer
  - Everybody does tests in different ways.
  - More or less rigorous.
- TDD suggests a «standard» way to consider Tests during development
  - Only the ways, not the tools.
  - Tools depend on the situation.
- Writing tests first forces to think about the needs before starting implementation.

Why TDD ? (2/4)

- Without TDD:
  - Lot of manual tests.
  - Replace them with automatic, faster tests.
  - Automatic tests are unstable and slow.
  - Maintenance hard and time consuming.
  - Because of a lack of time, Unit Tests are done for few cases.

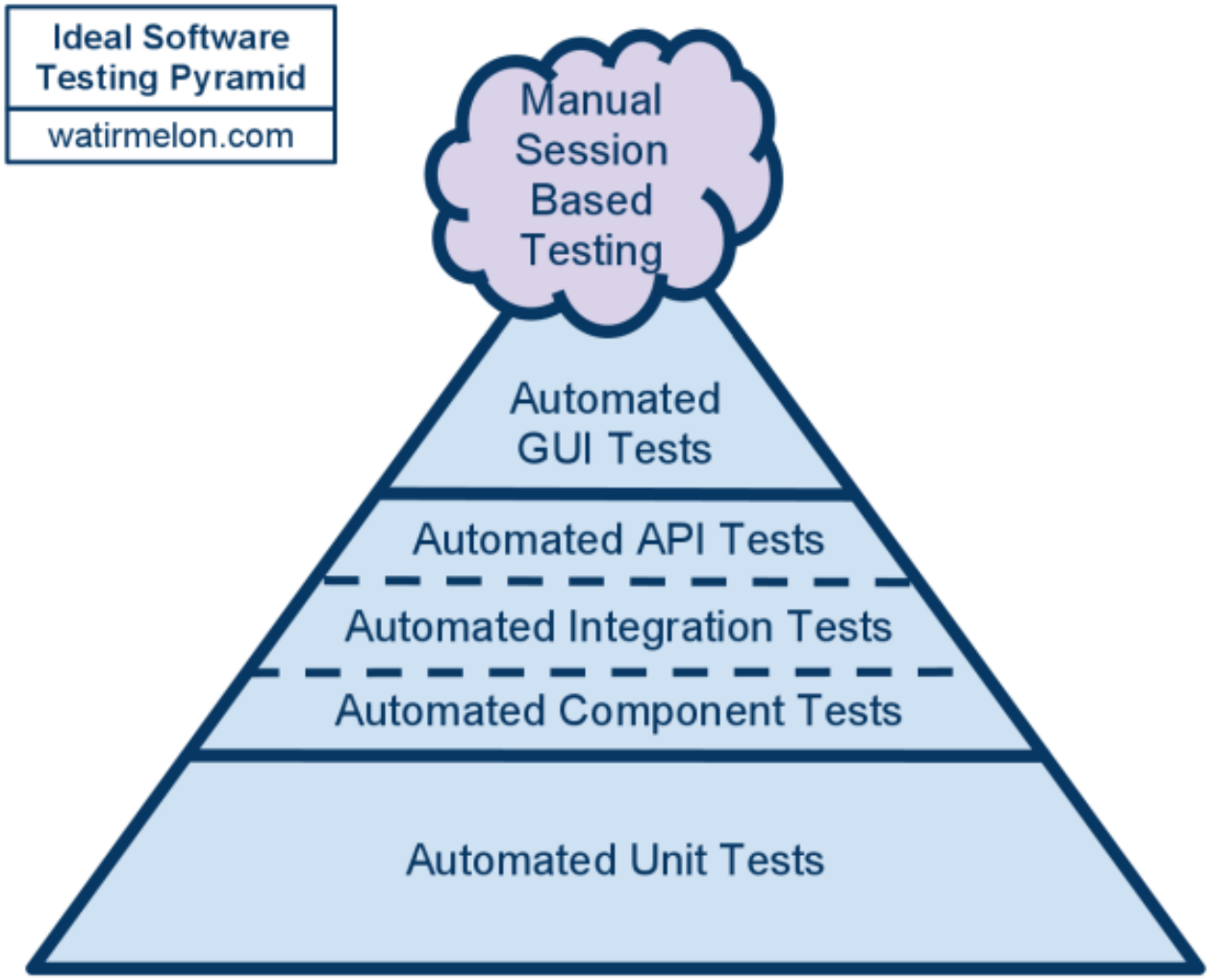
Why TDD ? (3/4)

Ice Cream Cone anti-pattern ! (defined by Alister Scott et Nathan Jones)



Why TDD ? (4/4)

Absolute pattern (defined by Mike Cohn)

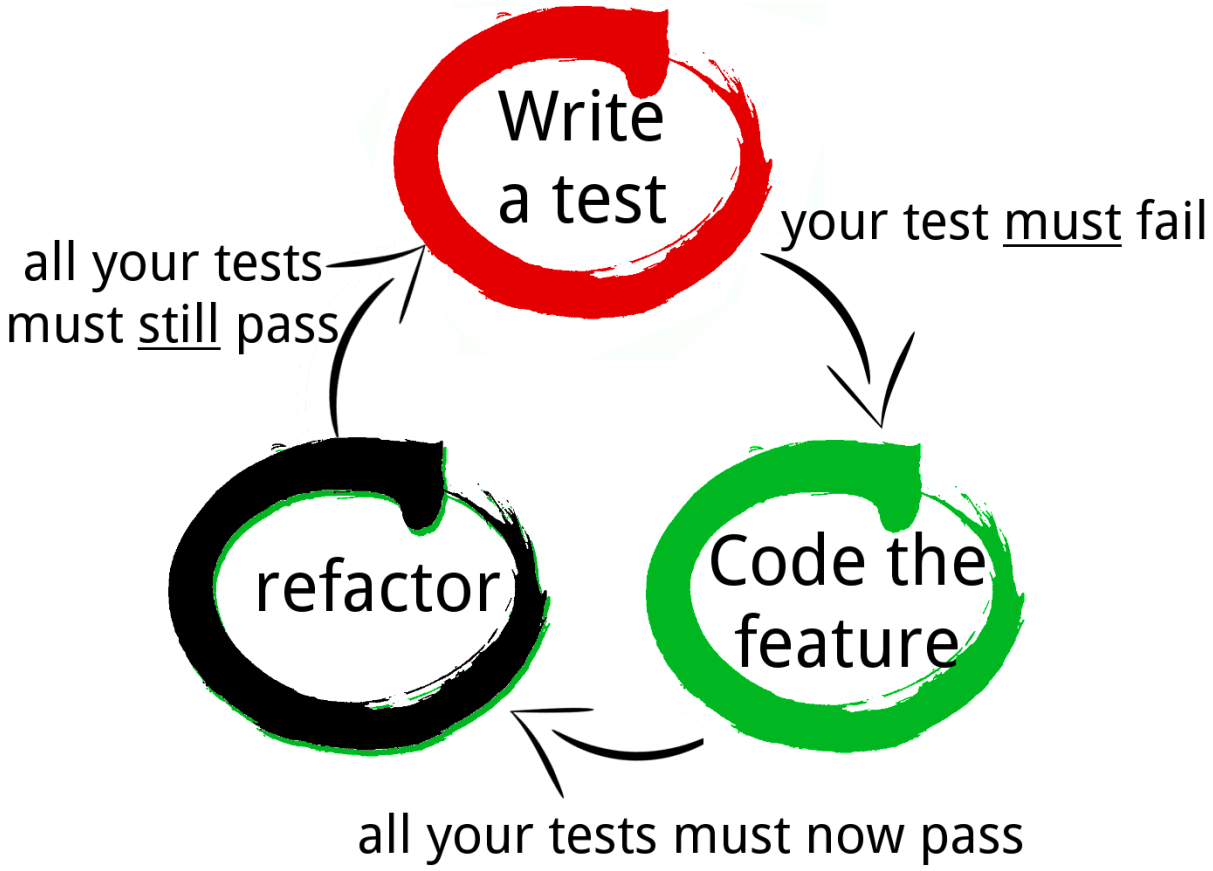


Tools requirements

- Needs:
  - Automated Test Unit.
  - 'true' or 'false' assertions.
  - Fast execution of one or more Test Units.
  - Rapid feedback for success or failure.
- Possible to do it manually...
  - Easier to use a generic tool.
  - Java: JUnit, TestNG.

TDD cycle: Introduction

- Described for the first time in « Test-Driven Development by Example », by Kent Beck, 2003



1 - Add a test

- With TDD, each new development starts by writing a new test « that must succeed ».
- This test must fail as long as the functionality is not implemented.
- Helps to understand the functional need and the specification from the beginning
  - Force to ask questions.

2 - Running the failing tests

- Run ***all*** the tests
  - Not only the ones that should fail.
- The new test ***must*** fail
  - For the targeted cause.
- The new test is ***the only one*** that must fail
  - The other tests must « succeed ».
  - No side effects on other tests.

3 - Writing the code

- The goal is to pass the test, even in an ugly way
  - For instance: don't factorize.
  - Improvements will be done in the coming steps.
- Focus on « passing the test »
  - Don't do any other development, it will not be tested.

3 - Writing the code

- Patterns to make test succeed
  - Fake it: return a constant, which will later be replaced by a variable.
  - Triangulation: with two example, the real implementation must be written.
  - Obvious implementation: If fast and simple, the real implementation may be directly written.
  - One to many: to implement an operation which takes a collection as a parameter, first start implementing for a single unit, then for the collection

4 - Running the passing tests

- Run ***all*** the tests
  - Not only the one that should pass.
- No tests must fail
  - Neither the new one, nor the old ones.

5 - Refactoring

- Now, the code may be cleaned up and refactored if needed
  - Possibility to factorize code and remove duplicated sections.
- Run all tests again. They must succeed
  - Certainty that nothing « broke » a functionality while refactoring.

TDD rules from Uncle Bob

Uncle Bob (Robert Martin) has defined 3 rules:

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

More information:
http://www.butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

TDD Cycle: Summary

- Repeat the cycle for each new functionality.
- Need to find a good size for each iteration.
- Iterations with different sizes
  - Depend on the developer, his experience et his self-confidence.
  - If iterations are too long (ie.: one day of test writing), cut up the functionality into smaller parts.

Emergent Design - Principles

- The architecture is not finalized before starting to code (**Design upfront**)
  - The tests help shape the architecture.
  - It remain as simple as possible.
  - Sufficient to pass tests.

Emergent Design - Practice

- Writing a test for a new functionality.
- If a method is too complex, create a dependency and delegate a part of treatment to it
  - **Divide and conquer**
- Mock dependencies to pass the test.
- Apply the same method to dependencies you only get simple classes.
- Don't hesitate to refactor as soon as necessary to make the architecture evolve.

Emergent Design - YAGNI Principle (1/2)

YAGNI (**Y**ou **A**in't **G**onna **N**eed **I**t)

- Just write the necessary code, to avoid the over-architecture trap.
- We're all tempted to add functionalities sooner rather than later, because we know how to add them and the system would look so much better with them...
- Derived from XP: Never add functionality too early.
- Adding flexibility more than necessary makes for a more complicated system

Focus on what is planned.

Emergent Design - YAGNI Principle (2/2)

- Writing unnecessary code too early:
  - Code base gets more cumbersome.
  - Source of bugs and unjustified complexity.
  - The implementation of a badly defined functionality will be inadequate in 90% of cases.
  - Longer development cycles without any direct justification (documentation and additional test).

YAGNI is not an excuse for lower code quality. YAGNI helps create more flexible code.

Emergent Design - Design Patterns

- Value Object
  - Immutable.
  - Two objects are equals if their corresponding fields are equals.
- Null Object
  - Empty implementation.
  - Avoid testing for null.
- Factory method
  - Create object via a method rather than using a constructor.
  - Encapsulation of object creation.

Emergent Design - Design Patterns

- Template method
  - Standardizes a sequence with default implementation
  - Specific behaviours are implemented in sub-classes.
- Composite
  - Same treatment for a group and a unit

Testing private methods ?

- One word: Never.
- A private method must not be tested.
  - It's an implementation detail.
  - They are tested indirectly through public methods.
  - If a private method is too complex to be tested through public methods.
    - Consider moving the private code into a new class.

TDD: Benefits (1/2)

- Positive effect on the code design
  - « Thinking with tests » helps to
    - Divide big, complex functionalities to make them easily testable.
    - Use the **Contract Design** principle (interfaces in Java).
- Better quality and more confidence
  - Each functionality is tested, therefore the entirety of the code is tested too
    - More global quality for the application.

TDD: Benefits (2/2)

- Identical or lower development time:
  - More code to write indeed.
  - BUT finding bugs earlier reduces costs: may cost a lot in case of late discovery.
- Less frequent use of the debugger
  - Code already tested « functionality by functionality » therefore almost « step by step »: less bad surprises.

