FITPED

GIT, testing

Peter Švec
Ján Skalka
Martin Drlík

www.fitped.eu

2021

Co-funded by the
Erasmus+ Programme
of the European Union

Work-Based Learning in Future
IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

GIT, Testing

Published on

November 2021

Authors

Peter Švec | Teacher.sk, Slovakia

Martin Drlík | Constantine the Philosopher University in Nitra, Slovakia

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Cyril Klimeš | Mendel University in Brno, Czech Republic

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Dušan Junas | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

Table of Contents

GIT

Version Control System

Chapter 1

1.1 Introduction

🕮 1.1.1

Version Control System (VCS) is a software that helps software developers to work together and maintain a complete history of all versions of programming code. VCSs are a category of software tools that help a software team manage changes to source code over time, keep track of every modification to the code in a special kind of database. If a mistake happens, developers can compare earlier versions of the code with the current version to fix the mistake while minimising disruption to all team members.

VCS has the following functions:

- allows developers to work simultaneously,
- does not allow overwriting each other's changes,
- maintains a history of every version.

🕮 1.1.2

Historically, there are two main types of VCS:

- Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in the worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, the distributed version control system (DVCS) comes into the picture.
- Distributed/Decentralized version control system (DVCS) clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server, and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require a network connection only to publish your changes and take the latest changes.

This tutorial is focused on distributed version control system Git.

🕮 1.1.3

The following are the benefits of Version Control Systems:

- track every individual change made by the developer,
- prevent concurrent work from conflicts,
- supports a developer's preferred workflow,
- works on any platform,
- facilitates a smooth and continuous flow of changes to the code rather than the frustrating of file locking,
- stores complete long-term change history of every file,
- allows creating and merging branches,
- allows easily to trace each change made to the software and connect it to project management and bug tracking software.

🕮 1.1.4

Git is the most widely used modern VCS. Git is a mature, open-source project with distributed architecture. It belongs to the DVCS. Git has the functionality, performance, security and flexibility that most teams and individual developers need. Git is a de facto standard. Moreover, many third-party software tools and services are already integrated with Git, including IDEs, issue and project tracking software, and code hosting services like GitHub or Bitbucket.

🕮 1.1.5

Git comes with several advantages. The most important are as follows:

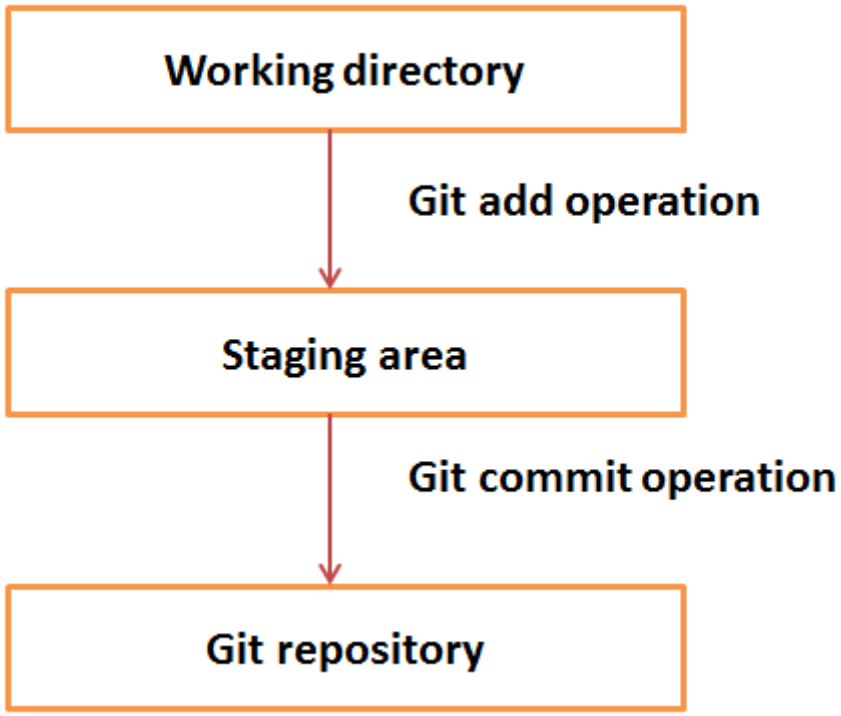
- As was mentioned before, Git belongs to the Free and open-source software. Git is released under GPL's open source license. It is available freely over the internet for download. It is possible to download its source code and perform any changes according to developers' requirements.
- Git is small because it does not require GUI, the core of Git is written in C, and most operations are done locally.
- Git supports implicit backup. It means that the chances of losing data are very rare when there are multiple copies of it. Each client contains the whole repository of source code, which can be used in the event of a crash or disk corruption.
- Because Git uses a cryptographic hash function SHA1 to name and identify objects within its database, Git is considered safe. Each operation is check-summed. Therefore, it is impossible to change file and its properties from the Git database without knowing Git.

- Git does not require powerful hardware because it does not rely on a central server. In the case of Git, similarly to other DVCS, developers do not interact directly with the server unless they need to push or pull changes.
- Using branches require copying, deleting and merging files is complicated and time-consuming. However, branch management with Git is very simple and fast. It takes only a few seconds to create, delete, and merge branches.

🕮 1.1.6

The basic workflow of changes (life cycle) in files consists of the following steps:

1. Files are modified in the working directory.
2. Files are added to the staging area.
3. Operation commit moves the files from the staging area to the local Git repository.



🕮 1.1.7

The file, several files or directories can be in one of the following states, which are available using git command

```
git status
```

- modified – changes have not been yet committed and stored in a local repository,
- staged – modified file or directory are prepared for the next commit,
- committed – changes are stored in a local repository,

🕮 1.1.8

Git can be easily installed on Windows, Linux or Mac OS. There are several ways how to download and install it, for example, from the following official websites (https://gitforwindows.org/, https://git-scm.com/book/en/v2/Getting-Started-Installing-Git).

🕮 1.1.9

Git has several global (system-wide) and local configuration variables, which can be set up using git config tool. Global configurations are stored in the .gitconfig file, which is located in the Git home directory. To set up these configuration variables as global, the --global option must be added. If --global option is omitted, the configuration variables are specific for the current Git repository, as --local option. To set up these values, the user must have root privileges the --system option must be used in commands.

Setting username, which is used by Git for each commit.

```
git config --global user.name <name>
```

Setting email id, which is used by Git for each commit.

```
git config --global user.email <email>
```

Listing current Git settings

To verify your Git settings of the local repository, git config –list command can be used as given below.

```
git config –list
```

It also often necessary to define or change the default text editor used for example by the commit command.

```
git config --system core.editor <editor>
```

where <editor> is a command that launches the desired text editor. The -- system option set the configuration for the entire system, meaning all users and repositories on the computer.

GIT Commands

Chapter 2

2.1 Git Commands

📖 2.1.1

Git contains many commands with a myriad of options and flags, which help a developer to solve many complicated situations. The git help command helps them to find the expected command quickly and provide all required information in a concise manner. This is the simplest version of the command

```
git help <verb>
```

where <verb> represents the search term.

