Astroinformatics I (Semester 1 2024)

Software Projects

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

July 8, 2024

Motivation

When writing code, we want to make sure that our code **works** as it is supposed to do.

We also want to make sure that code is **readable** so we can find possible errors more easily, and can also avoid such errors as far as possible.

Good coding practices, as well as guidelines for **debugging and testing** our code are integral parts of software development. Each program needs to be thoroughly tested before release to a user community - even if that "user community" is only you.

Good Coding Practices

Following coding best practices is important to **ensure readability, quality of code, and scalability** during software development activities.
You and others wo work with your code will benefit from that.

PEP 8 (Python Enhancement Proposal 8) is a style guide for Python code. Written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan, PEP 8 provides a set of recommendations to write more readable and consistent code. It covers everything from how to name variables, to the maximum number of characters that a line should have.

Good Coding Practices

Following coding best practices is important to **ensure readability, quality of code, and scalability** during software development activities.
You and others wo work with your code will benefit from that.

PEP 8 (Python Enhancement Proposal 8) is a style guide for Python code. Written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan, PEP 8 provides a set of recommendations to write more readable and consistent code. It covers everything from how to name variables, to the maximum number of characters that a line should have.

While not mandatory, much of the Python community uses PEP 8. So it is highly advisable to follow the guidelines. By doing that, you will improve your credentials as a professional programmer.

Good Coding Practices

Following coding best practices is important to **ensure readability, quality of code, and scalability** during software development activities.
You and others wo work with your code will benefit from that.

PEP 8 (Python Enhancement Proposal 8) is a style guide for Python code. Written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan, PEP 8 provides a set of recommendations to write more readable and consistent code. It covers everything from how to name variables, to the maximum number of characters that a line should have.

While not mandatory, much of the Python community uses PEP 8. So it is highly advisable to follow the guidelines. By doing that, you will improve your credentials as a professional programmer.

We won't cover here the complete PEP 8 but will give an overview of some of the most relevant coding practices.

Good Coding Practices

Indentation

Indentation refers to the spaces or tabs at the beginning of a code line. While indentation in other programming languages is only providing readability, indentation in Python is mandatory. Python uses indentation to open a block of code.

Good Coding Practices

Indentation

Indentation refers to the spaces or tabs at the beginning of a code line. While indentation in other programming languages is only providing readability, indentation in Python is mandatory. Python uses indentation to open a block of code.

PEP 8 recommendation:
Use 4 spaces per indentation level. Tabs should be used only in code that is already indented with tabs. Python disallows mixing tabs and spaces for indentation.
Continuation lines should align wrapped elements either vertically using Python's line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent, there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line:

Good Coding Practices

Indentation

```
# Correct:

# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Distinguish arguments by an extra level of indentation.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Good Coding Practices

Indentation

```
# Wrong:

# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Good Coding Practices

Maximum line length

PEP 8 recommends a maximum line length of 79 characters (72 for comments/ docstrings).

Good Coding Practices

Maximum line length

PEP 8 recommends a maximum line length of 79 characters (72 for comments/ docstrings).

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Backslashes may still be appropriate at times. For example, long, multiple with-statements could not use implicit continuation before Python 3.10, so backslashes were acceptable for that case:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Good Coding Practices

Source File Encoding

Code in the core Python distribution should always use UTF-8, and should not have an encoding declaration.

Good Coding Practices

Source File Encoding

Code in the core Python distribution should always use UTF-8, and should not have an encoding declaration.

Use non-ASCII characters sparingly, preferably only to denote places and human names. If using non-ASCII characters as data, avoid Unicode characters and byte order marks.

Good Coding Practices

Source File Encoding

Code in the core Python distribution should always use UTF-8, and should not have an encoding declaration.

Use non-ASCII characters sparingly, preferably only to denote places and human names. If using non-ASCII characters as data, avoid Unicode characters and byte order marks.

All identifiers (variable names, function names) must use ASCII-only identifiers, and should use English words wherever feasible. Remember that for special characters like symbols in titles, labels and legends, LaTeX can be used.

Good Coding Practices

Imports

Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Good Coding Practices

Imports

Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be on separate lines:

```
# Correct:
import os
import sys

# Wrong:
import sys, os
```

It's okay to use this though:

```
# Correct:
from subprocess import Popen, PIPE
```

Comments and Documentation

While **comments** are important for other developers you are working with to understand your code, **documentation** is intended to help a community of potential users how to use your software. This is a critical step: no matter how good your software is, if the documentation is not good enough, or even worse, missing, people will not use it.

Good Coding Practices

Comments

Comments are very important to make annotations for future readers of our code. Although it is difficult to define how the code should be commented on, there are certain guidelines that we can follow:

Good Coding Practices

Comments

Comments are very important to make annotations for future readers of our code. Although it is difficult to define how the code should be commented on, there are certain guidelines that we can follow:

Always make a priority of keeping the comments **up-to-date** when the code changes: Comments that contradict the code are worse than no comments.

Good Coding Practices

Comments

Comments are very important to make annotations for future readers of our code. Although it is difficult to define how the code should be commented on, there are certain guidelines that we can follow:

Always make a priority of keeping the comments **up-to-date** when the code changes: Comments that contradict the code are worse than no comments.

Ensure that your comments are **clear and easily understandable** to other speakers of the language you are writing in.
Python coders from non-English speaking countries: please write your comments in **English**, unless you are absolutely sure that the code will never be read by people who don't speak your language.

Good Coding Practices

Comments

There are two types of comments: block comments and inline comments.

Good Coding Practices

Comments

There are two types of comments: block comments and inline comments.

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. A block comment explains the code that follows it. Typically, you indent a block comment at the same level as the code block. Each line of a block comment starts with a # (hash) and a single space, as follows: They often consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.

```
# This is a block comment
print('Welcome to the coding session')
```

Good Coding Practices

Comments

On the other hand, **inline comments** appear at the same level as code. Inline comments should be separated by at least two spaces from the statement. Like block comments, inline comments begin with a single #, followed by a space and a text string. They should be used sparingly and they should not be used to state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, a comment like that is useful:

```
x = x + 1                 # Compensate for border
```

Good Coding Practices

Documentation

A **docstring** is a string literal that occurs as the first statement in a module, function, class, or method definition. Typically, you use a documentation string to automatically generate the code documentation. As such, a docstring can be accessed at run-time using the `obj.__doc__` special attribute of that object.

For consistency, docstrings are always enclosed in triple double quotes (""").

There are two forms of docstrings: one-liners and multi-line docstrings. One-liners are for really obvious cases. They should really fit on one line.

Good Coding Practices

Documentation

The following example illustrates a **one-line docstring** in the `multiply()` function:

```
def multiply(x, y):
    """ Return the product of two numbers """
    result = x * y
    return result
```

Documentation

On the other hand, a **multi-line docstring** can span multiple lines. Such a docstring should document the code function and command line syntax, environment variables, and files. Usage messages can be fairly elaborate and should be sufficient for a new user to use the command properly.

```
def calculate_salary(hours, price=20):
    """ Return the salary according to the hours worked

    Arguments:
    hours: total hours investe
    price: price of each hours worked. Minimum price is 20 dollars
    """
    salary = hours * price
    return salary
```

Good Coding Practices

Documentation

Especially if you are working on a new project, it's always a good practice to include a README file. These files are normally the main entry point for readers of your code. They include general information to both users and maintainers of the project.

