Software Engineering VU [194.020]

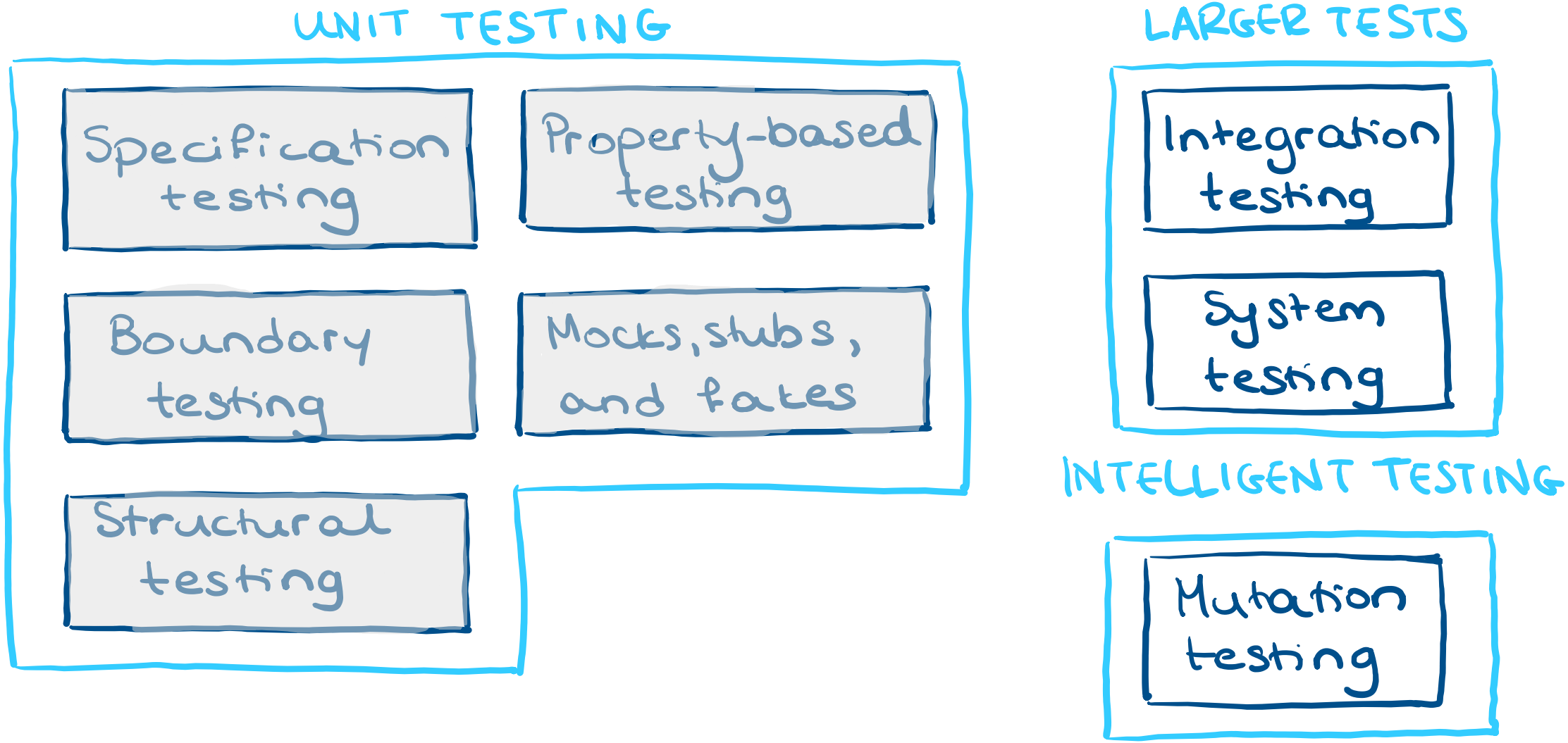
Maria Christakis

TU Wien

https://mariachris.github.io

Summary

Effective and systematic testing



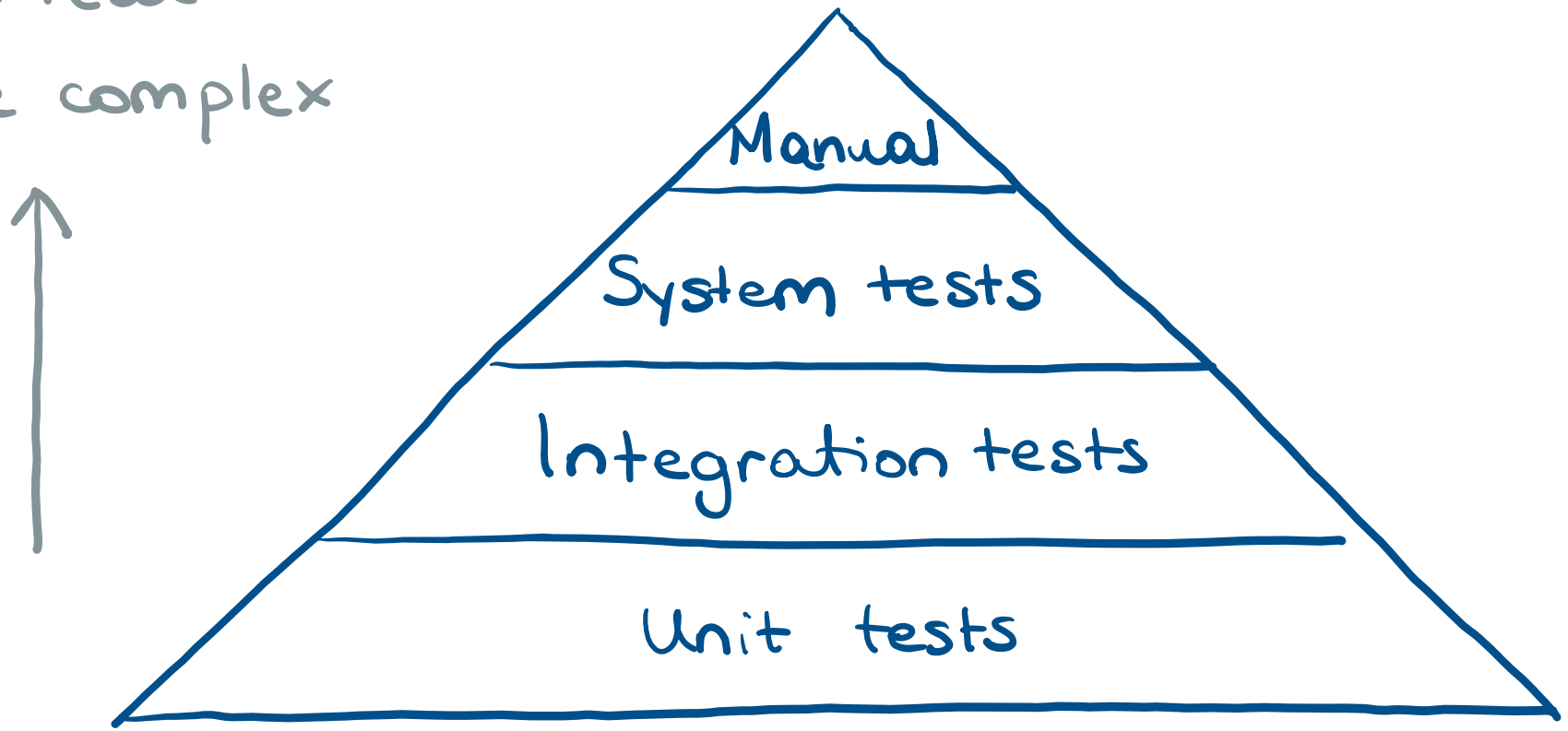
Testing

- Integration testing
- System testing
- Mutation testing
- Test code quality
- Wrap up

The testing pyramid

More real

More complex



Larger tests

- Some parts of the system may require the developer to write larger tests
  - Integration tests
  - System tests
- To devise larger tests, the developer uses the same techniques as for unit testing but looking at larger parts of the software system

When to use larger tests

- You have exercised each class individually, but the overall behavior is composed of many classes, and you want to see them work together
  - E.g., think of a set of classes that calculates the final cost of a shopping cart
  - The final cost is calculated by adjusting the delivery costs based on the number of items, charging extra for certain heavy items, etc.

