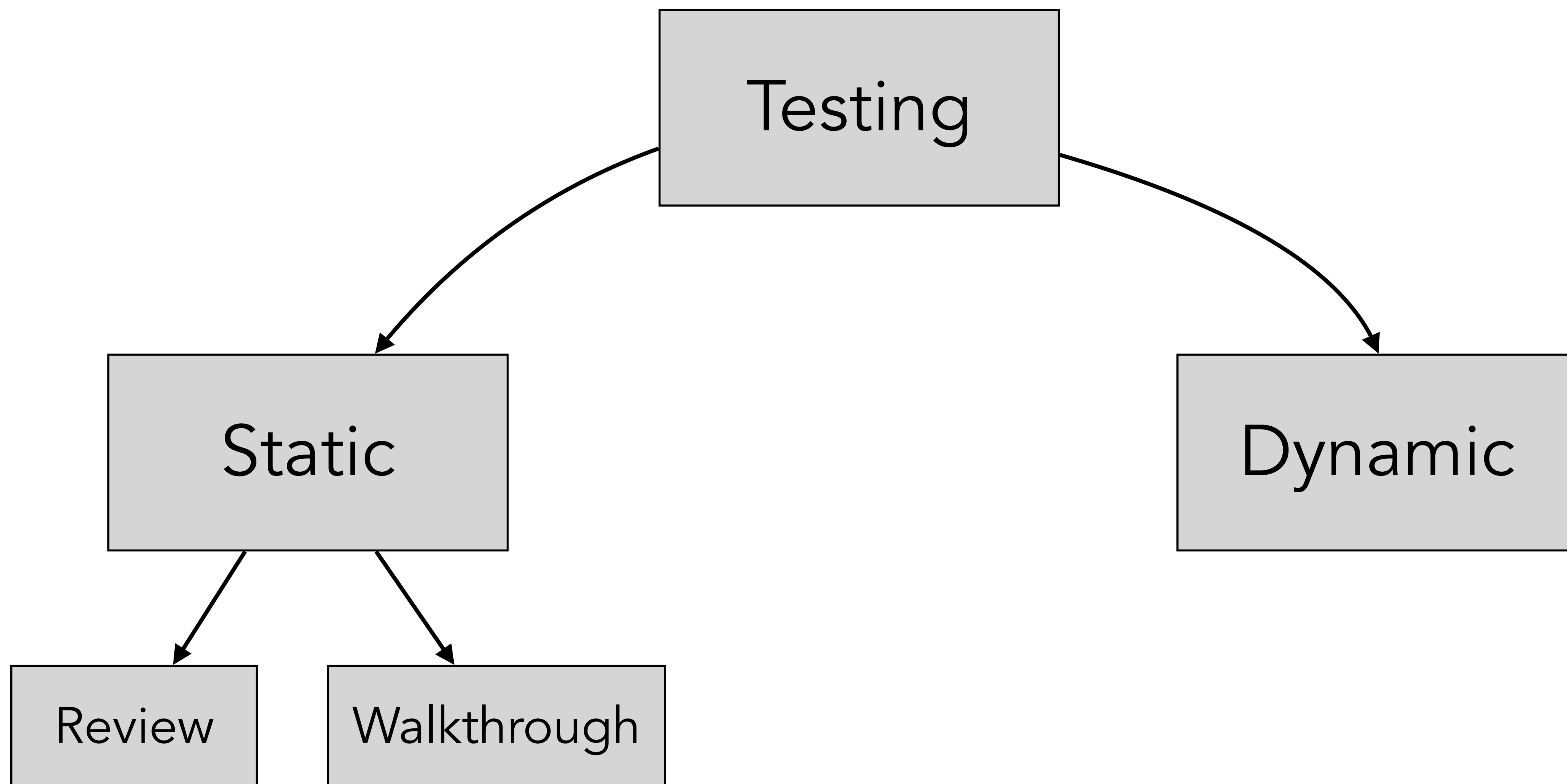
Software Testing, Verification and Validation

September 28, 2022
Week #3 — Lecture #2

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last week, we discussed **Static Testing**, a software testing technique which is used to check *faults* in a software application without executing its source code. It is concerned with the analysis of the static system representation (source code, documents, models, prototypes, etc.) to discover *faults*. This week, we will introduce **Dynamic Testing**.


Testing
Static
Dynamic
Review
Walkthrough

Dynamic Testing

Dynamic Testing

Dynamic Testing is a software testing technique which is used to check for functional behavior of software system, memory / cpu usage and overall performance of the system. Hence the name “Dynamic”. The main objective of this testing is to confirm that the software product works in conformance with the business requirements.

Dynamic testing executes the software and validates the output with the expected outcome. Dynamic testing is performed at all levels of testing and it can be either *black* or *white* box testing.

Static vs. Dynamic

Static	Dynamic
Testing was done without executing the program	Testing is done by executing the program
This testing does the verification process	Dynamic testing does the validation process
Static testing is about prevention of defects	Dynamic testing is about finding and fixing the defects
Static testing gives an assessment of code and documentation	Dynamic testing gives bugs/bottlenecks in the software system
Static testing involves a checklist and process to be followed	Dynamic testing involves test cases for execution
This testing can be performed before compilation	Dynamic testing is performed after compilation
Static testing covers the structural and statement coverage testing	Dynamic testing techniques are Boundary Value Analysis & Equivalence Partitioning.
Cost of finding defects and fixing is low	Cost of finding and fixing defects is high
Return on investment will be high as this process involved at an early stage	Return on investment will be low as this process involves after the development phase
More reviews comments are highly recommended for good quality	More defects are highly recommended for good quality.
May requires a large number of formal and/or informal meetings	Comparatively requires lesser meetings

