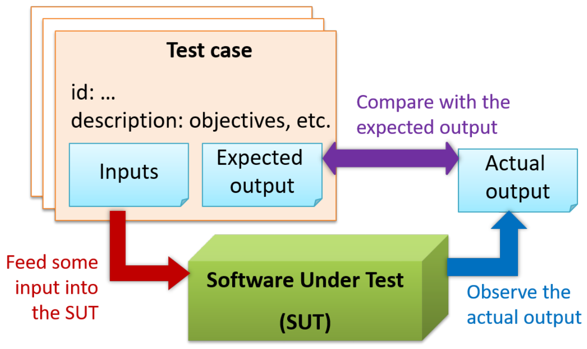
Testing

Introduction

What

★★☆☆ Can explain testing

> **Testing**: Operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. –- source: IEEE



Test case
id: ...
description: objectives, etc.
Inputs
Expected output
Compare with the expected output
Actual output
Feed some input into the SUT
Software Under Test (SUT)
Observe the actual output

When testing, you execute a set of *test cases*. A test case specifies how to perform a test. At a minimum, it specifies the input to the *software under test (SUT)* and the expected behavior.

Example: A minimal test case for testing a browser:

- **Input** – Start the browser using a blank page (vertical scrollbar disabled). Then, load `longfile.html` located in the `test data` folder.
- **Expected behavior** – The scrollbar should be automatically enabled upon loading `longfile.html`.

Test cases can be determined based on the specification, reviewing similar existing systems, or comparing to the past behavior of the SUT.

Other details a test case can contain + extra

For each test case you should do the following:

1. Feed the input to the SUT
2. Observe the actual output
3. Compare actual output with the expected output

A test case *failure* is a mismatch between the expected behavior and the actual behavior. A failure indicates a potential *defect* (or a bug), unless the error is in the test case itself.

Example: In the browser example above, a test case failure is implied if the scrollbar remains disabled after loading `longfile.html`. The defect/bug causing that failure could be an uninitialized variable.

A deeper look at the definition of testing + extra

Exercises

Testability

★★★☆ Can explain testability

***Testability* is an indication of how easy it is to test an SUT.** As testability depends a lot on the design and implementation, you should try to increase the testability when you design and implement software. The higher the testability, the easier it is to achieve better quality software.

Testing types

Regression testing

What

★★☆☆ Can explain regression testing

When you modify a system, the modification may result in some unintended and undesirable effects on the system. Such an effect is called a *regression*.

***Regression testing* is the re-testing of the software to detect regressions.** Note that to detect regressions, you need to retest all related components, even if they had been tested before.

Regression testing is more effective when it is done frequently, after each small change. However, doing so can be prohibitively expensive if testing is done manually. Hence, **regression testing is more practical when it is automated**.

Exercises

Developer testing

What

★★☆☆ Can explain developer testing

***Developer testing* is the testing done by the developers themselves** as opposed to professional testers or end-users.

Why

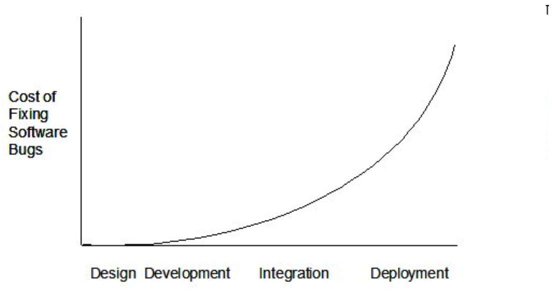
★★★☆ 🏆 Can explain the need for early developer testing

Delaying testing until the full product is complete has a number of disadvantages:

- **Locating the cause of a test case failure is difficult due to a large search space;** in a large system, the search space could be millions of lines of code, written by hundreds of developers! The failure may also be due to multiple inter-related bugs.
- **Fixing a bug found during such testing could result in major rework**, especially if the bug originated from the design or during requirements specification i.e. a faulty design or faulty requirements.
- **One bug might 'hide' other bugs**, which could emerge only after the first bug is fixed.
- **The delivery may have to be delayed** if too many bugs are found during testing.