When to use larger tests

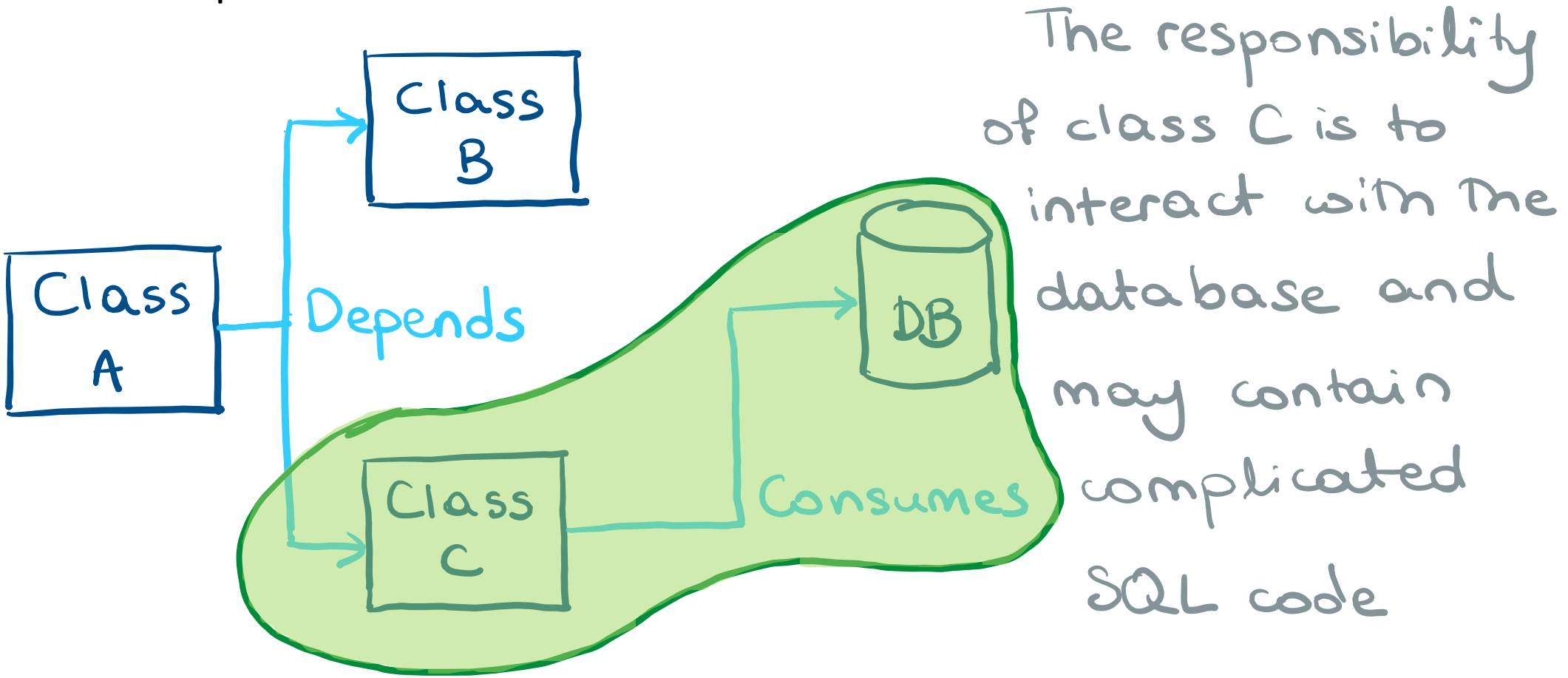
- The class you want to test is a component in a larger plug-and-play architecture
  - E.g., think of a plugin in your favorite IDE (IntelliJ)
  - You can develop the logic of the plugin, but many actions will only happen when IntelliJ calls the plugin and passes parameters to it

Integration testing: Definition

Integration testing is the test level we use to test the integration between our code and external parties

Integration testing: Definition

Integration testing is the test level we use to test the integration between our code and external parties



Integration testing: Example

- Consider the CK tool (https://github.com/mauricioaniche/ck)
  - Calculates code metrics for Java code, such as coupling between objects (CBO)
  - CBO counts the number of other classes that each class depends on

```
class A {
  private B b;

  public void action() {
    new C().method();
  }
}
```

A depends on B and C and CBO is 2

https://mazko.github.io/jsjavaparser/

Integration testing: Example

- CK relies on Eclipse JDT, a library that is part of the Eclipse IDE
  - JDT enables building abstract syntax trees (ASTs)
  - CK builds ASTs using JDT
  - CK then traverses them to calculate different metrics, like CBO

