DEGGENDORF INSTITUTE OF TECHNOLOGY

# Software Engineering

Prof. Dr. Christoph Schober

Department of Applied Computer Science,
Deggendorf Institute of Technology

*christoph.schober@th-deg.de*

## Software Testing

## Software Testing

### 1. Motivation

## Literature

[1] Vladimir Khorikov: *Unit Testing: Principles, Practises, and Patterns*. 1st ed. Manning Publications, 2020

[2] Ian Sommerville, *Engineering Software Productts: An Introduction to Modern Software Engineering*. 1st ed. Pearson Education, 2020

## Motivation

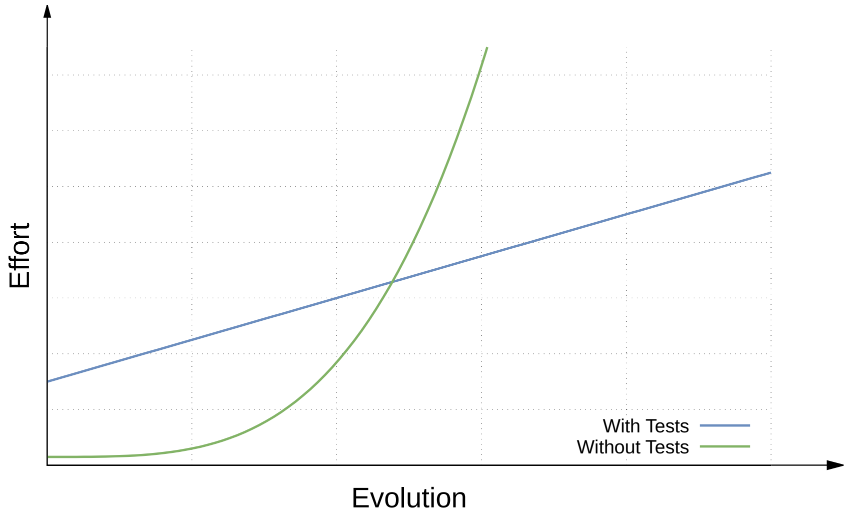
**Software Test**

A process of evaluating the functionality and performance of a software application or system.

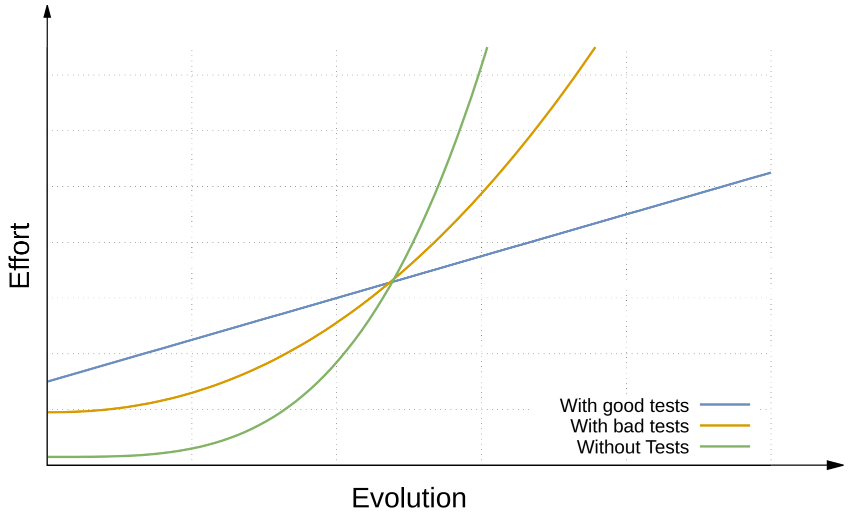
If the behavior of the program does not match the expected behavior, then this means that the program has bugs:

- *Programming errors:* Accidentally included faults in the code.
- *Understanding errors:* The expected behavior is misunderstood by the programmer.

## Motivation



## Motivation


Effort
Evolution
With good tests
With bad tests
Without Tests

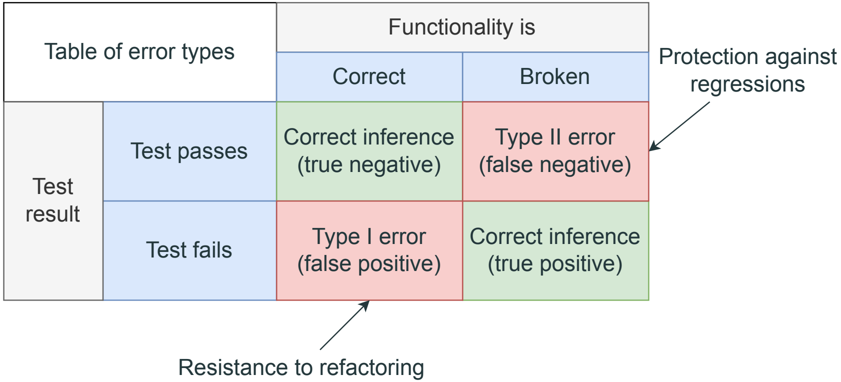
## Motivation

**The four pillars of a good test**

1. Protection against regressions[1]
2. Resistance to refactoring[2]
3. Fast feedback
4. Maintainability

[1]Code that worked before does not work anymore.
[2]Changing existing code without modifying its observable behavior.

