Software Engineering VU [194.020]

Maria Christakis

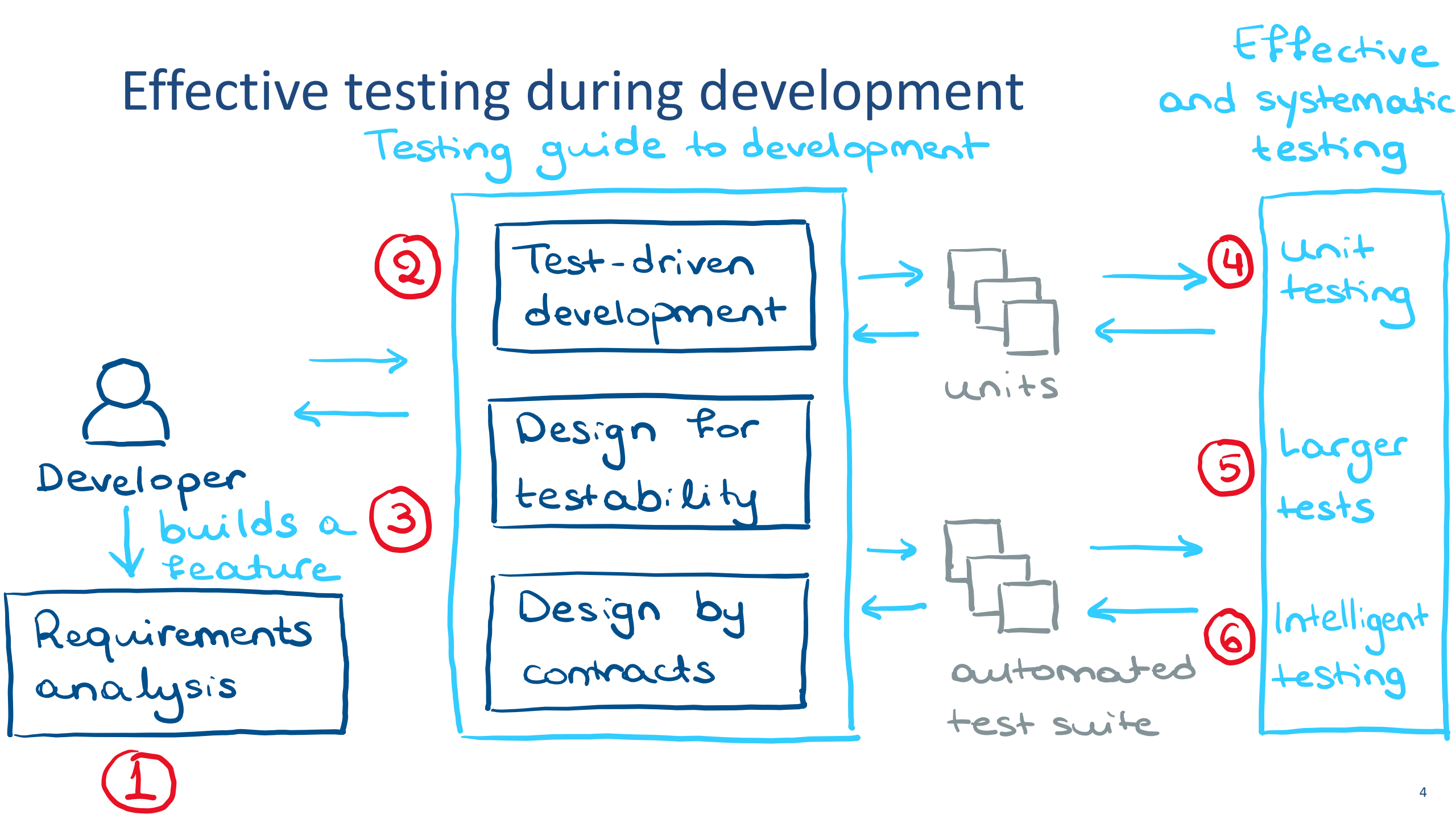
TU Wien

https://mariachris.github.io

Implementation

- Effective and systematic testing
- Developing for testability
- Test-driven development

Effective testing during development



Requirement

- The developer receives some requirement, e.g., in UML
- The developer builds up some understanding (requirement analysis)
- The developer starts writing code

Test-driven development

- The developer performs short test-driven development (TDD) cycles
  - These cycles give the developer rapid feedback about whether the code they just wrote makes sense
  - They also support the developer through the many refactorings that occur when a new feature is being implemented

Contracts and testability

- Requirements are often large and complex and are rarely implemented by a single class or method
- The developer creates several units (classes and methods) with different contracts that together implement the required functionality
- The developer should design the implementation with testability in mind – writing classes that are easy to test is challenging

Unit testing

- Once the developer believes the requirement is complete and is satisfied with the units, testing begins

- The first step is to exercise each unit
  - Domain testing
  - Boundary testing
  - Structural testing
  - …