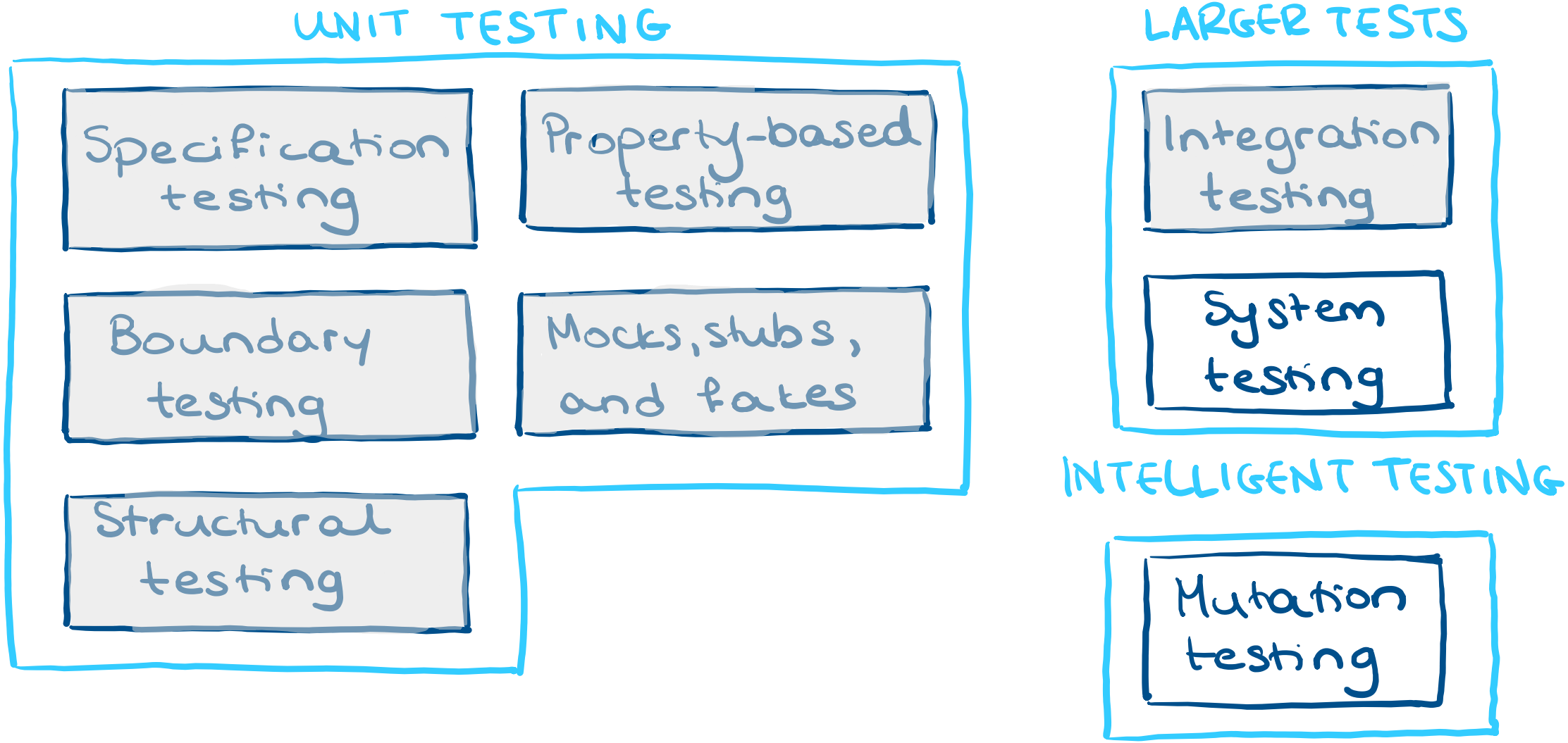
Integration testing: Example

- How can we write tests for the CBO class?
  - Start up JDT
  - Ask JDT to build an AST out of a small but real Java class
  - Use CBO to traverse the AST
  - Assert the expected result

Testing

- Integration testing
- System testing
- Mutation testing
- Test code quality
- Wrap up

Effective and systematic testing



System testing: Definition

System testing is testing the system in its entirety, as clients would

We do not care how the system works from the inside; we only care that, given input X, the system will provide output Y

System testing: Example

- Think of a web application for a pet clinic
  - Visit a web page that lists all scheduled appointments for today
  - Click the New Appointment button
  - Fill out the pet and owner name
  - Select an available time slot
  - Check that we go back to the appointments page
  - Check that the page shows the new appointment