Good Coding Practices

Documentation

Especially if you are working on a new project, it's always a good practice to include a README file. These files are normally the main entry point for readers of your code. They include general information to both users and maintainers of the project.

While concise, the README file should explain the project's purpose, the URL of the software's main source, and credit information.

Equally, you should always include a `setup.py` file, which ensures that the software or library has been packaged and distributed with Distulils, which is the standard for distributing Python modules.

Good Coding Practices

Documentation

Especially if you are working on a new project, it's always a good practice to include a README file. These files are normally the main entry point for readers of your code. They include general information to both users and maintainers of the project.

While concise, the README file should explain the project's purpose, the URL of the software's main source, and credit information.

Equally, you should always include a `setup.py` file, which ensures that the software or library has been packaged and distributed with Distulils, which is the standard for distributing Python modules.

Moreover, if your software requires other dependencies or packages to run, you should include a `requirements.txt` files, which describes the dependencies needed and their versions.

Good Coding Practices

Documentation

Especially if you are working on a new project, it's always a good practice to include a README file. These files are normally the main entry point for readers of your code. They include general information to both users and maintainers of the project.

While concise, the README file should explain the project's purpose, the URL of the software's main source, and credit information.

Equally, you should always include a `setup.py` file, which ensures that the software or library has been packaged and distributed with Distulils, which is the standard for distributing Python modules.

Moreover, if your software requires other dependencies or packages to run, you should include a `requirements.txt` files, which describes the dependencies needed and their versions.

Finally, you are free to add any other information you may think it's relevant to potential users. For example, a good practice is to provide examples of how your package or functions work.

Naming Conventions

There are a lot of different naming styles. It is helpful to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

Good Coding Practices

Naming Conventions

There are a lot of different naming styles. It is helpful to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

`b` (single lowercase letter)

`B` (single uppercase letter)

`lowercase`

`lower_case_with_underscores`

`UPPERCASE`

`UPPER_CASE_WITH_UNDERSCORES`

`CapitalizedWords` (or CapWords, or CamelCase - so named because of the bumpy look of its letters). This is also sometimes known as StudlyCaps.

Good Coding Practices

Naming Conventions

There are a lot of different naming styles. It is helpful to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

`b` (single lowercase letter)

`B` (single uppercase letter)

`lowercase`

`lower_case_with_underscores`

`UPPERCASE`

`UPPER_CASE_WITH_UNDERSCORES`

`CapitalizedWords` (or CapWords, or CamelCase - so named because of the bumpy look of its letters). This is also sometimes known as StudlyCaps.

Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus HTTPServerError is better than HttpServerError.

Good Coding Practices

Naming Conventions

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

`mixedCase` is allowed only in contexts where that's already the prevailing style, to retain backwards compatibility.

Good Coding Practices

Naming Conventions

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

`mixedCase` is allowed only in contexts where that's already the prevailing style, to retain backwards compatibility.

the complete guideline can be found at:
`https://peps.python.org/pep-0008/`

Version Control

The basic idea of **Version control systems** is to conserve the entire history of the software. There are many benefits:

Using a version control system facilitates development in a team.

It is easy to compare different versions. It is also possible to develop several versions in parallel (*branches*).

In case of encountering a bug, it is easy to detect when the feature was introduced and why.

Version Control

The basic idea of **Version control systems** is to conserve the entire history of the software. There are many benefits:

Using a version control system facilitates development in a team.

It is easy to compare different versions. It is also possible to develop several versions in parallel (*branches*).

In case of encountering a bug, it is easy to detect when the feature was introduced and why.

It is recommended to use a version control system even for small software projects.
With knowing how to use it, the overhead is only small, but the benefit large.

Version Control

There exists many version control systems:

Centralised: CVS, SVN, etc.

Distributed: git, mercurial, etc.

Version Control

There exists many version control systems:

Centralised: CVS, SVN, etc.

Distributed: git, mercurial, etc.

We focus here on **git**.

GVersion Control

The exact **workflow** varies depending on the system used, but generally follows this scheme:

- Creation of the main branch and initial import of the code
- Fork (copy) of the main branch into a feature branch
- Code development
- Review the changes
- Commit of the changes into the feature branch. (A commit saves the changes along with a comment describing them.)
- Repeat the last three steps until the feature is complete
- Checkout the main branch
- Merge the changes from the feature branch into the main branch
- Make a new release (if need be) of this main branch
- Fork the main branch to develop a new feature

Having this sort of workflow is especially important when working with a team

Version Control

What is Git?

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for:

- Tracking code changes
- Tracking who made changes
- Coding collaboration

What does Git do?

- Manage projects with Repositories
- Clone a project to work on a local copy
- Control and track changes with Staging and Committing
- Branch and Merge to allow for work on different parts and versions of a project
- Pull the latest version of the project to a local copy

What is GitHub?

Git is a version control system that allows developers to track changes in their code. GitHub is a web-based hosting service for git repositories. GitHub makes Git more user-friendly and also provides a platform for developers to share code with others.

In simple terms, you can use git without Github, but you cannot use GitHub without Git. Take note of the following illustration for further information on the distinctions between the two.

What is GitHub?

Git is a version control system that allows developers to track changes in their code. GitHub is a web-based hosting service for git repositories. GitHub makes Git more user-friendly and also provides a platform for developers to share code with others.

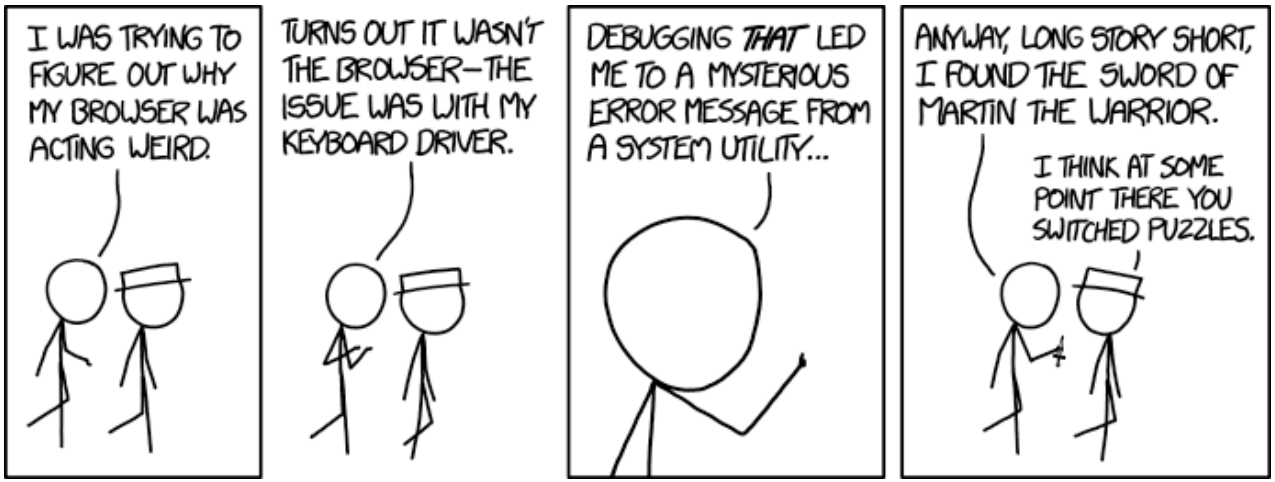
In simple terms, you can use git without Github, but you cannot use GitHub without Git. Take note of the following illustration for further information on the distinctions between the two.

We will continue with a GitHub tutorial in this week's **tutorial session**.