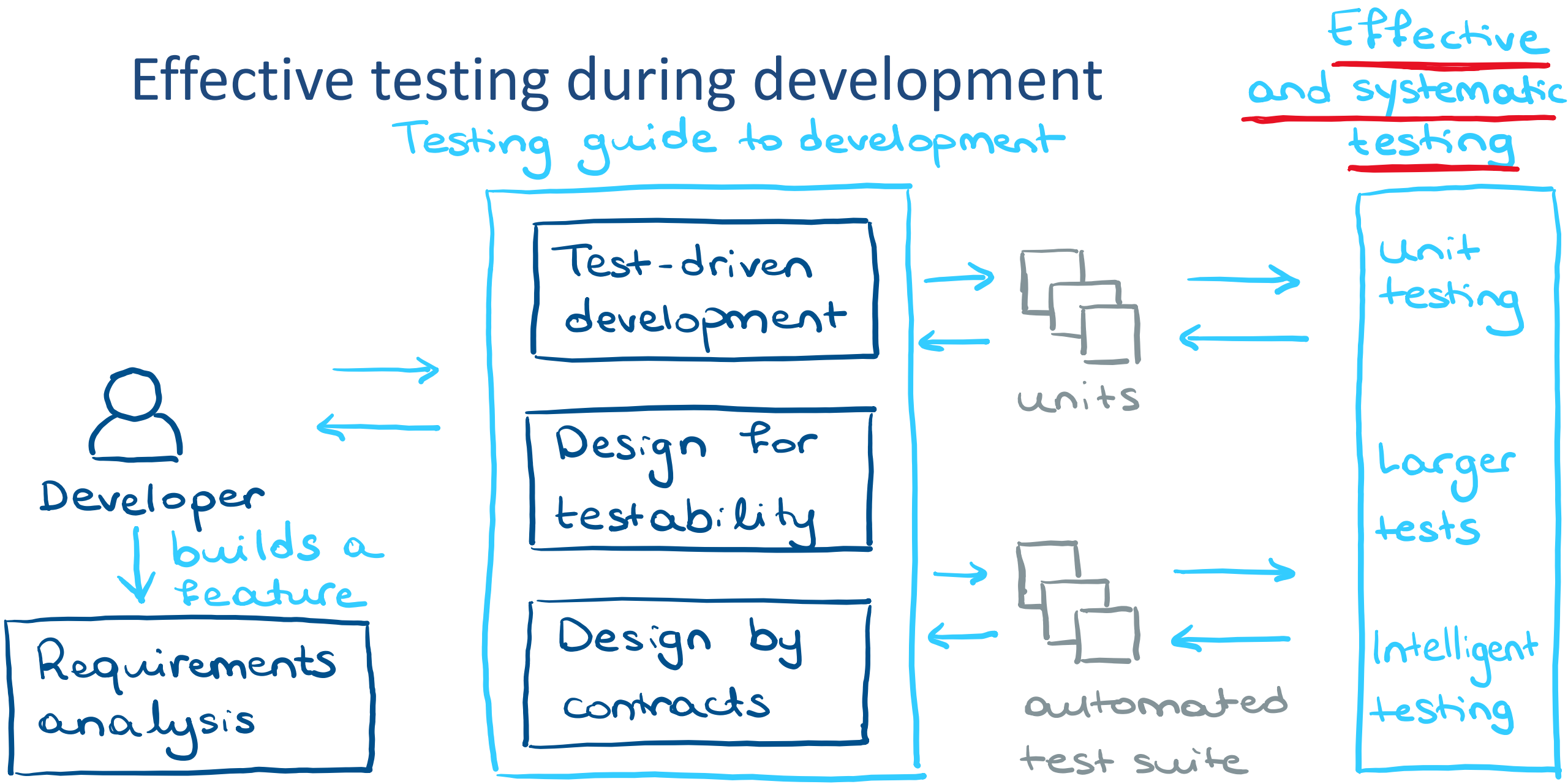
Larger tests

- 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

Intelligent testing

- When the developer has engineered tests using the various techniques, they apply automated, intelligent testing tools to look for tests that humans are not good at spotting
  - Test case generation
  - Mutation testing
  - …

Effective testing during development



Effective and systematic testing

- Being effective means we focus on writing the right tests
  - Testers want to maximize the number of detected bugs while minimizing the effort required to detect them
- Being systematic means that for a given piece of code, any developer should come up with the same test suite
  - We should be able to systematize our processes to reduce the dependency on the developer who is doing the job

Isn’t the cost of testing too high?

- Yes, but it’s worth it!
  - The cost of bugs that happen in production often outweighs the cost of prevention
  - Teams that produce many bugs spend a lot of time as follows: developers write bugs, customers find bugs, developers fix bugs, customers find other bugs, etc.
  - Once developers are used to engineering tests, they can do it much faster

Principles of software testing

1. Exhaustive testing is impossible

- We do not have the resources to completely test our programs
- Testing all possible situations of a system might be impossible even with unlimited resources
  - E.g., imagine 300 Boolean flags
- Prioritize what to test, i.e., write effective tests

Principles of software testing

2. Know when to stop testing

- Too few tests may leave us with a software system that is full of bugs
- Too many tests can be ineffective, costing time and money
- Use adequacy criteria to decide when to stop testing

Principles of software testing

3. Variability is important

- There is no single testing technique that you can always apply to find all bugs
  - Different testing techniques help reveal different bugs
  - This is known as the pesticide paradox: Every method you use to find bugs leaves a residue of subtler bugs against which the method is ineffectual
- Use different testing strategies to minimize the number of bugs

Principles of software testing

4. Bugs happen in some places more than others

- Bugs are not uniformly distributed or equally important

  We don't know the bug distribution in advance, but we know the importance

  - E.g., compare a Payment module with a Marketing module

- Watch and learn from the software system – data other than the source code may help prioritize the testing efforts

Principles of software testing

5. No matter what testing you do, it will never be perfect or enough

- Program testing can be used to show the presence of bugs, but never to show their absence
  - Our test suites, however large they may be, can never ensure that the software system is 100% bug free
- Set expectations (of developers and customers) – bugs will still happen