Therefore, it is better to do early testing, as hinted by the popular rule of thumb given below, also illustrated by the graph below it.

> The earlier a bug is found, the easier and cheaper to have it fixed.



Such early testing of partially developed software is usually, and by necessity, done by the developers themselves i.e. developer testing.

Exercises

Unit testing

What

★☆☆☆ 🏆 Can explain unit testing

Unit testing: testing individual units (methods, classes, subsystems, ...) to ensure each piece works correctly.

In OOP code, it is common to write one or more unit tests for each public method of a class.

Here are the code skeletons for a `Foo` class containing two methods and a `FooTest` class that contains unit tests for those two methods.

```
class Foo {
    String read() {
        // ...
    }

    void write(String input) {
        // ...
    }

}
```

```
class FooTest {

    @Test
    void read() {
        // a unit test for Foo#read() method
    }

    @Test
    void write_emptyInput_exceptionThrown() {
        // a unit tests for Foo#write(String) method
    }

    @Test
    void write_normalInput_writtenCorrectly() {
        // another unit tests for Foo#write(String) method
    }
}
```

Resources

Stubs

★★★☆ Can use stubs to isolate an SUT from its dependencies

A proper unit test requires the *unit* to be tested in isolation so that bugs in the dependencies cannot influence the test i.e. bugs outside of the unit should not affect the unit tests.

If a `Logic` class depends on a `Storage` class, unit testing the `Logic` class requires isolating the `Logic` class from the `Storage` class.

***Stubs* can isolate the SUT from its dependencies**.

Stub: A stub has the same interface as the component it replaces, but its implementation is so simple that it is unlikely to have any bugs. It mimics the responses of the component, but only for a limited set of predetermined inputs. That is, it does not know how to respond to any other inputs. Typically, these mimicked responses are hard-coded in the stub rather than computed or retrieved from elsewhere, e.g. from a database.

Consider the code below:

```
class Logic {
    Storage s;

    Logic(Storage s) {
        this.s = s;
    }

```

```java
    String getName(int index) {
        return "Name: " + s.getName(index);
    }
}
```

```java
interface Storage {
    String getName(int index);
}
```

```java
class DatabaseStorage implements Storage {

    @Override
    public String getName(int index) {
        return readValueFromDatabase(index);
    }

    private String readValueFromDatabase(int index) {
        // retrieve name from the database
    }
}
```

Normally, you would use the `Logic` class as follows (note how the `Logic` object depends on a `DatabaseStorage` object to perform the `getName()` operation):

```java
Logic logic = new Logic(new DatabaseStorage());
String name = logic.getName(23);
```

You can test it like this:

```java
@Test
void getName() {
    Logic logic = new Logic(new DatabaseStorage());
    assertEquals("Name: John", logic.getName(5));
}
```

However, this `logic` object being tested is making use of a `DataBaseStorage` object which means a bug in the `DatabaseStorage` class can affect the test. Therefore, this test is not testing `Logic` *in isolation from its dependencies* and hence it is not a pure unit test.

Here is a stub class you can use in place of `DatabaseStorage`:

```java
class StorageStub implements Storage {

    @Override
    public String getName(int index) {
        if (index == 5) {
            return "Adam";
        } else {
            throw new UnsupportedOperationException();
        }
    }
}
```

Note how the `StorageStub` has the same interface as `DatabaseStorage`, but is so simple that it is unlikely to contain bugs, and is pre-configured to respond with a hard-coded response, presumably, the correct response `DatabaseStorage` is expected to return for the given test input.

Here is how you can use the stub to write a unit test. This test is not affected by any bugs in the `DatabaseStorage` class and hence is a pure unit test.

```java
@Test
void getName() {
    Logic logic = new Logic(new StorageStub());
    assertEquals("Name: Adam", logic.getName(5));
}
```

In addition to Stubs, there are other type of replacements you can use during testing, e.g. *Mocks*, *Fakes*, *Dummies*, *Spies*.

Resources

- **Mocks Aren't Stubs** by Martin Fowler -- An in-depth article about how Stubs differ from other types of test helpers.