https://www.guru99.com/static-dynamic-testing.html

Test types

Test types

Level / Phase

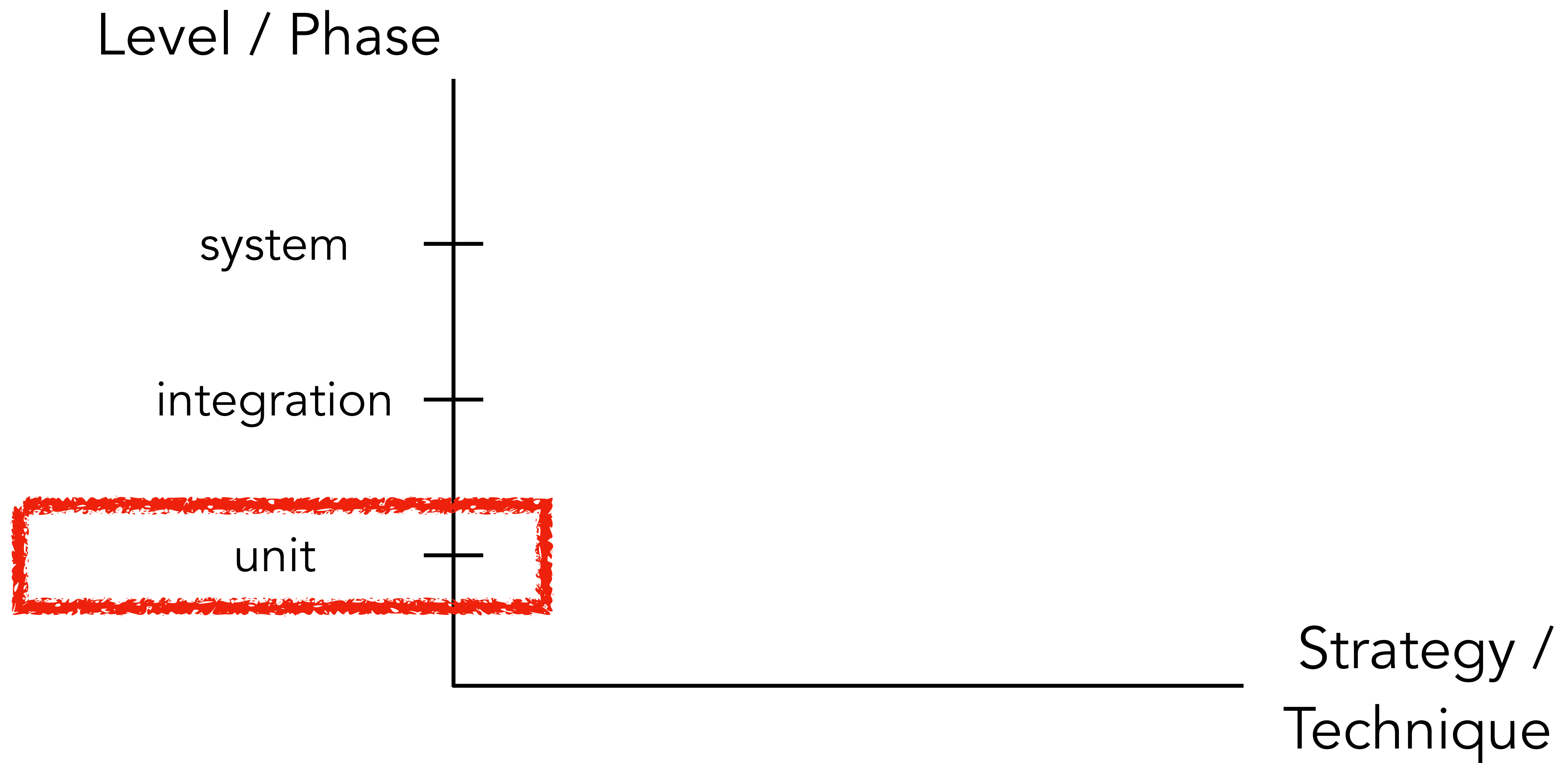
system

integration

unit

Strategy / Technique

Test types



Unit testing

In some situations, the goal of the tester is to test a single feature of the software, purposefully ignoring the other units of the systems. When we test units in isolation, we are doing what is called **unit testing**.

Defining a ‘unit’ is challenging and highly dependent on the context. A unit can be just one method or can consist of multiple classes. Here is a definition for unit testing by Roy Osherove

“A **unit test** is an automated piece of code that invokes a unit of work in the system. And a unit of work can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified.”

Roy Osherove, “The Art of Unit Testing”, 2nd edition, 2013

The anatomy of a unit test

(Arrange, Act, Assert — AAA)

```
public class CalculatorTests {

    public void test_sum_of_two_numbers() {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        org.junit.Assert.assertEquals(30, result);
    }
}
```

The anatomy of a unit test

(Arrange, Act, Assert — AAA)

```
public class CalculatorTests {

    public void test_sum_of_two_numbers() {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Arrange: This is is where you prepare and initialize all variables and objects that are needed by your system under test to work.

The anatomy of a unit test

(Arrange, Act, Assert — AAA)

```
public class CalculatorTests {

    public void test_sum_of_two_numbers() {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Arrange: This is is where you prepare and initialize all variables and objects that are needed by your system under test to work.

Act: In this section you will actually invoke the method that you are testing.

Vladimir Khorikov, “Unit Testing Principles, Practices, and Patterns” (Chapter 3), 2020

The anatomy of a unit test

(Arrange, Act, Assert — AAA)

```
public class CalculatorTests {

    public void test_sum_of_two_numbers() {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Arrange: This is is where you prepare and initialize all variables and objects that are needed by your system under test to work.

Act: In this section you will actually invoke the method that you are testing.

Assert: This section is used to validate the return value received from the *sum()* method. If the returned value is what is expected, then the test method will pass. If the returned value is not what was expected, then the test method will fail.

The anatomy of a unit test

(Given-When-Then)

```java
public class CalculatorTests {

    public void test_sum_of_two_numbers() {
        // Given
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // When
        double result = calculator.sum(first, second);

        // Then
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Unit testing, 👍 and 👎

Advantages 👍

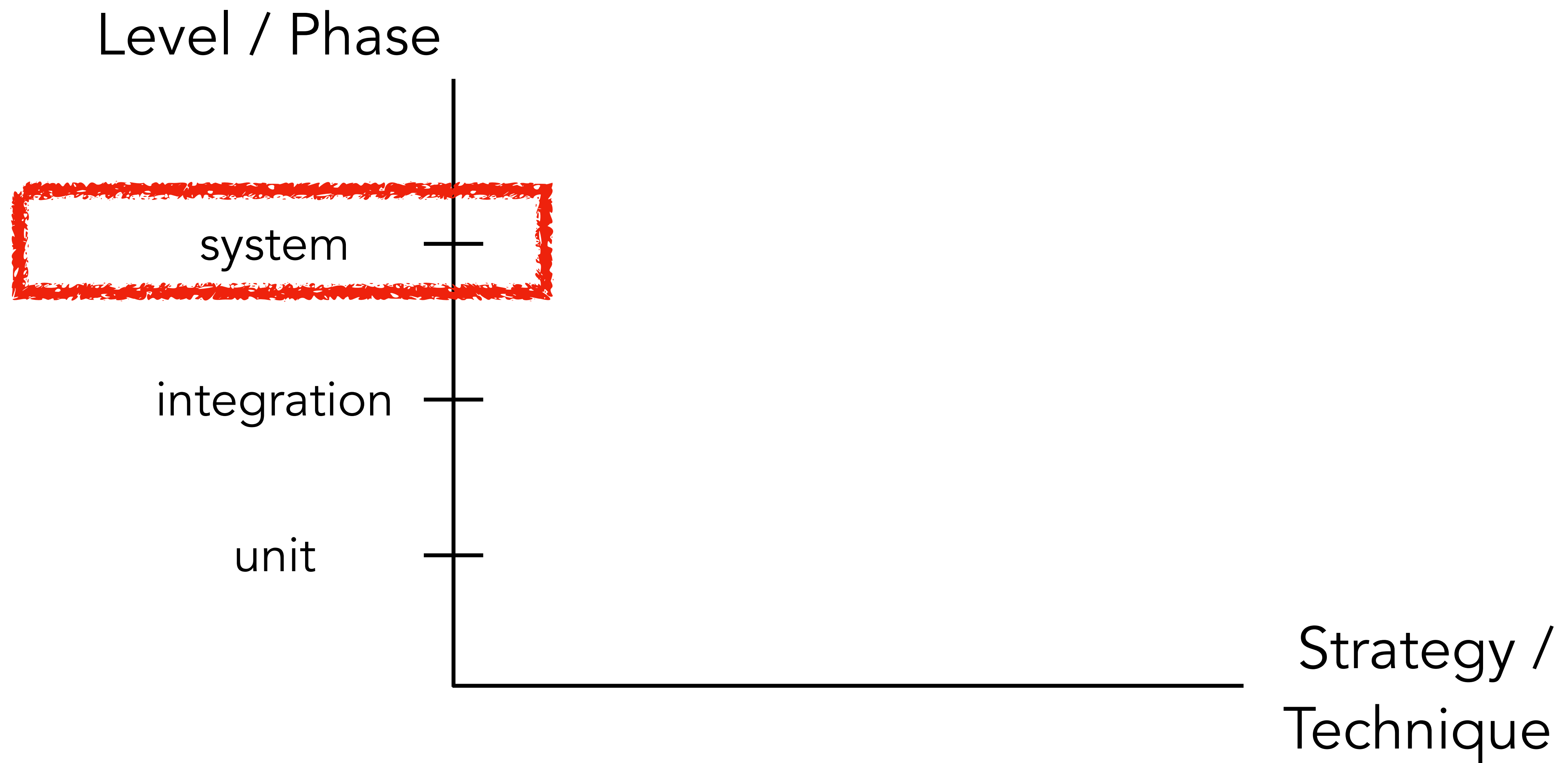
- Firstly, **unit tests are fast**. A unit test usually takes just a couple of milliseconds to execute. Fast tests give us the ability to test huge portions of the system in a small amount of time. Fast, automated test suites give developers constant feedback; this fast safety net makes developers feel more comfortable and confident in performing evolutionary changes to the software system they are working on.
- Secondly, **unit tests are easy to control**. A unit test tests the software by giving certain parameters to a method and then comparing the return value of this method to the expected result. The input values and expected result values are easy to adapt or modify in the test.
- Finally, **unit tests are easy to write**. Unit tests do not require complicated setup or additional work. A single unit is also often cohesive and small, making the job of the tester easier.