Testing vs. Debugging

Testing and debugging are two of the most common processes to ensure code quality. At first glance, they may appear similar, but they are indeed quite different.

Testing is the process by which we find defects in the functionality of the code. **Debugging** is locating their cause in the code and fixing them. Let's look at each in detail.

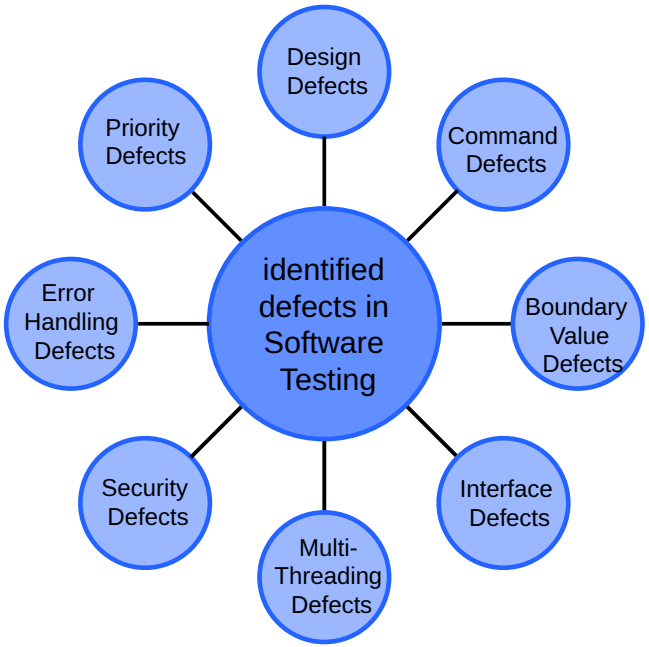


Software Testing

Software testing is a process of identifying defects in the software product. It is performed to validate the behavior of the software or the application compared to requirements.

In other words, software testing is a **collection of techniques to determine the accuracy** of the application under the predefined specification but it cannot identify all the defects of the software.

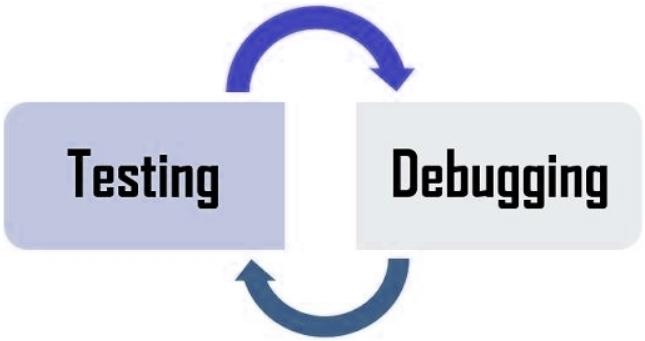
Each software needs to be tested before being put into production.



Debugging

In the software development process, debugging includes detecting and modifying code errors in a software program.

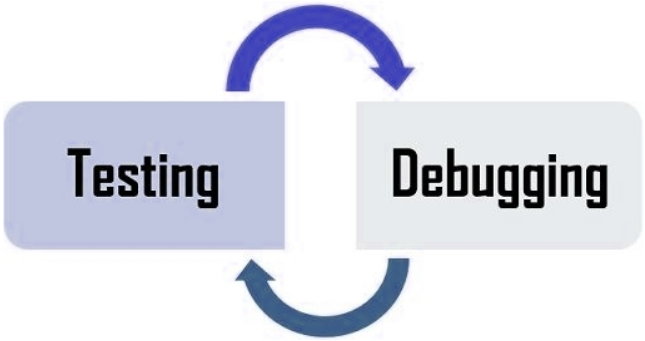
For doing so, the code needs to be analyzed rigorously. Often a **debugger tool** is used to step line-by-line through the source code. A debugger also allows developers to examine variables and evaluate expressions, thus enabling them to **locate the problem** quickly. The developer **changes** the code and then **re-checks** whether the defect has been deleted whenever the bug or error is found.



Debugging

example:

During testing, it was found that the login fails because the search for users is not successful. During debugging it was found that the wrong user account table is referenced. It was also found that this was causing another bug that was previously known: users were not notified to change their password once a month. Fixing the reference to the user table solves both the login problem and the password reminder problem.



Testing and Debugging Compared

	Testing	Debugging
Objective	The objective of testing is to identify defects in the software.	The purpose of debugging is to fix errors observed in testing.
Process	It is the process to identify the failure of code.	It is the process of fixing mistakes in the code.
Programming Knowledge	Most testing requires no knowledge of the source code.	Debugging requires a proper understanding of the source code.
Performed By	A tester performs testing. Testing can be carried out by insiders and outsiders.	Debugging is performed by a programmer. Only an insider does the debugging.
Design Knowledge	Design knowledge is not needed in the testing process.	Design knowledge is not needed but recommended as wrongly fixed bugs can cause other defects or maintenance problems.
Automation	Testing can be manual or automated.	Debugging is always manual.
Initiation	Testers can only initiate testing after developers write the code.	Debugging begins after a failed test case execution.

Software Testing

Software Testing is a method to check whether the actual software product matches expected **requirements**.
It involves manual or automated tools to evaluate one or more properties of interest. The **purpose** of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Software Testing

Software Testing is a method to check whether the actual software product matches expected **requirements**.
It involves manual or automated tools to evaluate one or more properties of interest. The **purpose** of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Why is Software Testing important?

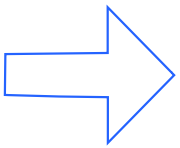
Software Testing is important because if there are any bugs or errors in the software, they can be **identified early** and solved before software is delivered. Properly tested software **ensures reliability, security and high performance** which further results in **time saving, cost effectiveness and customer satisfaction**.

Software Testing

Software Testing is a method to check whether the actual software product matches expected **requirements**.
It involves manual or automated tools to evaluate one or more properties of interest. The **purpose** of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Why Software Testing is Important?

Software Testing is important because if there are any bugs or errors in the software, they can be **identified early** and solved before software is delivered. Properly tested software **ensures reliability, security and high performance** which further results in **time saving, cost effectiveness and customer satisfaction**.



This as well applies to scientific code where you are the "customer" yourself!

