Software Testing

An Introduction with Examples

Paulo Maio
School of Engineering, Polytechnic of Porto
Portugal
pam@isep.ipp.pt

Topics

- Testing: Purpose and Benefits
- Verification & Validation
- Testing Methods
- Level of Testing
- Principles and Good Practices
- Unit and Integration Tests
  - Using Mock Objects
  - Example using Google Testing Framework
- Testing Myths and Facts

What is SW Testing?

- It is the Software Development Activity through which is **verified** and **validated** that the software product does what it is supposed to do, i.e., match both the specified and the expected requirements. It also tries to ensure the software product is **defect-free**, secure, reliable and has any other desired quality (e.g., performance)

- Activity devoted to control (and ensure) the quality of the software product being developed.

- It involves execution of software components using manual or automated tools to evaluate some properties of interest (e.g., a functional requirement, non-functional requirement, a given business rule).

As so, …

- Having written requirements is essential

- Requirements need to be translated to test cases / test scenarios
  - Some tests can be written as software programs and further executed by testing software (e.g., Google Testing Framework)

  → Automated Testing

  - Other tests are meant to be executed by humans (e.g.:, system' users)

  → Manual Testing

- Automated testing is preferable to manual testing

Some up-front testing terminology

- **Defects:** are issues that do not meet the acceptance criteria. It might not be a bug, but rather a design or content issue that does not match the requirements set forth by the client.
- **Quality:** measures the design of the software, how well the actual application conforms to that software, and how the software executes its purpose.
- **Test case:** are a set of structured steps or script that tell the tester exactly how the feature of function of the system should work. It should usually contain expected results and the conditions associated with these.
- **Test environment:** is the technical environment in which the software tester will be conducting and running the tests.
- **Test suites:** are the test cases that are compiled together for the system testing.

https://www.workwithloop.com/blog/the-most-common-software-testing-terminology

SW Testing – Major Benefits

- **Product Quality**: ensures a quality SW product is delivered to customers
- **Customer Satisfaction**: as any product, a SW product should satisfy their customers by meeting their expectations and needs (i.e., requirements)
- **Cost-Effective**: on-time testing helps saving money on the long term as fixing defects on an earlier development stage is cost-less
- **Security**: as long as customers are looking for trusted products, testing helps in removing risks and problems earlier

Verification vs. Validation (V&V)

Criteria	Verification	Validation
Definition	The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.	The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements.
Objective	To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.	To ensure that the product actually meets the user's needs and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.
Question	Are we building the product *right*?	Are we building the *right* product?
Evaluation Items	Plans, Requirement Specs, Design Specs, Code, Test Cases	The actual product/software.

Testing Methods

- White (a.k.a. Transparent) Box Testing
  - It is a Structural Testing
  - The internal structure/design/implementation of the item being tested **is known** to the tester

- Black (a.k.a. Opaque) Box Testing:
  - It is Behavioral Testing
  - The internal structure/design/implementation of the item being tested **is not known** to the tester

- Gray (a.k.a. Semi-transparent) Box Testing
  - The internal structure is partially known only

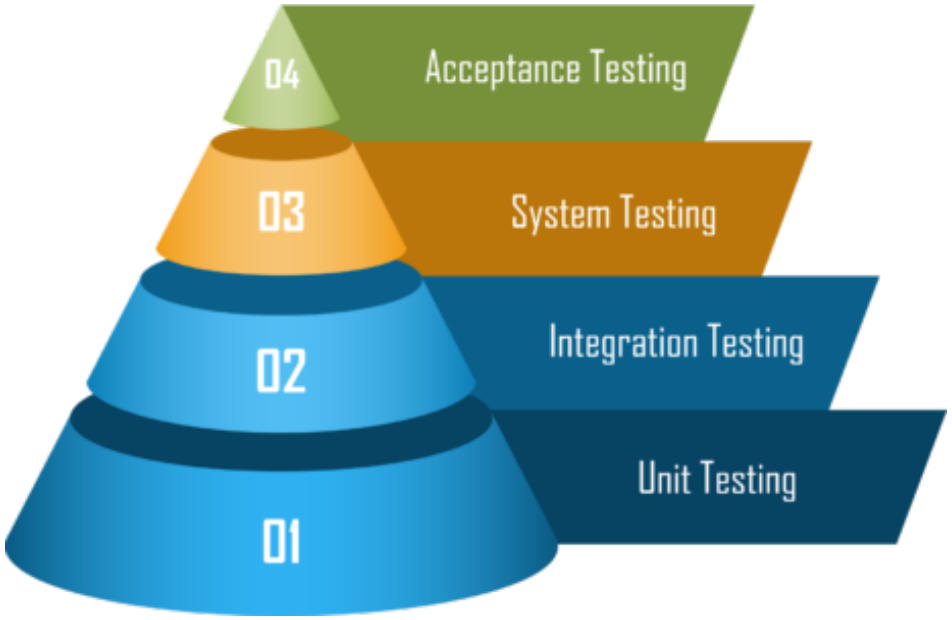
Testing Categories and Types

Category	Test Type/Method/Strategy (e.g.:)
Functional Testing	• Unit (or Component) Testing • Integration Testing • User Acceptance Testing • Localization and Globalization Testing • Interoperability Testing • …
Non-Functional (or Quality) Testing	• Usability Testing • Load Testing • Performance Testing • Scalability Testing • …
Maintenance Testing	• Regression Testing • Maintenance Testing