TDD: Drawbacks (1/2)

- Communication in the team is crucial: Test***er*** Driven Development
  - Code reviews and discussions are needed to make sure that code design stays consistent across the application.

TDD: Drawbacks (2/2)

- Know what should and should not be tested
  - Don't test language API nor frameworks: important costs and useless.
  - Don't test the database: a database always works.
  - Don't test the same functionality multiple times (in case of refactoring for instance).
- GUI are hard to test.

Conclusion

- Test: important matter and needs to be agreed by everybody.
- TDD: simple method but still « theoretical »
  - Needs practice.

Building Maintainable tests

Agenda

- Which tests?
- Test doubles
- Test coverage
- The TDD principles
- *Building Maintainable tests*
- SOLID principles in the context of TDD
- Using TDD for Refactoring & code legacy
- Going further
- Annexes

Goals

- Test code must be polished, just like production code
  - Needs to be cleaned, refactored et ***reviewed***.
  - Simple, understandable et ***maintainable***.
- Unmaintainable tests
  - Are not maintained.
  - Fail more often (code or functionality modifications).
  - Are not relevant.
  - Are ignored.

Given / When / Then

- Common pattern for Unit Tests
  - ***Given***: setup mocks and inputs.
  - ***When***: run the method to be tested.
  - ***Then***: check output.
- May be implicit or explicit
  - Must be visible and understandable.

Configuration

- Tests configuration must be as simple as possible
  - If it becomes too complex, tested component needs refactoring.
  - Use fixtures.
  - Mock only what is necessary (use the right Test Doubles).
- Avoid sharing datasets
  - High coupling between tests.
- Use frameworks only to make code simpler.

Tests fixtures - Issues

- Datasets can be verbose and repetitive
  - Every test method creating the same objects over and over.
  - Property values for an object may be not relevant (they just need to exist).
  - Adding a new property forces to rewrite tests.
- Dataset creation code makes tests verbose and obscure.
  - Actual test code is less visible.
  - Test objective is less clear.

Tests fixtures - Methods

- Method which instantiates an object with default values.
- Useful when we need an object and don't care about its property values.
- You may add a few parameters to make the object configurable
  - Useless if too many parameters (use constructor or builder).

Tests fixtures - Methods (example)

```
@Test
public void should_increment_basket_price() {
  Product product = aProduct(12.0);
  Basket basket = new Basket();

  basket.add(product);

  assertThat(basket.totalPrice(), is(12.0));
}

private Product aProduct(double price) {
  Product product = new Product();
  product.setId(1L);
  product.setName("Banana");
  product.setPrice(price);
  return product;
}
```

Tests fixtures - Builder

- Pattern **Builder** to instantiate an object.
- Useful when a high degree of customisation is needed
  - Use default values for fields that don't need customisation.

```
@Test
public void should_increment_basket_price() {
  // Using Product's builder
  Product product = aProduct().named("Lemon").thatCosts(12.0).build();

  Basket basket = new Basket();
  basket.add(product);

  assertThat(basket.totalPrice(), is(12.0));
}
```

Tests fixtures - Builder (exemple)

```
public class ProductBuilder {
  private Long id = 1L;
  private double price = 0.0;
  private String name = "Banana";

  public static ProductBuilder aProduct() {
    return new ProductBuilder();
  }
  public Product build() {
    Product product = new Product();
    product.setId(id);
    product.setName(name);
    product.setPrice(price);
    return product;
  }
  public ProductBuilder named(String name) {
    this.name = name;
    return this;
  }
  [...]
}
```

Testing for randomness

- Methods that handle dates or random values are hard to test
  - Unpredictable results.
  - Unable to have test constants (***Flaky test***).
- Solution: Externalize the random factors
  - Possibility to mock dependencies and "settle" randomness.
  - Tests become predictable.

Hiding implementation details

- Test behaviours, not implementations.
- No need to know how a method is implemented
  - Don't test private methods.
  - Don't check ***every*** dependency call (only the ones actually needed by the functionality).
- Modifying the implementation may break all the tests
  - Lots of time needed to maintain the tests.
  - Loss of motivation.
  - Tests are ignored.

Failing for only one reason

- A test must fail for one reason only
  - Few assertions per test.
  - One action per test.
- When a test fails, we know why and we can fix it
  - Improves readability.
  - Easier to debug.

Naming classes

- Being able to identify a class just by its name is very useful.
- No unique way to proceed.
- The most important: consistency and respecting conventions.
- Example (default configuration of Surefire/Failsafe)
  - Foo a service interface.
  - FooImpl its implementation.
  - FooImplTest its unit test.
  - [FooImplIT its integration test.]

Naming methods

- A method name must describe what it is testing.
- It must describe the test itself, not the class being tested
  - Helps to understand the context when it's failing.
  - Serves as documentation.
  - The naming itself doesn't matter
    - Be consistent and clear.
    - No "best" choice.
  - Example: should_increment_basket instead of testBasket.
- Depending on the tool, it's possible to add a text description for the test.

Structure

- If a class contains a lot of methods, the test class can become huge
  - Maybe the class is doing too many things.
  - I may need refactoring.
- Create multiple test classes if needed
  - Split by functionality
  - Be careful and use consistent naming.
  - Exemple: ProductServiceImpl_BasketTest.

Summary

- Test code is very important
  - Serves as documentation.
  - Serves as example.
  - Is updated often.
- They must be written with the same care as production code
  - Design, code reviews, patterns, maintainability.
  - Java: http://www.petrikainulainen.net/writing-clean-tests/

SOLID principles in the context of TDD

Agenda

- Which tests?
- Test doubles
- Test coverage
- The TDD principles
- Building Maintainable tests
- *SOLID principles in the context of TDD*
- Using TDD for Refactoring & code legacy
- Going further
- Annexes

SOLID Principles

- 5 principles that need to be respected in order to produce code that is:
  - More stable.
  - More robust.
  - More extensible.
- Principles from "Uncle" Bob Martin (early 2000's).
- Acronym created by Michael Feathers.

S: Single Responsibility Principle (SRP)

- A class or a method must do only one thing.
- Therefore, it must only have one reason to change.
- For TDD this means:
  - Easier tests.
    - A short ***given*** part.
    - Only one reason to fail.
    - Easy to understand.

O: Open/Close Principle (OCP) (1/2)

- ***Open for extension, closed to modification***.
- It must be possible to add functionalities to software, without modifying the content.
  - Encapsulating fields and using private methods.
  - Writing generic code.
  - Injection of specific components, tru to generic interfaces.
- Best example: plugins.

O: Open/Close Principle (OCP) (2/2)

- For TDD:
  - A system becomes extremely testable.
  - Interfaces can be mocked.
  - Less tests needed to have a good code coverage.

L: Liskov Substitution Principle (LSP)

- An object may be replaced by any inheriting object.
  - Code doesn't need to know which implementation he's really manipulating.
  - **Design by Contract**.
- For TDD:
  - An object can be replaced by a mock without breaking the system.

L: Liskov Substitution Principle (LSP) - Example

- Square inherits from Rectangle
  - Height and width of a rectangle may be changed independently.
  - Height and width of a square are always equal.
- In Square, we automatically update width when height is modified
  - LSP rule violation.
  - Rectangle's contract is not respected if we need to change both independently.

I: Interface Segregation Principle (ISP) (1/2)

- A client should not depends on methods he's not using
- "Junk" interfaces must be divided into smaller interfaces:
  - More specific.
  - More consistent.
  - Clearer about their role.
- The system is more decoupled and easier to maintain.

I: Interface Segregation Principle (ISP) (2/2)

- For TDD:
  - Smaller interfaces are easier to mock.
  - Tests are shorter, less complex and more understandable.

D: Dependency Inversion Principle (DIP)

- The code must depend on abstractions, not on concrete implementations.
- Details are dependant on the abstraction, not the opposite
  - Inversion of dependencies.
  - Code is decoupled.
- For TDD:
  - Each component may be tested in isolation.
  - Implementation details are tested independently from the calling code.

D: Dependency Inversion Principle (DIP) - Example

- Example: Communication with a database
  - The system depends on an interface.
  - There's a specific implementation of interface depending on the database type.

SOLID - Summary

- Doing TDD helps to write SOLID code
  - Helps writing decoupled code that is testable in isolation (DIP).
  - Not every test needs to be changed after a new functionality (SRP).
  - Tests need to be understandable easily and fast to write (SRP, ISP).
- Non SOLID code will be harder to harder
  - A system that is hard to test may not be SOLID.

Code Legacy & Refactoring

Agenda

- Which tests?
- Test doubles
- Test coverage
- The TDD principles
- Building Maintainable tests
- SOLID principles in the context of TDD
- *Using TDD for Refactoring & code legacy*
- Going further
- Annexes

Code legacy (1/2)

- What is a legacy code ?
  - Definition by Michael Feathers

> *Code legacy is code without tests*

- Hard do modify
- Hard to understand and to control the impact of a modification

Code legacy (2/2)

- A vicious cycle
  - No test harness.
  - Code is harder to modify due to lack of confidence
  - New code is added without any tests.
  - Code becomes more complex.
  - Code becomes harder to modify.
  - Less and less time is available for the project.
  - Harder to add tests and refactor code.
  - ...

Refactoring

- Definition
  - Any action that modifies a program structure without any modification of its behaviour.
- Refactoring dilemma

> *When we refactor, we need tests and to add tests, we often need to refactor.*

Why do I need to refactor ?

- To improve software design.
- To make maintenance easier.
- To facilitate modifications.
- To add flexibility.
- To improve code reusability.
- To find bugs faster.

When is refactoring needed ?

- Code duplication.
- Long methods.
- Too many parameters.
- Incorrect naming.
- Misinformation.
- Useless code.

TDD & Legacy code: How ?

- Identify needed modifications.
- Identify what you need to test.
- Break dependencies.
- Write tests.
- Perform modifications and refactor.

When should I break dependencies ?

- When a method can't be run through the test harness.
- When a class can't easily be instantiated in the test harness.
- A few technics
  - Extract an interface.
  - Inheritance and overriding.
  - Transform a method into a class.
  - ...

If a test harness can't be used

- "Sprout class / method" pattern
  - When the behaviour I need to code may be implemented in a separate class / method for which I can use TDD.
- Pattern "Wrap class / method"
  - When the behaviour I need to code may be put before or after an existing method, use TDD to develop it.
- Going further with "Working effectively with legacy code" by Michael Feathers.

Going further

Plan

- Which tests?
- Test doubles
- Test coverage
- The TDD principles
- Building Maintainable tests
- SOLID principles in the context of TDD
- Using TDD for Refactoring & code legacy
- *Going further*
- Annexes

BDD

- ***Behaviour Driven Development***
- Using the natural 'business' language to write tests.
  - Common language shared between developers and BA.
  - Tools help transform these sentences into code.
- Facilitates collaboration between developers and BA.
- Tests and specifications are fused.