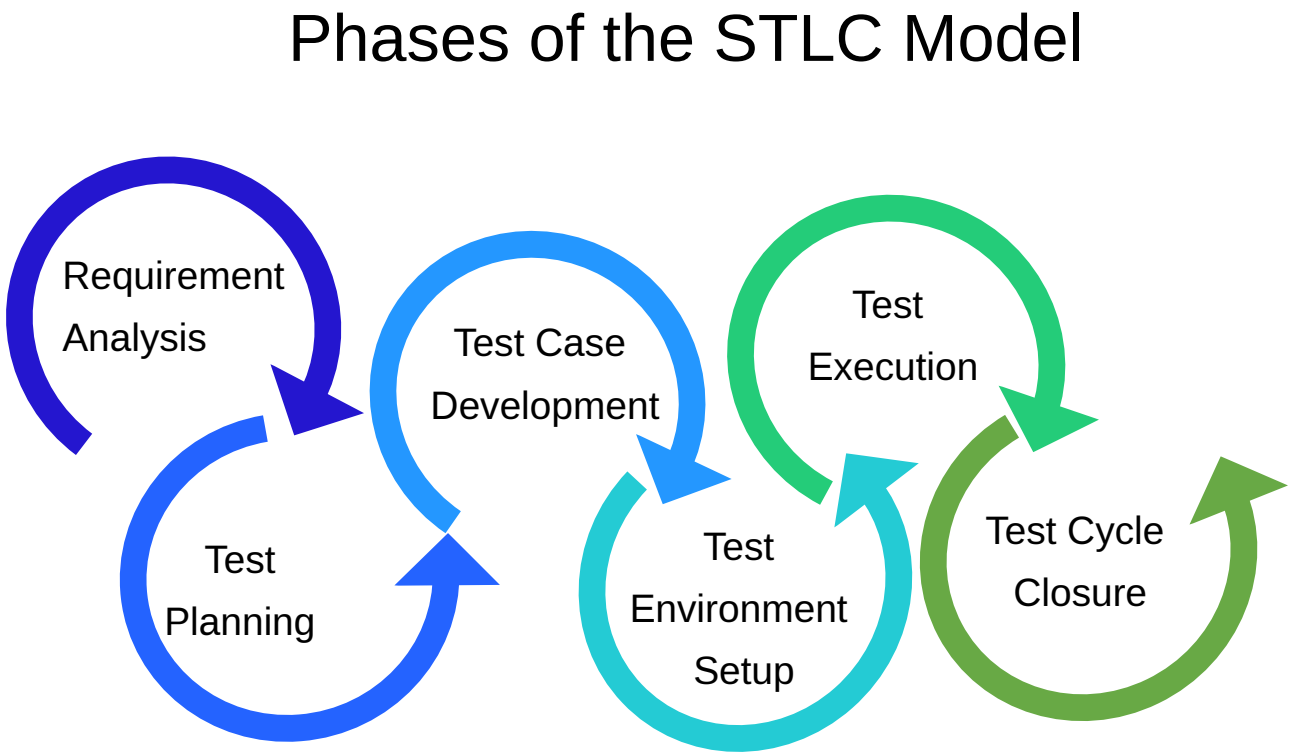
The Software Testing Life Cycle

The **Software Testing Life Cycle (STLC)** is a sequence of specific activities conducted during the testing process to ensure software quality goals are met.

STLC involves both **verification** and **validation** activities. It consists of a series of activities carried out methodologically to help certify your software product.

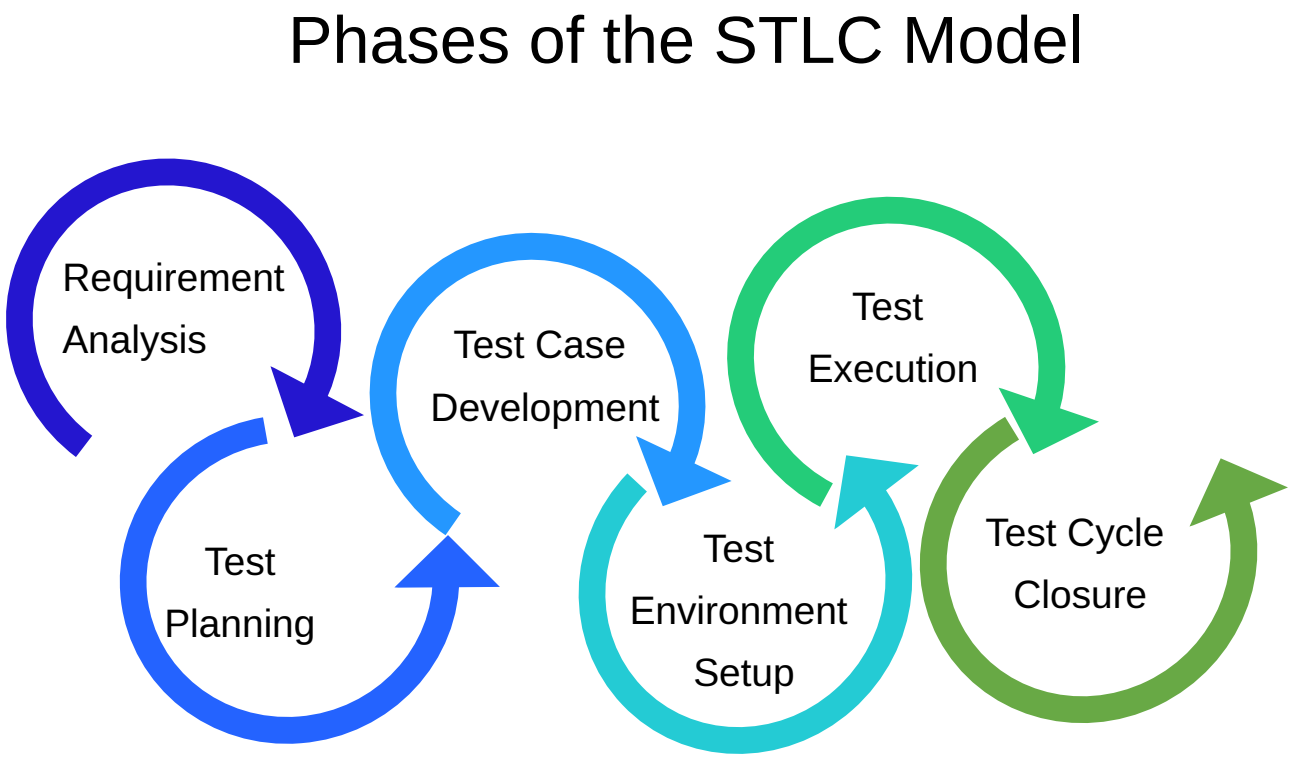
The Software Testing Life Cycle

There are following six major **phases** in every Software Testing Life Cycle Model (STLC Model):



The Software Testing Life Cycle

There are following six major **phases** in every Software Testing Life Cycle Model (STLC Model):



They have **Entry and Exit Criteria**!

The Software Testing Life Cycle

What are the **Entry and Exit Criteria** in STLC?

The **Entry Criteria** gives the prerequisite items that must be completed before testing can begin.
The **Exit Criteria** defines the items that must be completed before testing can be concluded

The Software Testing Life Cycle

What are the **Entry and Exit Criteria** in STLC?

The **Entry Criteria** gives the prerequisite items that must be completed before testing can begin.
The **Exit Criteria** defines the items that must be completed before testing can be concluded

We have Entry and Exit Criteria for all levels in the Software Testing Life Cycle (STLC).
In an ideal world, you will not enter the next stage until the exit criteria for the previous stage is met. But practically this is not always possible. So for this class, we will focus on activities and deliverables for the different stages in STLC life cycle.

The Software Testing Life Cycle

Phase 1: Requirement Analysis

Requirement Analysis, also known as Requirement Phase Testing, is the phase in which the test team studies the requirements to identify testable requirements. Requirements could be either functional or non-functional.

Activities in Requirement Analysis:

- Identify types of tests to be performed.
- Gather details about testing priorities and focus.
- Prepare Requirement Traceability Matrix (RTM).
- Identify test environment details where testing is supposed to be carried out.
- Automation feasibility analysis (if required).

The Software Testing Life Cycle

Phase 1: Requirement Analysis

Requirement Analysis, also known as Requirement Phase Testing, is the phase in which the test team studies the requirements to identify testable requirements. Requirements could be either functional or non-functional.

Deliverables of Requirement Phase Testing:

- RTM
- Automation feasibility report (if applicable)

The Software Testing Life Cycle

Phase 2: Test Planning

Test Planning in STLC is a phase in which a Senior QA manager determines the test plan strategy along with efforts and cost estimates for the project. Moreover, the resources, test environment, test limitations and the testing schedule are also determined. The Test Plan gets prepared and finalized in the same phase.