## Motivation



| Table of error types | | Functionality is | |
|---|---|---|---|
| | | Correct | Broken |
| Test result | Test passes | Correct inference (true negative) | Type II error (false negative) |
| | Test fails | Type I error (false positive) | Correct inference (true positive) |

Protection against regressions
Resistance to refactoring

$$\text{Test accuracy} = \frac{\text{Signal (number of bugs found)}}{\text{Noise (number of false alarms raised)}}$$

## Motivation

There are three types of software tests:

- **Unit tests:** Test small program components (units) in isolation.
- **Integration tests:** Test the aspects of features as integrated code units.
- **End-to-end tests:** Test a ready-to-ship version of the whole software system.

*Note:* In this lecture, we cover unit tests only. Integration and end-to-end tests are discussed in the 4th semester in *Software Engineering*.

## Motivation

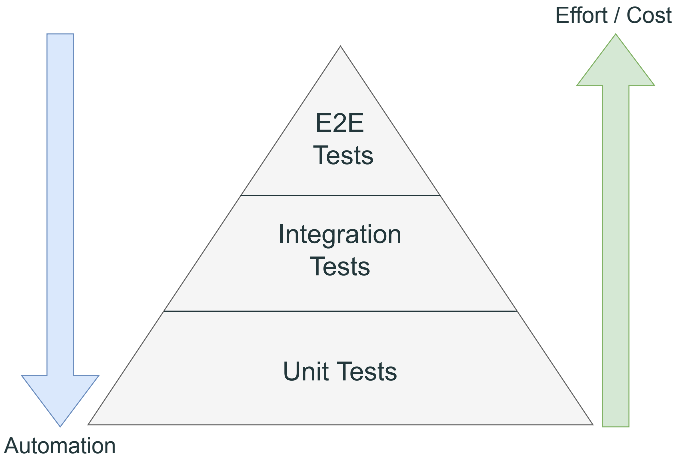
**Test automation**

Automated testing is based on the idea that tests should be executable. An executable test includes the input data to the unit that is being tested, the expected result, and a check that the unit returns the expected result.

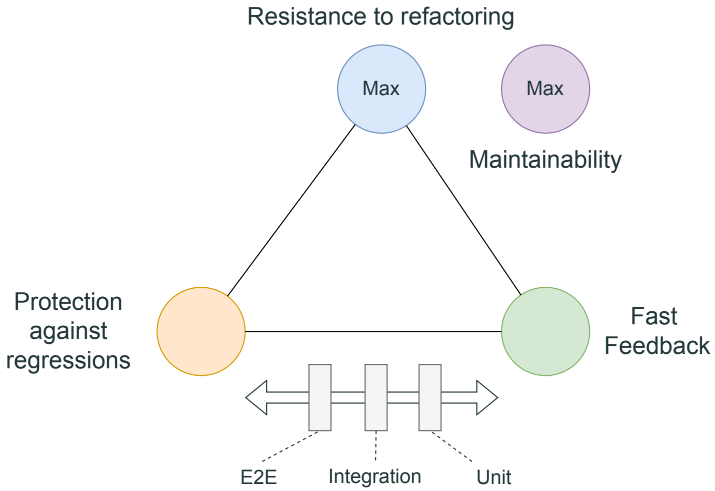
Popular software framework that support test automation:

- **Java:** jUnit, jBehave, TestNG
- **Python:** Unittest, PyTest, Behave
- **C++:** Catch2, Boost.Test

## Motivation


Effort / Cost
E2E Tests
Integration Tests
Unit Tests
Automation

## Motivation


Resistance to refactoring
Max
Max
Maintainability
Protection against regressions
Fast Feedback
E2E
Integration
Unit

## Motivation

**Pattern: AAA**

The AAA pattern provides a suitable test architecture:

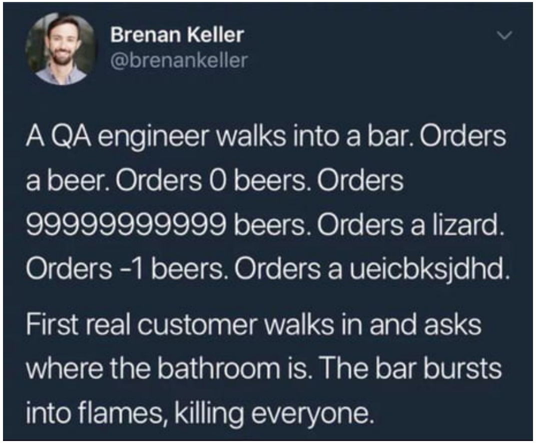
1. **Arrange:** Test setup, e.g., initialization of dependencies, definition of user inputs, generation of test data, etc.
2. **Act:** Test execution of the code to test[3].
3. **Assert:** Evaluation of the shown behavior.