How to write larger tests

- Use the requirement and its boundaries
- Use the structure of the code
- Use the properties it should uphold
- Use everything we used for unit tests when testing larger components
  - This means there will be many more tests to engineer

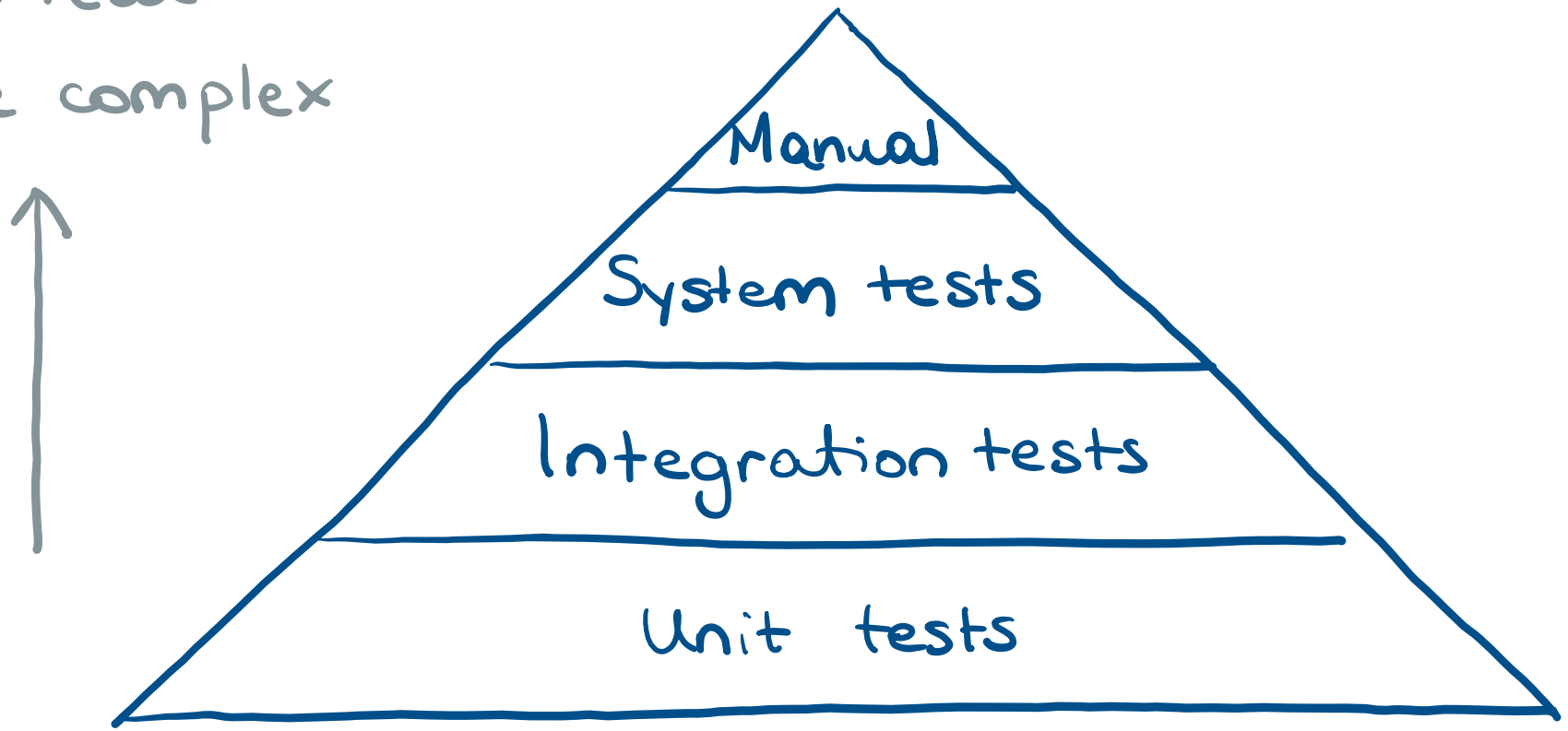
When to write larger tests

- Exercise everything at the unit level
- Exercise the most important behavior in larger tests
- Testing mantra: A good test is cheap to write but finds important bugs
  - Perform a simple cost/benefit analysis for larger tests
  - How much will it cost to write and run?
  - What bugs will it catch? Does it cover new functionality, not covered by unit tests?