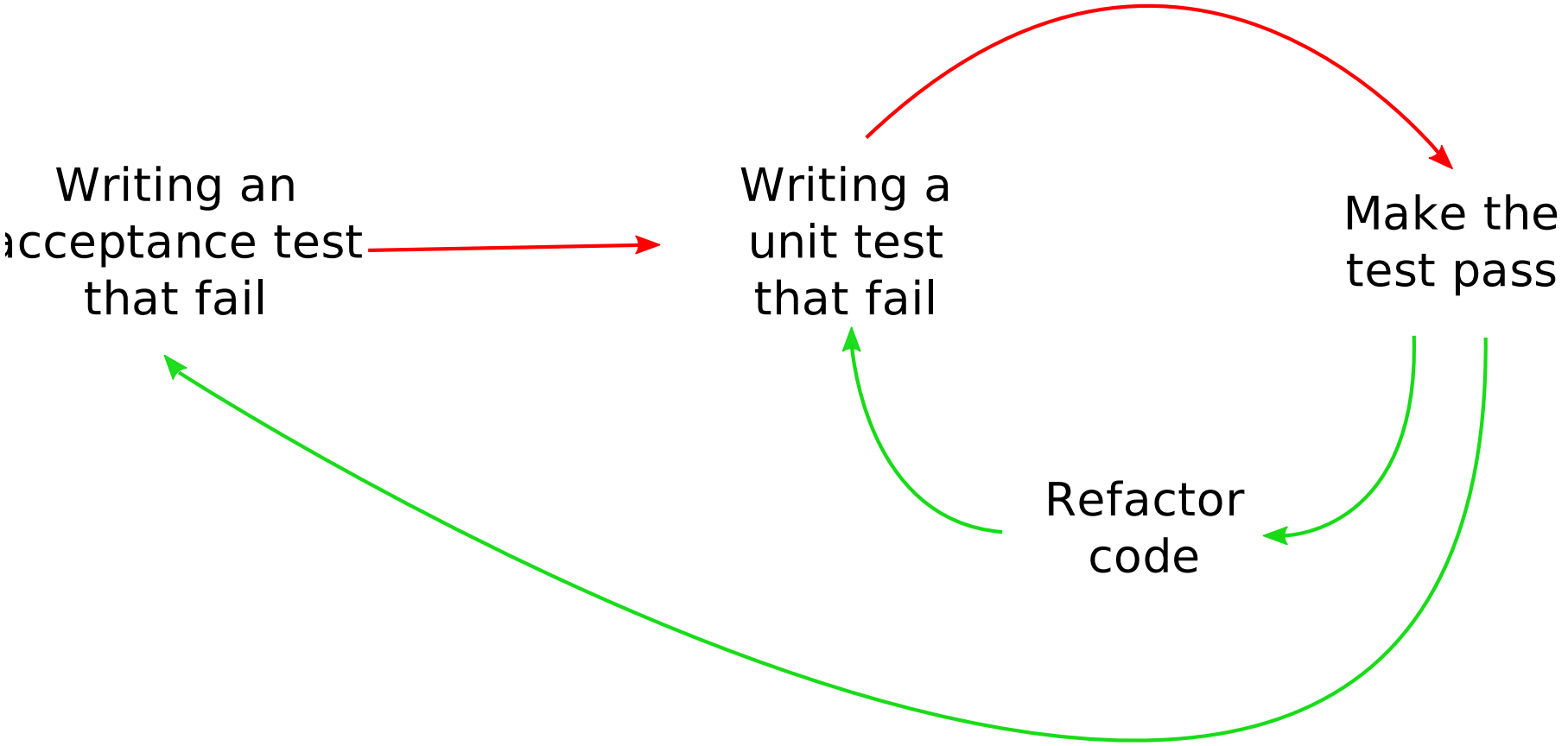
ATDD

- ***Acceptance Test Driven Development***
- Acceptance criteria definition
  - Collaboration between BA and developers to define criteria.
  - Tests are written to validate criteria.

BDD vs ATDD

- Tests focus on functionalities and not on implementation details.
- BDD Tests may be directly used while ATDD Tests are derived
  - BDD Tests are written in plain english and transformed into code.
  - ATDD Tests are derived from acceptance criteria (defined in a user story for instance).
- Both methods can be used with TDD.
  - They are integration tests.
  - Use double cycle.

Double cycle



Kata & Coding Dojo (1/2)

- Like with martial arts, we need training to improve.
- Katas are exercises made to train a specific topic (TDD, refactoring, ATDD, etc.)
  - It is possible to redo kata dozen of times with a different objective or methodology in mind.
  - Katas are usually very short (at the very most a couple hours).
  - There is never a "good answer", it's just a pretext for training.

Kata & Coding Dojo (2/2)

- A Coding Dojo is a meetup where developers come to train together.
- Learning is a collective process and can take many forms.
  - Katas resolution.
  - Peer programming.
  - Mob programming.
- In any case, the goal is to share knowledge.
  - Final code review.
  - Sharing solutions.

Continuous Integration

- Permanent checking to avoid regression
  - Checks out code from SCM.
  - Runs automated tests.
  - Informs and **stops chain** when a test fails.
- Static analysis of code quality.
- Continuous delivery and deployment
  - Ignored if tests are failing.

JUnit

Introduction (1/2)

- OpenSource Test Framework for developer
  - https://github.com/junit-team/junit
- Simple / Fast to learn and use
  - Make easy to build tests.
  - Help to run all tests: avoid regression.
- Available by default in IDE (Eclipse, IntelliJ, NetBeans).

Introduction (2/2)

- For Unit tests
  - Not only (DBUnit, Spring, AssertJ-Swing, ...).
- Good integration with build tools
  - Maven, Gradle, Ant, ...

@Test annotation

- @Test: mark a method to be run as a test.

```
public class TestExample {

  @Test
  public void dummyTest() {
      assertTrue(true);
  }

}
```

@Before et @After annotations

- @Before execute method before each tests to initialize objects for instance.
- @After execute method after each tests to clean execution context.

@Before et @After annotations

- Example:

```
public class TestExample {
  private List<String> fruits;

  @Before
  public void setUp() {
    this.fruits = Collections.emptyList();
  }

  @Test
  public void dummyTest() {
    [...]
  }

  @After
  public void cleanup() {
      this.fruits.clear();
  }
}
```

@BeforeClass et @AfterClass annotations

- @BeforeClass execute static method before running all the class tests. Use to initialize resources with high cost
- @AfterClass execute static method after running all the class tests to clean execution context.

@BeforeClass et @AfterClass annotations

- Example:

```
public class RepositoryTest {
  private static DatabaseConnection connection;

  @BeforeClass
  public static void setUp() {
    connection = DriverManager.getConnection(url, user, password);
  }

  @Test
  public void dummyTest() {
    [...]
  }

  @AfterClass
  public static void clean() {
    connection.close();
  }
}
```

@RunWith annotation

- @RunWith: define a test runner.
- Default runner is BlockJUnit4ClassRunner but there is different runners provided by JUnit:
  - Suite
  - Parameterized
  - Categories

@RunWith annotation - Suite Runner

- Suite runner execute test classes annotated with @SuiteClasses.

```
@RunWith(Suite.class)
@SuiteClasses({TestA.class, TestB.class})
public class ExampleSuiteTest {

}
```

@RunWith annotation - Parameterized Runner

- Parameterized runner execute parameterized tests.
  - @Parameters to specify datasets.
  - @Parameter to inject datasets before running test(s).

```
@RunWith(Parameterized.class)
public class Test Example {
  @Parameters
  public static Collection<String> parameters() {
    return Arrays.asList("A", "B", "C");
  }

  @Parameter
  public String parameter;

  @Test
  public void dummyTest() {
    System.out.println(parameter);
  }
}
```

@RunWith annotation - Categories Runner

- Categories runner execute tests annotated with @Category
- Warning: available in experimantal package org.junit.experimental

```
@RunWith(Categories.class)
@IncludeCategory(SlowerTest.class)
@Suite.SuiteClasses({ComponentTest.class})
public class SuiteTestExample {}

public class ComponentTest {

  @Category(FasterTest.class)
  @Test
  public void unitTest() {
     [...]
  }

  @Category(SlowerTest.class)
  @Test
  public void integrationTest() {
     [...]
  }
}
```

@Rule annotation

- A Rule help to execute code before and/or after each tests to customize the behaviour.
- Known Rule are:
  - Timeout
  - TemporaryFolder
  - ErrorCollector
  - Verifier
  - ExpectedException
  - TestWatcher
  - RuleChain

@Rule annotation- Timeout

- Timeout rule fail tests if time of execution is higher than the one specified.

```
public class TestExample {

  @Rule
  public TestRule globalTimeout = new Timeout(2, TimeUnit.SECONDS);

  @Test
  public void success() {
    [...]
  }

  @Test
  public void failed() {
    while(true) {
      [...]
    }
  }

}
```

@Rule annotation - TemporaryFolder

- TemporaryFolder rule helps to create folders / files and remove them automatically.

```
public class TestExample {

  @Rule
  public TemporaryFolder folder = new TemporaryFolder();

  @Test
  public void testUsingTempFolder() throws IOException, InterruptedException {

    File createdFile = folder.newFile("test-file.txt");
    File createdFolder = folder.newFolder("test-directory");
    [...]
  }

}
```

@Rule annotation - ErrorCollector

- ErrorCollector rule helps to catch errors and failing test without stopping test execution.

```
public class ExampleTest {

  @Rule
  public ErrorCollector collector= new ErrorCollector();

  @Test
  public void dummyTest() {
    [...]
    collector.addError(new Throwable("first thing went wrong"));
    [...]
    collector.addError(new Throwable("second thing went wrong"));
    [...]
  }

}
```

@Rule annotation - Verifier

- Verifier rule fail tests when assertions failed.

```
public class TestExample {
  private StringBuilder errorLog = new StringBuilder();

  @Rule
  public Verifier collector = new Verifier() {
    @Override
    protected void verify() {
      assertTrue(errorLog.toString().isEmpty());
    }
  };

  @Test
  public void success() {}

  @Test
  public void failed() {
    errorLog.append("test");
  }
}
```

@Rule annotation - ExpectedException

- ExpectedException rule validate exception type and content (message, code, etc.)
- More accurate than @Test(expected = Exception.class).

```
public class TestExample {
  @Rule
  public ExpectedException thrown = ExpectedException.none();

  @Test
  public void successThrowsNothing() {}

  @Test
  public void successthrowsNullPointerException() {
    thrown.expect(NullPointerException.class);
    throw new NullPointerException();
  }

  @Test
  public void failedNoThrowsNullPointerException() {
    thrown.expect(NullPointerException.class);
  }
}
```

@Rule annotation - TestWatcher

- TestWatcher rule help to be nofity when test state change.

```
public class RuleExample extends TestWatcher {
  @Override
  public Statement apply(Statement base, Description description) {
    return super.apply(base, description);
  }

  @Override
  protected void succeeded(Description description) {}

  @Override
  protected void failed(Throwable e, Description description) {}

  @Override
  protected void skipped(AssumptionViolatedException e, Description description)
{}

  @Override
  protected void starting(Description description) {}

  @Override
  protected void finished(Description description) {}
}
```

@Rule annotation - Custom Rule

- To define custom Rule, extend abstract class ExternalResource.
  - Preferred way to create custom rule
  - Can be compared to @Before et @After annotations.

```
public class CustomRule extends ExternalResource {
  private Connection connection;

  @Override
  protected void before() throws Throwable {
    this.connection = DriverManager.getConnection(url, user, password);
  }

  @Override
  protected void after() {
    this.connection.close();
  }
}
```

@Rule annotation - Custom Rule

- Can use TestRule.
- Be careful

```
public class RuleExample implements TestRule {
  private String message;

  public ConsoleOutRule(String s) {
    this.message = s;
  }

  @Override
  public Statement apply(final Statement base, Description description) {
    return new Statement() {
      @Override
      public void evaluate() throws Throwable {
        System.out.println("Start "+message);
        base.evaluate();
        System.out.println("End "+message);
      }
    };
  }
}
```

@Rule annotation- RuleChain

- RuleChain rule combine multiple rules using predefined order.

```
public class TestExample {

  @Rule
  public TestRule chain = RuleChain
                            .outerRule(new ConsoleOutRule("outer rule"))
                            .around(new ConsoleOutRule("around rule"));

  @Test
  public void dummyTest() {
      System.out.println("run dummyTest");
  }

}
```

@ClassRule annotation

- @ClassRule annotation apply rules before / after execution of all class tests.

```
public class TestExample {
  @ClassRule
  public static ExternalResource resource = new ExternalResource() {
    @Override
    protected void before() throws Throwable {
      System.out.println("before");
    }

    @Override
    protected void after() {
      System.out.println("after");
    }
  };

  @Test
  public void dummyTest1()  { System.out.println("dummyTest1"); }

  @Test
  public void dummyTest2()  { System.out.println("dummyTest2"); }
}
```

Assertions - assert

- To pass a test, we must do assertions about the expecting result.
- Some static methods are available
  - assertEquals & assertNotEquals
  - assertArrayEquals
  - assertSame & assertNotSame
  - assertNull & assertNotNull
  - assertTrue & assertFalse
- The signature is:
  - method(message, expected, actual)
  - method(expected, actual)

Assertions - assert

- Examples:

```
@Test
public void basic_junit_assert() {
  assertEquals("Orange", "Orange");
  assertNotEquals("Orange", "Banana");

  assertArrayEquals(
          new String[] { "Orange", "Banana" },
          new String[] { "Orange", "Banana" }
  );

  assertSame(new Orange(), new Orange());
  assertNotSame(new Orange(), new Banana());

  assertNull(null);
  assertNotNull("Banana");

  assertTrue(valid);
  assertFalse(invalid);
}
```

Assertions - assertThat

- public static <T> void assertThat(T actual, Matcher<? super T> matcher)
  - Type safe (by using Generics).
- ***Hamcrest*** is a Matcher library
  - New way to do assertions with Matcher.
  - Matchers make assertThat API clearer and more understandable than assert* methods.
  - May chain matchers.
  - Error messages are more accurate.
  - Hamcrest may be used with TestNG.