[3] called *System Under Test* (SUT)

# Software Testing

## 2. Unit-Tests

## Unit-Tests


Brenan Keller
@brenankeller

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

## Unit-Tests

**Unit-Test**

A unit test is an automated test that:

- Verifies a small piece of code (also known as *unit*),
- Does it quickly,
- And does it in an isolated manner.

## Unit-Tests

**Basic assumption**: If a unit shows a certain behavior for a small amount of characteristic user inputs, it will show the same behavior for a bigger amount of inputs having the same characteristics as well.

## Unit-Tests

**Pytest**

A popular Python testing framework.

- Installation:

```
pip install pytest
```

- Execute tests:

```
pytest
```

`pytest` will run all files of the form `test_*.py` or `*_test.py` in the current directory and its subdirectories. From those files, it collects tests from `test`-prefixed functions.

## Unit-Tests

**Running example**: The greatest common divisor.

Recursive implementation in Python:

```python
def gcd(x, y):
    if y == 0:
        return x
    return gcd(y, x % y)
```

## Unit-Tests

Respective unit test:

```python
def test_gcd_small_numbers(self):
  # Arrange
  x = 9
  y = 6
  expected_result = 3

  # Act
  actual_result = gcd(x,y)

  # Assert
  assert actual_result == expected_result
```

## Unit-Tests

`pytest` allows to use the standard Python `assert`[4] for verifying expectations and values in Python tests.

```python
x = "hello"
assert x == "hello"     # nothing happens
assert x == "goodbye"  # AssertionError is raised
```

To write assertions about raised exceptions, use `pytest.raises()` as follows:

```python
import pytest

def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

[4]Tests if a provided condition returns `True`, if not, the program will raise an `AssertionError`.

## Unit-Tests

**Test Fixtures**

A well defined test environment, in which software tests consistently yield the same results when executed repeatedly.

Examples:

- Input values and expected results.
- Setup of mocks and stubs
- Initializing a database with fixed tables and records.
- Text files with predefined content.

## Unit-Tests

*Note:* The reuse of test fixtures is **highly desirable**.

Warning:
A weak implementation supports coupling of tests - and thus violates *isolation* in the sense of the classical school.

*Hint:* The *factory method* and/or the *decorator* pattern may loosen the coupling.

## Unit-Tests

Test fixtures are created using the `@pytest.fixture` decorator on functions:

```python
@pytest.fixture
def arg_x():
    return 6
```

```python
@pytest.fixture
def arg_y():
    return 9
```

```python
@pytest.fixture
def exp_res():
    return 3
```

Apply test fixtures in pytest tests:

```python
def test_gcd(arg_x, arg_y, res): #arrange
   # act
   act_res = gcd(arg_x, arg_y)