📖 2.1.2

A new repository can be created using command

```
git init
```

This command creates a new .git subdirectory in the current directory, which is considered as a working directory of the project. Therefore, it is recommended to move to the directory, which should be tracked by the Git, before this initial command is executed. Git can be initialized in existing project directory using the following slightly modified command

📖 2.1.3

If the central repository has been already created and contains the source code of the project, the command git clone can be used for creating its local working copy. The original repository can be located on the local filesystem or on a remote machine accessible, supported protocols. Once a developer has obtained a working copy, all version control operations are managed through their local repository.

```
git clone <repo url>
```

Repository URL is available in the central repository and will be introduced later. As a convenience, cloning automatically creates a remote connection called "origin" pointing back to the original repository. This makes it very easy to interact with a central repository.

```
git clone <repo url> <directory>
```

this equivalent clones of a remote repository in the defined directory.

📖 2.1.4

In contrast with saving file in the operation system, saving file in Git consists of several steps. A commit command is at the end and therefore is often considered the Git equivalent of a "save". However, in the detailed view, there are three commands: git add, git status, and git commit, which are used in combination to save a snapshot of a Git project's current state.

📖 2.1.5

The *git add* command adds a change in the working directory to the staging area. It tells Git, which updates to a particular file should be included in the next commit. However, *git add* does not really affect the repository in any significant way. Changes are not actually recorded in the local repository until the command git commit runs.

```
git add <file> - stages a particular file to the staging area
for the next commit,
git add <directory> - stages all changes in a directory for
the next commit,
git add. – adds the whole project directory to the staging
area.
```

📖 2.1.6

The following command

```
git status,
```

is often used together with *git add* to view the state of the working directory and the staging area.

It returns the current state of the files in the working directory and staging area.

Other command

```
git log
```

shows detailed information about all the snapshots of the project. For example, it can show the whole history of a selected file

```
git log <file>
```

📖 2.1.7

The *git commit* command is together with *git add* the most frequent Git command. It captures a snapshot (commits) of the project's currently staged changes and can be thought of as snapshots along the timeline of a Git project. Committed snapshots represent stable, safe versions of a project. Git snapshots are always committed to the local repository. Git does not require to interact with the remote repository until the developer makes this decision. Each developer's local repository, where the commits are stored, is a buffer between the developer's contributions and the central remote repository.

The simplest form of commit command is as follows

```
git commit
```

However, this command launches the text editor and prompt developer for a commit message, which is a mandatory part of each commit. It is often easier for that reason to add the message directly to the commit command using with -m option

```
git commit -m "commit message"
```

If the -a option is included, this combination immediately creates a commit of all the staged changes and takes an inline commit message

```
git commit -am "commit message"
```

📖 2.1.8

A Git repository can be configured to ignore specific files or directories. This will prevent Git from saving changes to any ignored content stored in files. Git has multiple methods of configuration that manage the ignore list. Ignored files are usually built artefacts and machine-generated files that can be derived from the repository source. Some common examples are:

- dependency caches,
- compiled code, such as .class files
- build output directories, such as /bin, /out,
- files generated at runtime, such as .log, .lock, or .tmp
- hidden system files, such as .DS_Store or Thumbs.db
- personal IDE config files, such as .idea/workspace.xml

Ignored files are tracked in a special file named .gitignore that is checked in at the root of your repository. This file can be created using any text editor, which contains all required git ignore patterns based on the various symbols like **, *, !, and other regular expression wildcards

📖 2.1.9

Often it is necessary to compare changes between two input files. Git diff command is a multi-use Git command that runs a diff function on Git data sources like commits, branches, files, etc. It is often used along with git status and git log to analyse the current state of a Git repository.

```
git diff
```

This command has many options, which make it a very multi-purpose command. Their detailed description is available on the official documentation site (https://git-scm.com/book/en/v2)

📖 2.1.10

Sometimes it is necessary to temporarily shelves (stashes) changes made to the working directory and work on something else, and then come back and re-apply them later, without committing. The git command

```
git stash
```

takes staged and unstaged uncommitted changes save them away for later use, and then reverts them from working directory. After it is executed, the developer can make other changes, create new commits, switch branches. Finally, she can come back and re-apply stash with the command

```
git stash pop
```

📝 2.1.11

The abbreviation VCS stands for:

- Version Control System
- Virtual Control System
- Version Combination System
- Very Complicated System
- Value Commited System

📝 2.1.12

Mark the states in which files may be during the Git life cycle.

- staged
- commited
- modified
- aborted
- requested
- submitted

2.1.13

Which of the following Git commands can be used for creating a new repository?

- $ git init
- $ git clone
- $ git copy
- $ git start
- $ git create
- $ git download

Testing

Introduction to Software Testing

Chapter 3

3.1 Basic Testing Methods

🕮 3.1.1

Software testing is

- part of a more general verification and validation process, which also includes static validation techniques,
- the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not,
- executing a system to identify any gaps, errors, or missing requirements contrary to the actual requirements,
- intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- A process of analysing a software item to detect the differences between existing and required conditions and to evaluate the features of the software item.

Testing software often requires executing a program using artificial data. Testing can reveal the presence of errors, not their absence.

🕮 3.1.2

Testing is a part of all kinds of software development life cycles. Different types of stakeholders are involved in the process of software testing depending on the complexity of the project, used methodology and project management, the experience of the project team members. The following roles are often involved in the process:

- Software Tester,
- Software Developer,
- Project Manager,
- End-User,
- Software Quality Assurance Engineer,
- Quality Assurance Analyst.

🕮 3.1.3

Software Development Life Cycle (SDLC) often called the Software Development Process, is a process used by the software industry to design, develop and test high-quality software. The SDLC aims to produce high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates. It has the following phases in general:

- Planning and requirements analysis
- Defining requirements
- Designing the product architecture
- Building the product
- Testing the product
- Deployment of the product
- Maintenance and further development of the product

🕮 3.1.4

There are numerous SDLC models, which are often called Software Development Process Models. They are suitable for different situations. Their effective use depends on the complexity of the software product, which should be developed. They differ in the series of steps, which ensure success in the process of software development. The following are the most popular SDLC models:

- Waterfall model
- Iterative model
- Spiral model
- V-model
- Agile model
- Rapid Application Development Model
- Prototyping model.

The software testing is included in all SDLC models in different extend. It can create a separate phase, or testing can create an inseparable part of different phases of the SDLC.

🕮 3.1.5

Verification and Validation are two very similar terms, which closely relate to the software testing topic. They differ in the following aspects:

Verification:

1. "Are you building it right?"
2. Ensures that the software system meets all the functionality.
3. Verification takes place first and includes the checking for documentation, code, etc.
4. It is done by developers.
5. It has static activities, as it includes collecting reviews, walkthroughs, and inspections to verify software.
6. It is an objective process, and no subjective decision should be needed to verify software.

Validation:

1. Validation addresses the concern: "Are you building the right thing?"
2. Ensures that the functionalities meet the intended behaviour.
3. Validation occurs after verification and mainly involves the checking of the overall product.
4. It is done by testers.
5. It has dynamic activities, as it includes executing the software against the requirements.
6. It is a subjective process and involves subjective decisions on how well the software works.

🕮 3.1.6

When to Start Testing?