Assertions - assertThat vs assert*

- Equality comparison

```
  assertThat("Orange", is("Orange"));
  assertEquals("Orange", "Orange");

  assertThat("Orange", not(is("Banana")));
  assertNotEquals("Orange", "Banana");

  assertThat(new String[]{"Orange", "Banana"}, is(new String[]{"Orange",
"Banana"}));
  assertArrayEquals(new String[]{"Orange", "Banana"}, new String[]{"Orange",
"Banana"});
```

- Instances comparison

```
Orange orange = new Orange();
assertThat(orange, is(sameInstance(orange)));
assertSame(orange, orange);

Banana banana = new Banana();
assertThat(orange, not(sameInstance(banana)));
assertNotSame(orange, banana);
```

Assertions - assertThat vs assert*

- Conditional comparison

```
assertThat(valid, is(true));
assertTrue(valid);

assertThat(invalid, is(false));
assertFalse(invalid);
```

- Null comparison

```
assertThat(null, nullValue());
assertNull(null);

assertThat("Orange", notNullValue());
assertNotNull("Orange");
```

Assertions - assertThat

- Chain matchers

```
assertThat("Orange", is(allOf(notNullValue())));
assertThat("Orange", is(allOf(notNullValue(), instanceOf(String.class))));
assertThat("Orange", is(allOf(notNullValue(),
                              instanceOf(String.class),
                              equalTo("Orange")
                             )));

assertThat("Orange", is(anyOf(startsWith("Or"))));
assertThat("Orange", is(anyOf(startsWith("Or"), endsWith("ne"))));

List<String> fruits = Arrays.asList("Orange", "Banana");

assertThat(fruits, everyItem(is(anyOf(startsWith("Or"),
                                      endsWith("na")
                                ))));
assertThat(fruits, everyItem(both(is(anyOf(startsWith("Or"), startsWith("Ba"))))
                                 .and(endsWith("a"))));
```

Assertions - Custom Matcher

- May create custom Matcher to remove code duplication.
- Need to extend one of the following class:
  - BaseMatcher base class of all matchers.
  - TypesafeMatcher based on BaseMatcher, type safe and exlude null value.
  - TypesafeDiagnosingMatcher similar to TypesafeMatcher but need to use error message.
  - FeatureMatcher based on TypeSafeDiagnosingMatcher to create matcher for a specific functionality.

Assertions - Custom Matcher

```
public class WordLengthMatcher extends FeatureMatcher<String, Number> {

  public WordLengthMatcher(Matcher<? super Number> subMatcher, String
featureDescription, String featureName) {
    super(subMatcher, featureDescription, featureName);
  }

  @Override
  protected Number featureValueOf(String actual) {
    return actual.length();
  }

}

@Test
public void testCustomMatcher() throws Exception {
  WordLengthMatcher matcher = new WordLengthMatcher(IsEqual.<Number>equalTo(6),
"Only words with numbers of characters ", "Numbers of characters");
  assertThat("orange", matcher); // Success
  assertThat("apricot", matcher); // Failed  Expected: Only words with numbers of
characters <6> but: Numbers of characters was <7>
}
```

Assertions - Ignore & Assume

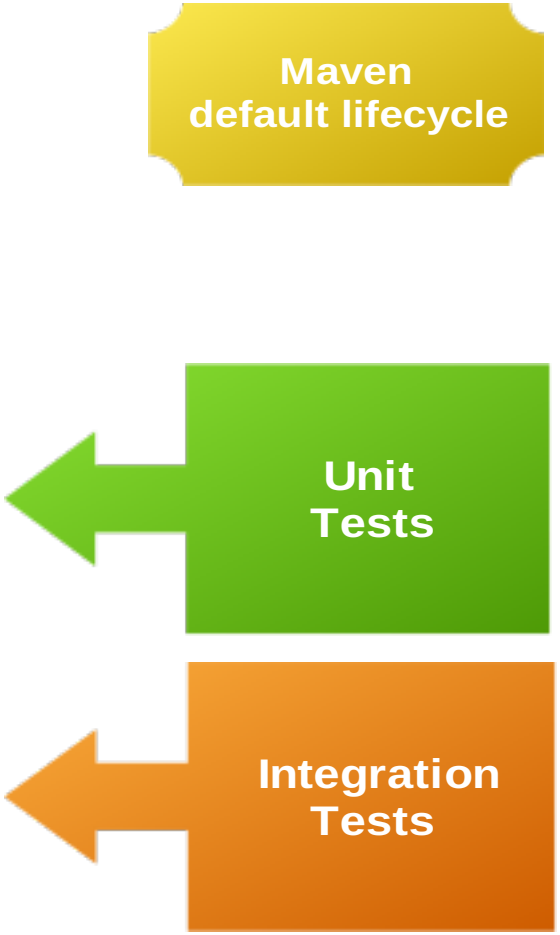
- JUnit ignore test annotated with @Ignore
- To ignore test based on assumptions, use:
  - static method void assumeThat(T actual, Matcher<T> matcher)
  - Hamcrest Matchers
- While running the test, if assumptions are not true the test is ignored.

```
@Test
public void testOnlyWorksUnderLinux() {
  assumeThat(File.separatorChar, is('/'));
  [...]
}
```

Maven

Cycle

- **validate**
- initialize
- generate-sources
- process-sources
- generate-resources
- process-resources
- **compile**
- process-classes
- generate-test-sources
- process-test-sources
- generate-test-resources
- process-test-resources
- test-compile
- process-test-classes
- **test**
- prepare-package
- **package**
- pre-integration-test
- **integration-test**
- post-integration-test
- verify
- **install**
- deploy



Cycle - Unit tests

- generate-test-sources: generate classes before test execution. Equal to generate-sources phase for tests.
- generate-test-resources: generate des resources before test execution. Equal to generate-resources phase for tests.
- test: running unit tests.

Cycle - Integration tests

- pre-integration-test: use to perform actions or run services before running integration tests
  - start embedded tomcat, selenium server, embedded database, etc.
- integration-test: running integration tests.
- post-integration-test: use to perform actions or stop services after running integration tests
  - stop embedded tomcat, selenium server, embedded database, etc.

Surefire

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven-surefire-plugin.version}</version>
</plugin>
```

- Surefire plugin is used to run unit tests during test phase.
- May generate report, useful on CI platform.
- Common test files pattern:
  - **/Test*.java, **/*Test.java et **/*TestCase.java
  - ie: MyClassTest.java, TestMyClass.java
  - Files must respect those patterns to be executed.
- To run:

```
mvn test
```

Failsafe

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>${maven-failsafe-plugin.version}</version>
</plugin>
```

- Failsafe is used to run integration tests during integration-test phase.
- May generate report, useful on CI platform.
- Common test files pattern:
  - **/IT*.java, **/*IT.java et **/*ITCase.java
- To run:

```
mvn verify
```

Failsafe & Surefire

- Most of the time, they are used with maven profiles
  - In order to run one or the other.
  - May generate different reports, depending on the need.
- Documentation
  - http://maven.apache.org/surefire/maven-surefire-plugin
  - http://maven.apache.org/surefire/maven-failsafe-plugin

AssertJ

Introduction

- Based on JUnit.
- Provide rich assertion.
- Provide ***fluent*** API.
- Provide chained assertions.

```
assertThat("toto").isEqual("toto");
```

Assertions - assert

- org.assertj.core.api.Assertions class provide multiple static methods using the following pattern:
  - assertThat(actual).<matcher>(expected)
- Each method exist with a signature according to a different type.
- Matchers depends on parameter type when calling assertThat(actual)
- Assertions may be chained (***fluent*** API).

```
assertThat("foobar")
  .isNotNull()
  .isNotEmpty()
  .startsWith("foo")
  .doesNotContain("goo");
```

Assertions - introspective assertions

- Compare specific attributes of objects.

```
Fruit redBanana = new Fruit();
redBanana.setName("Banana");
redBanana.setColor(Color.RED);
redBanana.setWeight(10);

Fruit yellowBanana = new Fruit();
yellowBanana.setName("Banana");
yellowBanana.setColor(Color.YELLOW);
yellowBanana.setWeight(20);

assertThat(redBanana)
  .isEqualToIgnoringGivenFields(yellowBanana, "color", "weight");
```

Assertions - introspective assertions

- Compare specific attributes of objects contained in Collection or Map.
- Extraction of one field

```
Fruit redBanana = new Fruit("Red Banana", Color.RED, 10);
Fruit greenBanana = new Fruit("Green Banana", Color.GREEN, 15);
Fruit yellowBanana = new Fruit("Yellow Banana", Color.YELLOW, 20);
List<Fruit> fruits = Arrays.asList(redBanana, greenBanana, yellowBanana);

// Untyped
assertThat(fruits).extracting("color")
  .containsExactly(Color.RED, Color.GREEN, Color.YELLOW);

// Typed
assertThat(fruits).extracting("color", Color.class)
  .containsExactly(Color.RED, Color.GREEN, Color.YELLOW);

assertThat(fruits).extracting(fruit -> fruit.getColor())
  .containsExactly(Color.RED, Color.GREEN, Color.YELLOW);
```

Assertions - assertions introspectives

- Extraction of multiple fields

```
Fruit redBanana = new Fruit("Red Banana", Color.RED, 10);
Fruit greenBanana = new Fruit("Green Banana", Color.GREEN, 15);
Fruit yellowBanana = new Fruit("Yellow Banana", Color.YELLOW, 20);
List<Fruit> fruits = Arrays.asList(redBanana, greenBanana, yellowBanana);

// Untyped
assertThat(fruits).extracting("color", "name") //
  .containsExactly(
    tuple(Color.RED, "Red Banana"),
    tuple(Color.GREEN, "Green Banana"),
    tuple(Color.YELLOW, "Yellow Banana")
  );

// Typed
assertThat(fruits) //
  .extracting(fruit -> fruit.getColor(), fruit -> fruit.getName()) //
  .containsExactly(
    tuple(Color.RED, "Red Banana"),
    tuple(Color.GREEN, "Green Banana"),
    tuple(Color.YELLOW, "Yellow Banana")
  );
```

Assertions - messages

- To customise error message:
  - .describedAs(...)
  - .as(...) alias to .describedAs(...)
  - .overridingErrorMessage(...)

```
assertThat("robert").describedAs("name").isEqualTo("hubert");
assertThat("robert").as("name").isEqualTo("hubert");

// org.junit.ComparisonFailure: [name] expected:<"[hu]bert"> but was:<"[ro]bert">

assertThat("robert").describedAs("name of %s", "user").isEqualTo("hubert");
assertThat("robert").as("name of %s", "user").isEqualTo("hubert");

// org.junit.ComparisonFailure:
// [name of user] expected:<"[hu]bert"> but was:<"[ro]bert">
```

Assertions - messages

- Examples:

```
assertThat("robert")
  .describedAs("name of %s", "user")
  .isEqualTo("robert")
  .as("name must be empty.")
  .isEmpty();

// java.lang.AssertionError: [name must be empty.] Expecting empty but was:
<"robert">
```

```
Fruit fruit = new Fruit();
fruit.setName("Banana");
fruit.setColor(Color.RED);

try {
  assertThat(fruit.getColor())
    .as("check %s's color", fruit.getName())
    .isEqualTo(Color.YELLOW);
} catch (AssertionError ex) {
  assertThat(ex)
    .hasMessage("[check Banana's color] expected:<[YELLOW]> but was:<[RED]>");
}
```

Assertions - Java 8

- WIth Java8, Type inference has been improved and may produce ambiguities.
- AssertJ suggest the use of org.assertj.core.api.AssertionsForInterfaceTypes.
- AssertionsForInterfaceTypes has same method than Asssertions.
- Assertions is a delegation on AssertionsForClassTypes.

Warning: StrictAssertions has been replaced by AssertionsForInterfaceTypes and AssertionsForClassTypes with version 3.2.0.

Custom assertions

- Inherit AbstractAssert.
- Create factory having static methods calling custom assertions.

Custom assertions

```
public class FruitAssert extends AbstractAssert<FruitAssert, Fruit> {
  public FruitAssert(Fruit actual) {
    super(actual, FruitAssert.class);
  }

  public FruitAssert hasColor(Color color) {
    // check that actual Fruit we want to make assertions on is not null.
    isNotNull();

    // overrides the default error message with a more explicit one
    String assertjErrorMessage =
        "\nExpecting color of:\n  <%s>\nto be:\n  <%s>\nbut was:\n  <%s>";

    // null safe check
    Color actualColor = actual.getColor();
    if (!Objects.areEqual(actualColor, color)) {
      failWithMessage(assertjErrorMessage, actual, color, actualColor);
    }

    // return the current assertion for method chaining
    return this;
  }
}
```

Custom assertions

- Example:

```
public class CustomAssertions {

 public static FruitAssert assertThat(Fruit actual) {
    return new FruitAssert(actual);
  }

}
```

```
assertThat(fruit).hasColor(Color.RED);
```

Assertions - Maven

- May generate assertions thanks to maven plugin.

```
<plugin>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-assertions-generator-maven-plugin</artifactId>
    <version>${assertj-assertions-generator-maven-plugin.version}</version>
</plugin>
```

- Use assertj:generate-assertions maven goal

```
mvn assertj:generate-assertions
```

Assertions - Maven

```
<plugin>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-assertions-generator-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>assertj-custom</id>
      <phase>generate-test-sources</phase>
      <goals><goal>generate-assertions</goal></goals>
    </execution>
  </executions>
  <configuration>
    <!-- target specific package -->
    <packages><param>com.zenika.project.pojo</param></packages>

    <!-- target specific class -->
     <classes><param>com.zenika.project.pojo.SimplePojo</param></classes>

    <!-- select which assertions entry point classes to generate -->
    <generateAssertions>true</generateAssertions>
    <generateBddAssertions>true</generateBddAssertions>
    <generateSoftAssertions>true</generateSoftAssertions>
    <generateJUnitSoftAssertions>true</generateJUnitSoftAssertions>
  </configuration>
</plugin>
```

Test doubles in JAVA

Mockito - Introduction

- Mock Framework open-source
  - http://mockito.org/
- Helps to configure test doubles (stub, mock ou spy)
  - Very useful for unit tests.