Principles of software testing

6. Context is king

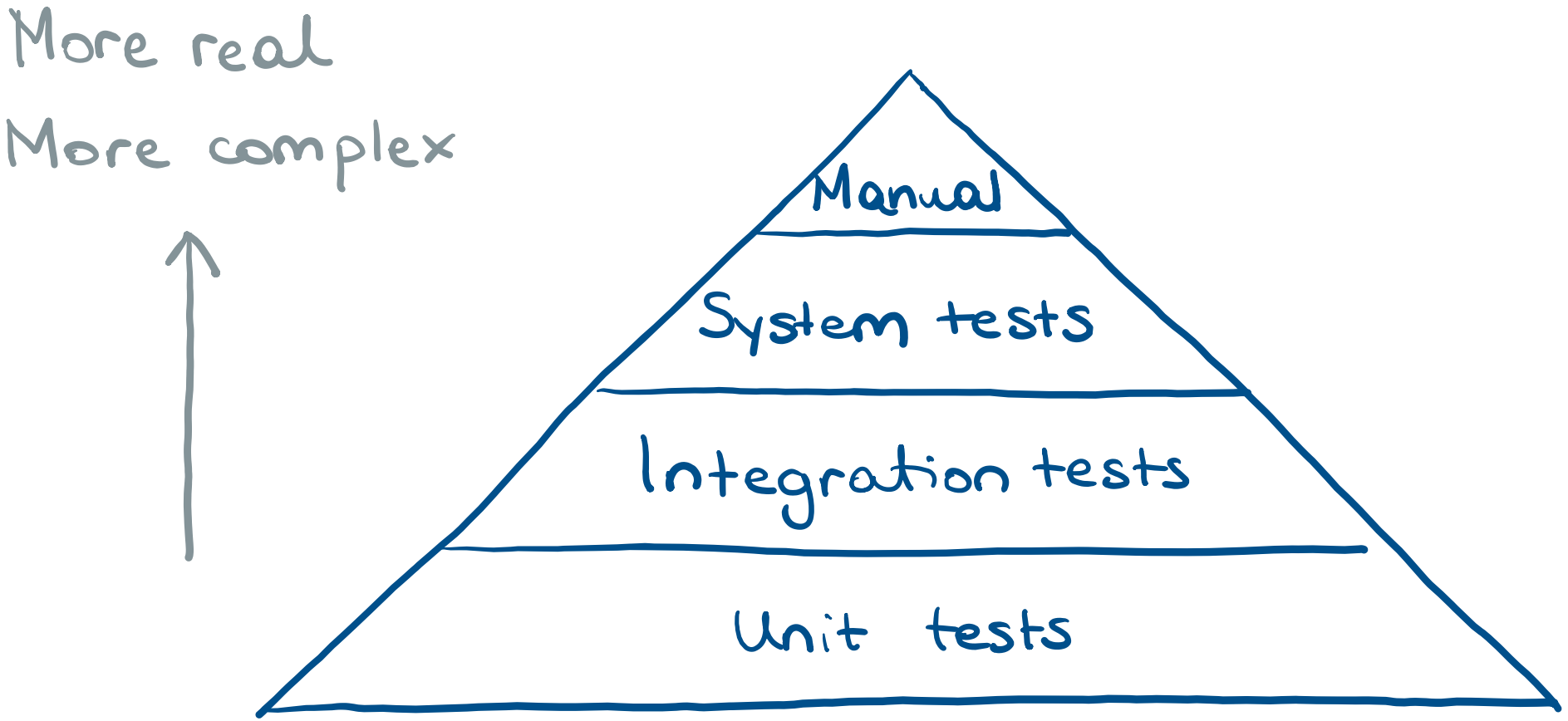
- Testing is context dependent
  - E.g., testing a mobile app is very different from testing software used in a rocket
- Take context into account when devising tests

Principles of software testing

7. Verification is not validation

- Verification is about having the system right; validation is about having the right system
  - Verification ensures that, given a specific requirement, the system implements it correctly
  - Validation focuses on, e.g., collaborating with customers to understand their needs
  - Testers face the absence-of-errors fallacy when they focus on verification but not on validation – we don't want systems that work flawlessly but are useless
- Do not underestimate validation

The testing pyramid



Unit testing: Definition

Unit testing is to test a single feature / unit in isolation, purposefully ignoring the other units of the system

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

A unit test typically tests the software by giving certain parameters to a method and then comparing the return value of this method to the expected result

Unit testing: Pros

- Unit tests are fast
  - Fast test suites give constant feedback about huge parts of the system in little time
- Unit tests are easy to control
  - The input values and expected result value are easy to adapt
- Unit tests are easy to write
  - No complicated setup is required