Testing should start as early as possible because early start to testing reduces cost and time to rework the product. Real start depends on the used development model. Testing is incorporated in every phase of SDLC:

- During the requirement gathering phase, the analysis and verification of requirements are also considered as testing.
- Reviewing the design in the design phase with the intent to improve the design is also considered as testing.
- Testing performed by a developer on completion of the code is also categorised as testing.

🕮 3.1.7

When to Finish Testing?

Any software can not be 100% tested. Testing is a never-ending process. Therefore, it is important to estimate how much testing is enough and consider the following circumstances:

- Deadline of the project
- Code coverage level of the source code
- Bug rate under a certain level
- Decisions of the project manager

🕮 3.1.8

Advantages of Software Testing

- Cost-effectiveness – early testing saves time and costs because the discovered problem does not affect the final implemented solution
- Software improvement – testing is a phase of all SDLC
- Automation reduces the testing time. However, it should be started after static validation, an inspection of the system.
- Software quality assurance helps to measure the following set of software properties like functionality, reliability, usability, efficiency, maintainability, portability.

🕮 3.1.9

Program Testing has the following parts

Validation testing demonstrates to the developer and the system customer that the software meets its requirements. A successful test shows that the system operates as intended.

Defect testing, which discovers situations in which the behaviour of the software is incorrect, undesirable or does not conform to its specification. It leads to defect testing. A successful test is a test that force the system to perform incorrectly and so shows a defect in the system

🕮 3.1.10

Software inspection is a formal evaluation technique in which software requirements, designs, or codes are examined in detail by a person or a group other than the author to detect faults, violations of development standards, and other problems Complementary verification technique to testing. It represents a formal technique that involves formal or informal technical reviews of any artefact by identifying any error or gap. Software inspection is considered an effective approach for discovering program errors. It is a static verification because it focuses on the analysis of the static system representation to discover problems.Simultaneously, it does not require the execution of a system and, therefore, t does not require additional costs for inspection of incomplete versions of a system. It can be applied to any software representations like the requirements specification, software architecture, database schema. Software inspection can consider a broader set of quality attributes like portability, maintainability, compliance with standards. It can check conformance with a specification, not with the customers' requirements. On the other hand, it does not check non-functional requirements like performance, usability.

📖 3.1.11

The software engineering defines the following two types of testing:

Manual testing covers testing a software manually, without any automated tool and scripts. Manual testing has several stages, which will be introduced later. The tester in the role of end-user test the software with the aim to identify any unexpected behaviour. The tester uses test cases and test scenarios to ensure the completeness of tests.

Automated testing (Test automation) requires the tester writes scripts and use specialised software to test the software product. Test automation is a logical replacement of the manual testing, in which the constantly repeating routines occur. It allows running the test scenarios repeatedly and incrementally.

📖 3.1.12

When to Automate?

Test Automation is useful in the following situations:

- project is large and complex,
- projects require testing the same areas repeatedly,
- requirements do not change very often.

📖 3.1.13

There are many specialised systems, which allow automated testing. The following methodology can be used to decide if automated testing can be used:

- Identifying areas within the software for automation
- Selection of appropriate tool for test automation
- Writing test scripts
- Development of test suits
- Execution of scripts
- Create result reports
- Identify any potential bug or performance issues

📝 3.1.14

Mark the correct statements about software verification.

- Verification answers the question of whether we developed the software correctly.
- Verification answers the question of whether we have developed the right software.
- Verification is done by software developers.
- Verification is done by end users.
- Verification is an objective process that requires no subjective decisions.
- Verification is a subjective process that assumes subjective decisions about how software works properly.
- Verification precedes software validation.
- Verification follows software validation.

3.1.15

Mark the correct statements about software validation.

- Validation answers the question of whether we developed the software correctly.
- Validation answers the question of whether we have developed the right software.
- Validation is done by software developers.
- Validation is done by end users.
- Validation is an objective process that requires no subjective decisions.
- Validation is a subjective process that assumes subjective decisions about how software works properly.
- Validation precedes software validation.
- Validation follows software validation.

3.1.16

Which of the following test types can be automated?

- unit tests
- integration test
- performance tests
- functional tests
- usability tests
- acceptance tests

3.1.17

The aim of testing is to ensure 100% error-free software.

- false
- true

📝 3.1.18

System testing is the level of software testing in which we test complete software using the black box method.

- true
- false

📝 3.1.19

Usability testing is a type of testing that is done from a developer perspective to see if the software is easy to use.

- false
- true

📝 3.1.20

Regression testing is a type of testing, in which we want to see if the changes we made to the source code have not retroactively affected the functionality of previously developed parts of the software.

- true
- false

3.2 Types of Testing

🕮 3.2.1

There are several types of testing:

- Smoke Testing
- Functional Testing
- Non-functional Testing

They all can be used on different levels of testing, which will be introduced later.

📖 3.2.2

Smoke Testing

Smoke testing, often called build verification testing, is a type of software testing that comprises of a non-exhaustive set of tests, which try to ensure that the most important functions of the software will work. The result of this testing is used to decide if a build is stable enough to proceed with further testing. This type of testing can uncover problems early. It can be used in integration, system and acceptance levels of testing.

📖 3.2.3

Functional Testing

Functional testing is a type of black-box testing. The software is tested using a set of tests with known inputs. The obtained outputs (results) are compared with expected ones. Functional testing has the following steps:

- The determination of the functionality that the intended application is meant to perform.
- The creation of test data based on the specifications of the application.
- The output based on the test data and the specifications of the application.
- The writing of test scenarios and the execution of test cases.
- The comparison of actual and expected results based on the executed test cases.

📖 3.2.4

Non-functional Testing

Non-functional testing involves testing important software non-functional requirements such as

- performance,
- security,
- user interface,
- compliance with standards.

Methods and Levels

Chapter 4

4.1 Testing Methods

🕮 4.1.1

Testing Methods

The following basic methods can be used based on the level of knowledge of the internal structure of the software, which is tested.

- Black-box testing
- White-box testing
- Grey-box testing
- Agile testing
- Ad-hoc testing

🕮 4.1.2

Black box Testing

Black box testing, known as behavioural testing, is testing without any knowledge of the internal structure, design or implementation. The tester has no access to the source code, but she interacts with the user interface of the software product. She provides a set of inputs and examines the outputs. The outputs must fulfil the tester's expectations. The main advantage of this approach is, that black-box testing is suitable for large code segments, does not require the access to source code, shows, how the software will be used by end-user, does not need testers with the knowledge of programming languages, operation systems and other whole SDLC. The main disadvantages of black-testing technique are limited coverage by tests, difficulties to design test cases and limited knowledge of the testers about the software product. This method attempts to find incorrect or missing functions, interface errors, errors in data structures, behaviour as well as performance. It is applicable to the integration, system and acceptance testing levels.

🕮 4.1.3

White box Testing

This testing method, also known as glass testing, requires the tester knows the internal structure, design or implementation of the software. In other words, she has access to the source code and can investigate the internal logic and structure of the code. This is simultaneously the main advantage of this method. Moreover, it allows code optimising, refactoring and the maximal coverage of the code due to the knowledge of the code. On the other hand, this technique requires skilled tester

and specialised tools like code analyser and debugging tools. This method is applicable to unit, integration and system testing levels.