- An ever-growing list of test types, methods and strategies is available here

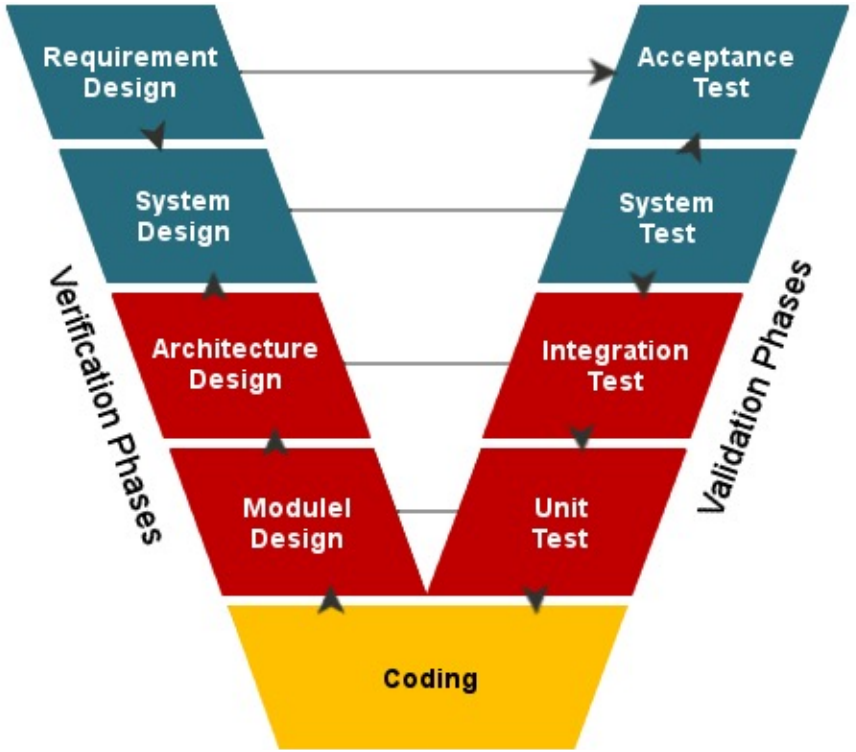
Levels of Testing

- Acceptance Testing
  - Alpha Testing
  - Beta Testing
- System Testing
  - Regression Testing
- Integration Testing
- Unit Testing

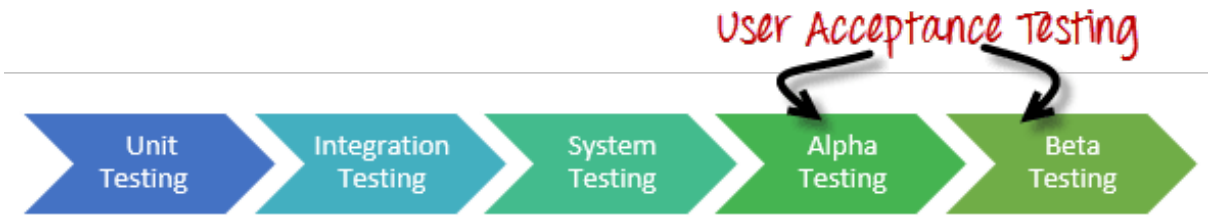


https://www.edureka.co/blog/software-testing-levels/

Testing Levels in Perspective



http://www.professionalqa.com/v-model



https://www.guru99.com/alpha-beta-testing-demystified.htm

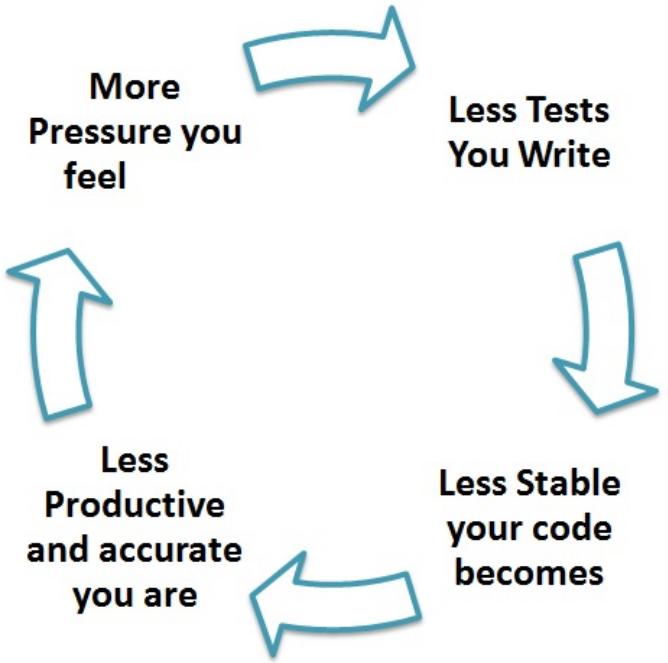
Unit Testing

- Aims to verify that each unit (or module) of the SW performs as expected
- Done by the developers during the construction activity
- When necessary, units are isolated to verify its correctness
- Follows a white box testing
  - Algorithms and logic
  - Data structures
    - Global
    - Local
  - Interfaces
  - Independent paths
  - Boundary conditions
  - Error handling

Module under Test (MUT).
For example:
- Method
- Class
- Component
- Layer
- Application

Unit Testing Myth → A Vicious Cycle

Myth: Unit test requires time, and I am always overscheduled. My code is rock solid! I do not need unit tests.