Exercises

Integration testing

What

★★☆☆ Can explain integration testing

***Integration testing* : testing whether different parts of the software *work together* (i.e. integrates) as expected.** Integration tests aim to discover bugs in the 'glue code' related to how components interact with each other. These bugs are often the result of misunderstanding what the parts are supposed to do vs what the parts are actually doing.

Suppose a class `Car` uses classes `Engine` and `Wheel`. If the `Car` class assumed a `Wheel` can support a speed of up to 200 mph but the actual `Wheel` can only support a speed of up to 150 mph, it is the integration test that is supposed to uncover this discrepancy.

How

★★★☆ Can use integration testing

Integration testing is not simply a case of repeating the unit test cases using the actual dependencies (instead of the stubs used in unit testing). Instead, integration tests are additional test cases that focus on the interactions between the parts.

Suppose a class `Car` uses classes `Engine` and `Wheel`. Here is how you would go about doing pure integration tests:

a) First, unit test `Engine` and `Wheel`.
b) Next, unit test `Car` in isolation of `Engine` and `Wheel`, using stubs for `Engine` and `Wheel`.
c) After that, do an integration test for `Car` by using it together with the `Engine` and `Wheel` classes to ensure that `Car` integrates properly with the `Engine` and the `Wheel`.

In practice, developers often use a hybrid of unit+integration tests to minimize the need for stubs.

Here's how a hybrid unit+integration approach could be applied to the same example used above:

(a) First, unit test `Engine` and `Wheel`.
~~(b) Next, unit test `Car` in isolation of `Engine` and `Wheel`, using stubs for `Engine` and `Wheel`.~~
(c) After that, do an integration test for `Car` by using it together with the `Engine` and `Wheel` classes to ensure that `Car` integrates properly with the `Engine` and the `Wheel`. This step should include test cases that are meant to unit test `Car` (i.e. test cases used in the step (b) of the example above) as well as test cases that are meant to test the integration of `Car` with `Wheel` and `Engine` (i.e. pure integration test cases used of the step (c) in the example above).

Note that you no longer need stubs for `Engine` and `Wheel`. The downside is that `Car` is never tested in isolation of its dependencies. Given that its dependencies are already unit tested, the risk of bugs in `Engine` and `Wheel` affecting the testing of `Car` can be considered minimal.

System testing

What

★★☆☆ Can explain system testing

> ***System testing*: take the *whole system* and test it *against the system specification*.**

System testing is typically done by a testing team (also called a QA team).

System test cases are based on the specified external behavior of the system. Sometimes, system tests go beyond the bounds defined in the specification. This is useful when testing that the system fails 'gracefully' when pushed beyond its limits.

Suppose the SUT is a browser that is supposedly capable of handling web pages containing up to 5000 characters. Given below is a test case to test if the SUT fails gracefully if pushed beyond its limits.

```
Test case: load a web page that is too big
* Input: loads a web page containing more than 5000 characters.
* Expected behavior: aborts the loading of the page
  and shows a meaningful error message.
```

This test case would fail if the browser attempted to load the large file anyway and crashed.

System testing includes testing against non-functional requirements too. Here are some examples:

- *Performance testing* – to ensure the system responds quickly.
- *Load testing* (also called *stress testing* or *scalability testing*) – to ensure the system can work under heavy load.
- *Security testing* – to test how secure the system is.
- *Compatibility testing, interoperability testing* – to check whether the system can work with other systems.
- *Usability testing* – to test how easy it is to use the system.
- *Portability testing* – to test whether the system works on different platforms.

Alpha and beta testing

What

★★★☆ Can explain alpha and beta testing

***Alpha testing* is performed by the users, under controlled conditions set by the software development team.**

***Beta testing* is performed by a selected subset of target users of the system in their natural work setting.**

An *open beta release* is the release of not-yet-production-quality-but-almost-there software to the general population. For example, Google's Gmail was in 'beta' for many years before the label was finally removed.

Dogfooding

What

★★★☆ Can explain dogfooding

***Eating your own dog food* (aka dogfooding), is when creators use their own product so as to test the product.**

Exercises

Exploratory versus scripted testing

What

★★★☆ Can explain exploratory testing and scripted testing

Here are two alternative approaches to testing a software: *Scripted* testing and *Exploratory* testing.