The testing pyramid

More real

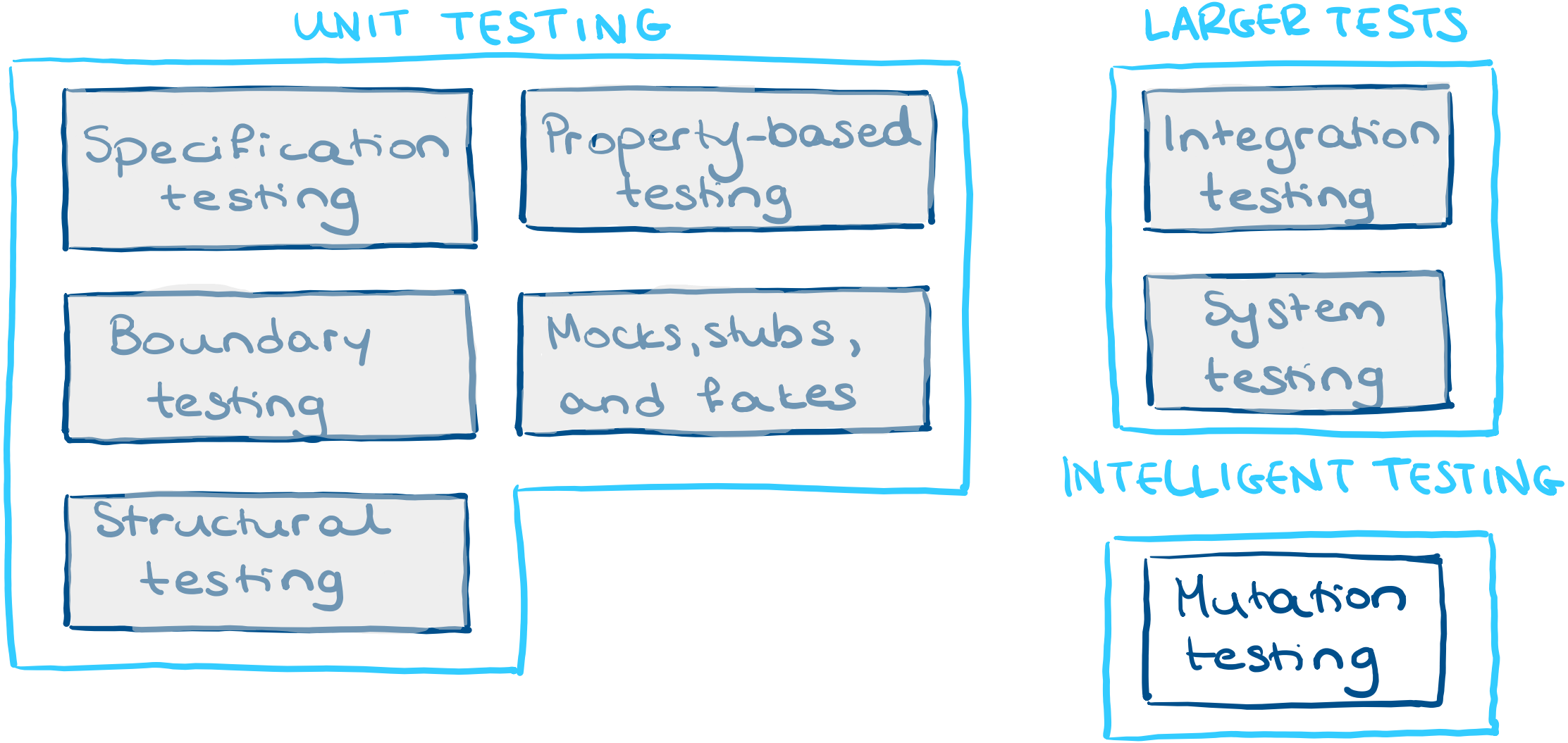
More complex



Testing

- Integration testing
- System testing
- Mutation testing
- Test code quality
- Wrap up

Effective and systematic testing



Mutation testing: Definition

- We insert a bug in the code and check whether the test suite breaks
  - The buggy version is called a mutant of the original version of the code
- If it breaks, great!
  - We say that the test suite kills the mutant
- If it does not break, we have found something to improve in our test suite
  - We say that the mutant survives
- A test suite achieves 100% mutation coverage if it kills all possible mutants
- Mutation testing measures the fault detection capability of a test suite

Mutation testing: Assumptions

- The competent programmer hypothesis assumes that the program is written by a competent programmer and that the implemented version is either correct or differs from the correct program by a combination of simple errors