The truth: Unit testing helps saving time and money.

https://www.guru99.com/unit-testing-guide.html

Why Unit Testing Is Necessary

- Increases developers understanding of the testing code base
  - Enables them to make changes quickly, even in other developers' code
  - Increases their confidence that no adverse effects on existing features were introduced
- Helps fixing defects early in the development cycle
- Early fixing defects is cheaper and, therefore, saves money
- Good unit tests serve as project documentation
- Facilitates code re-use. Migrate both code and tests to your new project. Tweak the code until the tests run again.

Integration Testing

- Aims to verify the interaction between two or more software modules when they are integrated
- Individual modules are combined and tested as a group
- Focus on checking data communication amongst integrated modules

Module Integration Approaches

Two common module integration approaches:

- Big Bang Integration Testing
  - All modules are integrated together at once and then tested as a single unit
  - Suitable for (very) small systems only
- Incremental Integration Testing
  - Two logically-related modules are integrated and tested together. Then other related module is integrated and tested incrementally until all related modules have been integrated and tested successfully.
  - Modules' integration might follow a:
    - Top-down approach
    - Bottom-up approach
    - Sandwich approach, i.e., a combination of top-down and bottom-up approach

Why Integration Testing Is Necessary

- One module can have an adverse effect on another
  - Subfunctions, when combined, may not produce the desired major function
  - Individually acceptable imprecision in calculations may be magnified to unacceptable levels
- Interfacing errors not detected in unit testing may appear
- Timing problems (in real-time systems) are not detectable by unit testing
- Resource contention problems are not detectable by unit testing

System Testing

- Aims to validate the complete and fully integrated software product
  - A.k.a. End-to-end Testing
- Determines if the software meets all the requirements defined in the software requirements specification document
- Usually takes a black box approach

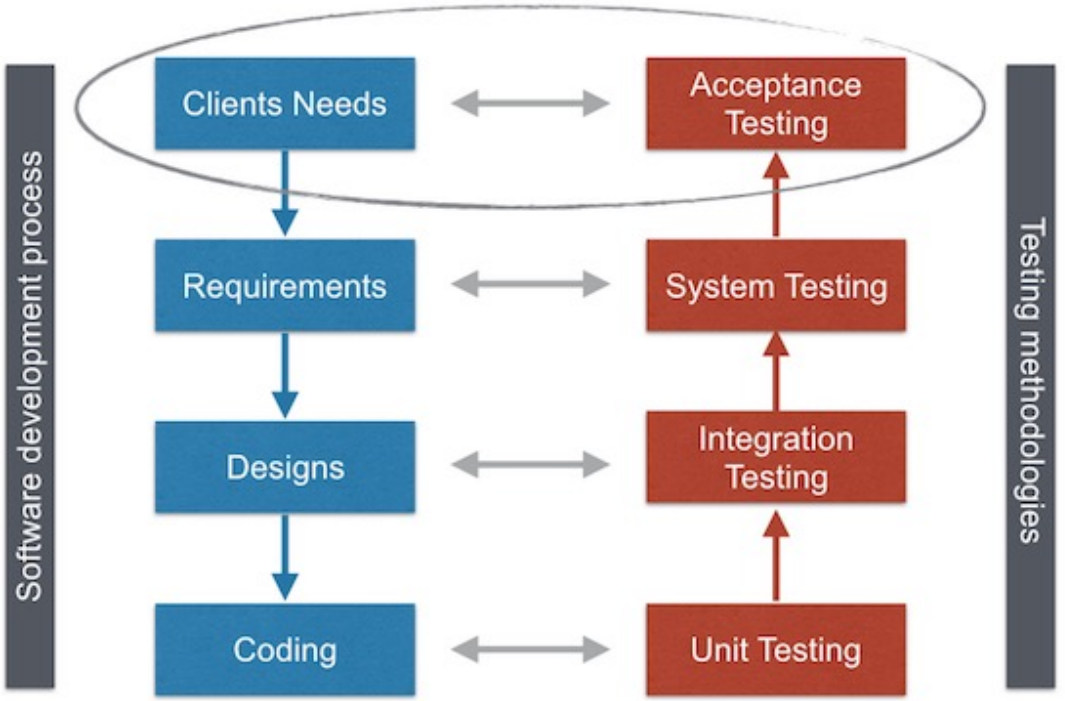
- Actually, system' testing is a series of different tests whose sole purpose is to exercise the full SW system under test

Regression Testing

- Aims to confirm that changes/updates to the SW product has not adversely affected previously validated/verified existing features
- It is required/useful when, for instance, there is a:
  - Change in requirements and code is modified according to the requirement
  - A new feature is added
  - A defect is fixed
- To ensure that no degradation of baseline functionality has occurred with modifications
- It is a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine
- Automated tests are regression tests

Acceptance Testing

- Performed by the SW customers (i.e.:, system' end users) to accept the software product before using it in a production environment



https://usersnap.com/blog/types-user-acceptance-tests-frameworks/

Acceptance Testing – Alpha Testing

Usually:

- It is carried out in a lab environment, near the end of the software development
- Testers are internal employees of the organization and preferable not directly related to the development team
- Testers are simulating real users by using black box and white box techniques
- Aims to identify all possible issues/bugs before releasing the product

Acceptance Testing – Beta Testing

Usually:

- A SW product beta version is released to a limited number of “real” system’ users to obtain feedback on the product quality in a "real environment”
- Reduces product failure risks and provides increased quality of the product through (direct) customer validation

- Direct feedback from customers is a seen as the major advantage of Beta Testing