Unit testing, 👍 and 👎

Disadvantages 👎

- Unit tests **lack "reality"**. A software system is rarely composed of a single class. The large number of classes in a system and the interaction between these classes can cause the system to behave differently in its real application than in the unit tests. Therefore, unit tests do not perfectly represent the real execution of a software system.
- **Some types of bugs are not caught**. Some types of bugs cannot be caught at unit test level. They only happen in the integration of the different components (which are not exercised in a pure unit test).

Test types



System testing

Unit tests do not exercise the system in its entirety (but this is not their goal). To get a more realistic view of the software, and thus perform more realistic tests, we should run the entire software system, with all its databases, front-end apps, and any other components it has.

When we test the system in its entirety, we are doing what is called **system testing**. In practice, instead of testing small parts of the system in isolation, system tests exercise the system as a whole.

System testing, 👍 and 👎

Advantages 👍

- The obvious advantage of system testing is **how realistic the tests are**. After all, the more realistic the tests are, the greater the chance that the system works when released.
- System tests also **capture the user's perspective** better than unit tests. In other words, system tests are a better simulation of how the final user interacts with the system.

System testing, 👍 and 👎

Disadvantages 👎

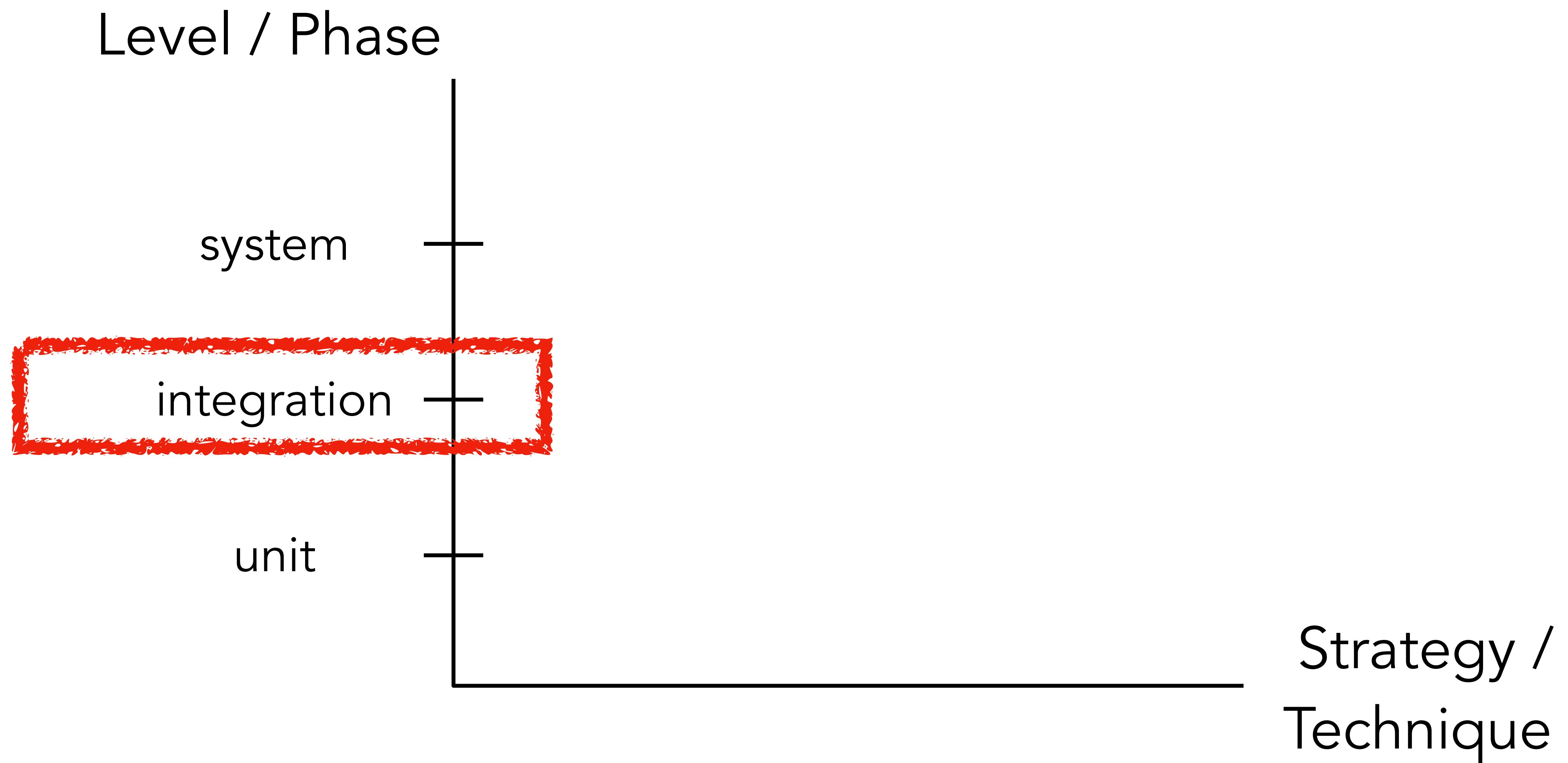
- **System tests are often slow when compared to unit tests**. Although we have not written any system tests up until now, try to imagine what all a system test has to do, including starting and running the whole system with all its components. The test also has to interact with the real application and actions might take a few seconds to happen. Imagine a test that starts a container with a web application and another container with a database. It then submits an HTTP request to a webservice that is exposed by this web app. This webservice then retrieves data from the database and writes a JSON response to the test. This obviously takes more time than running a simple unit test, which has virtually no dependencies.
- System tests are also **harder to write**. Some of the components (e.g., databases) might require complex setup before they can be used in a testing scenario. Think of not only connection and authentication, but also making sure that the database has all the data that is required by that test case. This takes additional code that is needed just for automating the tests.