   # assert
   assert act_res == exp_res
```

## Unit-Tests

One fixture can be composed of other fixtures:

```python
@pytest.fixture
def gcd_small_numbers(arg_x, arg_y, exp_res): # test fixtures from before
    return {
        "args": (arg_x, arg_y)
        "result": exp_res
    }
```

→ Loosely coupled[5] test fixtures.

[5]The components are detached (as much as possible) from each other

## Unit-Tests

Remember:

**Unit-Test**

A unit test is an automated test that:

- Verifies a small piece of code (also known as *unit*),
- Does it quickly,
- And does it in an isolated manner.

Questions:

- What exactly is *a small piece of code* that qualifies as unit?
- What exactly does *isolated manner* mean?

## Unit-Test

There are two approaches (schools) that have (vastly) different opinions on the *isolation* issue:

- **Classical[6] school:** Isolating the unit tests themselves from each other.
  $\rightarrow$ Isolates tests.
- **London school:** Isolatating the system under test from its collaborators.
  $\rightarrow$ Isolates units.

As a consequence, the different schools have a different takes on the scope of a unit and how to handle dependencies between units.

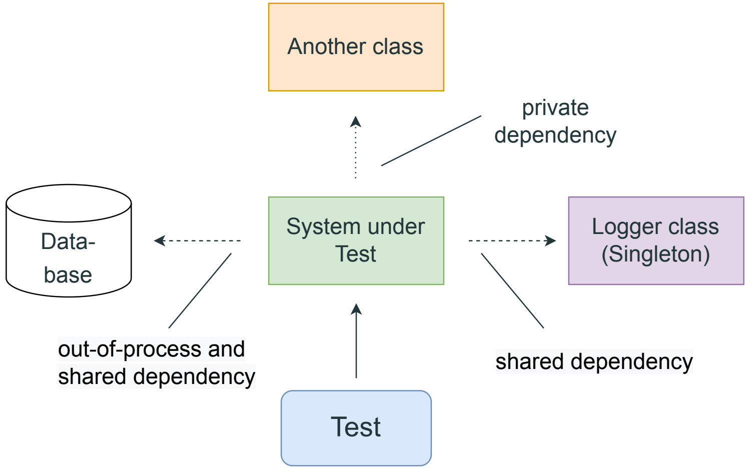
[6]Sometimes refered to as Detroid

## Unit-Tests

Types of dependencies:

- A *shared dependency* is a dependency that is shared between tests and provides means for those tests to affect each other's outcome.
- A *private dependency* is a dependency that is not shared.
- An *out-of-process dependencies* is a dependency that runs outside the application's execution process.

## Unit-Tests


Another class
private dependency
Data-base
System under Test
Logger class (Singleton)
out-of-process and shared dependency
shared dependency
Test

## Unit-Tests

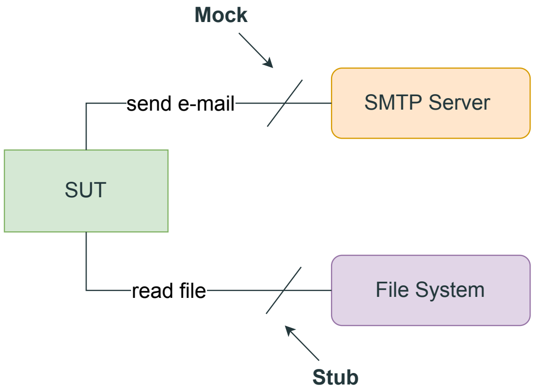
**Test Double**

An overarching term that describes all kinds of non-production-ready, fake dependencies in tests.

There are multiple variations of test doubles:

- *Mocks* help to emulate and examine outcoming interactions, which are calls the SUT makes to its dependencies to change their state.
- *Stubs* help to emulate incoming interactions, which are calls the SUT makes to its dependencies to get input data.
- *Fakes* are simplified implementations of a dependency.
- A *dummy* is a placeholder for unutilized functionality.

## Unit-Tests: Dependencies



## Unit-Tests

The package `pytest-mock`[7] is a plugin to `pytest` that provides a test fixture called `mocker`, which provides a powerful mechanism for mocking and stubbing.

- The method `mocker.patch(obj: str)` looks up an object in a given module and replaces that object with an instance of the class `unittest.Mock`[8].
- It allows to replace parts of the system under test with mock objects and make assertions about how they have been used.

[7] `https://pypi.org/project/pytest-mock/`

[8] `https://docs.python.org/3/library/unittest.mock.html`

## Unit-Tests

Among others, `unittest.Mock` provides the following methods:

- `assert_called_with()`: This method is a convenient way of asserting that the last call has been made in a particular way.
- `assert_not_called()`: Assert the mock was never called.
- `side_effect()`: This can either be a function to be called when the mock is called, an iterable, or an exception (class or instance) to be raised.

*Full documentation:* https://docs.python.org/3/library/unittest.mock.html

## Unit-Tests: Test Doubles

As an example, consider the following class:

```python
import os


class FileSystem:
    def rm(self, filename):
        os.remove(filename)

    def read(self, filename):
        with open(filename) as f:
            content = f.readlines()
        return content
```

→ Question: How to create unit tests for this class?

## Unit-Tests

Ideas for unit testing the class `FileSystem`:

- `rm()`: Replace the function `"os.remove"` with a test double (mock) that mimics the behavior of `os.remove`, but does not actually interact with the file system.
- `read()`: Replace the (builtin) function `open` with a test double (stub) that mimics the behaviour of `open` by returning predefined data, but does not actually interact with the file system.

## Unit-Tests

Use an instance of `Mock`[9] to implement a mock:

```python
def test_delete_file(mocker):
    # Arrange
    mocker.patch('os.remove')
    fs = FileSystem()

    # Act
    fs.rm('file')

    # Assert
    os.remove.assert_called_with('file')
```

[9] https://docs.python.org/3/library/unittest.mock.html

## Unit-Tests

Use an instance of `Mock`[10] to implement a stub:

```python
def test_read_file(mocker):
    # Arrange
    builtins.open = mocker.mock_open(read_data="Hi")
    fs = FileSystem()

    # Act
    content = fs.read('file')

    # Assert
    assert content == ["Hi"]
```

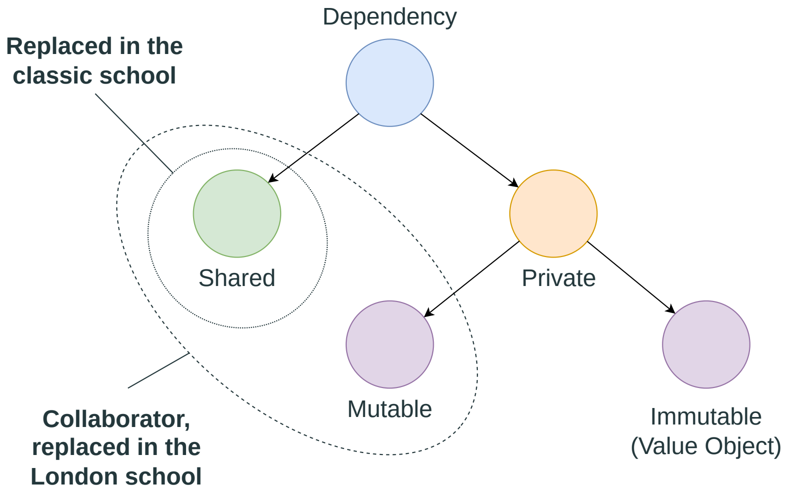
[10] https://docs.python.org/3/library/unittest.mock.html

## Unit-Tests

The differences between the schools, summed up by the approach to isolation, the size of a unit, and the use of test doubles:

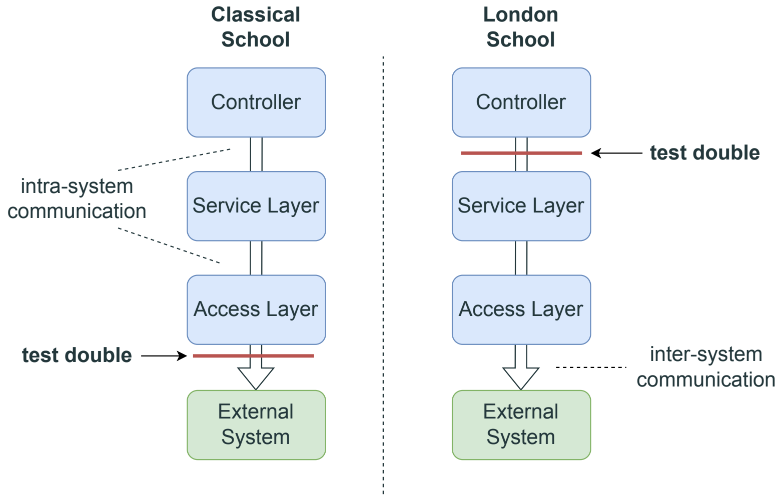
| School | Isolation of | A unit is | Uses test doubles for |
|---|---|---|---|
| London | Units | A class | All but immutable dependencies |
| Classical | Tests | Set of classes | Shared dependencies |

## Unit-Tests


Dependency
Replaced in the classic school
Shared
Private
Mutable
Immutable (Value Object)
Collaborator, replaced in the London school

# Unit-Tests

Unit tests in context of a multi-tier software architecture:



## Unit-Tests

**Self-reflect**: For which school would you decide? And Why?

## Unit-Tests

**Test Coverage**

Test coverage is a percentage measure of the degree to which the source code of a program is executed when the respective tests are run.

The `pytest-cov` package is a plugin for `pytest` to generate test coverage reports.

Usage:

```
pytest --cov=PATH
```

## Unit-Tests

Example (filesystem example):

```
pytest --cov=filesystem
```

Output:

```
tests/test_filesystem.py                [100%]