Activities in Test Planning:

- Preparation of test plan/strategy document for various types of testing
- Test tool selection
- Test effort estimation
- Resource planning and determining roles and responsibilities.

The Software Testing Life Cycle

Phase 2: Test Planning

Test Planning in STLC is a phase in which a Senior QA manager determines the test plan strategy along with efforts and cost estimates for the project. Moreover, the resources, test environment, test limitations and the testing schedule are also determined. The Test Plan gets prepared and finalized in the same phase.

Deliverables of Test Planning:

- Test plan/strategy document.
- Effort estimation document.

The Software Testing Life Cycle

Phase 3: Test Case Development

The Test Case Development Phase involves the creation, verification and rework of test cases and test scripts after the test plan is ready. Initially, test data is identified, created and reviewed and then reworked based on the preconditions. Then test cases for individual units are developed.

Activities in Test Case Development:

- Create test cases, automation scripts (if applicable)
- Review and baseline test cases and scripts
- Create test data (if test environment is available)

The Software Testing Life Cycle

Phase 3: Test Case Development

The Test Case Development Phase involves the creation, verification and rework of test cases and test scripts after the test plan is ready. Initially, test data is identified, created and reviewed and then reworked based on the preconditions. Then test cases for individual units are developed.

Activities in Test Case Development:

- Create test cases, automation scripts (if applicable)
- Review and baseline test cases and scripts
- Create test data (if test environment is available)

Deliverables of Test Case Development:

- Test cases/scripts
- Test data

The Software Testing Life Cycle

Phase 4: Test Environment setup

This phase decides the software and hardware conditions under which a work product is tested. This can be done in parallel with the Test Case Development Phase. A readiness check of the given environment is required.

Activities in Test Environment Setup:

- Understand the required architecture, environment set-up and prepare hardware and software requirement list for the Test Environment.
- Setup test Environment and test data
- Perform rediness check on the build

The Software Testing Life Cycle

Phase 4: Test Environment setup

This phase decides the software and hardware conditions under which a work product is tested. This can be done in parallel with the Test Case Development Phase. A readiness check of the given environment is required.

Activities in Test Environment Setup:

- Understand the required architecture, environment set-up and prepare hardware and software requirement list for the Test Environment.
- Setup test Environment and test data
- Perform rediness check on the build

Deliverables of Test Environment Setup:

- Environment ready with test data set up
- Readiness check result.

The Software Testing Life Cycle

Phase 5: Test Execution

The Test Execution Phase is carried out by the testers in which testing of the software build is done based on test plans and test cases prepared. The process consists of test script execution, test script maintenance and bug reporting. If bugs are reported then it is reverted back to development team for correction and retesting will be performed.

Activities in Test Execution:

- Execute tests as per plan
- Document test results, and log defects for failed cases
- Map defects to test cases in RTM
- Retest the Defect fixes
- Track the defects to closure

The Software Testing Life Cycle

Phase 5: Test Execution

The Test Execution Phase is carried out by the testers in which testing of the software build is done based on test plans and test cases prepared. The process consists of test script execution, test script maintenance and bug reporting. If bugs are reported then it is reverted back to development team for correction and retesting will be performed.

Deliverables of Test Execution:

- Completed RTM with the execution status
- Test cases updated with results
- Defect reports

The Software Testing Life Cycle

Phase 6: Test Cycle Closure

Test Cycle Closure phase, the completion of test execution, involves several activities like test completion reporting, collection of test completion matrices and test results. Testing team members meet, discuss and analyze testing artifacts to identify strategies that have to be implemented in future, taking lessons from current test cycle.

Activities in Test Cycle Closure:

- Evaluate cycle completion criteria based on Time, Test coverage, Cost,Software, Critical Business Objectives, Quality
- Prepare test metrics based on the above parameters.
- Document the learning out of the project
- Prepare Test closure report
- Qualitative and quantitative reporting of quality of the work product to the customer.
- Test result analysis: defect distribution by type and severity.

The Software Testing Life Cycle

Phase 6: Test Cycle Closure

Test Cycle Closure phase, the completion of test execution, involves several activities like test completion reporting, collection of test completion matrices and test results. Testing team members meet, discuss and analyze testing artifacts to identify strategies that have to be implemented in future, taking lessons from current test cycle.

Deliverables of Test Cycle Closure:

- Test Closure report
- Test metrics

Testing Concepts

After we have seen now how the general Software Testing Life Cycle looks like, we will take a look at a few **testing concepts** that implement this STLC.

Black Box Testing

Black Box Testing, also known as Behavioral Testing, is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths.

Black Box Testing **focuses on input and output** of software applications and it is entirely based on software requirements and specifications.

Black Box Testing

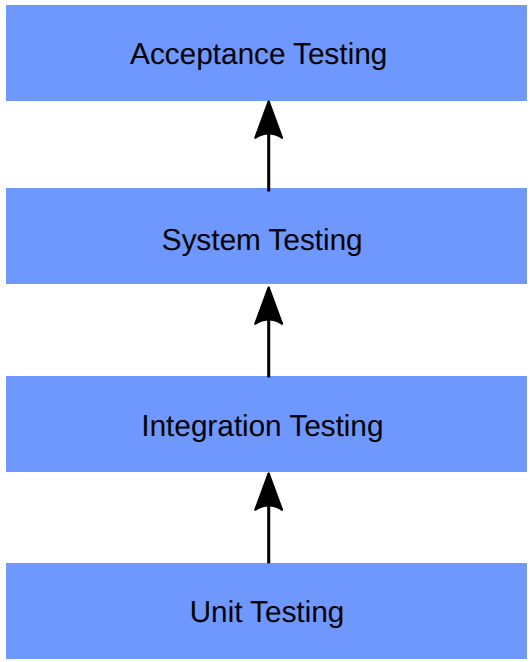
These are the general steps to follow when carrying out Black Box Testing in Software Engineering:

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether the software processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the software is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if there are any are fixed and re-tested.

Software Testing Hierarchy

As with almost any software engineering process, software testing has a prescribed order in which things should be done:

- **Unit testing** performed on each module or block of code during development. Unit Testing is normally done by the programmer who writes the code.
- **Integration testing** done before, during and after integration of a new module into the main software package. This involves testing of each individual code module.
- **System testing** done by a professional testing agent on the completed software product before it is used.
- **Acceptance testing** is beta testing done by the actual end users.



Unit Testing & Integration Testing

Unit Testing is a software development process in which the smallest testable parts of an application, called units, are individually scrutinized for proper operation. Software developers and sometimes QA staff complete unit tests during the development process.

Unit Testing & Integration Testing

Unit Testing is a software development process in which the smallest testable parts of an application, called units, are individually scrutinized for proper operation. Software developers and sometimes QA staff complete unit tests during the development process.