Seven Testing Principles

- Exhaustive testing is not possible
  - Focus on modules/features having a high-risk assessment
- Defect Clustering
  - Pareto Principle: ~80% of the problems are found in ~20% of the modules
- Pesticide Paradox
  - As, throw time, a pesticide become ineffective in insects, the same set of repetitive tests is useless to find new defects → Regularly review and add new test cases
- Testing shows a presence of defects
  - Reduces the probability of undiscovered defects but not their absence
- Absence of Error – fallacy
  - Absence of error does not mean SW is usable and useful. It still needs to meet users' needs
- Early Testing
  - As soon as requirements are defined, testing can start. It is cheaper fix defects detected earlier
- Testing is context dependent
  - Distinct application types implies adopting different testing approaches, techniques and types

https://www.guru99.com/software-testing-seven-principles.html

Good Testing Practices

- Ensure that testability is a key objective in your software design
- Write test cases for valid as well as invalid input conditions
- A good test case is one that has a high probability of detecting an undiscovered defect, not one that shows that the program works correctly
- A necessary part of every test case is a description of the expected result
- It is impossible to test your own program. Assign your best people to testing
- Thoroughly inspect the results of each test
- Avoid nonreproducible or on-the-fly testing

Testing Methods

Characterization

Black Box vs. White Box Testing (1/4)

Parameter	Black Box Testing	White Box Testing
Definition	Used to test the software without the knowledge of the internal structure of program or application.	Internal structure is known to the tester.
Alias	It is also known as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge	Not needed.	Required.

https://www.guru99.com/back-box-vs-white-box-testing.html

Black Box vs. White Box Testing (2/4)

Parameter	Black Box Testing	White Box Testing
Implementation knowledge	Implementation knowledge is not required.	Complete understanding needs to implement WhiteBox testing.
Automation	Test and programmer are dependent on each other, so it is tough to automate.	Easy to automate.
Objective	The main objective of this testing is to check the functionality of the SUT.	The main objective is to check the quality of the code.
Basis for test cases	Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by tester and developers.
Granularity	Granularity is low.	Granularity is high.

https://www.guru99.com/back-box-vs-white-box-testing.html

Black Box vs. White Box Testing (3/4)

Parameter	Black Box Testing	White Box Testing
Testing method	It is based on trial and error method.	Data domain and internal boundaries can be tested.
Time	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Algorithm test	Not the best method for algorithm testing.	Best suited for algorithm testing.
Code Access	Code access is not required.	Requires code access. Thereby, the code could be stolen if testing is outsourced.
Benefit	Well suited and efficient for large code segments.	It allows removing the extra lines of code, which can bring in hidden defects.

https://www.guru99.com/back-box-vs-white-box-testing.html

Black Box vs. White Box Testing (4/4)

Parameter	Black Box Testing	White Box Testing
Skill level	Low skilled testers can test the application with no knowledge of the implementation of programming language or operating system.	Need an expert tester with vast experience to perform transparent box testing.
Techniques	Equivalence partitioning divides input values into valid and invalid partitions and selecting corresponding values from each partition of the test data. Boundary value analysis checks boundaries for input values.	Statement Coverage validates whether every line of the code is executed at least once. Branch coverage validates whether each branch is executed at least once Path coverage method tests all the paths of the program.
Drawbacks	Update to automation test script is essential if you to modify application frequently.	Automated test cases can become useless if the code base is rapidly changing.

https://www.guru99.com/back-box-vs-white-box-testing.html

Automated vs. Manual Testing

A Comparison

Automated vs. Manual Testing (1/4)

Parameter	Automated Testing	Manual Testing
Definition	Uses automation tools to execute test cases.	Test cases are executed by a human tester and software.
Processing time	Is significantly faster than a manual approach.	Is time-consuming and takes up human resources.
Exploratory Testing	Does not allow random testing.	Is possible.
Initial investment	The initial investment is higher. Though the ROI is better in the long run.	The initial investment is comparatively lower. ROI is also lower in the long run.
Investment	Investment is required for testing tools as well as automation engineers	Investment is needed for human resources.
Cost-effective	Not cost effective for low volume.	Not cost effective for high volume.
Reliability	Is a reliable method, as it is performed by tools and scripts. There is no testing Fatigue.	Is not as accurate because of the possibility of the human errors.

https://www.guru99.com/difference-automated-vs-manual-testing.html

Automated vs. Manual Testing (2/4)

Parameter	Automated Testing	Manual Testing
UI Change	For even a trivial change in the UI of the AUT, Automated Test Scripts need to be modified to work as expected.	Small changes like change in id, class, etc. of a button wouldn't thwart execution of a manual tester.
Test Report Visibility	All stakeholders can login into the automation system and check test execution results.	Are usually recorded in an Excel or Word, and test results are not readily/ readily available.
Human observation	Does not involve human consideration. So it can never give assurance of user-friendliness and positive customer experience.	Allows human observation, which may be useful to offer user-friendly system.
Performance Testing	Load Testing, Stress Testing, Spike Testing, etc. have to be tested by an automation tool compulsorily.	Performance Testing is not feasible manually.
Programming knowledge	It is a must in automation testing.	No need for programming.

https://www.guru99.com/difference-automated-vs-manual-testing.html

Automated vs. Manual Testing (3/4)

Parameter	Automated Testing	Manual Testing
Parallel Execution	Can be executed on different operating platforms in parallel and reduce test execution time.	Can be executed in parallel but would need to increase your human resource which is expensive.
Batch testing	You can Batch multiple Test Scripts for nightly execution.	Cannot be batched.
Set up	Requires less complex test execution set up.	Needs have a more straightforward test execution setup
Engagement	Done by tools. Its accurate and never gets bored!	Repetitive Manual Test Execution can get boring and error-prone.
Ideal approach	Is useful when frequently executing the same set of test cases	Proves useful when the test case only needs to run once or twice.
Deadlines	Have zero risks of missing out a pre-decided test.	Has a higher risk of missing out the pre-decided test deadline.
Build Verification Testing (BVT)	Is useful for BVT	Executing the BVT is very difficult and time-consuming in manual testing.