Unit testing: Cons

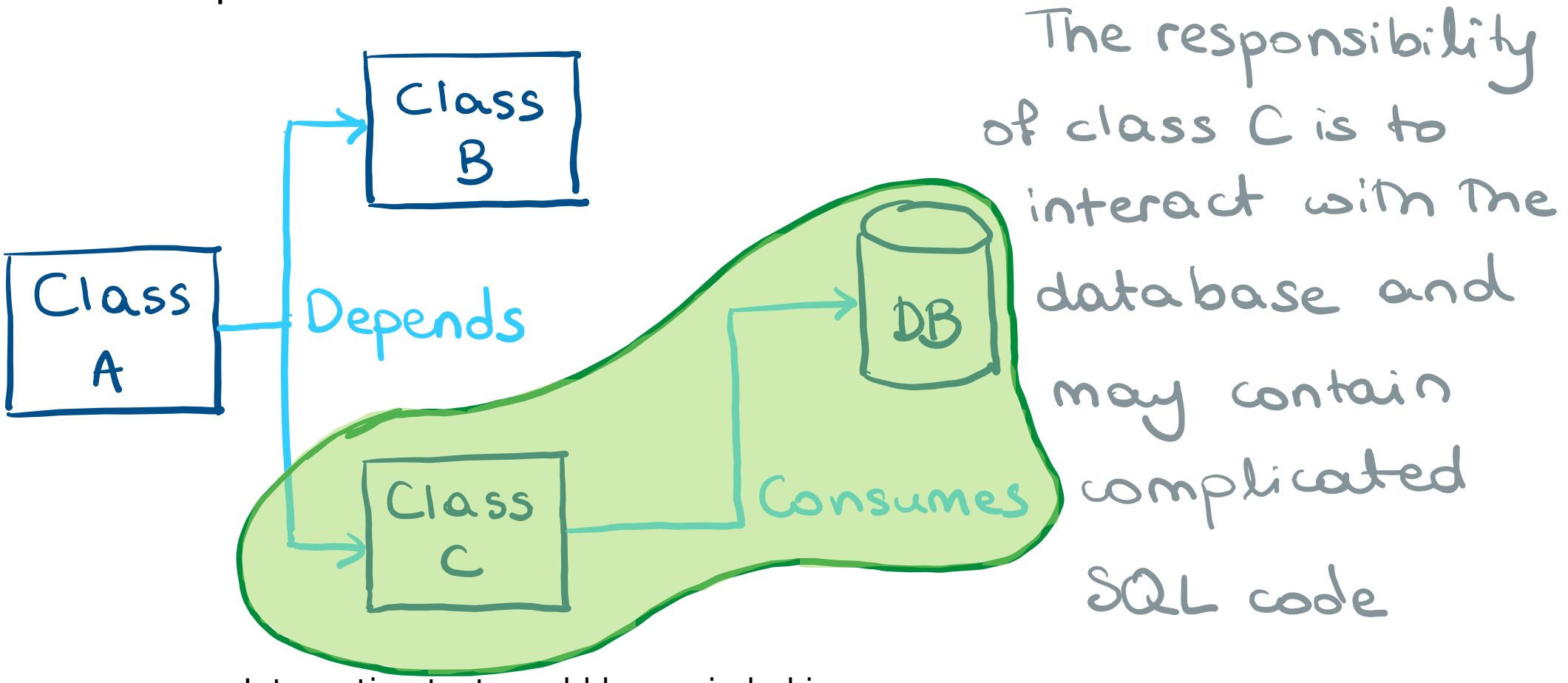
- Unit tests lack reality
  - The large number of classes in a system and their interaction can cause the system to behave differently in its real application
- Some types of bugs are not caught
  - Some bugs only happen in the integration of different components
  - E.g., think of multithreaded code where bugs only appear once threads are running together

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 test would be encircled in green

System testing: Definition

System testing is testing the system in its entirety

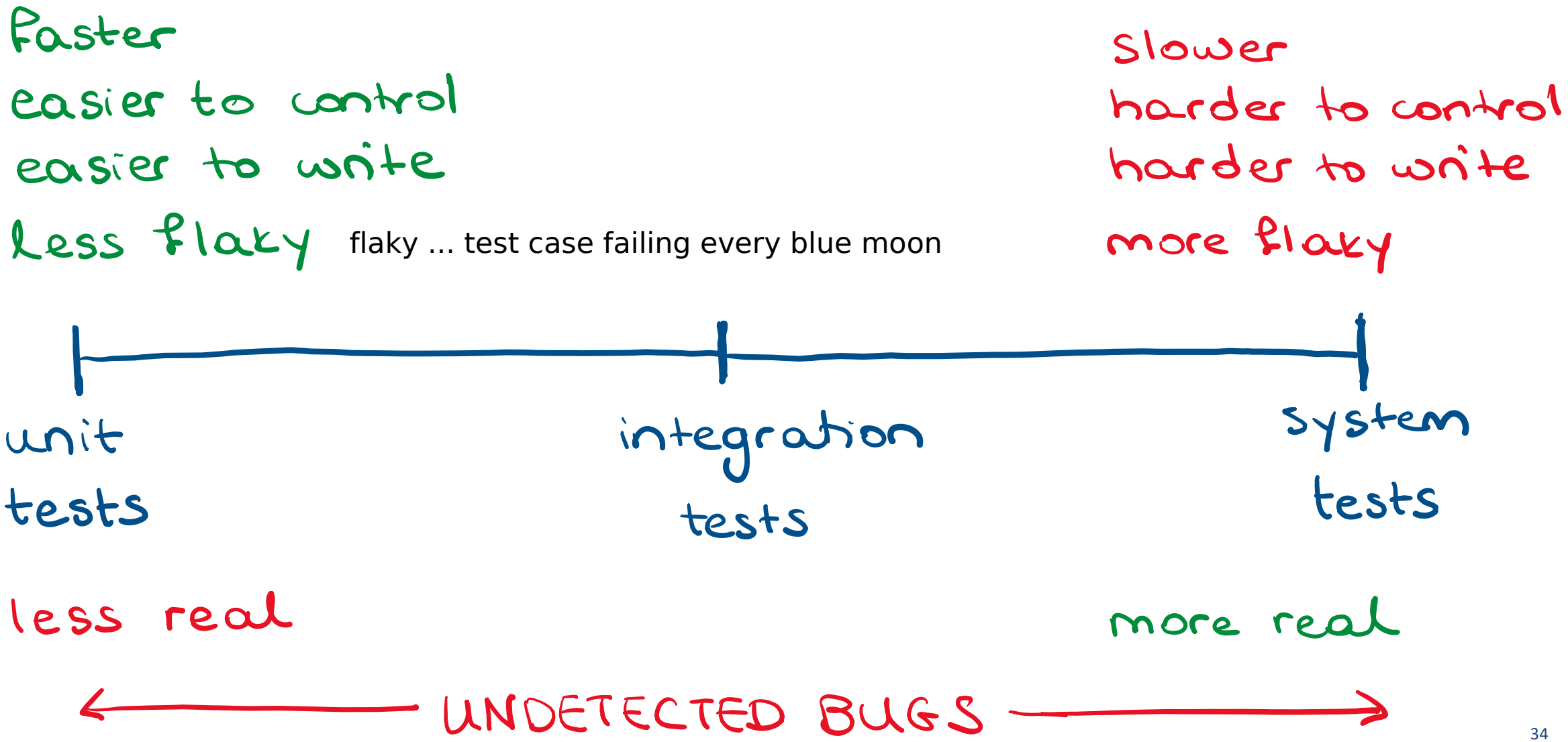
We do not care how the system works from the inside; we only care that, given input X, the system will provide output Y