Mockito - @Mock and @InjectMocks annotations

- @Mock annotation indicate that an object must be mocked
  - Possible to use on classes or interfaces.
- @InjectMocks annotation indicate that an object must be instanciated
  - All mocks will be injected through constructor, setters or reflection.

```
public class UserServiceTest {

   @Mock
   private UserDao userDao;

   @InjectMocks
   private UserServiceImpl userService;

   // ...
}
```

Mockito - Mocks initialization

- Using MockitoJUnitRunner JUnit runner
  - Mocks will be initialized before each tests method.

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UserDao userDao;

    @InjectMocks
    private UserServiceImpl userService;

    // ...
}
```

Mockito - Mocks initialization

- Cannot use multiple runners.
- Use initMocks static method to instantiate the object to test and do injections.
  - Most of the time called in @Before method.

```
public class UserServiceTest {
   @Mock
   private UserDao userDao;

   @InjectMocks
   private UserServiceImpl userService;

   @Before
   public void setUp() {
      initMocks(this);
   }

   // ...
}
```

Mockito - Mocks initialization

- Cannot use multiple runners.
- JUnit MockitoRule rule

```
@RunWith(FooBarJUnitRunner.class)
public class UserServiceTest {

  @Rule
  public final MockitoRule mockitoRule = MockitoJUnit.rule();

  @Mock
  private UserDao userDao;

  @InjectMocks
  private UserServiceImpl userService;

  // ...
}
```

Mockito - static method mock()

- Alternative to initialize mocks
  - Most of the time done in @Before method.
  - Need to instantiate manually the object to test.

```
public class UserServiceTest {
   private UserDao userDao;
   private UserServiceImpl userService;

   @Before
   public void setUp() {
      userDao = mock(UserDao.class);
      userService = new UserServiceImpl(userDao);
   }

   // ...
}
```

Mockito - when() ... thenXXX() methods

- Mockito helps to mock methods and
  - return value with thenReturn().
  - throw exception with thenThrow().
- This is the **given** part of unit tests.

```
@Test
public void should_return_result() {
   when(userDao.countFriends(eq(1L))).thenReturn(15);
   when(userDao.findUser(eq(12L))).thenThrow(new ResourceNotFound());

   assertThat(userService.countFriends(1L), is(15));
}
```

Mockito - verify() method

- Helps to verify that a method has been called and
  - to check the inputs.
  - to checks the number of times with times() (1 par défaut).
- This is the **then** part of unit tests.
- Don't overuse it.

```
@Test
public void should_return_result() {
   User user = aUser();
   userService.save(user);

   verify(userDao, times(1)).save(user);
}
```

Mockito - ArgumentCaptor

- Helps to capture the object instance as input of a method, just to check some fields.

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

   @Captor
   ArgumentCaptor<User> captor;

   @Test
   public void should_persist_user() {
      userService.save("Robert");
      verify(userDao).save(captor.capture());
      User savedUser = captor.getValue();
      assertThat(savedUser.login(), is("Robert"));
   }
}
```

Mockito - any() and eq() methods

- any(Class<?>) helps to indicate than the object used as input of a method doesn't matter.
- Use eq to test equality.

```
// will not work
verify(userDao).findByLoginAndDate("Robert", any(Date.class));

// will work
verify(userDao).findByLoginAndDate(eq("Robert"), any(Date.class));
when(userDao.findByLogin(any(String.class))).thenReturn(aUser());
```

Other tools

- There exists others libraries dedicated to test doubles:
  - **EasyMock** http://easymock.org
  - **PowerMock** https://github.com/jayway/powermock
  - **JMockit** http://jmockit.org
- The most popular and easiest to use is Mockito.

Tools for integration tests

DBUnit

- JUnit extension for testing database
  - http://dbunit.sourceforge.net
- Reuse database state across tests executions
  - Avoid test issues when modifying database state and make other tests failing
- Export / import data from / to XML
  - Help to check data values (less used).

DBUnit - Database Export

- To create XML to load into database:
  - with org.dbunit.dataset.xml.FlatXmlDataSet: code to write (see example).
  - with Jailer - http://jailer.sourceforge.net: IDE helps to filter graph relations and amount of data.

DatabaseExport (1/2)

```
import java.io.FileOutputStream;
import java.sql.Connection;

import org.dbunit.database.*;
import org.dbunit.dataset.*;
import org.dbunit.dataset.xml.FlatXmlDataSet;

public class DatabaseExport {
  public static void main(String[] args) throws Exception {
    // database connection
    Connection jdbcConnection = ConnectionManager.getInstance().getConnection();
    IDatabaseConnection connection = new DatabaseConnection(jdbcConnection);

    // full database export
    ITableFilter filter = new DatabaseSequenceFilter(connection);
    IDataSet fullDataSet = new FilteredDataSet(filter,
connection.createDataSet());

    FlatXmlDataSet.write(fullDataSet, new FileOutputStream("full-dataset.xml"));
  }
}
```

DatabaseExport (2/2)

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <transport ID="1" DEPARTURE_DATE="2007-01-01 15:30:00.0"
    ARRIVAL_DATE="2007-01-01 16:45:00.0" TOTAL_SEATS_NUMBER="120"
    SEATS_AVAILABLE="20" PRICE="240.0" DEPARTURE_CITY="Paris"
    ARRIVAL_CITY="Toulouse" />
  <transport ID="2" DEPARTURE_DATE="2007-03-05 12:13:00.0"
    ARRIVAL_DATE="2007-03-06 07:52:00.0" TOTAL_SEATS_NUMBER="127"
    SEATS_AVAILABLE="2" PRICE="540.0" DEPARTURE_CITY="New-York"
    ARRIVAL_CITY="Paris" />
  <reservation ID="1" RESERVATION_DATE="2006-10-17"
    TRANSPORT_ID="2"
    TRAVELER_NAME="Tiger" TRAVELER_FIRST_NAME="Scott"
    TRAVELER_MAIL="scott.tiger@nowhere.com" />
  <reservation ID="2" RESERVATION_DATE="2006-10-18"
    TRANSPORT_ID="2"
    TRAVELER_NAME="Blanc" TRAVELER_FIRST_NAME="Didier"
    TRAVELER_MAIL="didier.blanc@ailleurs.fr" />
</dataset>
```

TransportServiceImplTest (1/2)

```
public class TransportServiceImplTest {
  private TransportServiceImpl service;
  private Connection cx;

  @Before
  protected void setUp() throws Exception {
    TransportDaoImpl transportDao = new TransportDaoImpl();
    this.service = new TransportServiceImpl(transportDao);

    this.cx = ConnectionManager.getInstance().getConnection();

    // reinit database with data
    IDatabaseConnection connection = new DatabaseConnection(this.cx);

    FlatXmlDataSetBuilder builder = new FlatXmlDataSetBuilder();
    FlatXmlDataSet dataset = builder.build(new FileInputStream("full-
dataset.xml"));
    DatabaseOperation.CLEAN_INSERT.execute(connection, dataset);
    this.cx.commit();
  }

  @After
  protected void tearDown() throws Exception {
    cx.close();
  }
```

TransportServiceImplTest (2/2)

```
...
  @Test
  public void testFindAllTransports() {
    List<Transport> result = service.findAllTransports(cx);
    assertEquals(2, result.size());
  }
  @Test
  public void testGetTransportById() {
    Transport result = service.getTransportById(cx, Long.valueOf(2));
    assertNotNull(result);
    assertEquals(2, result.getId().longValue());
  }
  @Test
  public void testReserverPlace()
          throws NoSeatsAvailableException, SQLException {
    Transport transport = service.getTransportById(cx, Long.valueOf(2));
    int oldValue = transport.getAvailableSeats();
    service.book(cx, transport);
    cx.commit();
    transport = service.getTransportById(cx, new Long(2));
    int newValue = transport.getAvailableSeats();

    assertEquals(oldValue - 1, newValue);
  }
}
```

SubEtha SMTP

- Fake SMTP server (send mails).
- Test mail sending without external server.
- Test content and metadata of mails.
- https://github.com/voodoodyne/subethasmtp

Example

```
@Test
public void testSendMail() {
  Wiser smtpServer = new Wiser(port);
  smtpServer.start();

  String mailBody = "Mail content";
  mailService.sendMail("foo@bar.com", "fii@bar.com", "Title", mailBody);

  List<WiserMessage> messages = smtpServer.getMessages();
  assertEquals(1, messages.size());

  for (WiserMessage message) {
    LOG.info("Envelope Receiver: {}", wiserMessage.getEnvelopeReceiver());
    LOG.info("Envelope Sender: {}", wiserMessage.getEnvelopeSender());

    MimeMessage mimeMessage = message.getMimeMessage();
    LOG.info("Content type: {}", mimeMessage.getContentType());
    LOG.info("Subject: {}", mimeMessage.getSubject());
    assertEquals(mailBody + System.lineSeparator(), mimeMessage.getContent());
  }

  smtpServer.stop();
}
```

Other tools

- There exists a wide variety of libraries helping to test or fake systems:
  - **Spring Test DBUnit**: annotation to ease DBUnit initialization
    - https://springtestdbunit.github.io/spring-test-dbunit/
  - **Spring Test**: mockMvc, Spring profiles, WebServices mock, ...
    - http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#testing
  - **Arquillian**: faking EJB container
    - http://arquillian.org
  - **WireMock**: faking Web Services
    - http://wiremock.org
  - **AssertJ-DB**, **DBSetup**, ...

Cucumber JVM

Cucumber

- Test framework to write readable test using human language.
- Follow the movement of ATDD, BDD, executable specifications, etc.
- Example:

```
Feature: Dates with different date formats
  This feature shows you can have different date formats, as long as you annotate
the
  corresponding step definition method accordingly.

  Scenario: Determine past date
    Given today is 2011-01-20
    When I ask if Jan 19, 2011 is in the past
    Then the result should be yes
```

Cucumber & BDD: Behavior Driven Development

- Objective: Write story into human readable and executable specification.
- Story

```
As financial director
I had liked to filter company results by month and activity area
Thus I can control more easily company financial results
```

- Feature

```
Feature: To control company financial results
         a user
         must be able to visualize all results

  Scenario: Display results by month
    Given I am logged to the system
    When I select 2015 as a year
    And I select september as a month
    Then the system display 456 785 € for Acme company under activity hydraulic
maintenance
```

Cucumber & Specification by Example

- Often specifications are not clear or enough accurate.
- A way to specify each stories with concrete examples.

```
Scenario Outline: Display results by month
  Given I am logged to the system
  When I select <year> as a year
  And I select <month> as a month
  Then the system display <result> €

Examples: results
   | year   | month     | result    |
   | 2015   | september | 456 785   |
   | 2015   | may       |  52 666   |
```

Structure of tests 1/4

- To create a test you need 3 items:
  - feature: text file (ie. displayResults.feature).
  - step: simple java class (ie. DisplayResultsStep.java).
  - runner: one JUnit runner per class (ie. RunCucumberTest.java).
- example of files structure

```
test
├── java
│   └── functional
│       ├── display
│       │   └── DisplayResultsStep.java
│       └──RunCucumberTest.java
└── resources
    └── functional
        └── displayResults.feature
```

Structure of tests 2/4: Feature file

- File displayResults.feature.
- A simple text file with .feature extension.

```
Feature: To control company financial results
         a user
         must be able to visualize all results

  Scenario: Display results by month
    Given I am logged to the system as john.doe
    When I select 2015 as a year
    And I select september as a month
    Then the system display 456 785 € for Acme company under activity hydraulic
maintenance
```

Structure of tests 3/4: Step file

- Using of regular expressions to extract data from feature.

```
@Given("^I am logged to the system as (.+)$")
public void login(String user) {
  application.login(user);
}

@When("^When I select (\\d+) as a year$")
public void select_year(int year) {
  application.selectYear(year);
}

@Then("^the system display (\\d+) € for (.+) company under activity (.+)$")
public void my_change_should_be_(double result, String company, String activity)
{
  application.apply();
  assertThat(application.getResult()).isEqualTo(result);
}
```

Structure of tests 4/4: Runner file

- Simple java class acting as a glu between Feature and JUnit.