Integration Testing is defined as a type of testing where software modules are integrated logically and tested as a group. A typical software project consists of multiple software modules, coded by different programmers. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated

Unit Testing & Integration Testing

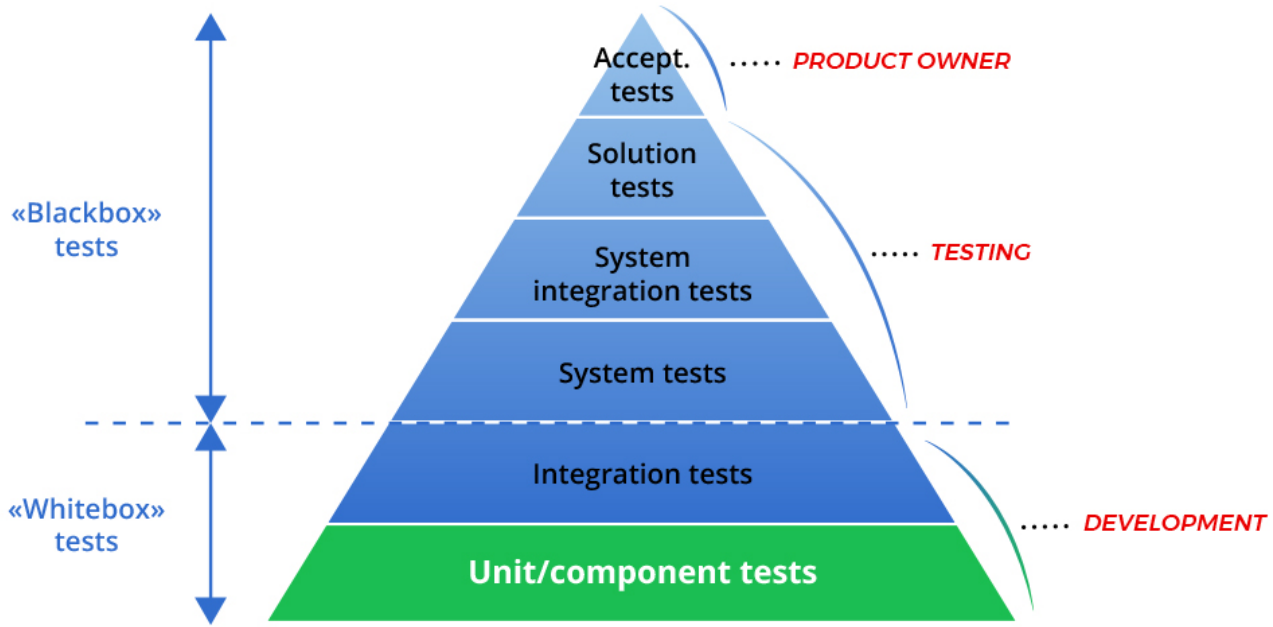
Unit Testing is a software development process in which the smallest testable parts of an application, called units, are individually scrutinized for proper operation. Software developers and sometimes QA staff complete unit tests during the development process.

Integration Testing is defined as a type of testing where software modules are integrated logically and tested as a group. A typical software project consists of multiple software modules, coded by different programmers. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated

Integration Testing focuses on checking data communication amongst these modules. Hence it is also termed as **I & T** (Integration and Testing), **String Testing** and sometimes **Thread Testing**.

Unit Testing

A unit test is a type of software test that focuses on **components** of a software product. The **objective** of a unit test is to test an entity in the code, ensure that it is coded correctly with no errors, and that it returns the expected outputs for all relevant inputs.



Unit Testing

Error-based unit tests should preferably be built by the developers who originally designed the code. Techniques include:

- Fault seeding: putting known bugs into the code and testing until they are found.
- Mutation testing: changing certain statements in the source code to see if the test code can detect errors. Mutation tests are expensive to run, especially in very large applications.
- Historical test data: uses historical information from previous test case executions to calculate the priority of each test case.

Unit Testing

Structural unit testing is a white box testing technique in which a developer designs test cases based on the internal structure of the code. The approach requires identifying all possible paths through the code. The tester selects test case inputs, executes them, and determines the appropriate output.

Primary structural testing techniques include:

- Statement, branch, and path testing: each statement, branch, or path in a program is executed by a test at least once. Statement testing is the most granular option.
- Conditional testing: allows a developer to selectively determine the path executed by a test, by executing code based on value comparisons.
- Expression testing: tests the application against different values of a regular expression.

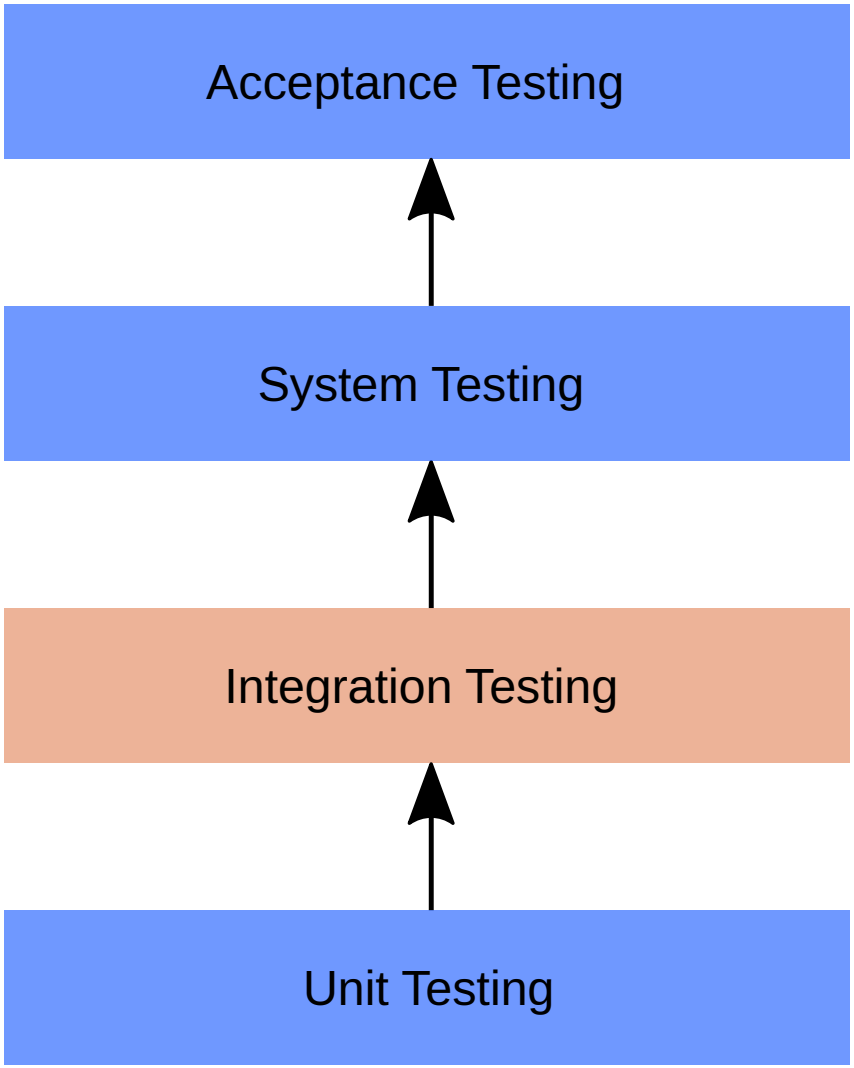
Unit Testing

Functional unit testing is a black box testing technique for testing the functionality of an application component.

Main functional techniques include:

- Input domain testing: tests the size and type of input objects and compares objects to equivalence classes.
- Boundary value analysis: tests are designed to check whether software correctly responds to inputs that go beyond boundary values.
- Syntax checking: tests that check whether the software correctly interprets input syntax.
- Equivalent partitioning: a software testing technique that divides the input data of a software unit into data partitions, applying test cases to each partition.

Integration Testing



Integration Testing

Why do Integration Testing?