Manual testing: Definition

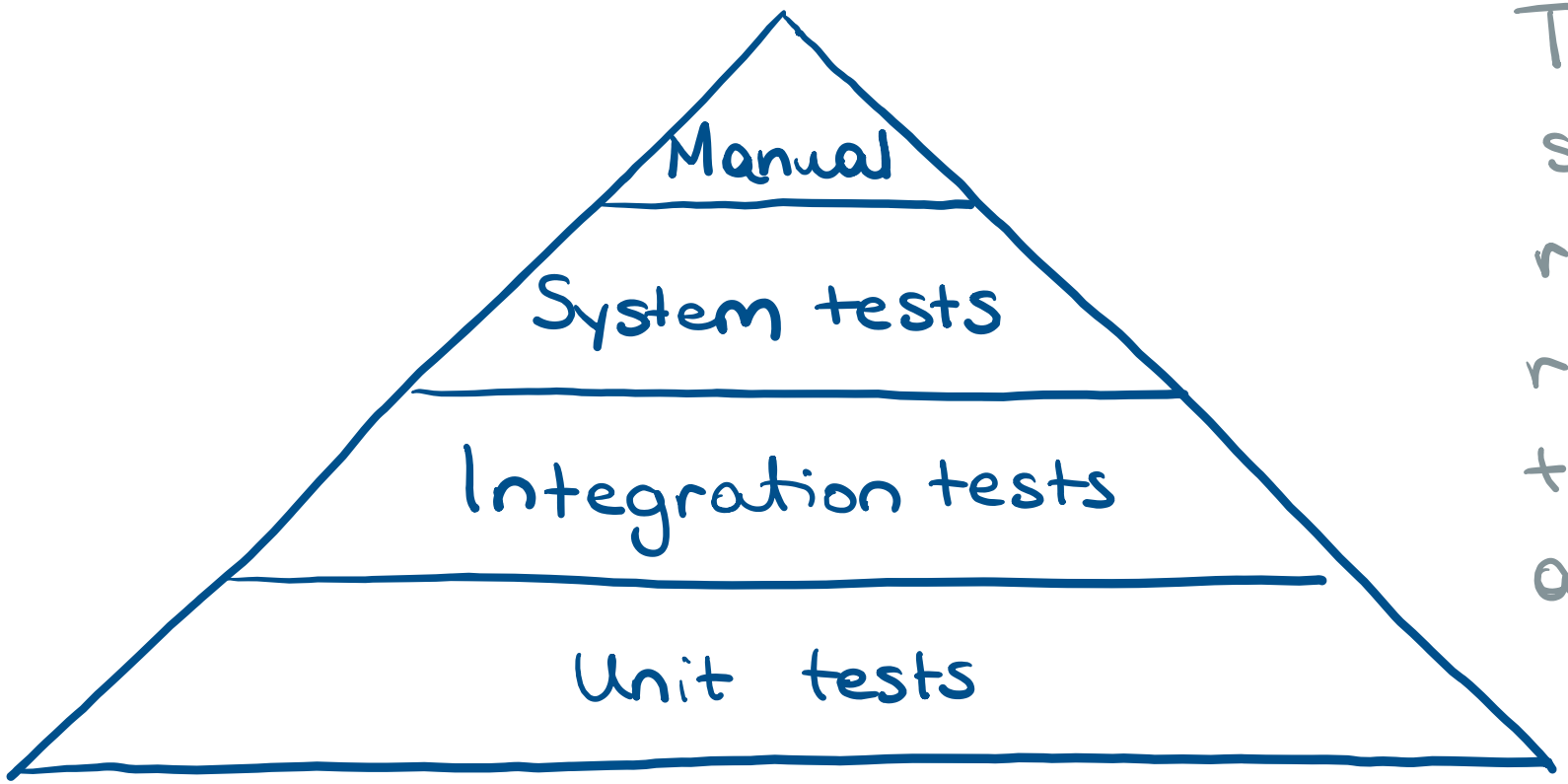
Manual testing is not automated and refers to manually exploring the software system

Manual tests typically focus on validation

Unit, integration, system testing: Pros and Cons



When to use each test level



The size of the slice in the pyramid represents the relative number of tests to carry out at each test level

Why favor unit tests?

- Because of their advantages
- Unit testing fits very well with the way developers work
  - To implement a new feature, they write separate units that will eventually work together to deliver larger functionality
  - While developing each unit, it is easy to ensure that it works as expected

Implementation

- Effective and systematic testing
- Developing for testability
- Test-driven development

Testability: Definition

Testability is how easy it is to write automated tests for the system, class, or method under test

If code is hard to test, it will likely remain untested

What is the right time to think about testability? All the time, and especially during implementation!