System testing, 👍 and 👎

Disadvantages 👎

- Lastly, **system tests tend to become flaky**. A flaky test is a test that presents an erratic behavior: if you run it, it might pass or it might fail for the same configuration. Flaky tests are an important problem for software development teams. After all, having a test that might pass when there is a *fault* or one that might fail when there are none harms the productivity of the development team. It is easy to imagine how a system test can become flaky. Think of a system test that exercises a web app. After clicking a button, the HTTP POST request to the web app took half a second more than usual (due to small variations we often do not control in real-life scenarios; tomcat decided to do a full garbage collection at that very second, for example). The test was not expecting it to happen and thus, it failed. If the test is executed again, the web app might now take its usual time to respond and the test will pass on this try. There are just too many uncertainties in a system test that can lead to unexpected behavior.

Test types



Integration testing

Unit and system testing are the two extremes of test levels. As we saw, unit tests focus on the smallest parts of the system while system tests focus on the whole system at once. However, sometimes we need something in between.

Integration testing is the test level we use when we need something more integrated (or less isolated) than a unit test but without the need of exercising the entire system.

Integration testing

"Software systems commonly rely on database systems. To communicate with the database, developers often create a class whose only responsibility is to interact with this external component (think of Data Access Objects - DAO - classes). These DAOs might contain complicated SQL code. Thus, a tester feels the need to test these SQL queries.

However, note that the tester does not want to test the entire system, only the integration between the DAO class and the database. The tester also does not want to test the DAO class in complete isolation; after all, the best way to know whether a SQL query works is to actually submit it to the database and see what the database returns back. This is an example of an integration test."

The goal of integration testing is to test multiple components of a system together, focusing on the interactions between them instead of testing the system as a whole. Are they communicating correctly? What happens if component A sends message X to component B? Do they still present correct behavior?

Integration testing, 👍 and 👎

Advantages 👍

- The advantage of integration tests is that, while not fully isolated, deriving tests just for a specific integration is easier than deriving tests for all the components together. Therefore, the effort of writing such tests is a little more than the effort required for unit tests but less than the effort for system tests.

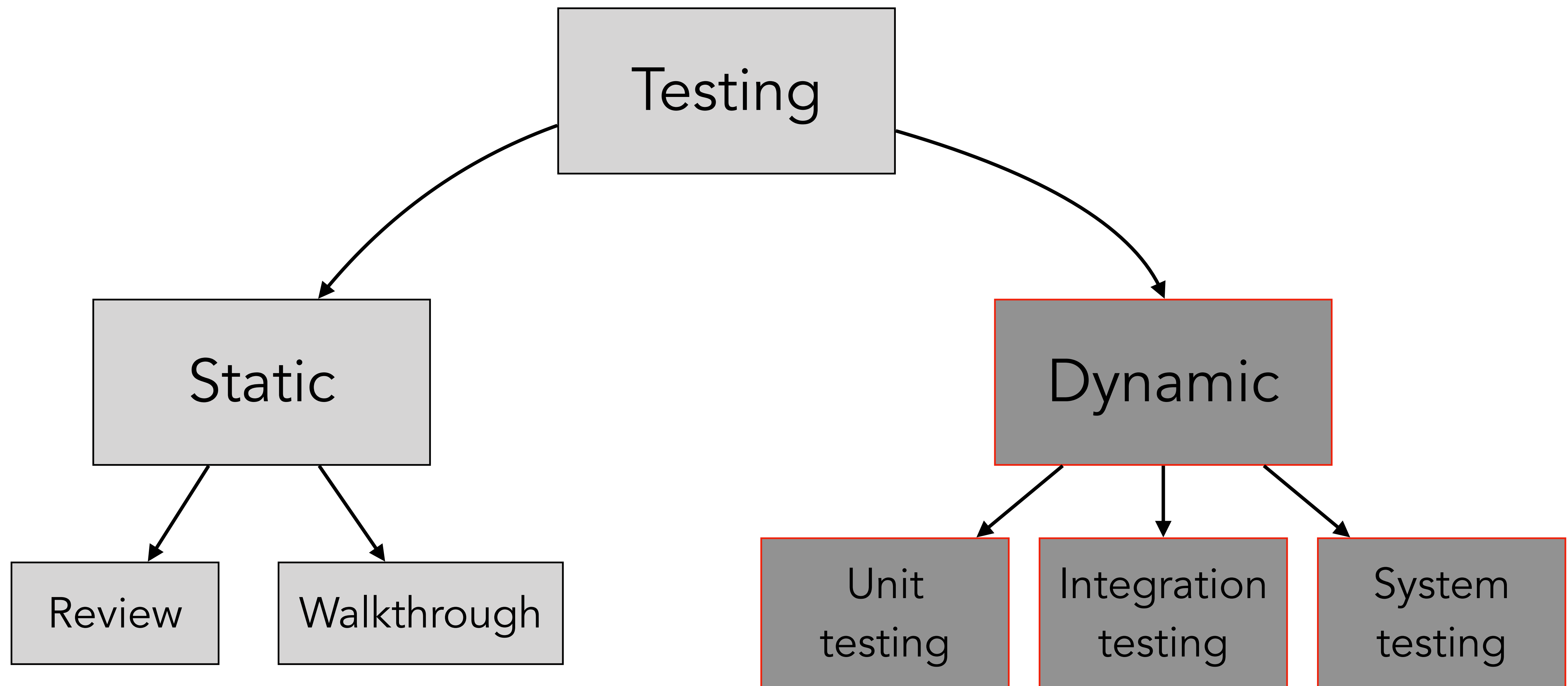
Integration testing, 👍 and 👎

Disadvantages 👎

Note that the more integrated our tests are, the more difficult they are to write. In the example, setting up a database for the test requires effort. Tests that involve databases usually need to:

- make use of an isolated instance of the database just for testing purposes (as you probably do not want your tests to mess with production data),
- update the database schema (in fast companies, database schemas are changing all the time, and the test database needs to keep up),
- put the database into a state expected by the test by adding or removing rows,
- and clean everything afterwards (so that the next tests do not fail because of the data that was left behind by the previous test).

The same effort happens to any other type of integration test you can imagine (e.g., web services, file reads and writes, etc.)



Unit Testing: Under Unit Testing, individual units or modules are tested by the developers. It involves testing of source code by developers.
Integration Testing: Individual modules are grouped together and tested by the developers. The purpose is to determine what modules are working as expected once they are integrated.
System Testing: System Testing is performed on the whole system by checking whether the system or application meets the requirement specification document.