Although each software module is unit tested, defects still exist for various reasons like

- A Module, in general, is designed by an individual software developer whose understanding and programming logic may differ from other programmers. Integration Testing becomes necessary to verify the software modules work in unity .
- At the time of module development, there are wide chances of change in requirements by the clients. These new requirements may not be unit tested and hence system integration Testing becomes necessary.
- Interfaces of the software modules with the database could be erroneous
- External Hardware interfaces, if any, could be erroneous
- Inadequate exception handling could cause issues.

Integration Testing

Integration Test Case differs from other test cases in the sense it focuses mainly on the **interfaces** between the modules. Here priority is to be given for the integrating links rather than the unit functions which are already tested.

Integration Testing

Integration Test Case differs from other test cases in the sense it focuses mainly on the **interfaces** between the modules. Here priority is to be given for the integrating links rather than the unit functions which are already tested.

example:

We have an application with 3 modules, being them 'Login Page', 'To-Do list', 'Archive To-Do list'. They are integrated logically. Unit tests were already carried out. We have to test how the data flow between 'To-Do list' and 'Archive To-Do list' works, and whether the 'Login Page' and 'To-Do list' work together.

Types of Integration Testing

In the **Incremental Testing** approach, testing is done by integrating two or more modules that are logically related to each other and then tested for proper functioning of the application. Then the other related modules are integrated incrementally and the process continues until all the logically related modules are integrated and tested successfully.

Types of Integration Testing

In the **Incremental Testing** approach, testing is done by integrating two or more modules that are logically related to each other and then tested for proper functioning of the application. Then the other related modules are integrated incrementally and the process continues until all the logically related modules are integrated and tested successfully.

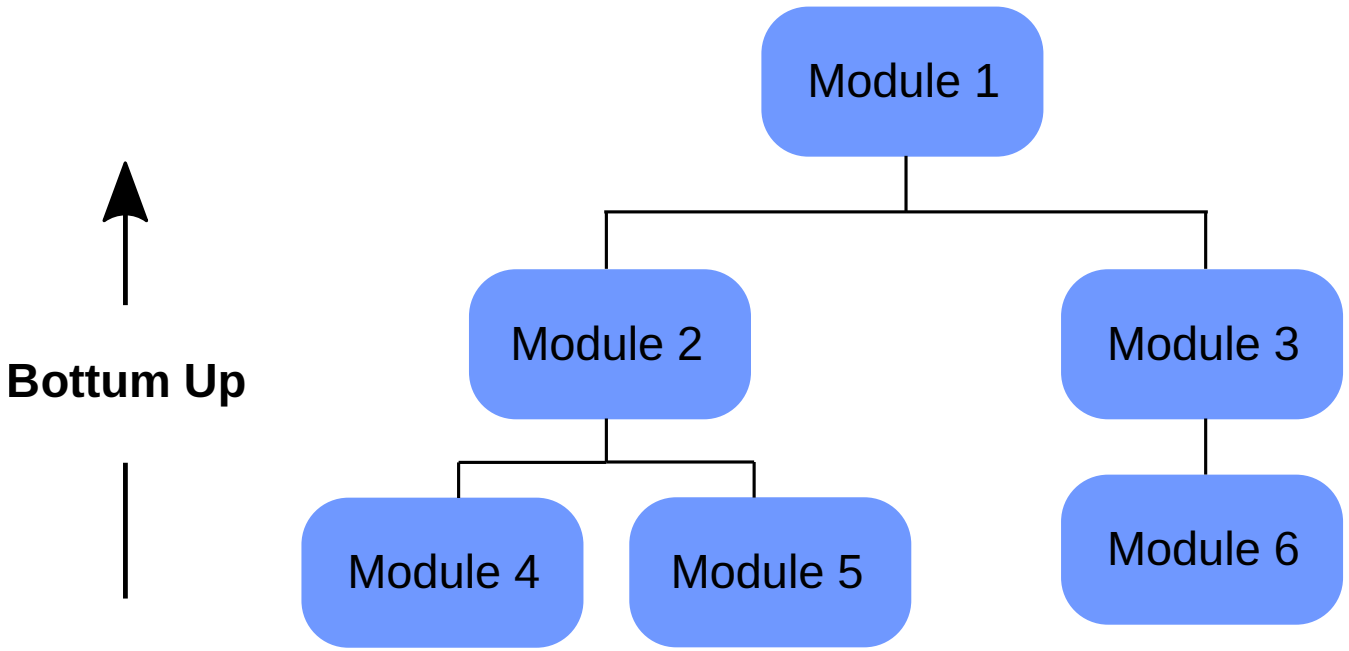
Software Engineering defines a variety of incremental strategies to execute Integration testing:

- Top-Down Approach
- Bottom-Up Approach
- Sandwich Approach: Combination of Top Down and Bottom Up

In the following we will see how the different strategies are executed and what are their limitations and advantages.

Bottom-Up Integration Testing

Bottom-Up Integration Testing is a strategy in which the **lower level modules are tested first**. These tested modules are then further used to facilitate the testing of higher level modules. The process continues until all modules at top level are tested. Once the lower level modules are tested and integrated, then the next level of modules are formed.



Bottom-Up Integration Testing

Advantages:

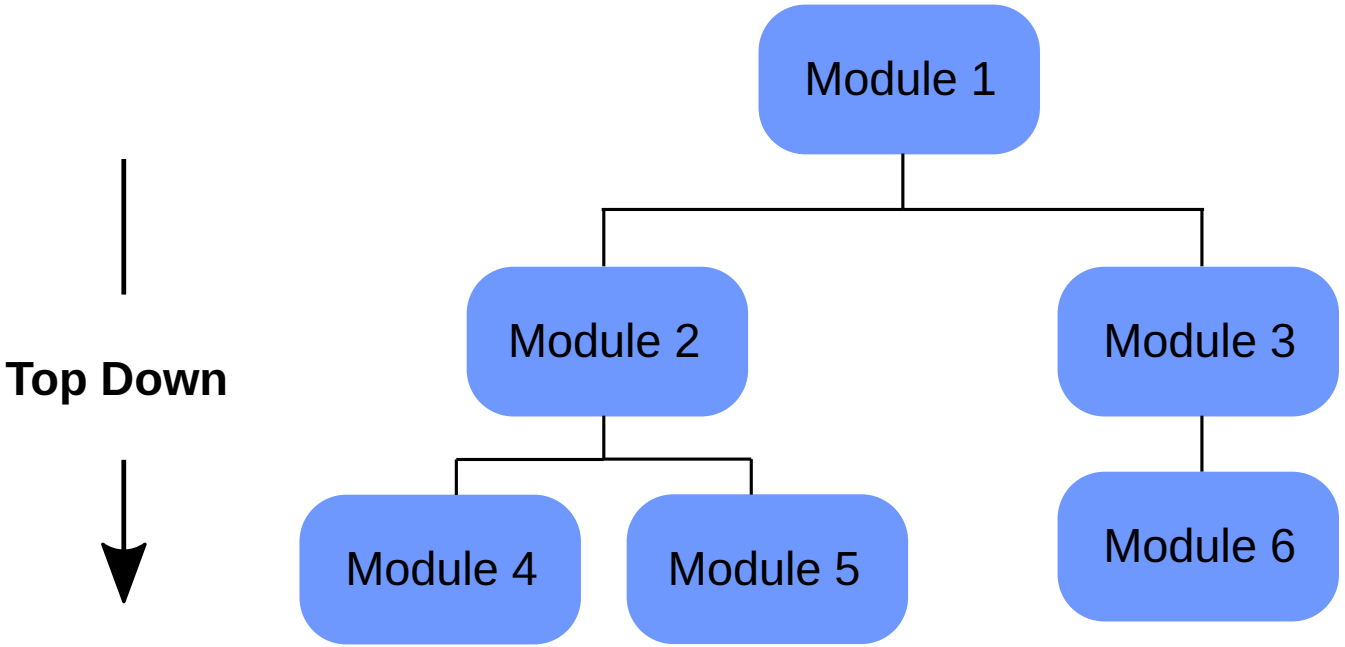
- Fault localization is easier.
- No time is wasted waiting for all modules to be developed unlike Big-bang approach

Disadvantages:

- Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
- An early prototype is not possible

Top-Down Integration Testing

Top Down Integration Testing is a method in which integration testing takes place from top to bottom following the control flow of software system. The higher level modules are tested first and then lower level modules are tested and integrated in order to check the software functionality. Stubs are used for testing if some modules are not ready.