```
@RunWith(Cucumber.class)
@CucumberOptions(format={"pretty", "html:target/cucumber/html"})
public class RunCucumberTest {

}
```

Passing variables from feature to step: basic (1/3)

- Catch variables thanks to regular expressions.
- Feature:

```
I select 2015 as a year and september as a month
```

- Step:

```
@Given("^I select (\\d+) as a year and (.+) as a month")
public void select(int year, String month) {}
```

Passing variables from feature to step: list (2/3)

- Feature:

```
Given the following animals: cow, horse, sheep
```

- Step:

```
@Given("^Given the following animals: (.*)$")
public void animals(List<String> animals) {

}
```

- Feature:

```
Given the following animals:
|cow|
|horse|
|sheep|
```

- Step:

```
@Given("^Given the following animals: (.*)$")
public void animals(List<String> animals) {

}
```

Passing variables from feature to step: Datatable (2/3)

- Feature:

```
The following names exist:
| name   | email                | twitter          |
| Aslak  | aslak@cucumber.io    | @aslak_hellesoy  |
| Julien | julien@cucumber.io   | @jbpros          |
| Matt   | matt@cucumber.io     | @mattwynne       |
```

- Step:

```
@Given("^The following names exist:$")
public void users(DataTable users) { }
```

Example

- Feature:

```
Scenario Outline: Feeding a cow
  Given the cow weight <weight> kg
  When we compute its needs
  Then it should be <energy> MJ for energy
  And <proteins> kg for proteins

  Examples:
    | weight| energy   | proteins   |
    |   450 |  26500   |        215 |
    |   600 |  37000   |        305 |
```

- Step

```
@Given("^the cow weight ([^\"]*) kg")
public void given_cow_weight(int weight) { }

@When("^we compute its needs$")
public void when_cow_compute_needs() {}

@Then("^it should be ([^\"]*) MJ for energy$")
public void then_vache_check_energy(int energy$) {}