🕮 4.1.4

Grey box Testing

The grey box testing method has limited knowledge of the internal structure and logic of the tested software product. The tester usually must design documents and the database. Therefore, she can write better test scenarios. The combination of best practices of white box and black box methods is considered the main advantage of this method. It relies on interface definition and functional specifications. The tests are realised from the end-user's point of view, not a developer. It is primarily used in integration testing level.

🕮 4.1.5

Agile Testing

Agile testing represents a testing method, which follows the principles of agile development methods. This testing method does not require any special approach and techniques. It still needs all proven software testing methods and levels, but their use depends predominantly on the tester or developer decision and other priorities of the agile team. Agile testing is built upon very simple, strong and reasonable processes like the process of conducting the daily meeting or preparing the daily build. Simultaneously, it attempts to leverage tools, especially for test automation, as much as possible. Testing itself is in the middle of interest. As a result, this method does not elaborate on any plan or documentation.

🕮 4.1.6

Ad-hoc Testing

Ad-hoc testing, sometimes called as Random Testing or Monkey Testing, is a software testing method without any planning and documentation. All tests are conducted informally and randomly without any formal procedure or expected results. This method is normally used during Acceptance Testing. Surprisingly, this method can be very useful in finding errors, which is hard to find using other more systematic, step-by-step approach. The success of the method, therefore, depends on the creativity and previous experience of the tester.

4.1.7

Which level of testing tests the smallest part of the software?

- unit testing
- integration testing
- system testing
- debugging
- source code testing
- acceptance testing

4.1.8

Which testing method does the unite testing mostly use?

- white box
- black box
- brut force testing
- grey box
- yellow box
- divide et impera

4.1.9

Who participates in acceptance testing?

- end-users
- developer
- customer
- manager
- CIO
- researcher

4.1.10

Which level of testing use black box testing method?

- integration testing
- acceptance testing
- unit testing
- system testing

📝 4.1.11

Which level of testing use white box testing method?

- integration testing
- acceptance testing
- unit testing
- system testing

4.2 Testing Levels

🕮 4.2.1

Unit Testing

Unit testing is a level of testing, where individual units/components are tested with the aim to verify that these units behave as expected. Unit testing belongs to the white-box testing method. A unit is the smallest tested part of the software, for example, method of a class in OOP. Unit testing is performed mainly by the developer, who can also be the author of the source code or other member of the development team. The developer uses test data. The main aim is to validate that part of the source code is correct in terms of requirements and functionality. It is impossible to cover all source code and evaluate all possible execution paths of the software.

Unit testing has the following benefits:

- increase confidence in changing code,
- code is easier to reuse,
- development is faster,
- the cost of fixing a bug is smaller,
- small units are easier to understand.

🕮 4.2.2

Integration Testing

Integration testing means testing of combined parts of the software with the aim to determine if the parts work correctly. The purpose of this second level of testing is to expose faults in the interaction between integrated units. Integration testing can use bottom-up, top-down and big bang approach. While the first one begins with unit testing, followed by a combination of module testing and builds, the second one the modules are tested first and then the lower-level modules and units are

tested. A big bang is an approach, where all or most of units are combined together and tested at one run. Integration testing uses any of black-box, white-box or grey-box testing methods.

🕮 4.2.3

System Testing

System testing is the third level of software testing. It tests the whole system to verify if it meets functional and technical specifications. After all the components are integrated, the software is tested with the aim to fulfil the specified quality standards. This kind of test enables to test, verify and validate not only business requirements but also the software architecture in the environment, which is very close to the production environment. Black box testing method is usually used form system testing.

🕮 4.2.4

Acceptance Testing

Acceptance testing is closely joined to quality assurance. Acceptance testing is the fourth and last level of software testing. It verifies whether the software meets previously defined specifications and satisfies the business requirements of the customer. A system is tested for acceptability. Acceptance tests are intended to point out any bugs, which will result in the software crash, but it also points out small mistakes, errors and differences. It usually uses the black-box testing method. It does not follow a strict procedure and is rather ad-hoc.

🕮 4.2.5

Alpha Testing

Alpha test is realised by the development teams in the first stage of testing. It means that combined unit, integration and system testing are considered together as alpha testing. The software is tested for spelling mistakes, broken links.

🕮 4.2.6

Beta Testing

The beta testing (also called pre-release testing) follows alpha testing after it has been successfully finished. A selected group of future users tests the software. It is important to distribute the testing to a wide range of future users, who will test installation procedure, typography, navigation and flow of tasks, etc. Simultaneously, they provide important feedback, identify hidden problems and test their fixes.

Advanced Testing

Chapter 5

5.1 Non-functional Types of Testing

📖 5.1.1

Regression Testing

All changes in the software can cause problems in other areas of the software. For that reason, the regression testing focuses on verification, if the change has not resulted in another functionality violation. In other words, regression testing ensures that this change has not caused problems, which are not covered by the tests. During regression testing, new test cases are not created but previously created test cases are re-executed. Regression testing can be used during any level of testing, mainly during system testing.

📖 5.1.2

Usability Testing

Usability testing is a black-box technique used for identification of errors and consequent implementations of the software improvements by observing the users' behaviour. This testing is done from a user perspective with the aim to find out, if the software is easy to use. It is focused on the efficiency of use, ability to learn, ability to memorise, errors and safety and satisfaction of the users. This type of testing can be performed during system and acceptance testing levels.

📖 5.1.3

Security Testing

Security testing belongs to the critical and inevitable kinds of non-functional testing. Its aim is to identify any problems with the security and vulnerability of the software. Depending on the nature of the software, security testing tries to ensure integrity, availability, correct authorisation and authentication, and save software against different kinds of attacks and flaws.

📖 5.1.4

Portability Testing

Portability testing is focused on testing software for reusability, transferring software between computers and different versions of operating systems and middleware.

🕮 5.1.5

Compliance Testing

Compliance testing, sometimes called a conformance testing or regulation testing, is a type of testing to determine the compliance of a system with internal or external standards. The method and type of testing to be conducted during compliance testing depends on the specific regulation / standard being assessed.

🕮 5.1.6

Performance Testing

Performance testing is a type of non-functional testing focused on determining how a software performs in terms of responsiveness and stability under different conditions. It covers load tests, stress tests, endurance and spike tests. While Load testing test the behaviour of the software by application maximum load of input data, stress testing tests the behaviour under abnormal conditions like losing resources. Performance testing tries to identify any bottlenecks related to the software performance like network delay, load balancing between servers, database transaction delay using quantitative and qualitative measures. It tests speed, capacity, stability and scalability of the software.

JUnit Testing

Introduction

Chapter 6

6.1 Introduction

🕮 6.1.1

JUnit is a simple open-source framework for Java source testing (www.junit.org). It is intended to verify if piece of code works as is expected. It uses principles based on comparison of expected and obtained outputs.

JUnit is a special tool that allows to write Java tests using a simple interface.

It can test functions, methods, classes, packages, subsystems and supports automated testing.

JUnit is intended to run tests that have already been prepared - after editing code and making changes.

🕮 6.1.2

The libraries for JUnit are shipped with IntelliJ IDEA, but are not included in the classpath of your project or module by default.