1. **Scripted testing:** First write a set of test cases based on the expected behavior of the SUT, and then perform testing based on that set of test cases.

2. **Exploratory testing:** Devise test cases on-the-fly, creating new test cases based on the results of the past test cases.

Exploratory testing is 'the simultaneous learning, test design, and test execution' [source: bach-et-explained] whereby the nature of the follow-up test case is decided based on the behavior of the previous test cases. In other words, running the system and trying out various operations. It is called *exploratory testing* because testing is driven by observations during testing. Exploratory testing usually starts with areas identified as error-prone, based on the tester's past experience with similar systems. One tends to conduct more tests for those operations where more faults are found.

Here is an example thought process behind a segment of an exploratory testing session:

> "Hmm... looks like feature x is broken. This usually means feature n and k could be broken too; you need to look at them soon. But before that, you should give a good test run to feature y because users can still use the product if feature y works, even if x doesn't work. Now, if feature y doesn't work 100%, you have a major problem and this has to be made known to the development team sooner rather than later..."

Exploratory testing is also known as *reactive testing, error guessing technique, attack-based testing,* and *bug hunting*.

Exercises

When

★★★☆ Can explain the choice between exploratory testing and scripted testing

Which approach is better – **scripted or exploratory? A mix is better.**

The success of exploratory testing depends on the tester's prior experience and intuition. Exploratory testing should be done by experienced testers, using a clear strategy/plan/framework. Ad-hoc exploratory testing by unskilled or inexperienced testers without a clear strategy is not recommended for real-world non-trivial systems. While **exploratory testing may allow us to detect some problems in a relatively short time, it is not prudent to use exploratory testing as the sole means of testing a critical system**.

Scripted testing is more systematic, and hence, likely to discover more bugs given sufficient time, while exploratory testing would aid in quick error discovery, especially if the tester has a lot of experience in testing similar systems.

> In some contexts, you will achieve your testing mission better through a more scripted approach; in other contexts, your mission will benefit more from the ability to create and improve tests as you execute them. I find that most situations benefit from a mix of scripted and exploratory approaches. --[source: bach-et-explained]

Exercises

Acceptance testing

What

★★☆☆ Can explain acceptance testing

***Acceptance testing* (aka *User Acceptance Testing (UAT)*: test the system to ensure it meets the user requirements.**

Acceptance tests give an assurance to the customer that the system does what it is intended to do. Acceptance test cases are often defined at the beginning of the project, usually based on the use case specification. Successful completion of UAT is often a prerequisite to the project sign-off.

Acceptance vs System Testing

★★★☆ Can explain the differences between system testing and acceptance testing

Acceptance testing comes after system testing. Similar to system testing, acceptance testing involves testing the whole system.

Some differences between system testing and acceptance testing:

System Testing	Acceptance Testing
Done against the system specification	Done against the requirements specification
Done by testers of the project team	Done by a team that represents the customer
Done on the development environment or a test bed	Done on the deployment site or on a close simulation of the deployment site
Both negative and positive test cases	More focus on positive test cases

Note: *negative* test cases: cases where the SUT is not expected to work normally e.g. incorrect inputs; *positive* test cases: cases where the SUT is expected to work normally

Requirement specification versus system specification

The requirement specification need not be the same as the system specification. Some example differences:

Requirements specification	System specification
limited to how the system behaves in normal working conditions	can also include details on how it will fail gracefully when pushed beyond limits, how to recover, etc. specification
written in terms of problems that need to be solved (e.g. provide a method to locate an email quickly)	written in terms of how the system solves those problems (e.g. explain the email search feature)
specifies the interface available for intended end-users	could contain additional APIs not available for end-users (for the use of developers/testers)

However, **in many cases one document serves as both a requirement specification and a system specification.**

Passing system tests does not necessarily mean passing acceptance testing. Some examples:

- The system might work on the testbed environments but might not work the same way in the deployment environment, due to subtle differences between the two environments.
- The system might conform to the system specification but could fail to solve the problem it was supposed to solve for the user, due to flaws in the system design.