Testing Pyramid

Testing Pyramid

We discussed three different test levels: unit, system, and integration. A question that pragmatic software developers might ask themselves is:

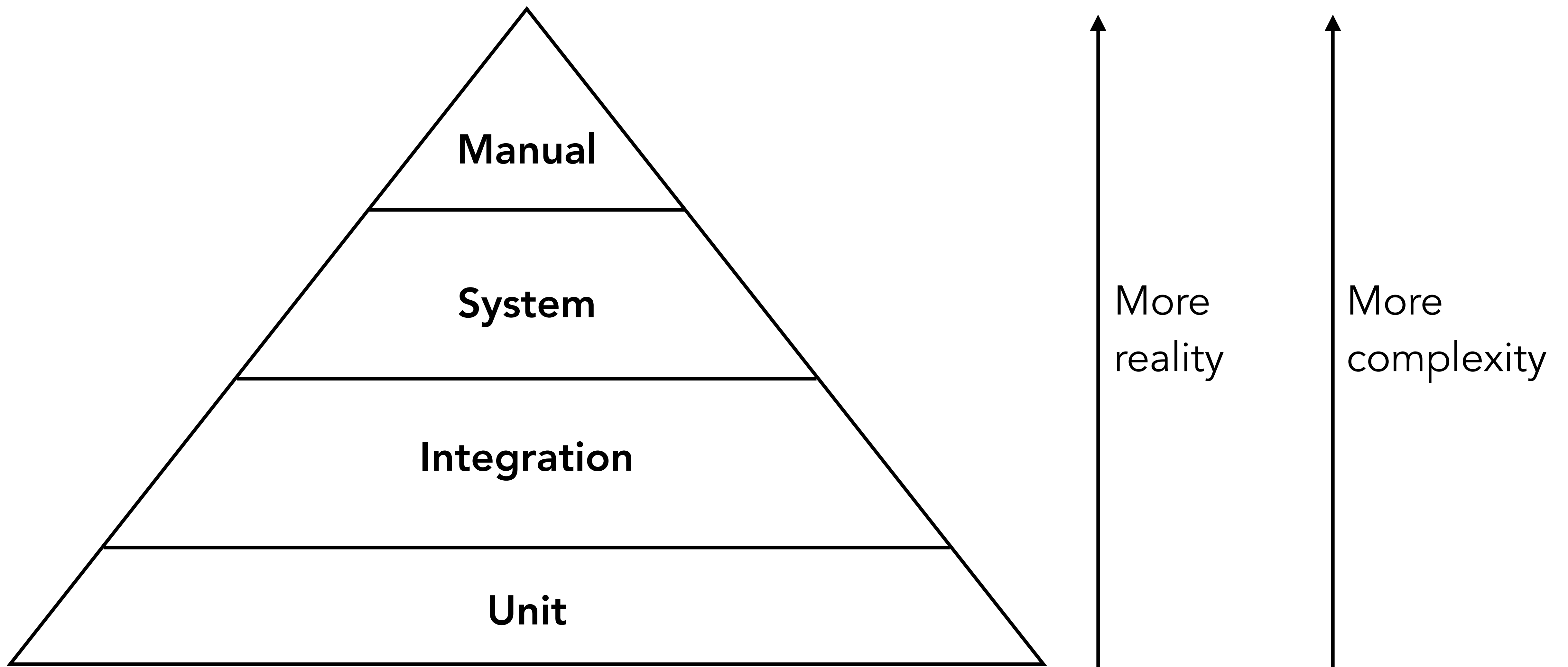
How much should I do of each?

Testers have to decide whether to invest more in unit testing or in system testing as well as determine which components should be tested via unit testing and which components should be tested via system testing. A wrong decision might have a considerable impact on the quality of the system: a wrong level might cost too much resources and might not find sufficient faults.

While we still have no clear empirical answer to this question, practitioners have been proposing different ways to make this decision.

One of the most famous diagrams that could help us in this discussion is the so-called **testing pyramid**.

Testing Pyramid



by Martin Fowler, https://martinfowler.com/bliki/TestPyramid.html

When should I write unit tests?

When the component is a single piece of business logic of the software system.

If we think of enterprise / business systems, most of them are about "transforming data". Such business logics is often expressed by means of entity classes (e.g., an Invoice class and an Order class) exchanging messages. Business logic often does not depend on external services and so it can easily be tested and fully exercised by means of unit tests. Unit tests give testers full control in terms of the input data, as well as full observability in terms of asserting that the behavior was as expected.

If you have a piece of code that deals with specific business logic but you are not able to test it via unit tests (e.g., it is only possible to exercise that business logic with the full system running), it is probably because of previous design or architectural decisions that prevent you from writing unit tests. The way you design your classes has a high impact on how easy it is to write unit tests for your code. We will discuss more about design for testability in a future lecture class.

When should I write integration tests?

Whenever the component under test interacts with an external component (e.g., a database or a web service) integration tests are appropriate.

Following our example in the integration testing section, a Data Access Object class is better tested at the integration level.

Again, note that integration tests are more expensive and harder to set up than a unit test. Therefore, making sure that the component that performs the integration is solely responsible for that integration and nothing else (i.e., no business rules together with integration code), will reduce the cost of the testing.

When should I write system tests?

As we know, system tests are very costly. This makes it impossible for testers to re-test their entire system at system level. Therefore, the suggestion here is to use a risk-based approach. What are the absolutely critical parts of the software system under test? In other words, what are the parts of the system on which a *fault* would have a high impact? These are the ones where the tester should focus on with system tests.

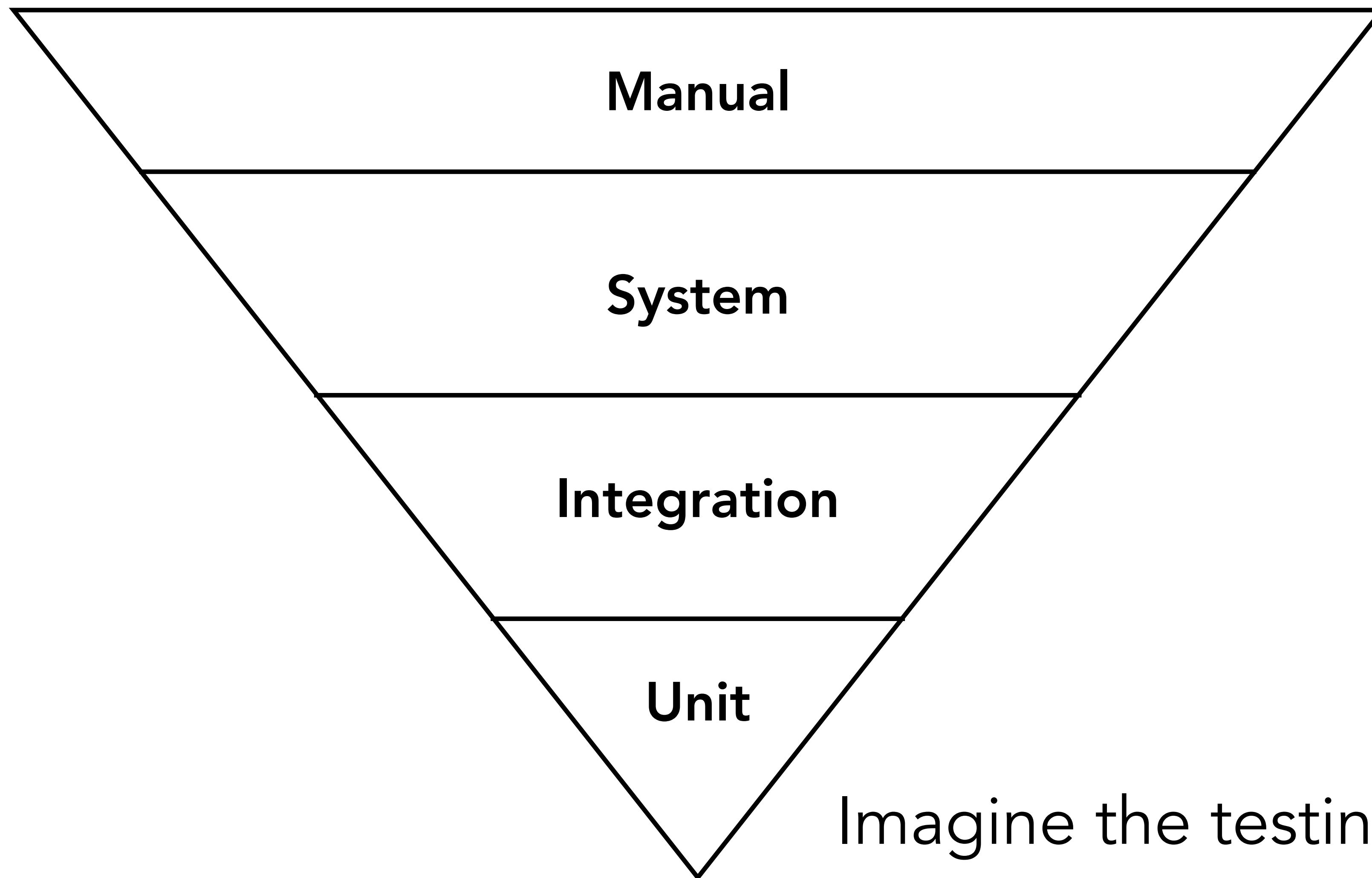
Of course, such critical parts must also be tested at other levels. Remember: a single technique is usually not enough to identify all *faults*.