Automated vs. Manual Testing (4/4)

Parameter	Automated Testing	Manual Testing
Framework	Uses frameworks like Data Drive, Keyword, Hybrid to accelerate the automation process.	Does not use frameworks but may use guidelines, checklists, stringent processes to draft certain test cases.
Documentation	Acts as a document provides training value especially for automated unit test cases. A new developer can look into a unit test cases and understand the code base quickly.	Manual Test cases provide no training value
Test Design	Automated Unit Tests enforce/drive Test Driven Development Design.	Do not drive design into the coding process
Devops	Help in BVT and are an integral part of DevOps Cycle	Defeats the automated build principle of DevOps
When to Use?	Is suited for Regression Testing, Performance Testing, Load Testing or highly repeatable functional test cases.	Is suitable for Exploratory, Usability and Adhoc Testing. It should also be used where the AUT changes frequently.

https://www.guru99.com/difference-automated-vs-manual-testing.html

Unit/Integration Testing and Mock Objects

Example using Google Testing Framework + DemoTasks Project

Testing Unit – StringUtils

```
// StringUtils.h

class StringUtils {
public:
   static wstring toUpperCase(const wstring &value);
   static wstring toLowerCase(const wstring &value);
   static wstring leftTrim(const wstring &value);
   static wstring rightTrim(const wstring &value);
   static wstring trim(const wstring &value);
   static bool ensureNotNullOrEmpty(const wstring &value);
   static bool ensureNotNullOrEmpty(const wstring &value, int minLength);
};
```

StringUtils – Testing "*LeftTrim*"

```
class StringUtilsFixture : public ::testing::Test {

protected:
   virtual void SetUp(){
      // Add here some testing set up code
   }

   virtual void TearDown() {
      // Add here some testing tear down code
   }
   StringUtils * utils;
};
```

```
TEST_F(StringUtilsFixture, LeftTrim){
   wstring expected = L"name  ";
   wstring original1 = L"   name  ";
   wstring original2 = L"name  ";
   wstring original3 = L" name  ";

   wstring result = utils->leftTrim(original1);
   EXPECT_EQ(result, expected);
   result = utils->leftTrim(original2);
   EXPECT_EQ(result, expected);
   result = utils->leftTrim(original3);
   EXPECT_EQ(result, expected);
}
```

StringUtils – Testing "*RightTrim*" and "*Trim*"

```
TEST_F(StringUtilsFixture, RightTrim){
   wstring expected = L"  name";
   wstring original1 = L"  name  ";
   wstring original2 = L"  name";
   wstring original3 = L"  name ";

   wstring result = utils->rightTrim(original1);
   EXPECT_EQ(result, expected);
   result = utils->rightTrim(original2);
   EXPECT_EQ(result, expected);
   result = utils->rightTrim(original3);
   EXPECT_EQ(result, expected);
}
```

```
TEST_F(StringUtilsFixture, Trim){
   wstring expected = L"name";
   wstring original1 = L"  name  ";
   wstring original2 = L"  name";
   wstring original3 = L"name  ";

   wstring result = utils->trim(original1);
   EXPECT_EQ(result, expected);
   result = utils->trim(original2);
   EXPECT_EQ(result, expected);
   result = utils->trim(original3);
   EXPECT_EQ(result, expected);
}
```

Notice that each unit test is checking either valid and invalid values.
Despite checking (in)valid values, it is also important to check, for instance, boundary values.

Some Considerations

- Standard libraries have been already extensively tested
- Such external dependencies do not need to be tested again

- *StringUtils* has no other dependencies than to standard libraries
  - As so, our tests are exclusively focusing on our base code

→ Tests to *StringUtils* are unitary tests

Another Testing Unit – Category

```
class Category {
private:
   wstring code;
   wstring description;
   Category();
   bool isCodeValid(const wstring &code);
   bool isDescriptionValid(const wstring &description);
public:
   Category(const wstring &code, const wstring &description);
   const wstring& getCode() const;
   const wstring& getDescription() const;
   Result changeDescription(const wstring &newDescription);
   bool hasCode(const wstring& code) const;
   bool operator == (const Category &param) const;
   bool operator < (const Category &param) const;
};
```

Category – Testing Acceptance Criteria (1/2)

```
TEST_F(CategoryFixture, CreateWithEmptyCode){
   EXPECT_THROW(new Category(L"",L"Category One"),std::invalid_argument);
}

TEST_F(CategoryFixture, CreateWithCodeHavingOneChar){
   EXPECT_THROW(new Category(L"1",L"Category One"),std::invalid_argument);
}

TEST_F(CategoryFixture, CreateWithCodeHavingFourChars){
   EXPECT_THROW(new Category(L"C001",L"Category One"),std::invalid_argument);
}

TEST_F(CategoryFixture, CreateWithValidData){
   EXPECT_NO_THROW(new Category(L"C0001",L"Category One"));
}
```

Acceptance Criteria:

- AC01-1. Category code cannot be empty nor have less than five chars.
- AC01-2. Category description cannot be empty.