Good, testable code costs more than bad code, but it ensures quality

Separating infrastructure from domain code

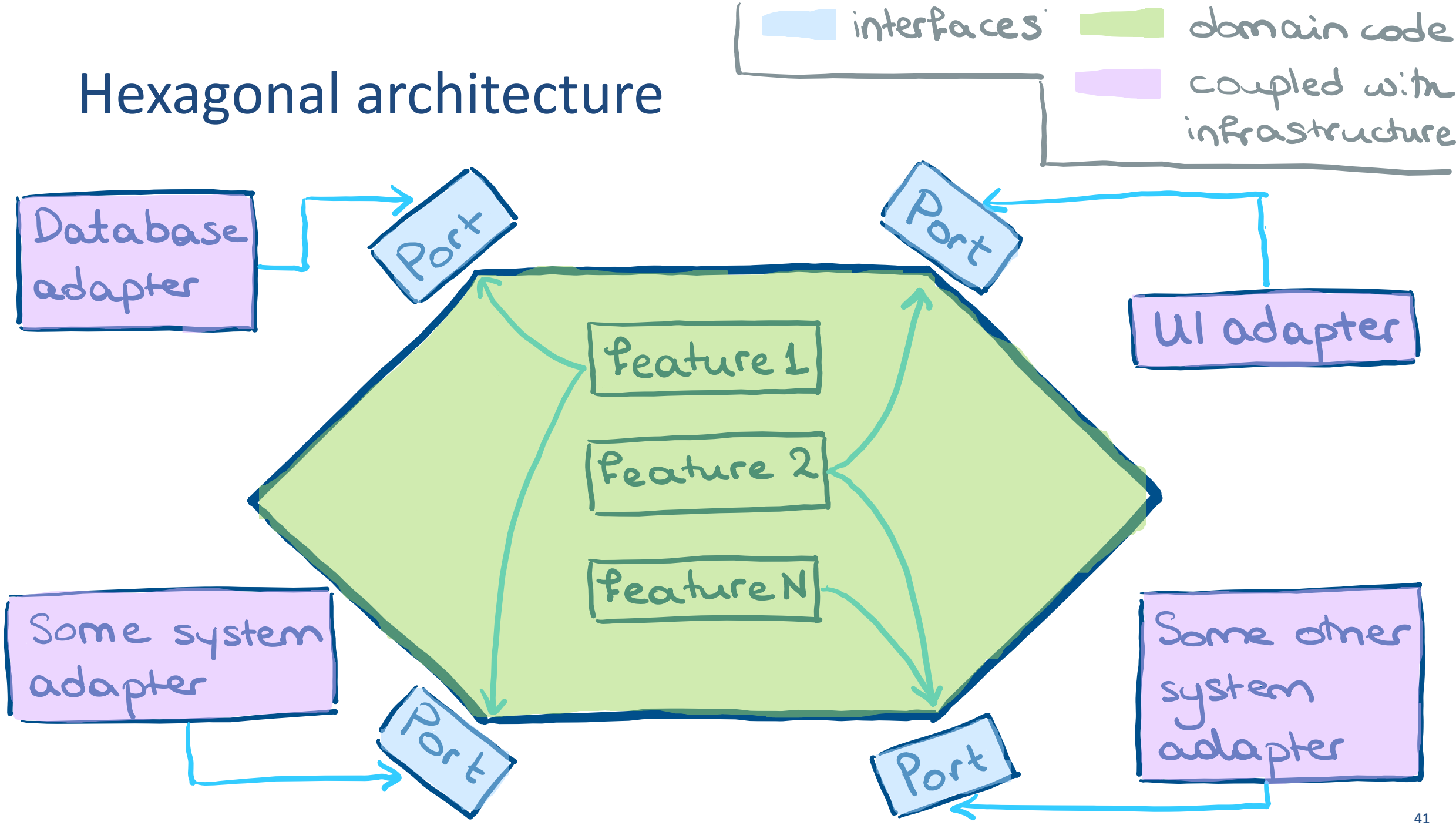
- The domain is where the core of the system lies
- The infrastructure is all the code that handles external dependencies
- When domain and infrastructure code are mixed, the system is harder to test
  - Simpler code is easier to write and test
  - There are fewer possibilities and corner cases

Hexagonal architecture: Definition

To enforce clear separation of responsibilities, we use hexagonal architecture (or the ports and adapters pattern)

- The domain depends on ports, not directly on the infrastructure
  - Ports are interfaces that define what the infrastructure can do and enable the application to get information from or send information to something else
- Adapters are very close to the infrastructure
  - Adapters are the implementations of the ports that talk to the infrastructure; they know how the infrastructure works and how to communicate with it