When should I perform manual tests?

Manual testing has lots of disadvantages, but is sometimes impossible to avoid. Even in cases where automation is fully possible, manual exploratory testing can be useful.

On the other hand, those who apply the testing pyramid try to avoid the so-called ice-cream cone anti-pattern.

When should I perform manual tests?



Imagine the testing pyramid upside down. In this new version, manual testing has the largest area, which means more effort on manual testing (!!!).

When should I perform manual tests?

Unfortunately, it is common to see development teams relying mostly on **manual tests** in their quality assurance processes. Often, these teams also have a large number of system tests. This is not because they believe system tests are more efficient, but because the system was badly designed, so that it is impossible to carry out unit and integration tests.

Testing pyramid at Google

Winters, T., Manshreck, T., Wright, H.
“Software Engineering at Google: Lessons Learned from Programming Over Time”
O'Reilly, 2020, chapters 11 and 12.

Test types

Level / Phase

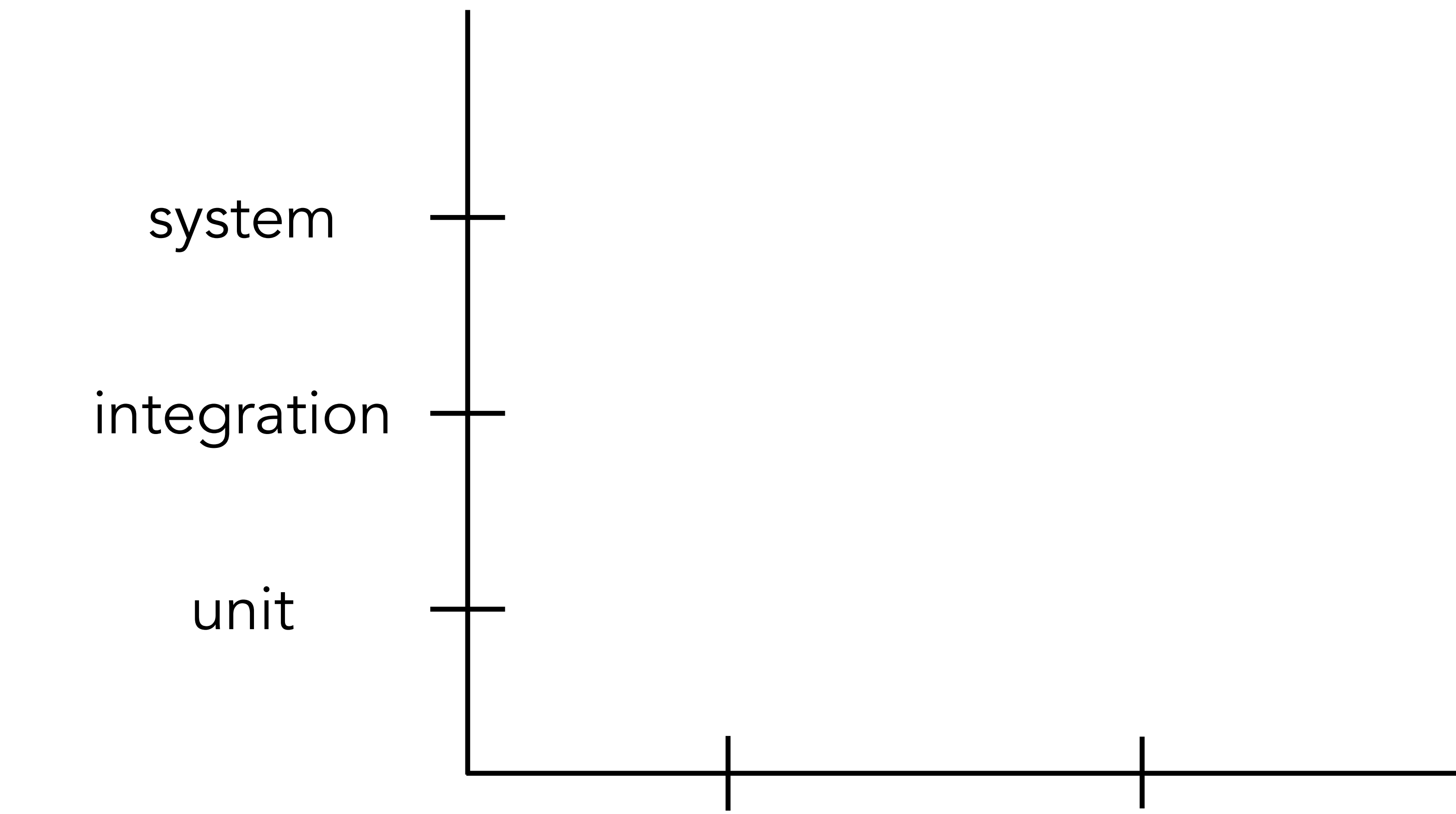
system

integration

unit

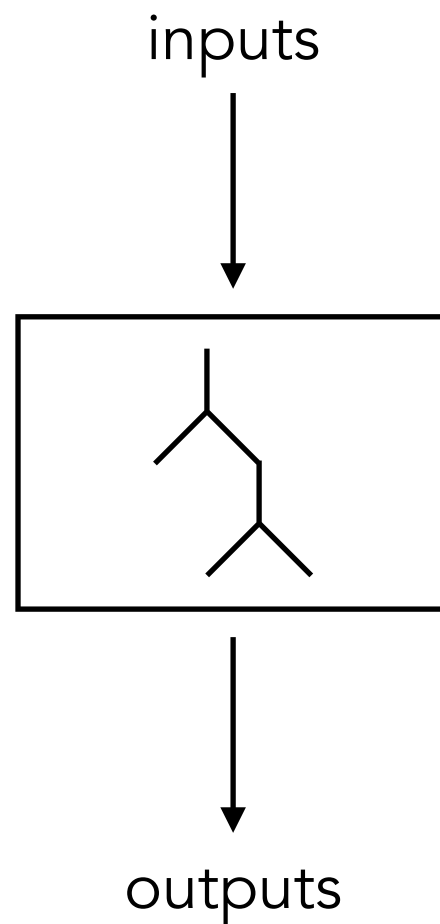
Strategy / Technique

Test types



Test types

White-box testing



Knowledge sources

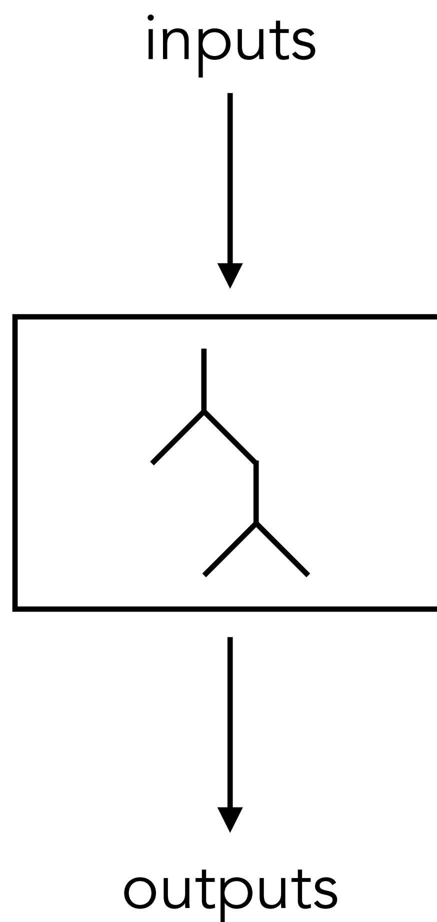
Control flow graphs

Data flow graphs

Cyclomatic complexity

…

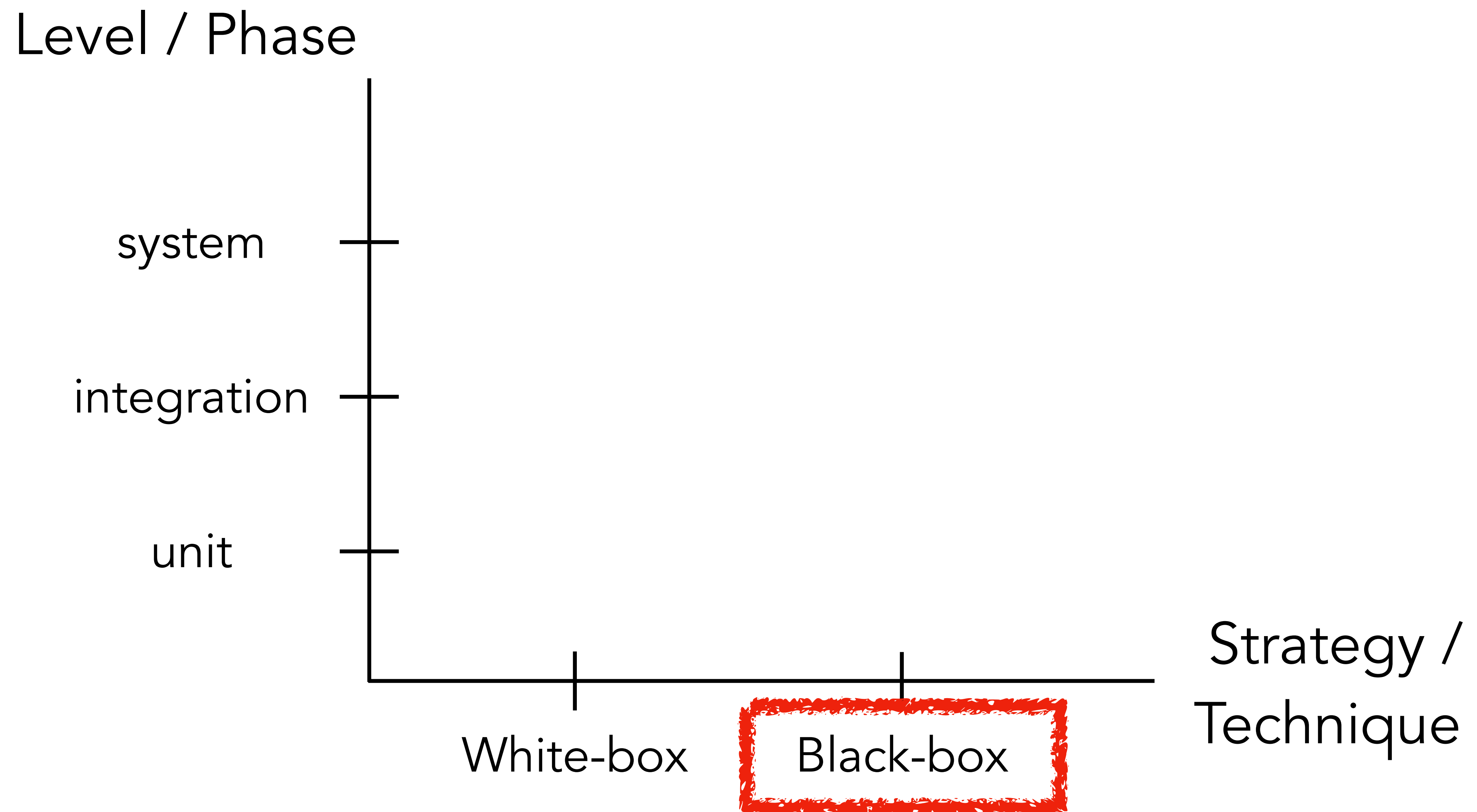
White-box testing



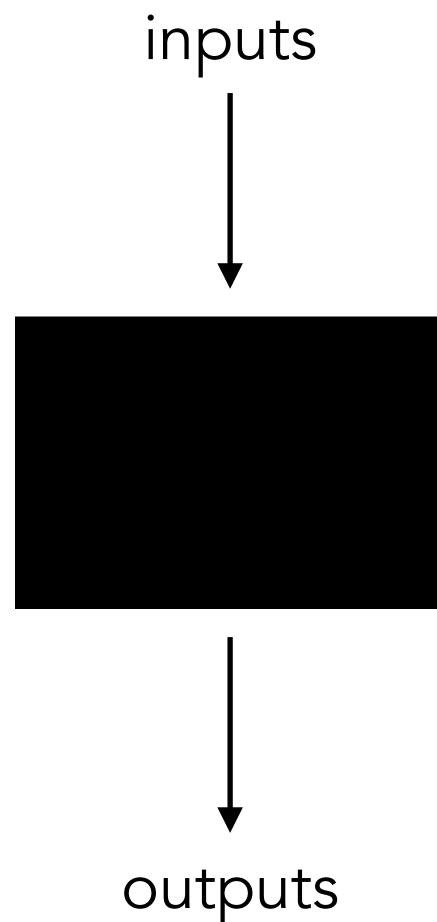
Techniques

Control flow testing/coverage
Data flow testing/coverage
Class testing/coverage
Mutation testing
…