Category – Testing Acceptance Criteria (2/2)

```
TEST_F(CategoryFixture, CreateWithEmptyDescription){
   EXPECT_THROW(new Category(L"C0001",L""),std::invalid_argument);
}

TEST_F(CategoryFixture, ChangingToInvalidDescription){
   Category cat(L"C0001",L"Category One");
   EXPECT_TRUE(cat.changeDescription(L"").isNOK());
   EXPECT_EQ(cat.getDescription(),L"Category One");
}

TEST_F(CategoryFixture, ChangingToValidDescription){
   Category cat(L"C0001",L"Category One");
   EXPECT_TRUE(cat.changeDescription(L"Changed Category").isOK());
   EXPECT_EQ(cat.getDescription(),L"Changed Category");
}
```

Acceptance Criteria:

- AC01-1. Category code cannot be empty nor have less than five chars.
- AC01-2. Category description cannot be empty.

Category – Testing Equal Operator

```
TEST_F(CategoryFixture, CheckingEqualOperatorUsingMemoryAddress){
   Category cat1(L"C0001",L"Category One");
   Category cat2 = cat1;
   EXPECT_TRUE(cat1 == cat2);
}

TEST_F(CategoryFixture, CheckingEqualOperatorUsingSameCodeValue){
   Category cat1(L"C0001",L"Category One");
   Category cat2(L"C0001",L"Category Two");
   EXPECT_TRUE(cat1 == cat2);
}

TEST_F(CategoryFixture, CheckingEqualOperatorUsingDistinctCodeValues){
   Category cat1(L"C0001",L"Category One");
   Category cat2(L"C0002",L"Category Two");
   EXPECT_FALSE(cat1 == cat2);
}
```

…

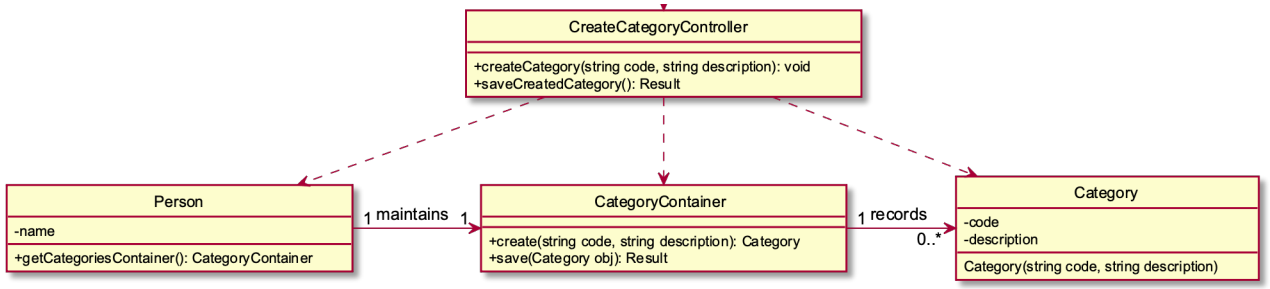
Some Considerations (cont.)

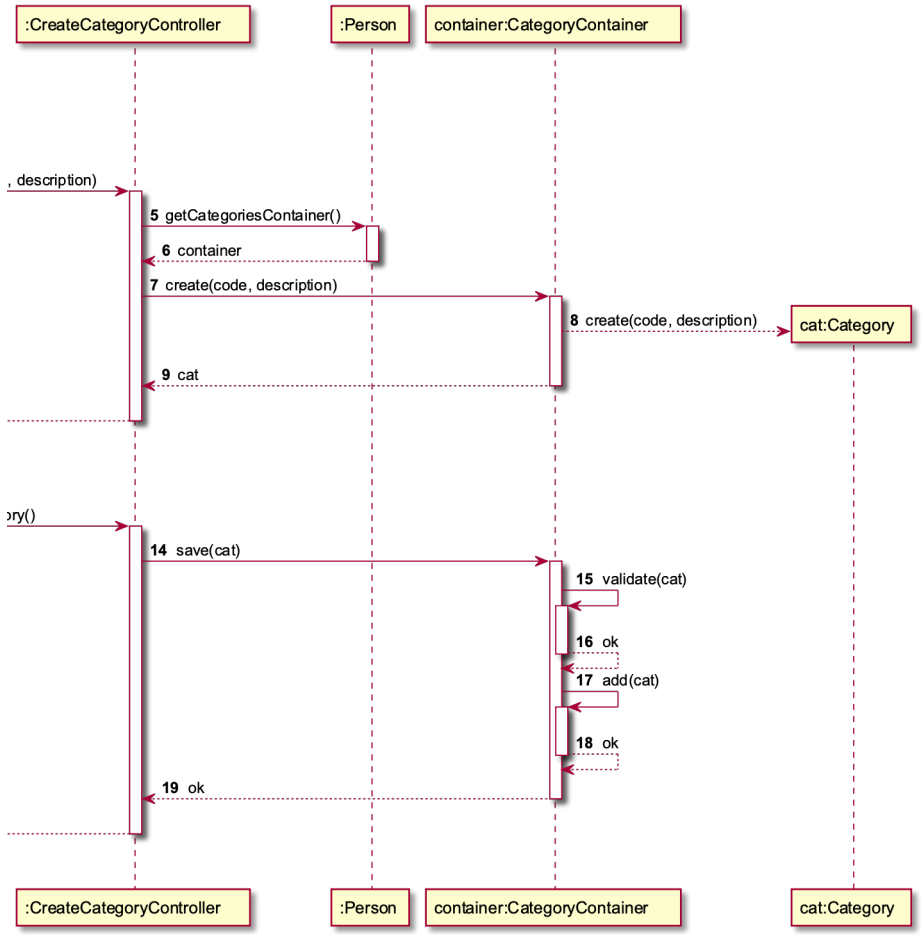
- Are tests made to the *Category* class unitary?
  - Yes! It has no other dependencies than to standard libraries
  - Like the *StringUtils* class
- What if the *Category* class had a dependency to the *StringUtils* class? Would it still be unitary tests?
  - Yes, if the considered unit becomes the **two classes together**
  - Yes, if *StringUtils* is considered as being a standard library class (which, in fact, is its purpose)
  - No, otherwise. It will be Integration Tests.