---------- coverage: platform linux ----------
Name                       Stmts   Miss  Cover
----------------------------------------------
filesystem/__init__.py         0      0   100%
filesystem/filesystem.py       8      0   100%
----------------------------------------------
TOTAL                          8      0   100%
```

## Unit-Tests

**Best practice:**

- Test the observable behavior of business logic (skip trivial code).
- Test all edge[11] and corner[12] cases.
- Test with `null`, respectively `None` inputs.
- When dealing with data structures, test with 0-element, 1-element, and many-element inputs.
- Test each error case at least once.
- Reach at least 75% test coverage.

[11] Extreme operating parameter (minimum or maximum)
[12] Multiple parameters are simultaneously at extreme levels

## Software Testing

### 3. Integration Tests

## Integration Tests

### Integration Test

An integration test is an automated test that:

- Verifies a distinct software functionality (also known as *feature*).
- May depend on out-of-process dependencies.
- Does not run quickly.

*Note:* Integration tests are typically targeted at classes in the service and/or controller layer.

## Integration Tests

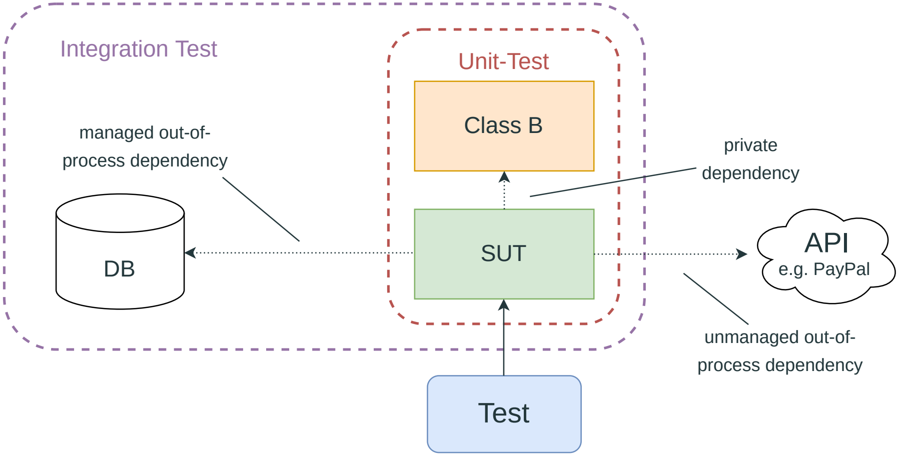
All out-of-process dependencies fall into two categories:

- *Managed dependencies*, over which developers have full control.
  Examples: Database, SMTP server.

- *Unmanaged dependencies*, over which developers don't have full control.
  Examples: PayPal API, Salesforce CRM.

$\rightarrow$ Integration tests replace *unmanaged dependencies* by test doubles.

# Integration Tests



## Integration Tests

**Testcontainers**[13][14]

Framework that facilitates the provisioning of managed out-of-process dependencies using container-based virtualization.

Benefits:

- Create and manage Docker containers for test purposes from source code.
- Integration with popular testing frameworks such as JUnit, TestNG, or Pytest.

$\rightarrow$ Developers can ensure that tests are running against ”the real thing”, leading to more reliable and accurate testing.

[13] `https://www.testcontainers.org/`

[14] `https://github.com/testcontainers/testcontainers-python`

# Integration Tests

Example: PostgreSQL database container

```python
import sqlalchemy
from testcontainers.postgres import PostgresContainer

def testcontainer_postgres():
    with PostgresContainer("postgres:12.4") as pg:
        url = pg.get_connection_url()
        engine = sqlalchemy.create_engine(url)
        result_set = engine.execute("select version()")
        for row in result_set:
            print("server version:", row[0])
```

## Integration Tests

Example: SMTP server using Docker compose

`docker-compose.yml`:

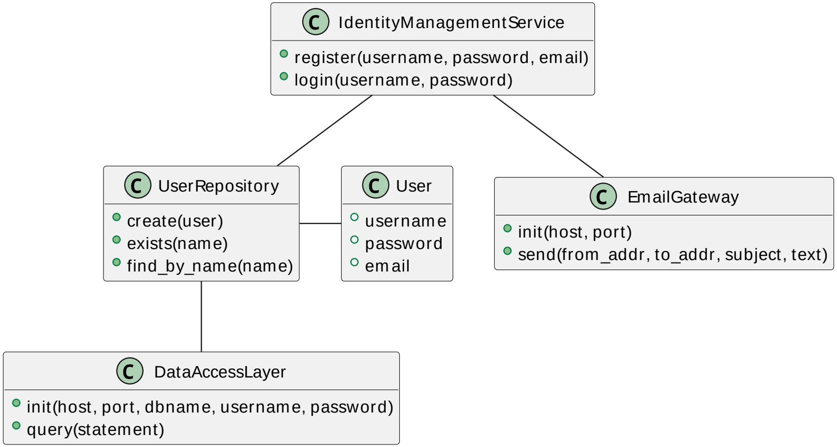
```yaml
services:
    smtp:
        image: mailhog/mailhog
        ports:                  # random port forwarding required for parallelism
            - "1025"            # smtp server
            - "8025"            # webui
```

Instantiation:

```python
def smtp_server_container():
    with DockerCompose(filepath=".") as dc:
        host = compose.get_service_host("smtp", 1025)
        port = compose.get_service_port("smtp", 1025)
```

## Integration Tests

Example: https://mygit.th-deg.de/aw-public/integration-test-examples


IdentityManagementService
register(username, password, email)
login(username, password)
UserRepository
create(user)
exists(name)
find_by_name(name)
User
username
password
email
EmailGateway
init(host, port)
send(from_addr, to_addr, subject, text)
DataAccessLayer
init(host, port, dbname, username, password)
query(statement)

## Integration Tests

Consider different user journeys:

- *Happy Path*: The ideal path that a user takes when interacting with a product.
- *Angry Path*: The path that a user takes when they encounter frustrating or unpleasant experiences when interacting with a product.
- *Scary Path*: The path that a user takes when they encounter unexpected or alarming experiences when interacting with a product.
- *Abandoned Path*: The path that a user takes when they abandon their interaction with a product before completing their intended task or goal.

## Integration Tests

Best practices:

- Use real-world scenarios.
- Test functionality, not components (we have unit-tests for that).
- Consider different perspectives in terms of user journeys.
- Run integration tests in parallel (e.g., with `pytest-xdist`).

# Software Testing

## 4. E2E Tests

## E2E Tests

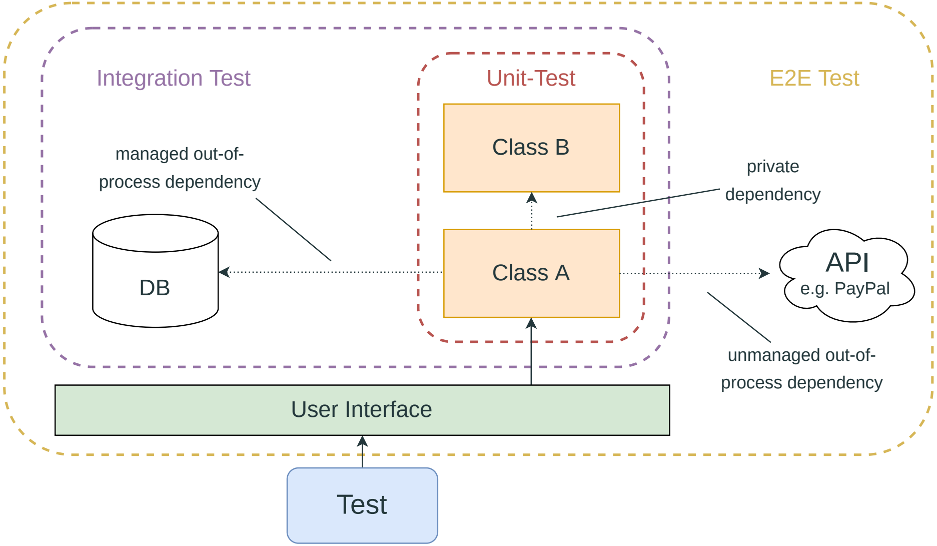
**E2E Test**

An E2E test is an automated test that:

- Verifies the system as a whole.
- Ensures that a functional or non-functional requirement is met.
- Validates the application and user flow $\rightarrow$ establishes acceptance.
- Runs slowly.

*Note:* E2E tests are targeted towards a ready-to-ship (i.e., packaged) version of the software system and operate on user interface level.

## E2E Tests


Integration Test
Unit-Test
E2E Test
Class B
managed out-of-process dependency
private dependency
DB
Class A
API
e.g. PayPal
unmanaged out-of-process dependency
User Interface
Test

## E2E Tests

**E2E tests do not make use of test doubles at all:**

- Many services provide a test/sandbox environment to support E2E testing. Example: PayPal Sandbox[15]
- If an unmanaged out-of-process dependency does not provide a sandbox, procuring a dedicated instance for testing purposes is highly recommended.

$\rightarrow$ Running tests against a dependency in production leads to inevitable pain.

[15] https://sandbox.paypal.com

## E2E Tests

**Behavior-Driven Development (BDD)**

A software development methodology that emphasizes collaboration between developers, testers, and business stakeholders to ensure that the software meets the desired behavior and functionality.

Main approach:

- Define E2E tests based on the expected behavior of the system.
- Use natural language specifications that can be understood by both technical and non-technical team members.
- Involve all stakeholders to define and verify the functionality of the system.

## E2E Tests

Example: A given-when-then scenario