Top-Down Integration Testing

Advantages:

- Possibility to obtain an early prototype.
- Critical Modules are tested on priority; major design flaws could be found and fixed first.

Disadvantages:

- Needs many Stubs.
- Modules at a lower level are tested inadequately.

Entry and Exit Criteria of Integration Testing

The entry and exit criteria to Integration Testing phase in any software development model are the following:

Entry Criteria:

- Unit tested components/modules
- All high prioritized bugs fixed and closed
- All modules to be code completed and integrated successfully.
- Integration test plan, test case, scenarios documented.
- Required test environment to be set up for integration testing

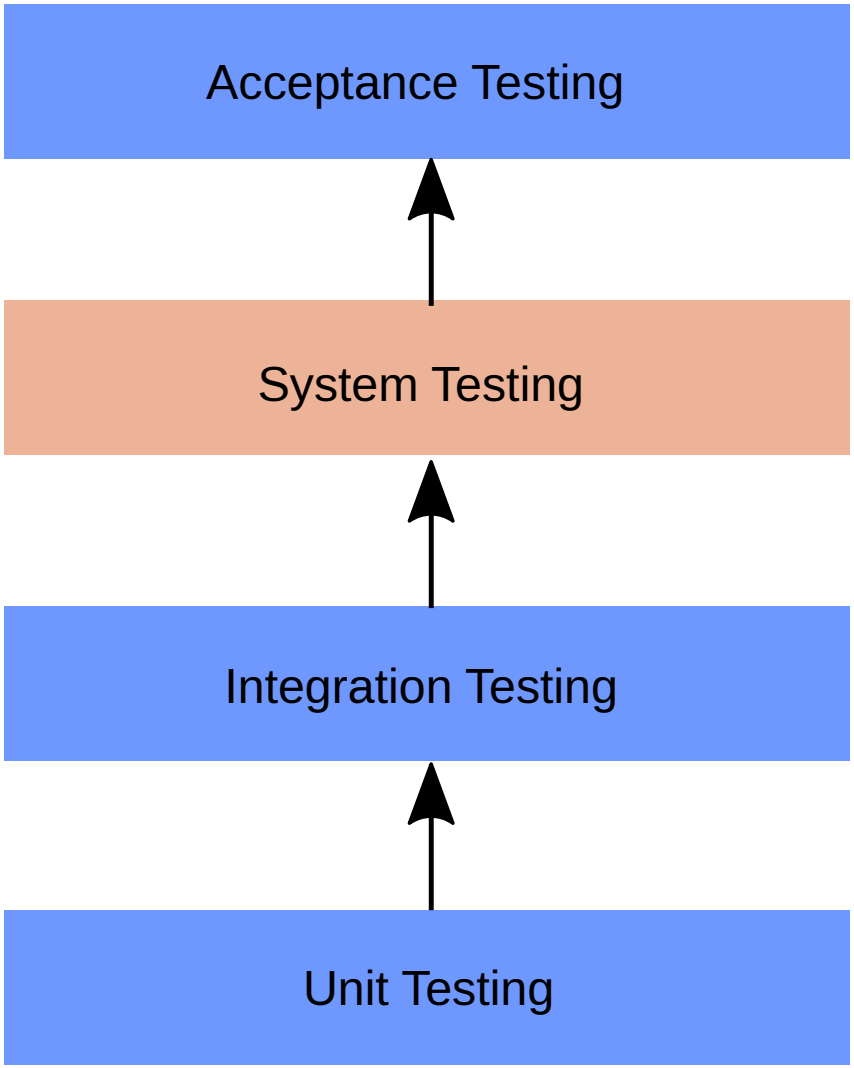
Exit Criteria:

- Successful testing of integrated application
- Executed test cases are documented
- All high prioritized bugs fixed and closed
- Technical documents to be submitted followed by release notes.

Best Practices/ Guidelines for Integration Testing

- First, determine the Integration Test Strategy that could be adopted and later prepare the test cases and test data accordingly.
- Study the Architecture design of the Application and identify the Critical Modules. These need to be tested on priority.
- Obtain the interface designs from the Architectural team and create test cases to verify all of the interfaces in detail. Interface to database/external hardware/software application must be tested in detail.
- After the test cases, its the test data which plays the critical role.
- Always have the mock data prepared, prior to executing. Do not select test data while executing the test cases.

System Testing



System Testing

System Testing is a level of testing that validates the complete and fully integrated software product. The purpose of system testin is to evaluate the end-to-end system specifications. Usually, the software is only one element of a larger computer-based system. Ultimately, the software is interfaced with other software/hardware systems. System Testing is defined as a series of different tests whose sole purpose is to exercise the full computer-based system.

System Testing involves testing the code for the following:

- Testing the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole. This is also called End to End testing scenario.
- Verify thorough testing of every input in the application to check for desired outputs.
- Evaluating the user experience with the application.

Regression Testing

So far we have handled testing as something that happens after our software is complete.

Regression Testing is a type of testing in the software development cycle that runs after every change to ensure that the change introduces no unintended breaks. Regression testing addresses a common issue that developers face: the emergence of old and new bugs with the introduction of new changes.

Regression Testing

So far we have handled testing as something that happens after our software is complete.

Regression Testing is a type of testing in the software development cycle that runs after every change to ensure that the change introduces no unintended breaks. Regression testing addresses a common issue that developers face: the emergence of old and new bugs with the introduction of new changes.

Regression testing is put in place to minimize such risks. Regression testing **provides overall stability** to the software application by keeping a check on the functionality of the existing features. It enables professionals to verify whether the previously developed and tested code stays operational when new features or code changes are introduced.

Software Performance Testing

So far we covered functional software testing techniques: They are put into place to ensure that our code runs as it is supposed to be, without bugs.

Software performance testing is a non-functional software testing technique that determines how the stability, speed, scalability, and responsiveness of an application holds up under a given workload. It's a key step in ensuring software quality, but unfortunately, is often seen as an afterthought, in isolation, and to begin once functional testing is completed, and in most cases, after the code is ready to release.

The goals of performance testing include evaluating application output, processing speed, data transfer velocity, network bandwidth usage, maximum concurrent users, memory utilization, workload efficiency, and command response times.

Summary

We have seen many ways on how to ensure the quality of our code using software testing mechanisms and debugging.

It is especially important that test cases, as well as test data (especially important for scientific software), need to be created carefully. Regarding **mock astronomical catalogs** with images, light curves and spectra, techniques like data augmentation and other ways on how to genereate test data, many resources can be found online.