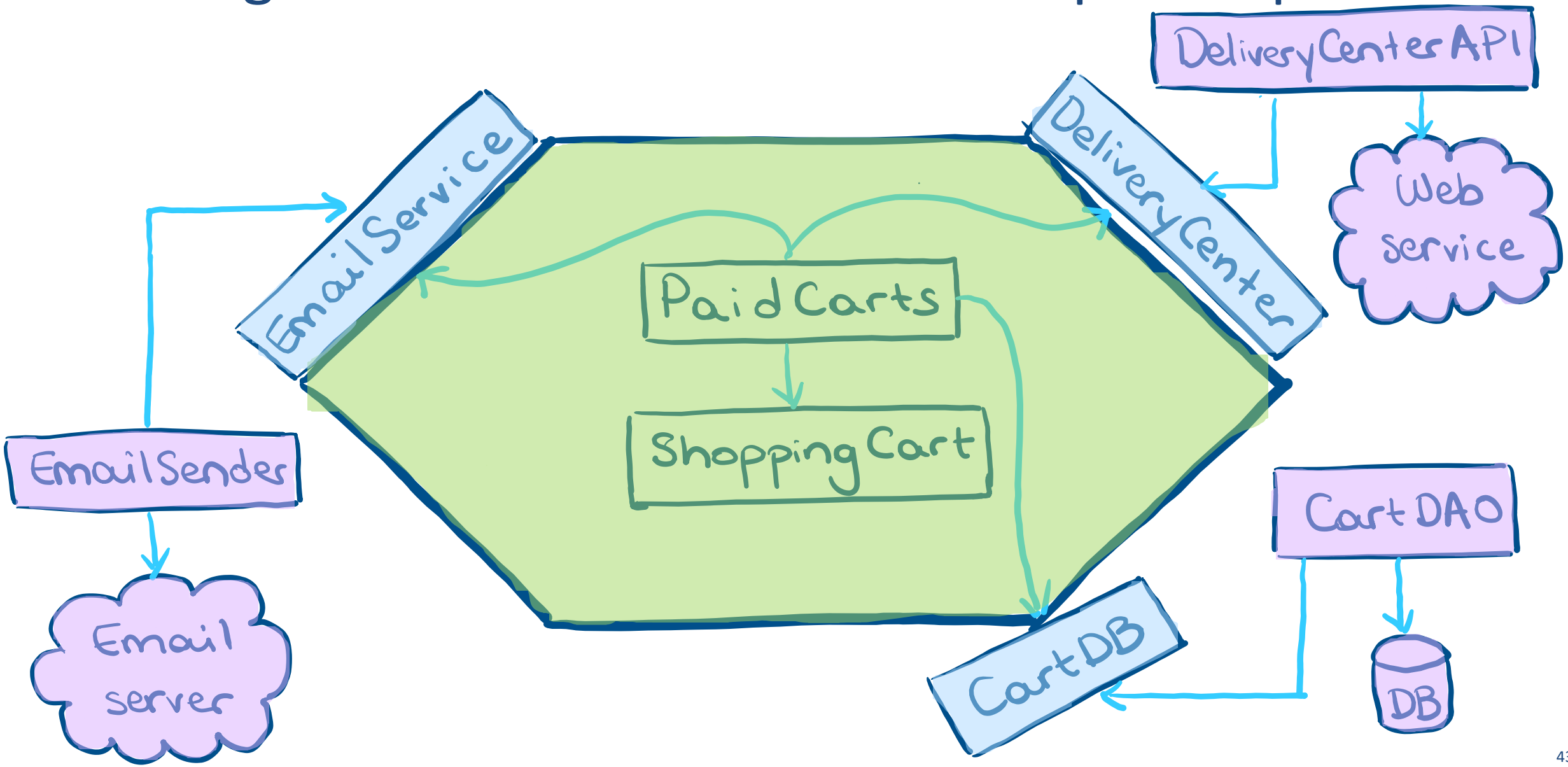
Hexagonal architecture



Hexagonal architecture: Online shop example

- For all the shopping carts that were paid today, the system should:
  - Notify the delivery center to send the goods to the customer
  - Set the status of the shopping cart as ready for delivery and persist its new state in the database
  - Send an email to the customer about when the delivery will happen

Hexagonal architecture: Online shop example



Why does this pattern improve testability?

- We can easily control what to test
- If the domain classes depend only on ports, we can exercise their behavior by implementing fake ports
  - E.g., method `deliver` of `DeliveryCenter` could return the current `LocalDate`
  - We will check if `DeliverySenderAPI` does its job properly in its own test suite
  - We will use integration testing to check communication with external dependencies

Dependency injection: Definition

Dependency injection is an implementation strategy that allows to easily separate domain from infrastructure code

Code that instantiates its dependencies hinders our ability to control the internals of the class and write unit tests

Dependency injection is as easy as receiving the dependencies via the constructor or via setters

Dependency injection: Pros

- It enables to fake the dependencies during testing
- It makes all dependencies more explicit
- It offers separation of concerns: classes do not need to know how to build the dependencies
- The class becomes more extensible: any new dependency can just be passed via the constructor

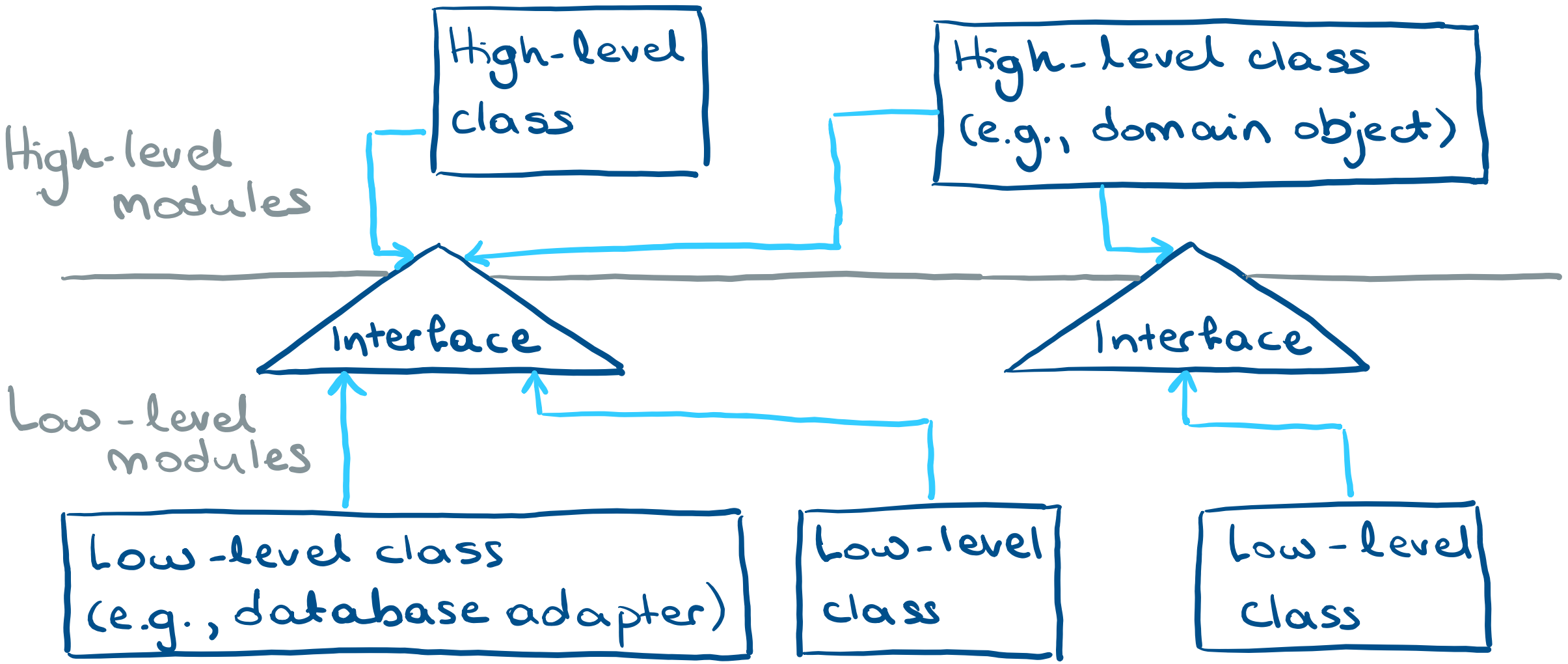
Dependency inversion principle

- High-level modules (such as domain code) should not depend on low-level modules, but on abstractions (such as interfaces)