@Then("^([^\"]*) kg for proteins$")
public void then_vache_check_protein(int proteins) {}
```

Selenium

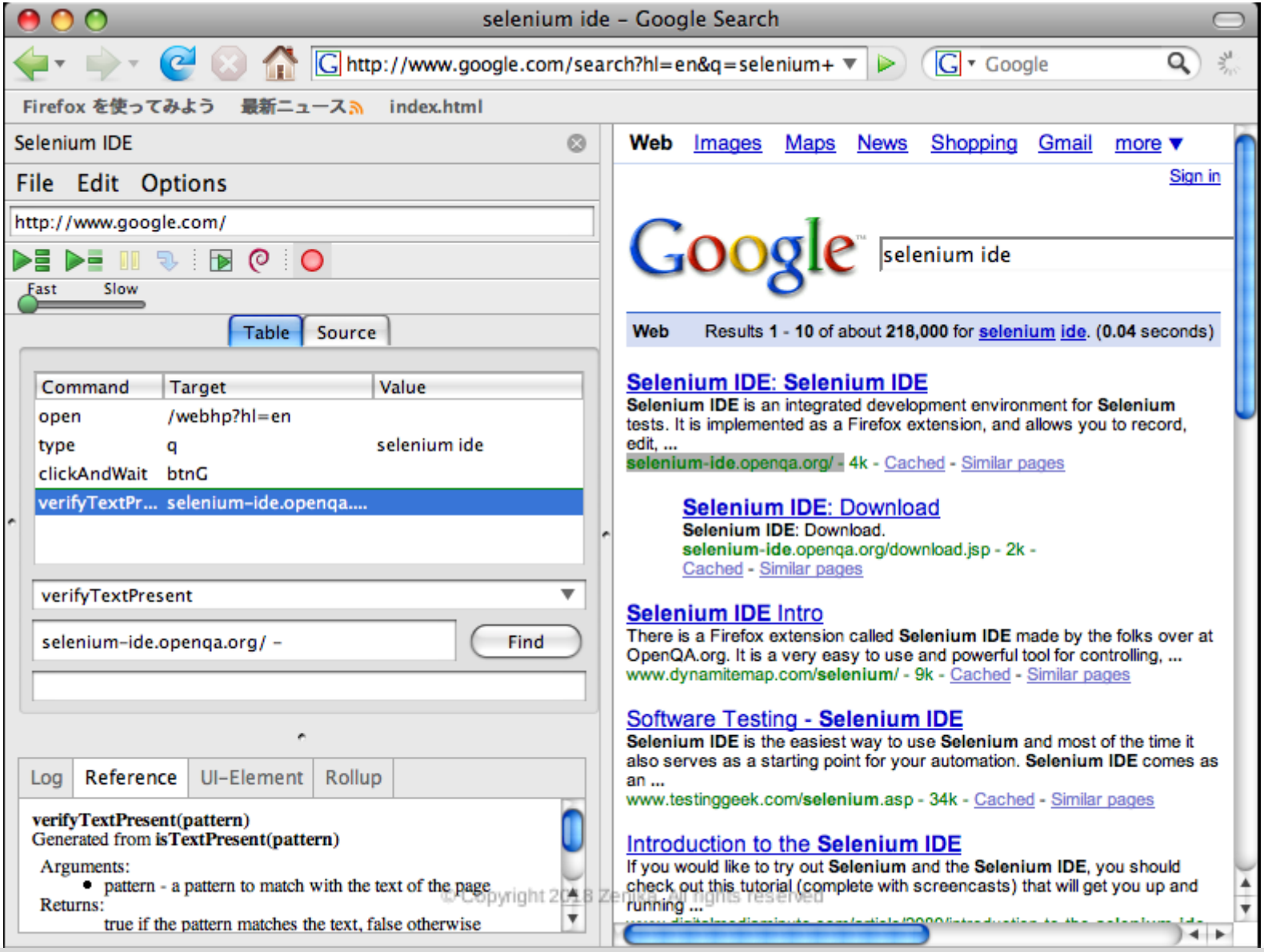
Introduction to functional testing

- The goal is not to test each component
  - From technical API to functional use cases.
- Test from the point of view of users
  - Usually tests done by QA team.
  - Often done « manually » by using the application.
- Tools depends on GUI, often one specific per GUI.

Selenium (1/2)

- Tools to test Web GUI
  - http://selenium.openqa.org/
- Multi-browsers.
- Compatible with JUnit.
- Recording tool available on Firefox.

Selenium (2/2)



Load testing

Load testing (1/3)

- Goals:
  - Test application behaviour when heavily used (dozens, hundreds or thousands of users): performance evaluation.
  - Identify eventual issues
    - Bottlenecks.
    - Misconfigurations.
    - Critical resources.
    - etc.

Load testing (2/3)

Beware

- The goal is not to check the application works well
  - This is the goal of unit tests, integration tests and functional tests.

Load testing (3/3)

- Must not be written before ending development