- The coupling effect says that a complex bug is caused by a combination of small bugs, so if your test suite can catch simple bugs, it will also catch complex ones

Mutation testing tool for Java

- Check out Pitest: https://pitest.org/
- Mutation operations: https://pitest.org/quickstart/mutators/
- Small example: https://pitest.org/quickstart/basic_concepts/
- Reports:

https://codesoapbox.dev/wp-content/uploads/2022/06/image-1.png

https://codesoapbox.dev/wp-content/uploads/2022/06/image-5.png

Mutation testing: Pros and Cons

\+ Highly beneficial in revealing additional weaknesses in the test suite, e.g., compared to branch coverage

– Very costly – it requires generating many mutants and executing the whole test suite with each one

Tip:

Apply mutation testing in smaller, more sensitive parts of the system; it may give valuable insights about what else to test

Testing

- Integration testing
- System testing
- Mutation testing
- Test code quality
- Wrap up

Principles of maintainable test code

- Tests should be fast
  - Slower test suites force us to run them less often
  - Use stubs or mocks to replace slow test components
  - Redesign the production code so slower code can be tested separately
  - Move slower tests to a different test suite that can run less often, e.g., when you modify production code that has a slow test tied to it, or before committing

Principles of maintainable test code

- Tests should be cohesive, independent, and isolated
  - A test should test a single functionality of the system
  - Complex test code reduces understanding and makes maintenance more difficult
  - A test should not depend on others to succeed, e.g., by setting up the state for it
  - Each test should set up the state it needs and then clean it up

Principles of maintainable test code

- Tests should have a reason to exist
  - Tests should either help find bugs or document behavior
  - You don’t want tests that only increase code coverage
  - You must maintain all your tests
  - The perfect test suite is the one that can detect all the bugs with the minimum number of tests

Principles of maintainable test code

- Tests should be repeatable and not flaky
  - A repeatable tests gives the same result no matter how many times it is executed
  - It is hard to know whether a flaky test is failing because the behavior is buggy or because it is flaky
  - Developers may lose their trust in the test suite and deploy their system even though the tests fail

Principles of maintainable test code

- Tests should be repeatable and not flaky
  - A test can become flaky for many reasons
    - Because it depends on external or shared resources – e.g., it depends on a database that might not be available, may contain data that the test does not expect, may be shared with another developer running the test suite
    - Due to improper time-outs – e.g., when testing a web application, the web service might be slower than normal, and the test might fail if it doesn't wait long enough
    - Because of a hidden interaction between test methods – e.g., when a test does not clean its state well enough

Principles of maintainable test code

- Test should have strong assertions
  - Assertions should be as strong as possible to fully validate the behavior and break if there is any slight change in the output
  - E.g., think of a method `calculatePrice()` in a `ShoppingCart` that changes two properties, `finalPrice` and `taxPaid`
  - If your tests only ensure the value of the `finalPrice` property, a bug may happen in the way `taxPaid` is set, and the tests will not notice it

Principles of maintainable test code

- Tests should break if the behavior changes
  - If you break the behavior and the test suite is still green, something is wrong with it
  - This may happen because of weak or missing assertions
  - The TDD cycle allows developers to always see their tests break

Principles of maintainable test code

- Tests should have a single and clear reason to fail
  - Your test code should help you understand what causes a bug
  - A test should exercise only one behavior of the system
  - It should have a name that indicates this behavior
  - Anyone should be able to understand the input values and variable names
  - The assertions and expected values should be clear

Principles of maintainable test code

- Tests should be easy to write
  - If tests are hard to write, you will give up
  - Writing unit tests is easy most of the time
  - It gets complicated when the code under test requires too much infrastructure, e.g., a database
  - Invest time and effort in writing good test infrastructure