Exercises

Test automation

What

★★☆☆ 🏆 Can explain test automation

An automated test case can be run programmatically and the result of the test case (pass or fail) is determined programmatically. Compared to manual testing, automated testing reduces the effort required to run tests repeatedly and increases precision of testing (because manual testing is susceptible to human errors).

Resources

Automated Testing of CLI Apps

★★☆☆ 🏆 Can semi-automate testing of CLIs

A simple way to semi-automate testing of a CLI (Command Line Interface) app is by using input/output re-direction.

- First, you feed the app with a sequence of test inputs that is stored in a file while redirecting the output to another file.
- Next, you compare the actual output file with another file containing the expected output.

Let's assume you are testing a CLI app called `AddressBook`. Here are the detailed steps:

1. Store the test input in the text file `input.txt`.

   Example `input.txt`

2. Store the output you expect from the SUT in another text file `expected.txt`.

   Example `expected.txt`

3. Run the program as given below, which will redirect the text in `input.txt` as the input to `AddressBook` and similarly, will redirect the output of AddressBook to a text file `output.txt`. Note that this does not require any code changes to `AddressBook`.

```
java AddressBook < input.txt > output.txt
```

   - The way to run a CLI program differs based on the language.
     e.g., In Python, assuming the code is in `AddressBook.py` file, use the command `python AddressBook.py < input.txt > output.txt`
   - If you are using Windows, use a normal command window to run the app, not a PowerShell window.

More on the `>` operator and the `<` operator + extra

4. Next, you compare `output.txt` with the `expected.txt`. This can be done using a utility such as Windows' `FC` (i.e. File Compare) command, Unix's `diff` command, or a GUI tool such as *WinMerge*.

```
FC output.txt expected.txt
```

Note that the above technique is only suitable when testing CLI apps, and only if the exact output can be predetermined. If the output varies from one run to the other (e.g. it contains a time stamp), this technique will not work. In those cases, you need more sophisticated ways of automating tests.

Test Automation Using Test Drivers

★★★☆ Can explain test drivers

A test driver is the code that 'drives' the SUT for the purpose of testing i.e. invoking the SUT with test inputs and verifying if the behavior is as expected.

`PayrollTest` 'drives' the `Payroll` class by sending it test inputs and verifies if the output is as expected.

```java
public class PayrollTest {
    public static void main(String[] args) throws Exception {

        // test setup
        Payroll p = new Payroll();

        // test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        // automatically verify the response
        if (p.totalSalary() != 6400) {
            throw new Error("case 1 failed ");
        }

        // test case 2
        p.setEmployees(new String[]{"E001"});
        if (p.totalSalary() != 2300) {
            throw new Error("case 2 failed ");
        }

        // more tests...

        System.out.println("All tests passed");
    }
}
```

Test Automation Tools

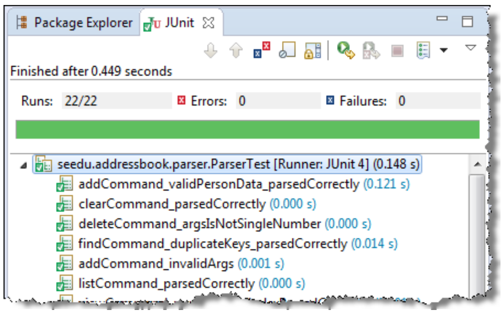
★★★☆ Can explain test automation tools

JUnit is a tool for automated testing of Java programs. Similar tools are available for other languages and for automating different types of testing.

This is an automated test for a `Payroll` class, written using JUnit libraries.

```java
@Test
public void testTotalSalary() {
    Payroll p = new Payroll();

    // test case 1
    p.setEmployees(new String[]{"E001", "E002"});
    assertEquals(6400, p.totalSalary());

    // test case 2
    p.setEmployees(new String[]{"E001"});
    assertEquals(2300, p.totalSalary());

    // more tests...
}
```

Most modern IDEs have integrated support for testing tools. The figure below shows the JUnit output when running some JUnit tests using the Eclipse IDE.