To add the necessary library to the classpath, you can use the general procedure of adding a dependency to a module. The corresponding libraries are located in the following directories:

- JUnit libraries (**hamcrest-core-1.3.ja**r and **junit-4.12.jar**): <IntelliJ IDEA directory>\lib.

IntelliJ IDEA can add the necessary library to the classpath automatically. The corresponding features are available when creating a test for a class or when writing the code for a test.

🕮 6.1.3

Create a class Calculator with methods:

- sum - adds two integers obtained as parameters and returns the result as an integer,
- multi - multiplies two integers obtained as parameters and returns the result as an integer,

Write tests to verify that the class you created is working properly.

📖 6.1.4

To create new class is easy - we create it in new project:

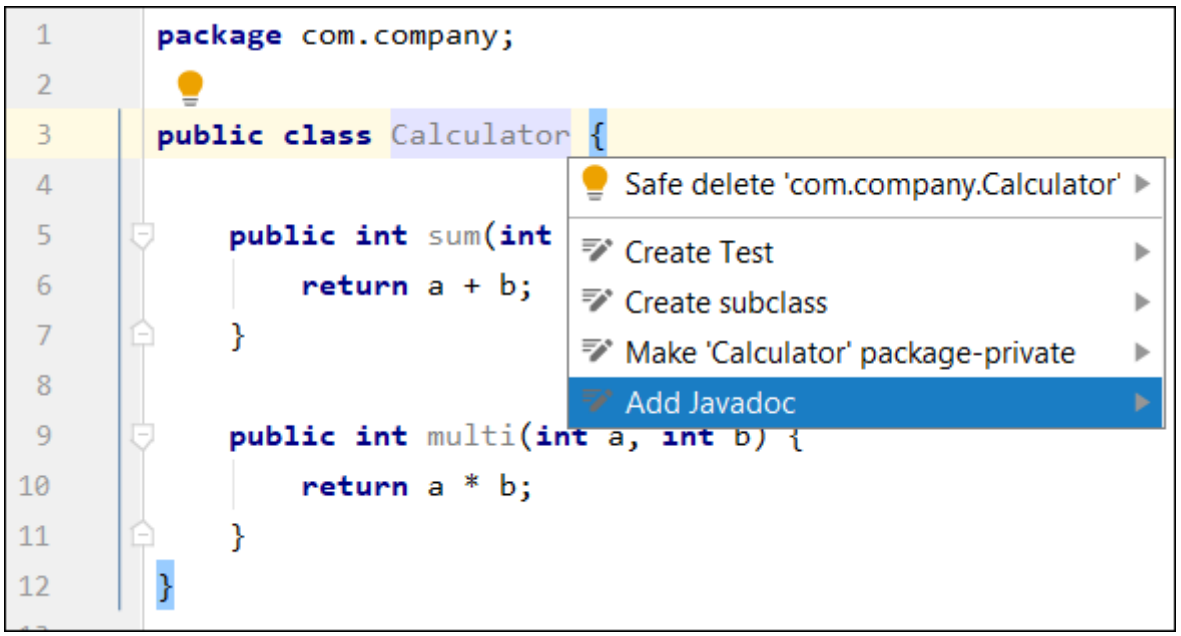
```
public class Calculator {

    public int sum(int a, int b) {
        return a + b;
    }

    public int multi(int a, int b) {
        return a * b;
    }
}
```

📖 6.1.5

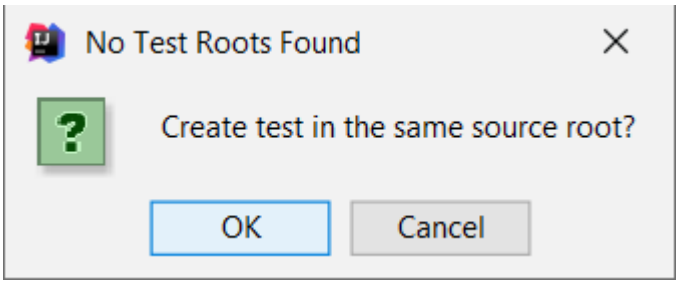
Ta add tests we can proceed in several ways:

- we can place the test files directly into package with code (it is not the best solution)
- we can create separated group for tests.

To initialise tests we should press **Alt+Enter** in the name of our class (or select *Show content action* in popup menu).

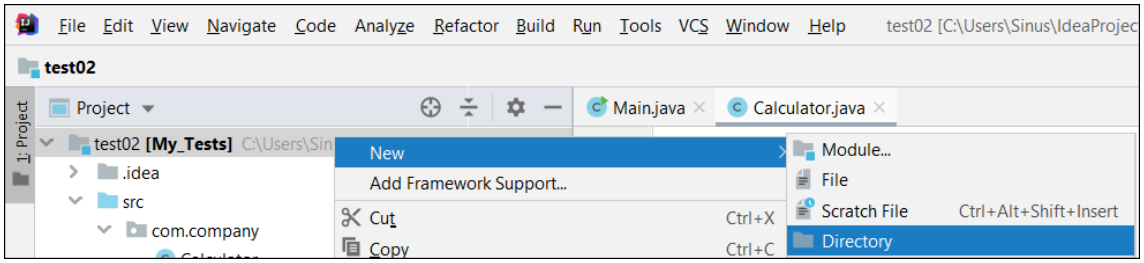


The warning after **Create test** option selection is that the application doesn't have place for roots

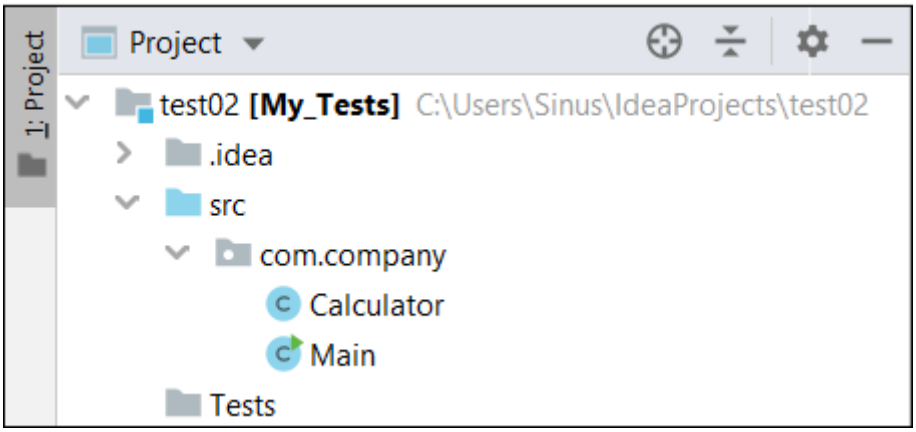


We cancel dialogue, and:

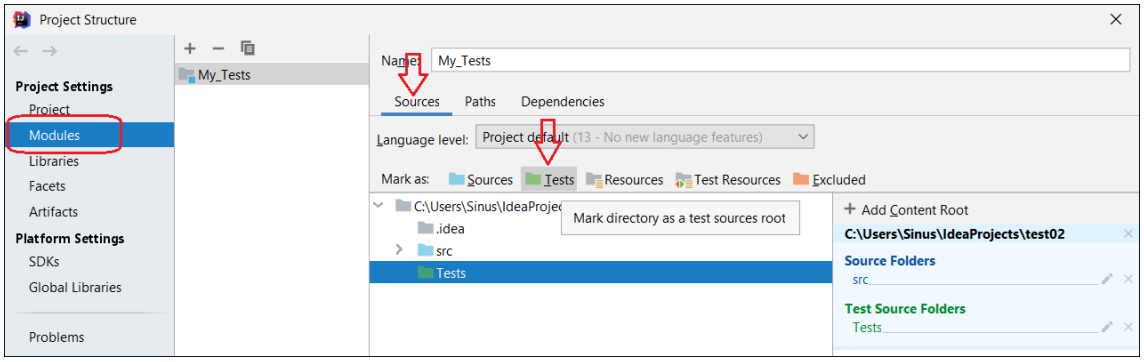
We create new folder in project structure (e.g. **Tests**)



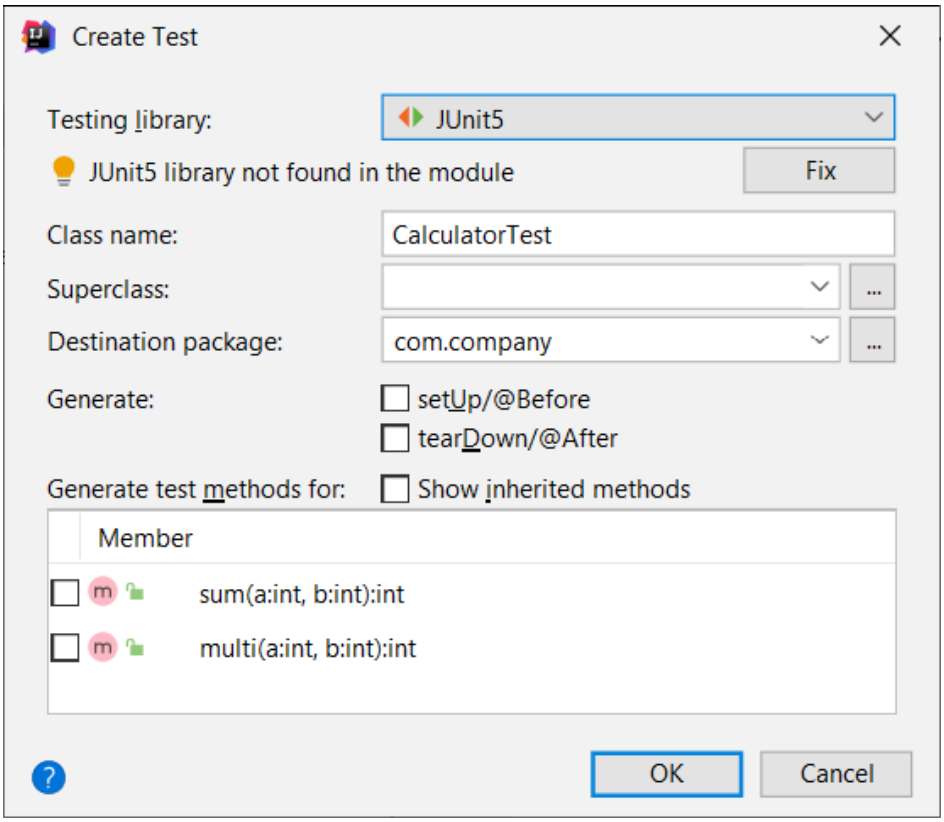
The result is new folder:



We mark this created folder in project structure (**File -> Project structure**) in the **Module Group** on tab **Sources** to **Tests**:

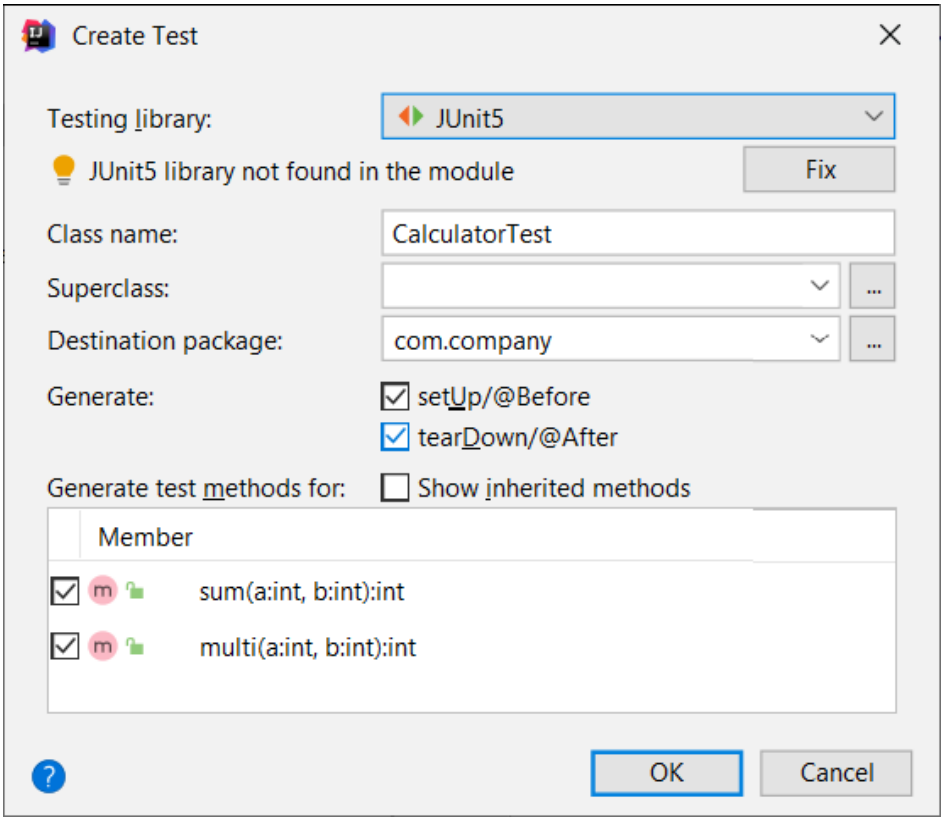


Next use of **Alt + Enter** opens window for test parameters settings.



If we prepare first test, we probably need to install library and use Fix button to solve actual situation.

After install (the window is still opened) we set following selections:



🕮 6.1.6

The result of test dialogue activity bring some code:

```
package com.company;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

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

class CalculatorTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void sum() {
    }

    @Test
    void multi() {
    }
}
```

Every test class consists of optional parts defined by notations (the method names are not important):

- **@BeforeEach** - must be performed **before each test** in the class to set the parameters needed for the test,
- **@AfterEach** - must be performed **after each test** in the class (e.g. reset parameters, etc.),
- **@Test** - the method of testing itself

6.2 First test

🕮 6.2.1

The program has to work to every time with an independent instance. We can achieve it with new independent calculator create before every test.

We use **@BeforeEach** notation method:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }
```

🕮 6.2.2

The use of method with **@AfterEach** notation is not necessary. We can let it empty.

🕮 6.2.3

The tests are realised in methods with notation **@Test**. We can prepare its content on ourself, but the idea of test writing is to prepare easy understandable code - we use the methods in test class for testing in methods with the same name.

The most commonly used method for testing is the **AssertEquals** method, which compares the expected value with the result obtained from the tested class.