- Is it possible to do unitary tests on classes with dependencies to other classes not (considered as) belonging to standard libraries?

CreateCategoryController – How to unitary test it?

- It has dependencies to:
  - *Person* – invoking a method
  - *CategoryContainer* – invoking a method
  - *Category* – just knowledge (no methods are invoked) → Weak dependency





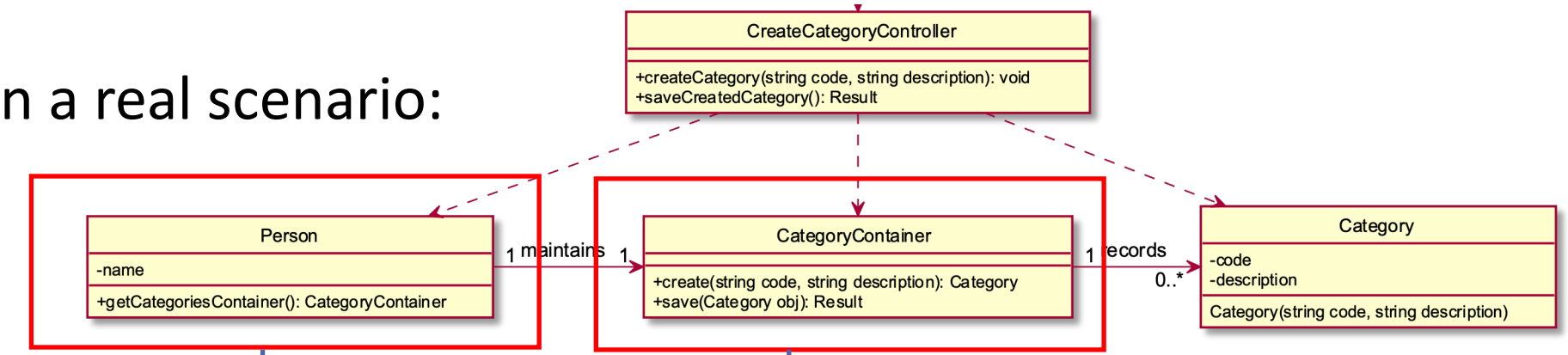
Mock Objects – What is it?

- **Mocking** is making a replica or imitation of something
- **Mock Object** is an object imitating another object
- Typically, **Mock Objects** are used
  - In automated tests of complex objects (i.e., with dependencies to other objects)
  - To isolate the behavior of the object being tested from its dependencies
  - To simulate the behavior of the objects that the object being tested is dependent

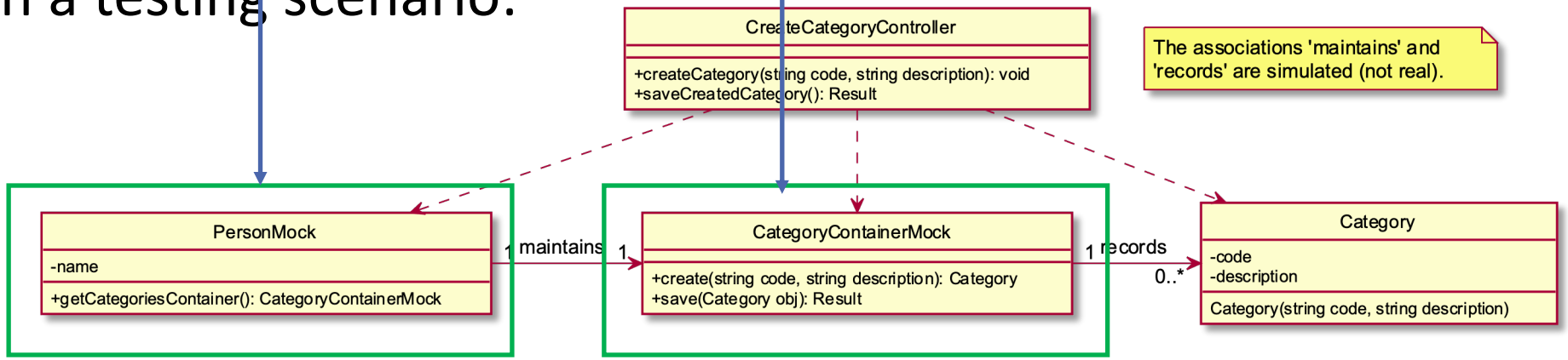
  → As so, **Mock Objects** are used to replace real objects.
- Example:
  - A *Person* object can be replaced by a *PersonMock* object
  - A *CategoryContainer* object can be replaced by *CategoryContainerMock* object

Unit Testing *CreateCategoryController* (1/4)