- Abstractions should not depend on details, but details should depend on abstractions

Same as hexagonal architecture

Dependency inversion principle



Implementation

- Effective and systematic testing
- Developing for testability
- Test-driven development

Test-driven development (TDD)

- Write a test for the next small feature we want to implement
  - The test of course fails because the feature is not yet implemented
- Implement the feature
  - The test passes
- Refactor the code we wrote

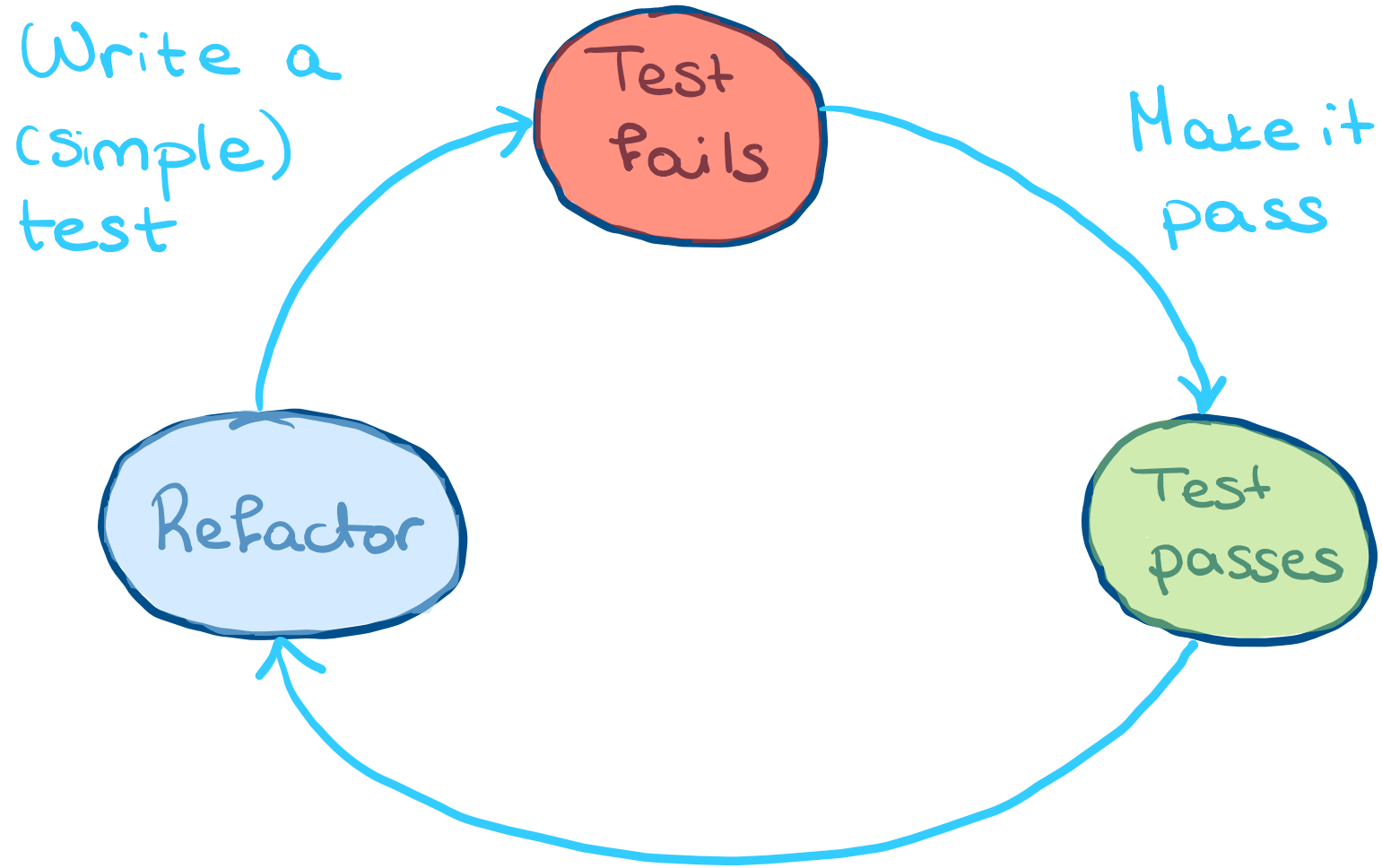
TDD: Example

Convert Roman numerals to integers

- Roman numerals represent numbers with 7 symbols
  - I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000
- All numbers are represented using two rules, starting from right to left
  - Digits of higher or equal value on the left are added
  - Digits of lower value on the left are subtracted
  - E.g., XV = 10 + 5 = 15 and XXIV = 10 + 10 – 1 + 5 = 24

TDD differently

red-green-refactor cycle



TDD: Pros

- Looking at requirements first
  - The tests are basically executable requirements
- Control over the pace of writing production code
  - If we are confident about the problem, the tests can be more complicated
- Quick feedback about problems
  - It is easier to identify new problems as they arise because we have only written a small amount of code since the last time everything was under control

TDD: Pros

- Testable code
  - We think from the beginning about how to (easily) test production code
- Feedback about design
  - The tests are often the first client of the code
  - E.g., a test method instantiates the class under test, invokes a method passing all its parameters, and asserts that the method produces the expected results
  - If the client is hard to write, perhaps there is a better way to design the code

Suggested reading

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

- Sections 1.2.1, 1.2.5, 1.2.6, 1.3, 1.4
- Introduction of Chapter 7
- Sections 7.1, 7.2
- Sections 8.1, 8.2