```
assertEquals(5, calc.sum(2, 3));
```

- the first parameter is the expected value
- the second parameter is the value obtained as a result of the test class

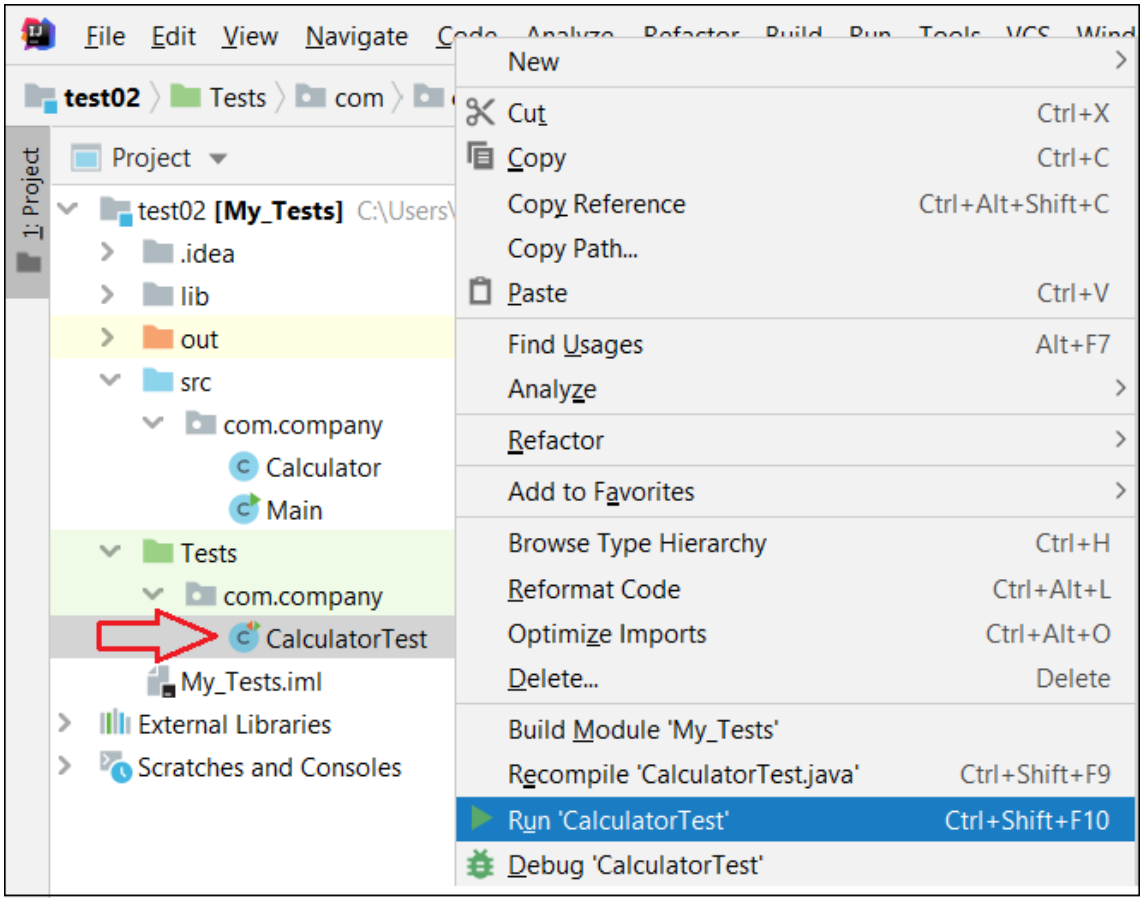
The code with tests for sum has the following form:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
```

```
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void sum() {
        assertEquals(5, calc.sum(2, 3));
        assertEquals(-3, calc.sum(-8, 5));
    }
```

📖 6.2.4

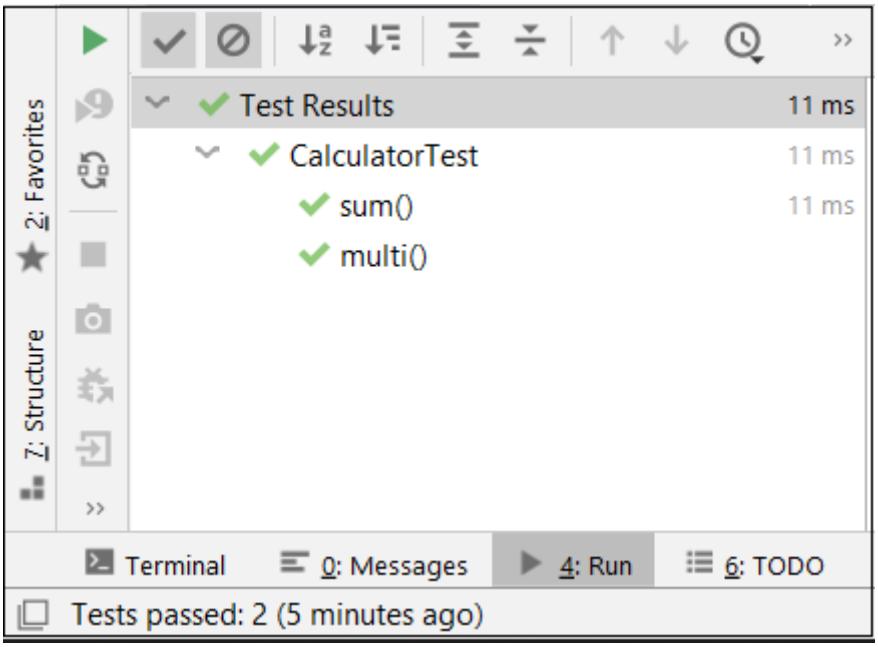
To run created test you have to start it as follow:



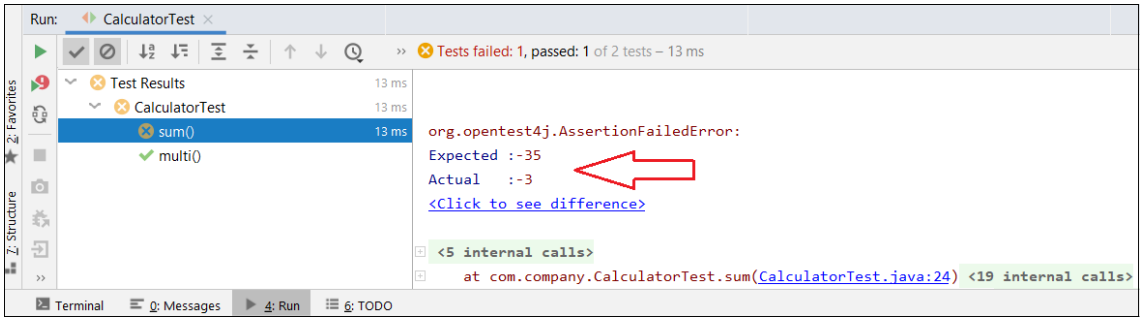
Using context menu you can **Run CalculatorTest**. Or you can run application using button on the toolbar.