  - Unlike other tests.
- Before developments
  - Maybe used on POC to check performances.
- One kind of tool is adapted to one kind of usage
  - Some are multipurpose.

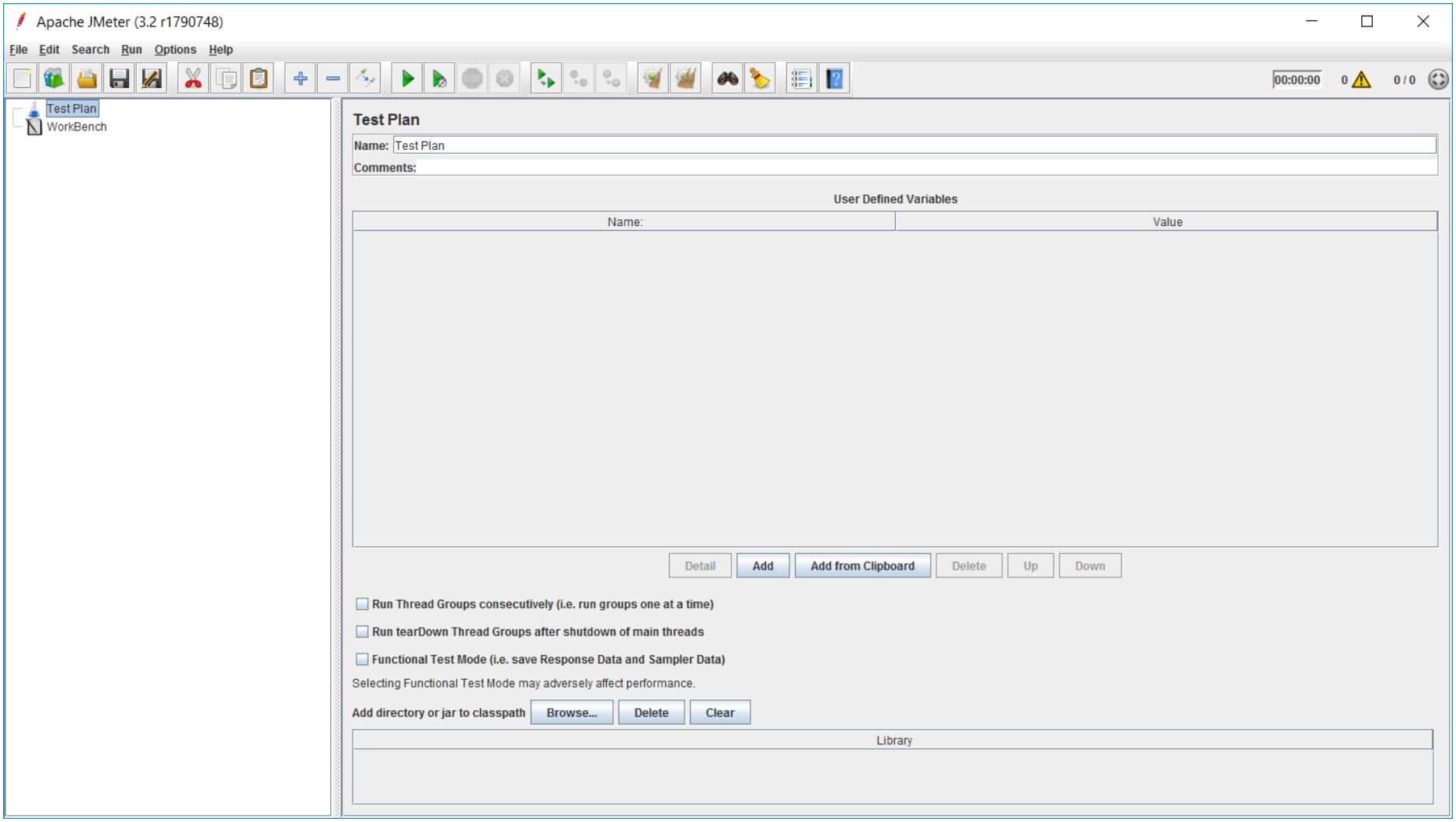
JMeter (1/2)

- A Java application dedicated to load testing
  - http://jmeter.apache.org/
  - At first, only for web applications (HTTP & FTP), bot new abilities now (JDBC, LDAP, JMS, etc.).

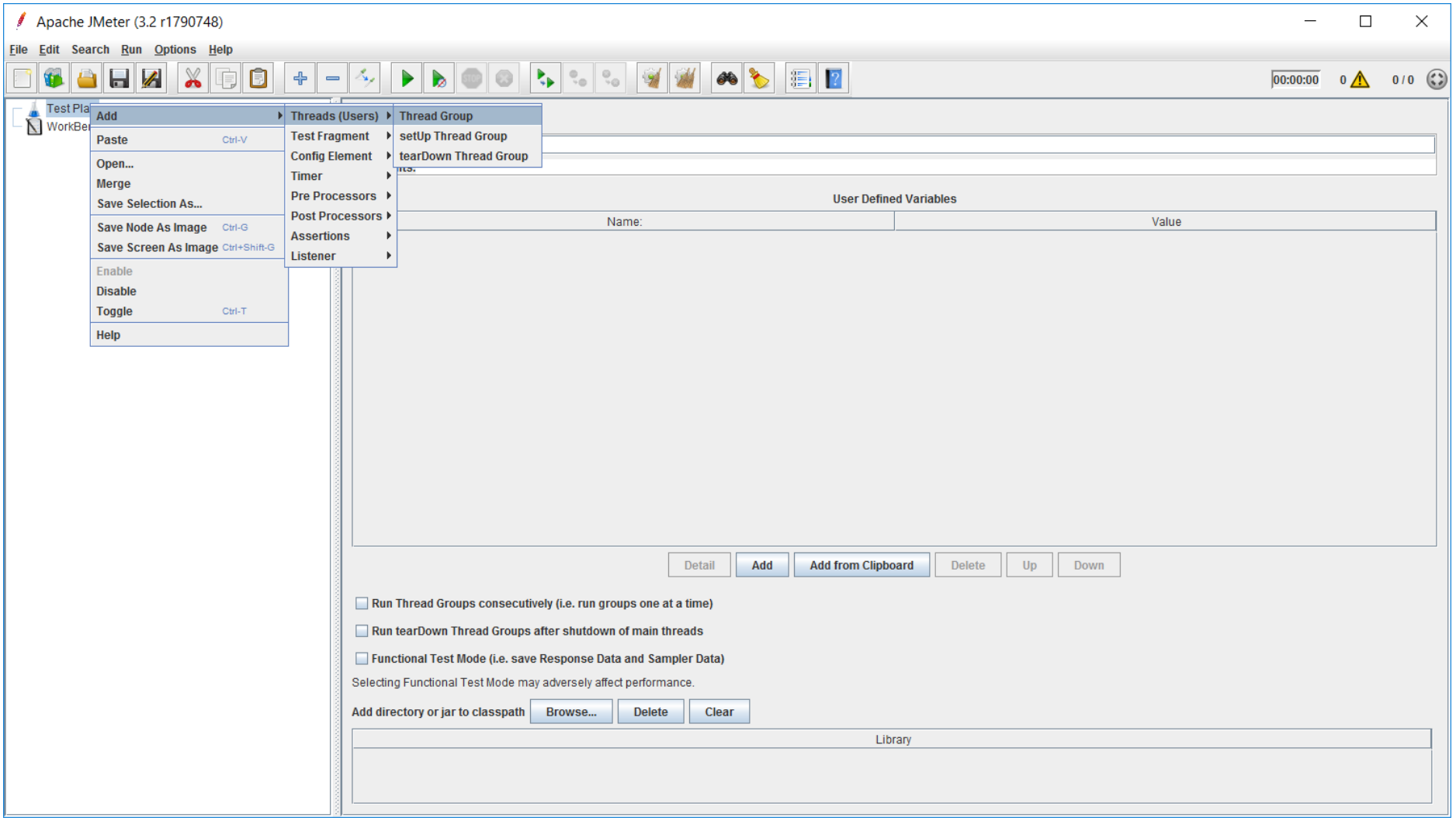
JMeter (2/2)

- Use Swing as GUI
  - Easy launching and data handling through files.
- Principles
  - Save bunch of requests.
  - Redo configuration (how many, delay, etc.).
  - Results with tools.

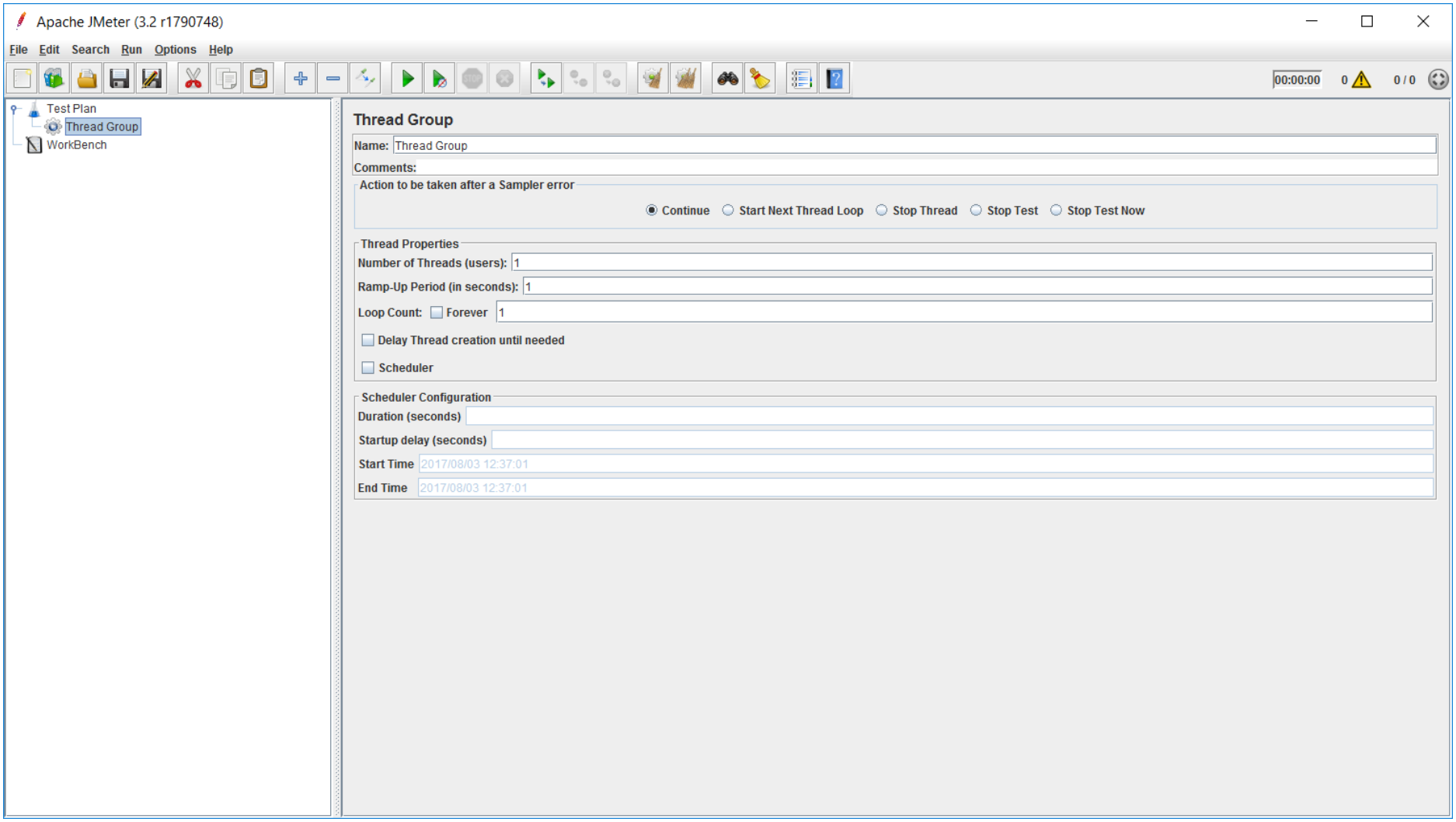
Example - Interface



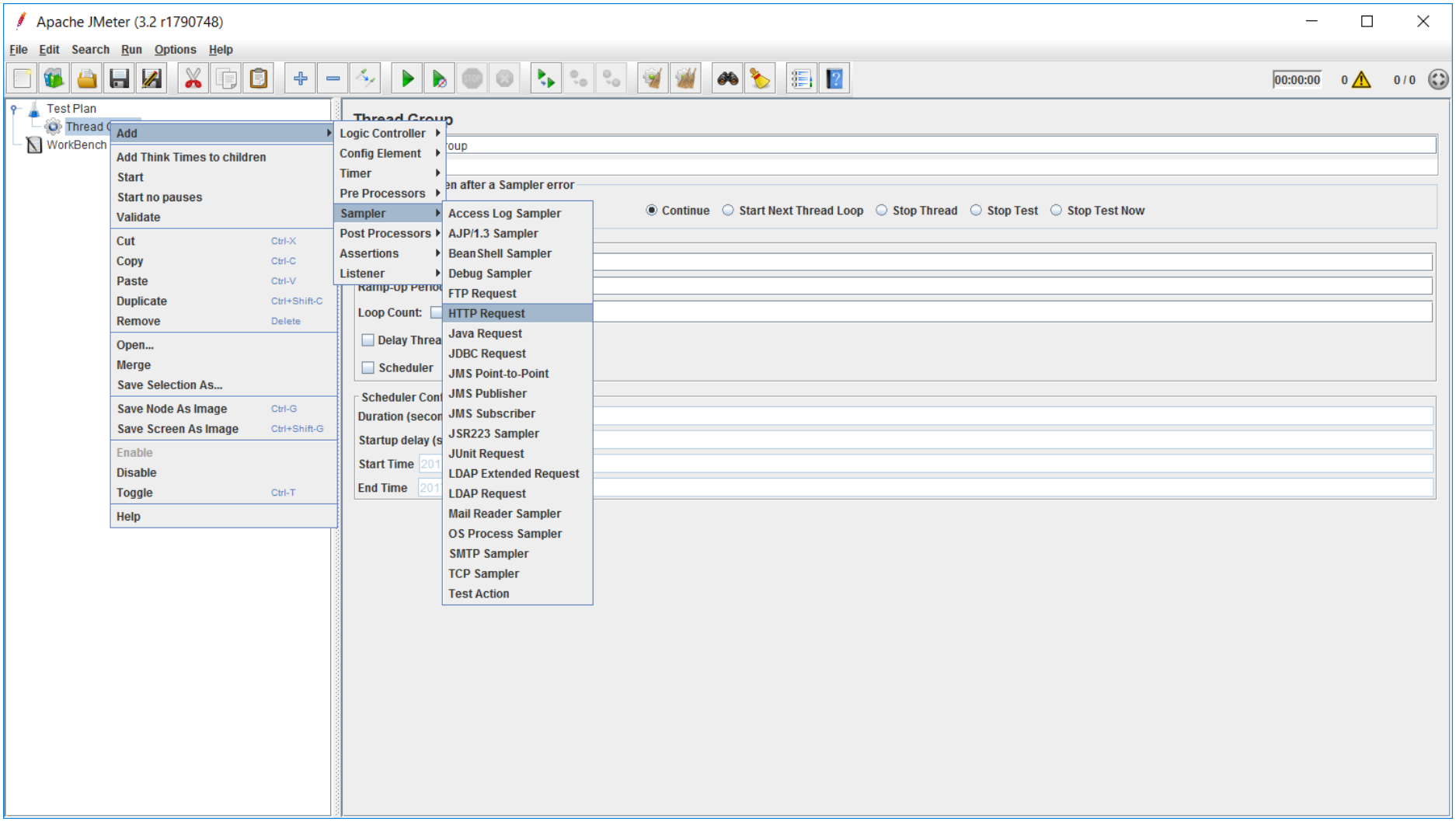
Example - Add a use case



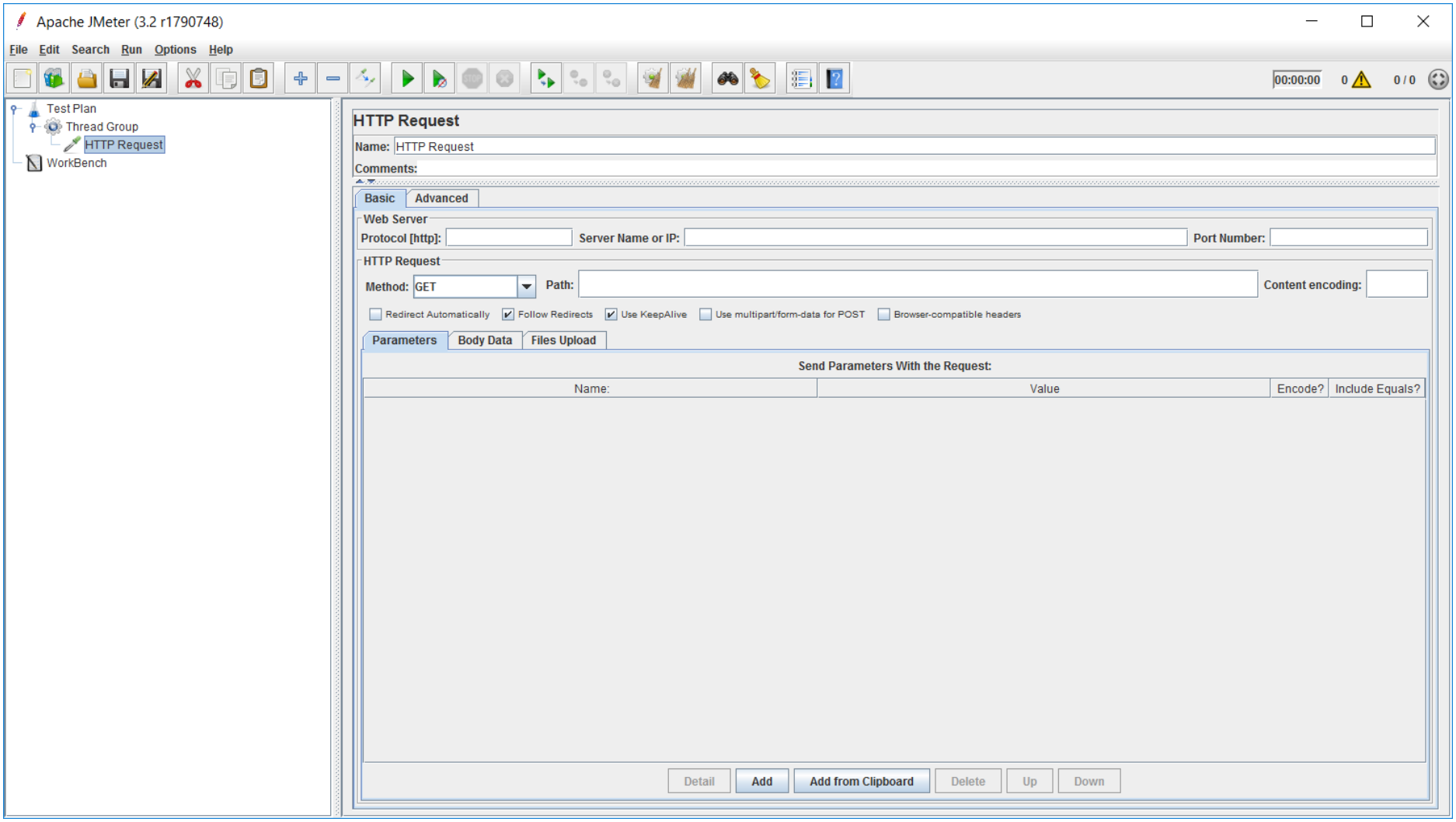
Example - Scenario



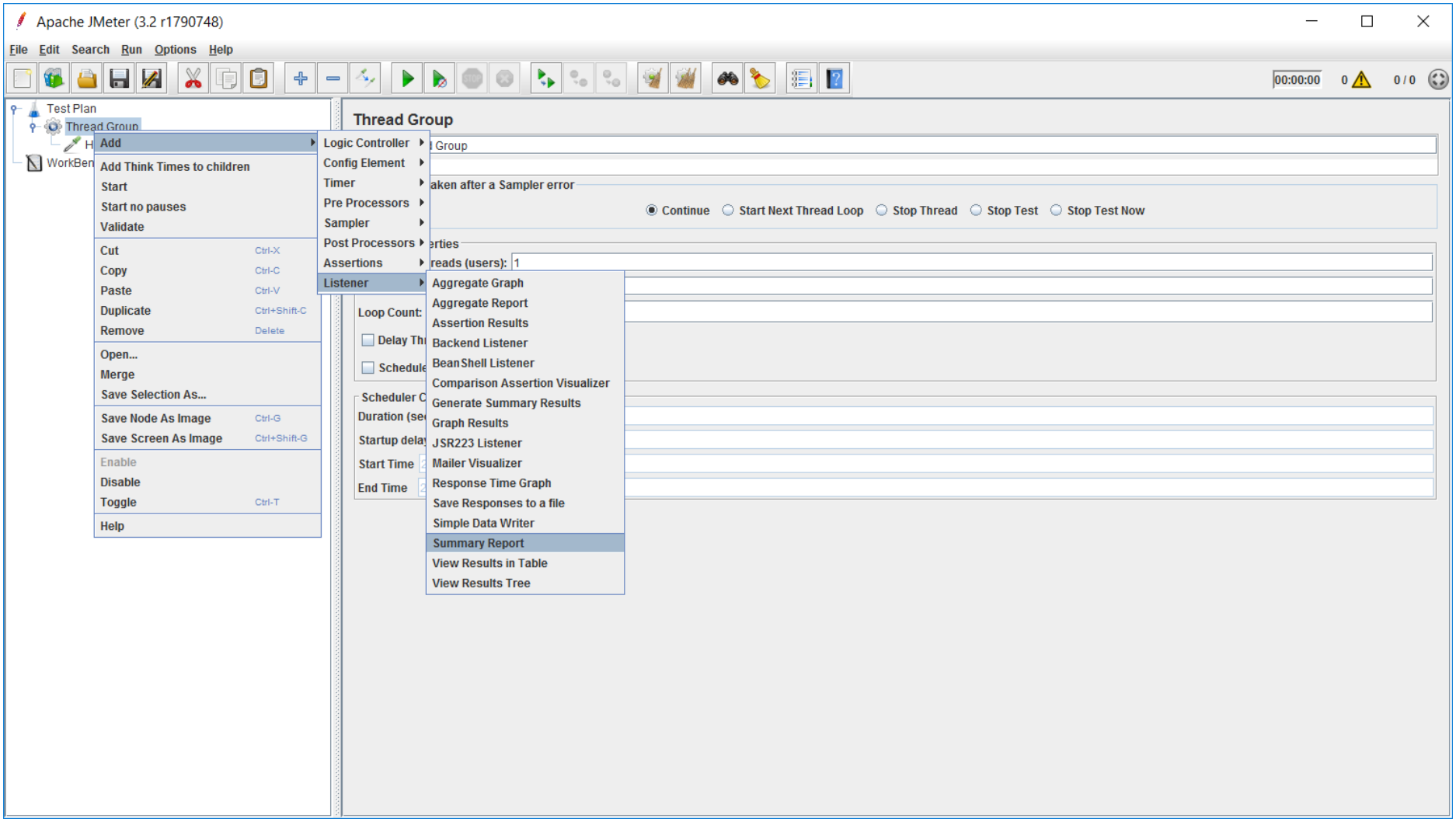
Example - Add HTTP request



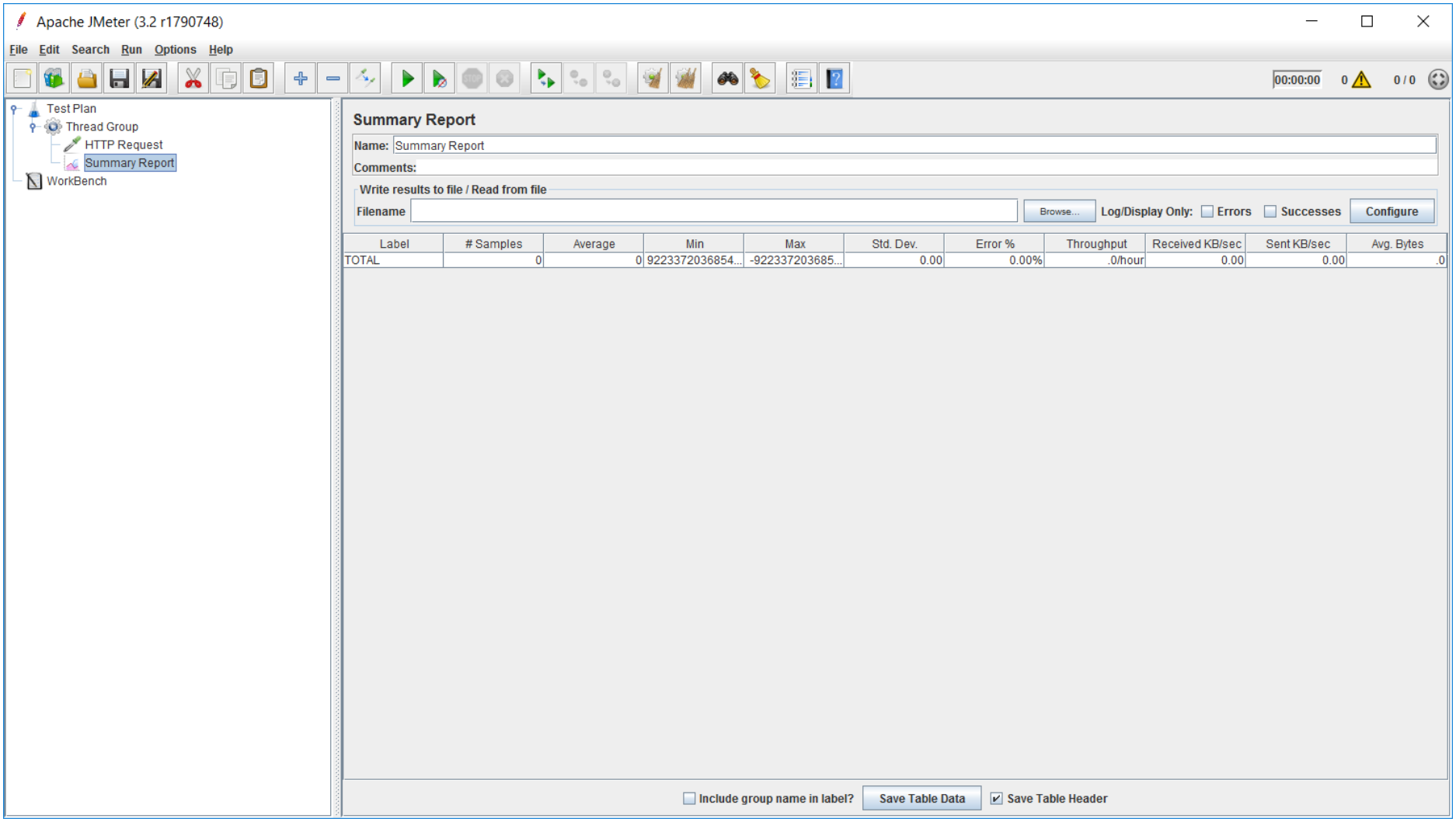
Example - HTTP request



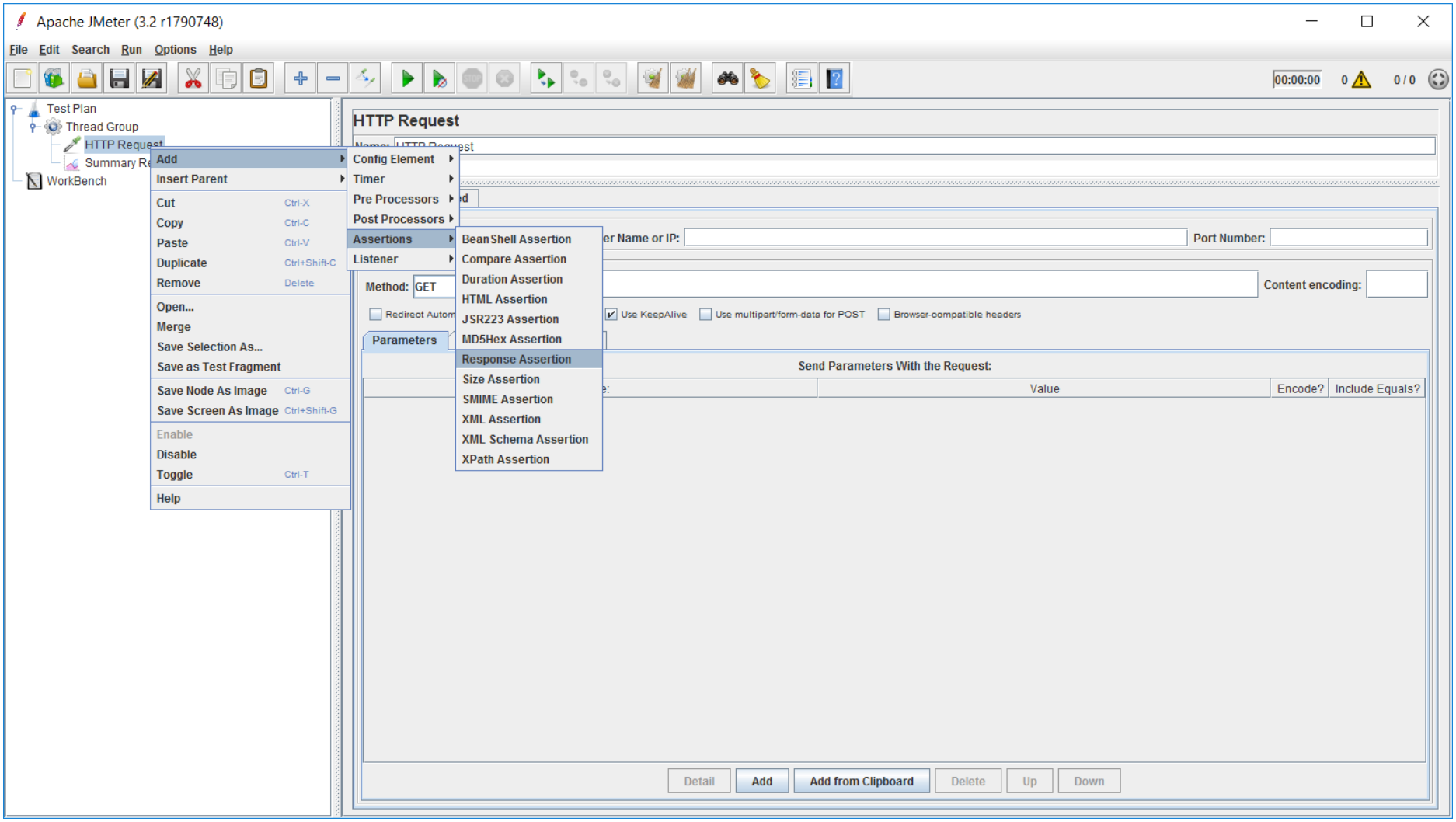
Example - Add report



Example - Aggregated report



Example - Add assertion



Example - Assertion