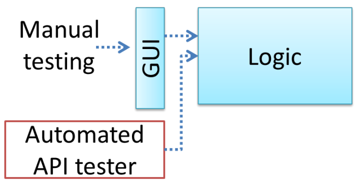
```
Finished after 0.449 seconds

Runs: 22/22    Errors: 0    Failures: 0

seedu.addressbook.parser.ParserTest [Runner: JUnit 4] (0.148 s)
    addCommand_validPersonData_parsedCorrectly (0.121 s)
    clearCommand_parsedCorrectly (0.000 s)
    deleteCommand_argsIsNotSingleNumber (0.000 s)
    findCommand_duplicateKeys_parsedCorrectly (0.014 s)
    addCommand_invalidArgs (0.001 s)
    listCommand_parsedCorrectly (0.000 s)
```

Automated Testing of GUIs

★★★☆ Can explain automated GUI testing

If a software product has a GUI (Graphical User Interface) component, all product-level testing (i.e. the types of testing mentioned above) need to be done using the GUI. However, **testing the GUI is much harder than testing the CLI (Command Line Interface) or API**, for the following reasons:

- Most GUIs can support a large number of different operations, many of which can be performed in any arbitrary order.
- GUI operations are more difficult to automate than API testing. Reliably automating GUI operations and automatically verifying whether the GUI behaves as expected is harder than calling an operation and comparing its return value with an expected value. Therefore, automated regression testing of GUIs is rather difficult.
- The appearance of a GUI (and sometimes even behavior) can be different across platforms and even environments. For example, a GUI can behave differently based on whether it is minimized or maximized, in focus or out of focus, and in a high resolution display or a low resolution display.



Moving as much logic as possible out of the GUI can make GUI testing easier. That way, you can bypass the GUI to test the rest of the system using automated API testing. While this still requires the GUI to be tested, the number of such test cases can be reduced as most of the system will have been tested using automated API testing.

There are testing tools that can automate GUI testing.

Some tools used for automated GUI testing:

- **TestFX** can do automated testing of JavaFX GUIs
- **Visual Studio** supports the 'record replay' type of GUI test automation.
- **Selenium** can be used to automate testing of web application UIs

Demo video of automated testing of a web application

Exercises

Test coverage

What

★★☆☆ Can explain test coverage

***Test coverage* is a metric used to measure the extent to which testing exercises the code** i.e., how much of the code is 'covered' by the tests.

Here are some examples of different coverage criteria:

- **Function/method coverage** : based on functions executed e.g., testing executed 90 out of 100 functions.
- **Statement coverage** : based on the number of lines of code executed e.g., testing executed 23k out of 25k LOC.
- **Decision/branch coverage** : based on the decision points exercised e.g., an `if` statement evaluated to both `true` and `false` with separate test cases during testing is considered 'covered'.
- **Condition coverage** : based on the boolean sub-expressions, each evaluated to both true and false with different test cases. Condition coverage is not the same as the decision coverage.

`if(x > 2 && x < 44)` is considered one decision point but two conditions.

For 100% branch or decision coverage, two test cases are required:

- `(x > 2 && x < 44) == true` : [e.g. `x == 4`]
- `(x > 2 && x < 44) == false` : [e.g. `x == 100`]

For 100% condition coverage, three test cases are required:

- `(x > 2) == true` , `(x < 44) == true` : [e.g. `x == 4`] [see note 1]
- `(x < 44) == false` : [e.g. `x == 100`]
- `(x > 2) == false` : [e.g. `x == 0`]

Note 1: A case where both conditions are `true` is needed because most execution environments use a *short circuiting* behavior for compound boolean expressions e.g., given an expression `c1 && c2`, `c2` will not be evaluated if `c1` is `false` (as the final result is going to be `false` anyway).

- **Path coverage** measures coverage in terms of possible paths through a given part of the code executed. 100% path coverage means all possible paths have been executed. A commonly used notation for path analysis is called the *Control Flow Graph (CFG)*.
- **Entry/exit coverage** measures coverage in terms of possible *calls to* and *exits* from the operations in the SUT.

Exercises

How

★★★☆ Can explain how test coverage works

Measuring coverage is often done using *coverage analysis tools.* Most IDEs have inbuilt support for measuring test coverage, or at least have plugins that can measure test coverage.