The result of run should be:

- all tests passed is successfull:



- some tests failed:



The environment shows you:

- expected value: value what was written by user as expected value of tested function
- actual value: value returned by tested function

... and now you can looking for where is the mistake.

🕮 6.2.5

Let's go to finish our tests.

We add new asserts for second tested method:

```
class CalculatorTest {
    Calculator calc;
```

```
    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void sum() {
        assertEquals(5, calc.sum(2, 3));
        assertEquals(-3, calc.sum(-8, 5));
    }

    @Test
    void multi() {
        assertEquals(8, calc.multi(2, 4));
        assertEquals(40, calc.multi(-8, -5));
        assertEquals(0, calc.multi(0, -5));
    }
}
```

The final test show us that our tests passed and we have **probably** correct code.

🕮 6.2.6

If the expected value differs from the obtained value, an error (**AssertionError**) is generated, usually with a message why it is unsuccessful:

```
org.opentest4j.AssertionFailedError:
Expected :1
Actual   :0
<Click to see difference>

   at
org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:
55)
   at
org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.j
ava:195)
```

```
   at
org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.j
ava:152)
   at
org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.j
ava:147)
```

It should be noted that the system cannot identify that this is a tester error and if the tester makes a mistake when entering the expected value, the test tool attributes a program error.

It should be noted that the test lists only and mismatch between the expected and the obtained value.

Accuracy in Tests

Chapter 7

7.1 Real numbers in test

🕮 7.1.1

Modify the previous program and tests to use decimal/real numbers.

The original task was to create a Calculator class with methods:

- sum - adds two integers obtained as parameters and returns the result as an integer,
- multi - multiplies two integers obtained as parameters and returns the result as an integer,

Write tests to verify that the modified class you created is working properly.

🕮 7.1.2

The modified class has following form:

```
public class Calculator {

    public double sum(double a, double b) {
        return a + b;
    }

    public double multi(double a, double b) {
        return a * b;
    }
}
```

🕮 7.1.3

The accuracy of real numbers processing is often a problematic part of calculation using digital computers.

According with this fact, the tests used in programming, support the parameter accuracy as the third parameter in **assertEquals**, e.g.:

```
assertEquals(8, calc.multi(2, 4), 0.001);
```

If the result of inspected method and the result set by tester differ by less than the specified accuracy, the test is passed.

🕮 7.1.4

We prepare the tests for modified Calculator:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void sum() {
        assertEquals(5.5, calc.sum(2.5, 3),0.0001);
        assertEquals(-3.01, calc.sum(-8.01, 5),0.0001);
    }

    @Test
    void multi() {
        assertEquals(6.25, calc.multi(2.5, 2.5),0.0001);
        assertEquals(40, calc.multi(-8.0001, -5),0.01);
        assertEquals(0, calc.multi(0, -5.99),0.0001);
    }
}
```

🕮 7.1.5

Look to the second test in multiplication:

```
    @Test
    void multi() {
        assertEquals(6.25, calc.multi(2.5, 2.5),0.0001);
        assertEquals(40, calc.multi(-8.0001, -5),0.01);
        assertEquals(0, calc.multi(0, -5.99),0.0001);
```

```
    }
}
```

Even though the result of multiplication is 40.0005, the test ignores the difference between the expected result (40) and the actual result. The difference is considered to be irrelevant because the acceptable deviation is 0.01.

🕮 7.1.6

Be carefully:

if we allow big inaccuracy, system will pass this test as well:

```
assertEquals(-5, calc.sum(-8, 5), 3);
```

Exceptions

Chapter 8

8.1 Division by zero

🕮 8.1.1

Enrich the class **Calculator** to the division and solve situation with division by zero.

We can add new method to get quotient:

```
public class Calculator {

    public double sum(double a, double b) {
        return a + b;
    }

    public double multi(double a, double b) {
        return a * b;
    }

    public double quotient(double a, double b) {
       if (b!=0)
            return a / b;
    }
}
```

The result of new method is decimal value - it is defined via type of method.

To return some text in the form of "do not divide by zero" is therefore quite problematic.

But we can create a “managed” exception that can be caught and handled in the code using the Calculator.

🕮 8.1.2

The exception generation is common method in work with classes and methods. We can apply it using keyword **throw**.

```
public double quotient(double a, double b) {
    if (b!=0)
        return a / b;
    else
        throw new IllegalArgumentException("zero division");
}
```

This exception is catch in application using:

```
public static void main(String[] args) {
    Calculator calc = new Calculator();

    try {
        calc.quotient(4,0);
    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
```

The output is:

```
java.lang.IllegalArgumentException: zero division
```

📖 8.1.3

Exceptions testing is important and common. This type of testing needs special method to process the returned exceptions.The method **assertThrows** is used to assert that the supplied executable will throw an exception of the **expectedType**. If there is no exception of expectedType, the method will fail.

The definition of assertThrow consists of two parameters:

```
public static void assertThrows(Class<? extends Throwable>
expectedType, Executable executable)
```

Second part is defined as executable. We can use the lambda notation and set this parameter as

```
() -> method()
```

The final code has following form:

```
assertThrows(IllegalArgumentException.class, () ->
calc.quotient(2, 0));
```

When writing tests, keep in mind that we separately test the values for the standard result and the values giving the exception.

```
@Test
public void quotient_common() {
    assertEquals(2,calc.quotient(2, 1));
    assertEquals(2,calc.quotient(3, 1.5),0.0001);
}
```

```
@Test
public void quotient_exception() {
    assertThrows(IllegalArgumentException.class, () ->
calc.quotient(2, 0));
}
```

Testing Methods

Chapter 9

9.1 Testing methods

🕮 9.1.1

In addition to the methods described above, we also have others available:

- **assertArrayEquals()** - return **true** if two arrays contain the same elements
- **assertNotEquals()** - return **true** if two values are **not** the same
- **assertNotNull()** - return **true** if the value is not null
- **assertNotSame()** - return **true** if two references don't address to the same object
- **assertSame()** - return **true** if two references address to the same object (compares with ==)
- **assertTrue()** - return **true** if result of expression or method is true
- **assertFalse()** - return **true** if result of expression or method is false

🕮 9.1.2

Example of assertions:

```
public class TestAssertions {
  @Test
  public void testAssertions() {
     String str1 = new String ("abc"), str2 = new String
("abc");
     String str3 = null;
     String str4 = "abc", str5 = "abc";

     int val1 = 5, val2 = 6;
     String[] expectedArray = {"one", "two", "three"};
     String[] resultArray = {"one", "two", "three"};
     //Check that two objects are equal
     assertEquals(str1, str2);
     //Check that a condition is true
     assertTrue (val1 < val2);
     //Check that a condition is false
     assertFalse(val1 > val2);
     //Check that an object isn't null
     assertNotNull(str1);
     //Check that an object is null
     assertNull(str3);
     //Check if two object references point to the same object
     assertSame(str4,str5);
```

```
     //Check if two object references not point to the same
object
     assertNotSame(str1,str3);
     //Check whether two arrays are equal to each other.
     assertArrayEquals(expectedArray, resultArray);
  }
}
```

PRISCILLA

priscilla.fitped.eu