```
Feature: Account holder withdraws cash

Scenario: Account has sufficient funds
    Given the account balance is $100
          and the card is valid
          and the machine contains enough money
     When the Account Holder requests $20
     Then the ATM should dispense $20
          and the account balance should be $80
          and the card should be returned
```

## E2E Tests

Benefits of *Behavior-Driven Development*:

- Ideal for beginners, no prior knowledge required.
- All development effort relates to business goals (no *over-engineering*).
- Focus on the desired behavior and the user experience of the software.
- Improved communication between development, quality assurance, and other domain experts.
- Specification serves as *living documentation*.
- **Clear definition of acceptance criteria.**

## E2E Tests

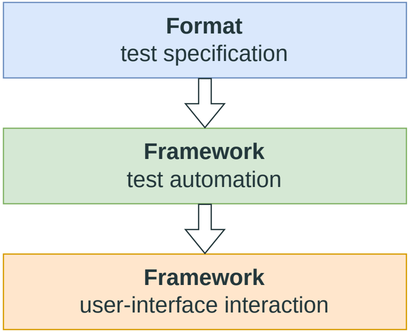
Drawbacks of *Behavior-Driven Development*:

- Poorly written specifications complicate software engineering.
- Involving multiple roles and stakeholders may lead to longer cycle-times.
- Refactoring of legacy projects is complex and costly.
- The process is rather conceptual and is therefore subject to interpretation.
- Requires active teams with strong communication skills that are committed to engaging frequently with customers.

# E2E Tests

To put *Behavior-Driven Development* into practice, 3 technologies are required:

1. A format to create a machine-readable specification of the system.
2. A framework to execute the specification in context of an automated software test.
3. A framework to interact with the system's user interface.



## E2E Tests

A BDD tech-stack based on Python for web applications:

| | |
|---|---|
| Gherkin[16] | De-facto standard format for BDD specifications based on the *given-when-then* structure. |
| Behave[17] | Testing framework to read and execute *Gherkin* specification in context of Python software tests. |
| Selenium[18] | Framework to automate browser inputs. |

[16] https://cucumber.io/docs/gherkin/reference/
[17] https://github.com/behave/behave
[18] https://www.selenium.dev/

## E2E Tests

A BDD tech-stack based on Python for CLI applications:

| | |
|---|---|
| Gherkin[19] | De-facto standard format for BDD specifications based on the *given-when-then* structure. |
| Behave[20] | Testing framework to read and execute *Gherkin* specification in context of Python software tests. |
| Pyexpect[21] | Implementation of the expect pattern to allow concise assertions on CLI outputs. |

[19] https://cucumber.io/docs/gherkin/reference/
[20] https://github.com/behave/behave
[21] https://pypi.org/project/pyexpect/

## E2E Tests

A BDD tech-stack based on JavaScript for web applications:

| | |
|---|---|
| Gherkin[22] | De-facto standard format for BDD specifications based on the *given-when-then* structure. |
| Cucumber[23] | Testing framework to read and execute *Gherkin* specification in context of JavaScript software tests. |
| Cypress[24] | Framework to automate browser inputs. |

[22] https://cucumber.io/docs/gherkin/reference/
[23] https://www.npmjs.com/package/@cucumber/cucumber
[24] https://www.cypress.io/

## Example: Gherkin specification

```
Feature: As a prospective student, I want to download and look at module
  handbooks, so that I can make an informed decision on my study programme.

Scenario: Download and view the module handbook for the study programme AIN-B
  Given I opened the website of the DIT
  When I switch the language to EN
    And hover over the navigation element Students
    And click on the navigation element Organization & documents
    And expand Faculty of Computer Science under the section Module Handbooks
    And click on Artificial Intelligence
  Then I see the module handbook file modulhandbuch_ki.pdf
```

# E2E Tests

## Example: behave framework

```python
from behave import given, when, then

@given('I opened the website of the DIT')
def step_impl(context):
    pass

@when('I switch the language to {lang}')
def step_impl(context, lang):
    pass

@then('I see the module handbook file {file}')
def step_impl(context):
    pass
```

## E2E Tests

### Example: Selenium

```python
@when('click on {text}')
def step_impl(context, text):
    # find element on the website
    link = context.driver.find_element(
        By.XPATH, "//a[contains(text(),'" + text + "')]")

    # move mouse to the element
    context.action_chains.move_to_element(link).perform()

    # click the link
    link.click()
```

## E2E Tests

Run **behave** from the command line:

```
behave
```

Output:

```
Feature: Showing off behave

Scenario: Run a simple test...

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.001s
```

## E2E Tests

**Code examples** (self study):

- https://mygit.th-deg.de/aw-public/e2e-test-examples
- https://mygit.th-deg.de/aw-public/ci-cd-examples

# Software Testing

## 5. Summary

## Summary

### Summary

You should have acquired the following competencies:

- Embrace the four pillars of a good test.
- Distinguish between the different types of tests.
- Differentiate between the different schools of testing and their respective stand on the *isolation* issue.
- Develop good (!) unit tests in Python.
- Develop good (!) integration tests in Python.
- Develop good (!) E2E tests in Python.