Test types



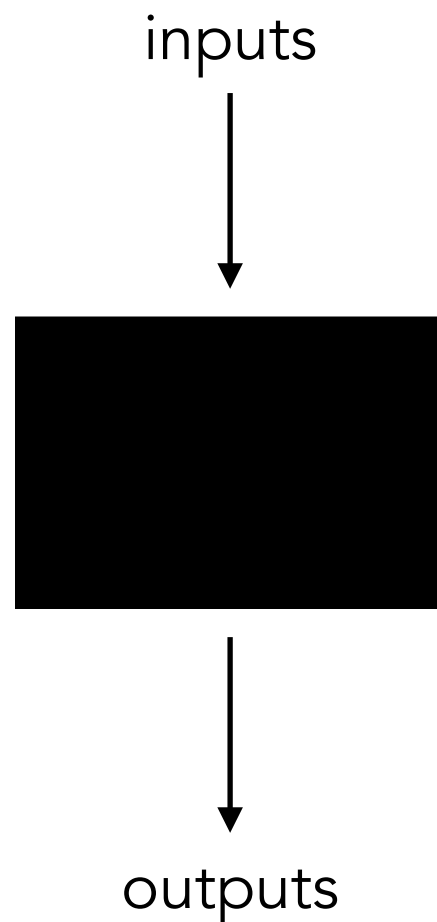
Black-box testing



Knowledge sources

Requirements document
Specifications
User manual
Models
Domain knowledge
Intuition
Experience
…

Black-box testing



Techniques

Equivalence class partitioning
Category partition
Boundary value analysis
Cause effect graphing
Error guessing
Random testing
State-transition testing
Scenario-based testing
…

Equivalence Class Partitioning

Programs are usually too complex to be tested with just a single test. There are different cases in which the program is executed and its execution often depends on various factors, such as the input to the program.

Equivalence Class Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc. In this technique, input data units are divided into equivalent partitions that can be used to derive test cases which reduces time required for testing because of small number of test cases.

Let's use a small program as an example. The specification below talks about a program that decides whether a given year is a leap year or not.

Equivalence Class Partitioning

example

Given a specific year as an input, the program should return true if the provided year is a leap year and false if it is not.

A year is a leap year if:

- the year is divisible by 4;
- and the year is not divisible by 100;
- except when the year is divisible by 400 (because then it is a leap year)

Equivalence Class Partitioning

example

Given a specific year as an input, the program should return true if the provided year is a leap year and false if it is not.

A year is a leap year if:

- the year is divisible by 4;
- and the year is not divisible by 100;
- except when the year is divisible by 400 (because then it is a leap year)

By looking at the requirements above, we can derive the following classes/partitions:

- Year is divisible by 4, but not divisible by 100 (i.e., leap year, TRUE)
- Year is divisible by 4, divisible by 100, divisible by 400 (i.e., leap year, TRUE)
- Year is not divisible by 4 (i.e., not leap year, FALSE)
- Year is divisible by 4, divisible by 100, but not divisible by 400 (i.e., not leap year, FALSE)

Equivalence Class Partitioning

The partitions above are not tests that we can implement directly because each partition might be instantiated by an infinite number of inputs. For example, for the partition "year not divisible by 4", there are infinitely many numbers that are not divisible by 4 which we could use as concrete inputs to the program. So how do we know which concrete input to instantiate for each of the partitions?

Each partition exercises the program in a certain way. In other words, all input values from one specific partition will make the program behave in the same way. Therefore, any input we select should give us the same result. We assume that, if the program behaves correctly for one given input, it will work correctly for all other inputs from that class. This idea of inputs being equivalent to each other is called **equivalence partitioning**. Thus, it does not matter which precise input we select and one test per partition should be enough.

Equivalence Class Partitioning

- Year is divisible by 4, but not divisible by 100 (i.e., leap year, TRUE)
- Year is divisible by 4, divisible by 100, divisible by 400 (i.e., leap year, TRUE)
- Year is not divisible by 4 (i.e., not leap year, FALSE)
- Year is divisible by 4, divisible by 100, but not divisible by 400 (i.e., not leap year, FALSE)

With the classes we derived above, we could write 4 tests (i.e., one test for each class/partition). As any input can be used for a given partition, the following inputs can be used for the partitions:

- 2016, divisible by 4, not divisible by 100.
- 2000, divisible by 4, also divisible by 100 and by 400.
- 39, not divisible by 4.
- 1900, divisible by 4 and 100, not by 400.

Category-Partition

So far we have derived partitions by just looking at the specification of the program. We basically used our experience and knowledge to derive the tests. We will now discuss a more systematic way of deriving these partitions: the **Category-Partition** method.

This method provides us with a systematic way of deriving tests, based on the characteristics of the input parameters. It also reduces the number of tests to a practical number.

Category-Partition

recipe

1. Identify the parameters, or the input for the program. For example, the parameters your classes and methods receive.

2. Derive characteristics of each parameter. For example, an `int` year should be a positive integer number between 0 and infinite.

    - Some of these characteristics can be found directly in the specification of the program.

    - Others might not be found from specifications. For example, an input cannot be `null` if the method does not handle that well.

3. Add constraints in order to minimize the set of tests.

    - Identify invalid combinations. For some characteristics it might not be possible to combine them with other characteristics.

    - Exceptional behavior does not always have to be combined with all of the values of the other inputs. For example, trying a single `null` input might be enough to test that corner case.

4. Generate combinations of the input values. These are the tests.

Category-Partition

example

Requirement: Christmas discount

The system should give a 25% discount on the cart when it is Christmas season. The method has two input parameters: the total price of the products in the cart, and the date. When it is not Christmas, it just returns the original price; otherwise it applies the discount.

Category-Partition

example

1. We have two parameters:

    - The current date

    - The total price

2. For each parameter we define the characteristics as:

    - Based on the requirements, the only important characteristic is that the date can be either Christmas or not.

    - The price can be a positive number, or in certain circumstances it may be 0.

3. The number of characteristics and parameters is not too large in this case. Constraint, negative prices are not allowed.

4. We combine the other characteristics to get the following tests:

    - Positive price at Christmas

    - Positive price and not at Christmas

    - Empty cart (i.e., price 0) at Christmas

    - Empty cart (i.e., price 0) not at Christmas

    - Negative price at Christmas or not, does not really matter as negative prices are not allowed

References

- Chapter 2 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Vocke, Ham. The Practical Test Pyramid (2018), https://martinfowler.com/articles/practical-test-pyramid.html.
- Fowler, Martin. TestingPyramid (2012). https://martinfowler.com/bliki/TestPyramid.html
- Wikipedia. Exploratory testing. https://en.wikipedia.org/wiki/Exploratory_testing. Last access on March, 2020.
- Winters, T., Manshreck, T., Wright, H. Software Engineering at Google: Lessons Learned from Programming Over Time. O'Reilly, 2020. Chapters 11 and 12.
- Graham, D., Van Veenendaal, E., & Evans, I. (2008). Foundations of software testing: ISTQB certification. Cengage Learning EMEA. Chapter 4.
- Pezzè, M., & Young, M. (2008). Software testing and analysis: process, principles, and techniques. John Wiley & Sons. Chapter 10.
- Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), 676-686.