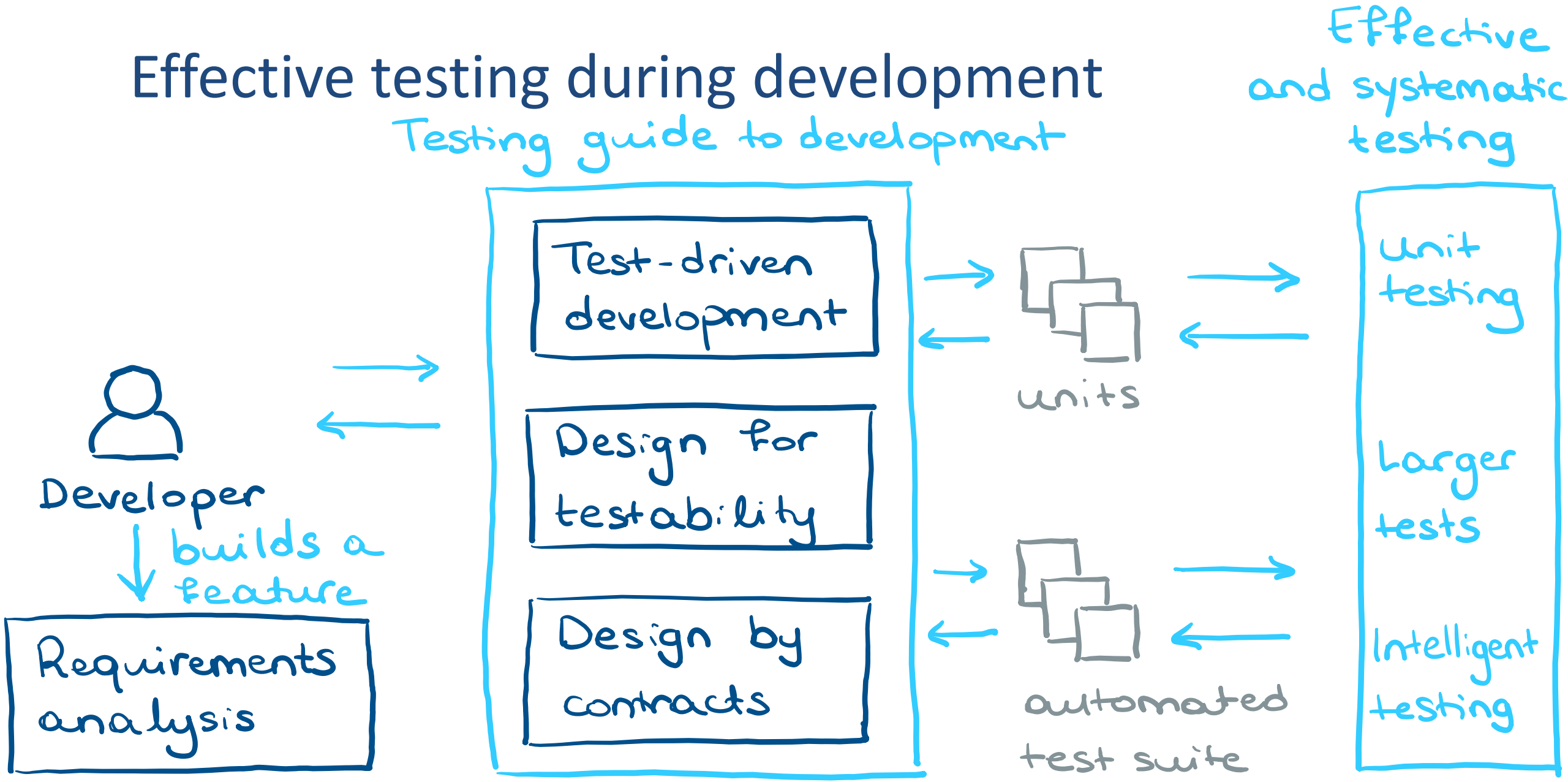
Principles of maintainable test code

- Tests should be easy to change and evolve
  - Production code will change, which will force the tests to change
  - When implementing tests, we should make sure that changing them is not too painful
  - E.g., if you see the same code snippet in 10 different test methods, consider extracting it
  - The more your tests know about how the production code works, the harder it is to change them

Testing

- Integration testing
- System testing
- Mutation testing
- Test code quality
- Wrap up

Effective testing during development



Wrap up

- Although the flow for effective and systematic testing looks linear, it is an iterative process
- The more you test your code using different techniques, the greater the chances of revealing new bugs
- Bugs will still happen – the software systems of today are very complex with dozens of different components working together
- Intelligent testing is the way to go, that is, having computers explore software systems for us
  - Taught in detail in Advanced Software Engineering

Suggested reading

From Book: "Effective Software Testing A Developer's Guide" by M. Aniche

- Sections 9.1, 9.1.2
- Introduction of Section 9.3
- Sections 9.4.1, 9.4.2
- Section 3.11
- Section 10.1
- Sections 11.1, 11.2