Coverage analysis can be useful in improving the quality of testing e.g., if a set of test cases does not achieve 100% branch coverage, more test cases can be added to cover missed branches.

Measuring code coverage in Intellij IDEA (watch from `4 minutes 50 seconds` mark)

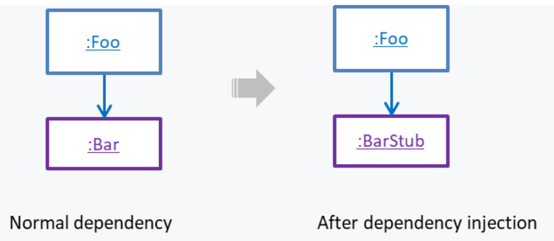
Dependency injection

What

★★★☆ Can explain dependency injection

***Dependency injection* is the process of 'injecting' objects to replace current dependencies with a different object.** This is often used to inject stubs to isolate the SUT from its dependencies so that it can be tested in isolation.

A `Foo` object normally depends on a `Bar` object, but you can inject a `BarStub` object so that the `Foo` object no longer depends on a `Bar` object. Now you can test the `Foo` object in isolation from the `Bar` object.



How

★★★★ Can use dependency injection

Polymorphism can be used to implement dependency injection, as can be seen in the example given in [Quality Assurance → Testing → Unit Testing → Stubs] where a stub is injected to replace a dependency.

Here is another example of using polymorphism to implement dependency injection:

Suppose you want to unit test `Payroll#totalSalary()` given below. The method depends on the `SalaryManager` object to calculate the return value. Note how the `setSalaryManager(SalaryManager)` can be used to inject a `SalaryManager` object to replace the current `SalaryManager` object.

```java
class Payroll {
    private SalaryManager manager = new SalaryManager();
    private String[] employees;

    void setEmployees(String[] employees) {
        this.employees = employees;
    }

    void setSalaryManager(SalaryManager sm) {
        this.manager = sm;
    }

    double totalSalary() {
        double total = 0;
        for (int i = 0; i < employees.length; i++) {
            total += manager.getSalaryForEmployee(employees[i]);
        }
        return total;
    }
}


class SalaryManager {
    double getSalaryForEmployee(String empID) {
        // code to access employee’s salary history
        // code to calculate total salary paid and return it
    }
}
```

During testing, you can inject a `SalaryManagerStub` object to replace the `SalaryManager` object.

```java
class PayrollTest {
    public static void main(String[] args) {
        // test setup
        Payroll p = new Payroll();
        // dependency injection
        p.setSalaryManager(new SalaryManagerStub());
        // test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        assertEquals(2500.0, p.totalSalary());
        // test case 2
        p.setEmployees(new String[]{"E001"});
        assertEquals(1000.0, p.totalSalary());
        // more tests ...
    }
}

```

```
class SalaryManagerStub extends SalaryManager {
    /** Returns hard coded values used for testing */
    double getSalaryForEmployee(String empID) {
        if (empID.equals("E001")) {
            return 1000.0;
        } else if (empID.equals("E002")) {
            return 1500.0;
        } else {
            throw new Error("unknown id");
        }
    }
}
```

Exercises

TDD

What

★★★☆ Can explain TDD

***Test-Driven Development(TDD)* advocates writing the tests before writing the SUT, while evolving functionality and tests in small increments**. In TDD you first define the precise behavior of the SUT using test cases, and then write the SUT to match the specified behavior. While TDD has its fair share of detractors, there are many who consider it a good way to reduce defects. One big advantage of TDD is that it guarantees the code is testable.

Exercises

How

★★★★ Can follow TDD

Note that TDD does not imply writing all the test cases first before writing functional code. Rather, proceed in small steps:

1. Decide what behavior to implement.
2. Write test cases to test that behavior.
3. Run those test cases and watch them fail.
4. Implement the behavior.
5. Run the test cases.
6. Keep modifying the code and rerunning test cases until they all pass.
7. Refactor code to improve quality.
8. Repeat the cycle for each small unit of behavior that needs to be implemented.