- In a real scenario:



- In a testing scenario:



Unit Testing *CreateCategoryController* (2/4)

```
class PersonMock : public Person
{
public:
   PersonMock(): Person(L"Joe"){
   };

   MOCK_METHOD(shared_ptr<CategoryContainer>, getCategoriesContainer, (), (override));
};


class CategoryContainerMock : public CategoryContainer
{
public:

   MOCK_METHOD(shared_ptr<Category>, create, (const wstring&, const wstring& ),(override));

   MOCK_METHOD(Result, save, (shared_ptr<Category>),(override));
};
```

Unit Testing *CreateCategoryController* (3/4)

```
TEST_F(CreateCategoryControllerFixture, SavingWithoutCreatingCategory){
    shared_ptr<PersonMock> person = make_shared<PersonMock>();

    CreateCategoryController controller(person, L"BlaBla");

    Result result = controller.saveCreatedCategory();

    EXPECT_TRUE(result.isNOK());
}
```

Creating an instance of *PersonMock.*

Passing the mock instance to the controller being tested.

Acting on the object being tested, and assertions are made.
Like what is done on other tests not using mocks.

Unit Testing *CreateCategoryController* (4/4)

```
TEST_F(CreateCategoryControllerFixture, SavingAfterCreatingCategory){
  shared_ptr<CategoryContainerMock> container = make_shared<CategoryContainerMock>();
  EXPECT_CALL(*container, create(_,_))
        .Times(1)
        .WillOnce(Return(make_shared<Category>(L"C0001", L"Category1")));
  EXPECT_CALL(*container, save(_))
     .Times(1)
     .WillOnce(Return(Result::OK()));

  shared_ptr<PersonMock> person = make_shared<PersonMock>();
  EXPECT_CALL(*person, getCategoriesContainer())
        .Times(2)
        .WillRepeatedly(Return(container));

  CreateCategoryController controller(person,L"BlaBla");
  controller.createCategory(L"C0001", L"Category 1");
  Result result = controller.saveCreatedCategory();
  EXPECT_TRUE(result.isOK());
}
```

Creating an instance of *CategoryContainerMock* and configuring the behavior of each mock method.

Creating an instance of *PersonMock* and configuring its behavior.

Acting on the object being tested, and assertions are made.

Some Considerations (cont.)

- Is it possible to do unitary tests on classes with dependencies to other classes not (considered as) belonging to standard libraries?
  - Yes! As seen on the previous example.
  - For that, you need to adopt **Mock Objects**.
- Do I need to mock all the dependencies of the object being tested?
  - No! Just mock the ones from which you want to isolate the behavior of.
  - However, notice that objects not being mocked are part of the unit being tested. Otherwise, it is an integration test.
- Mocks can be used in integration tests too?
  - Yes! E.g.: In the previous example, if you have just mocked the *CategoryContainer* and not the *Person*. In that case, it will be an integration test between the *CreateCategoryController* and *Person* classes.

More Myths and Facts

More Myths (1/2)

MYTH: ***Quality Control = Testing.***	**FACT**: Testing is just one component of software quality control. Quality Control includes other activities such as Reviews.
MYTH: ***The objective of Testing is to ensure a 100% defect- free product.***	**FACT**: The objective of testing is to uncover as many defects as possible while ensuring that the software meets the requirements. Identifying and getting rid of all defects is impossible.
MYTH: ***Testing is easy.***	**FACT**: Testing can be difficult and challenging (sometimes, even more so than coding).
MYTH: ***Anyone can test.***	**FACT**: Testing is a rigorous discipline and requires many kinds of skills.

More Myths (2/2)

MYTH: ***There is no creativity in testing.***	**FACT**: Creativity can be applied when formulating test approaches, when designing tests, and even when executing tests.
MYTH: ***Automated testing eliminates the need for manual testing.***	**FACT**: 100% test automation cannot be achieved. Manual Testing, to some level, is always necessary.
MYTH: ***When a defect slips, it is the fault of the Testers.***	**FACT**: Quality is the responsibility of all members/ stakeholders, including developers, of a project.
MYTH: ***Software Testing does not offer opportunities for career growth.***	**FACT**: Gone are the days when users had to accept whatever product was dished to them; no matter what the quality. With the abundance of competing software and increasingly demanding users, the need for software testers to ensure high quality will continue to grow. Software testing jobs are hot now.

References & Bibliography

- Nuno Silva et. al.; “Introduction to Software Testing”; ARQSI of LEI-ISEP; Porto, Portugal; 2021.
- What are the Different Levels of Software Testing?
  - https://www.edureka.co/blog/software-testing-levels/
- 5 Types Of User Acceptance Testing
  - https://usersnap.com/blog/types-user-acceptance-tests-frameworks/
- Alpha Testing Vs Beta Testing: What's the Difference?
  - https://www.guru99.com/alpha-beta-testing-demystified.html
- Black Box Testing Vs. White Box Testing: Key Differences
  - https://www.guru99.com/back-box-vs-white-box-testing.html

References & Bibliography

- Automation Testing Vs. Manual Testing: What's the Difference?
  - https://www.guru99.com/difference-automated-vs-manual-testing.html
- 7 Principles of Software Testing: Learn with Examples
  - https://www.guru99.com/software-testing-seven-principles.html
- Software Testing Fundamentals (STF)!
  - http://softwaretestingfundamentals.com
- The Most Common Software Testing Terminology
  - https://www.workwithloop.com/blog/the-most-common-software-testing-terminology