Summary of the ALCOR Academy Training Programme

Mike Mugglin

Content

Motivation

- I had a great time in the last 2.5 month and I wanted to recap the key points of this training programme

- I just wanted us to remember the concepts/guidelines/methods/… we learned

Walking

TDD

Classic TDD

Main characteristics

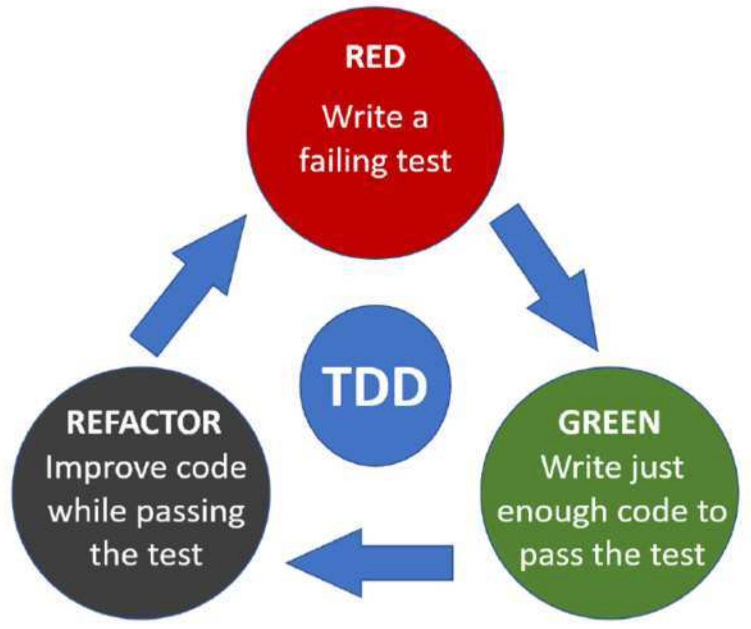
- ➔ Design happens during refactoring
- ➔ Usually tests are state-based
- ➔ When refactoring the unit under test can grow to multiple classes
- ➔ Mocks are rarely used, usually only for external systems isolation
- ➔ No upfront design assumptions. Design emerges completely from code, hence it solves over-engineering problems.
- ➔ Easy to understand due to state-based tests and no design upfront.
- ➔ Often used with the 4 rules of simple design
- ➔ Good for exploration when only input/output couples are known (black box)
- ➔ Great when we can't rely on a domain expert or domain language (algorithms, data transformation, etc.)

Main problems

- ➔ Exposing state for test purposes only
- ➔ Refactoring step is often skipped by inexperienced practitioners and left as the final big refactoring step
- ➔ The public interface of the units under test are created accordingly to the criteria "I think I will need this public methods", which doesn't always fit well with the rest of the system
- ➔ Can be slow and wasteful when we know that a class has too much responsibility and other classes could be extracted early on

TDD

The three laws of TDD / baby steps


RED
Write a failing test
TDD
REFACTOR
Improve code while passing the test
GREEN
Write just enough code to pass the test

1) You are not allowed to write any production code unless it is for making a failing unit test pass
2) You are not allowed to write any more of a unit test than is sufficient to fail. (compilation failure is a failure)
3) You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Refactoring -> use the *Rule of Three*:
Extract duplication only when you see it for the third time

Lesson 1 TRAINING PROGRAMME ALCOR academy

TDD

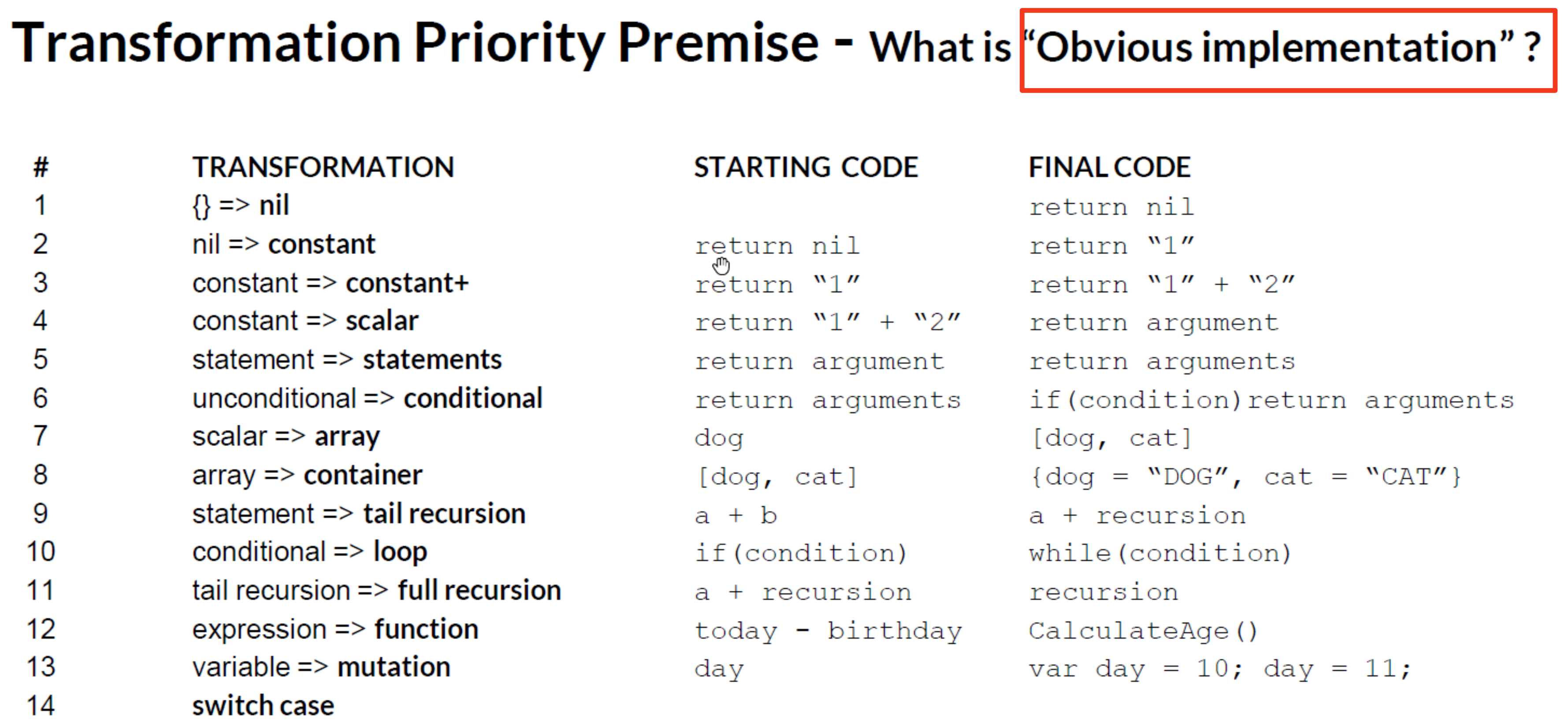
Baby steps - the three ways forward

1) **Fake implementation**
   When you hard code exactly the value you need to pass the test

1) **Obvious implementation**
   When you are sure of the code you need to write. This is what you will be using more often to move forward quickly.

1) **Triangulation with the next test**
   When you want to generalize a behaviour but are not sure how to do it. Starting with fake implementation and then adding more tests will force the code to be more and more generic. Complete one dimension first and then move on the next one with another test case.

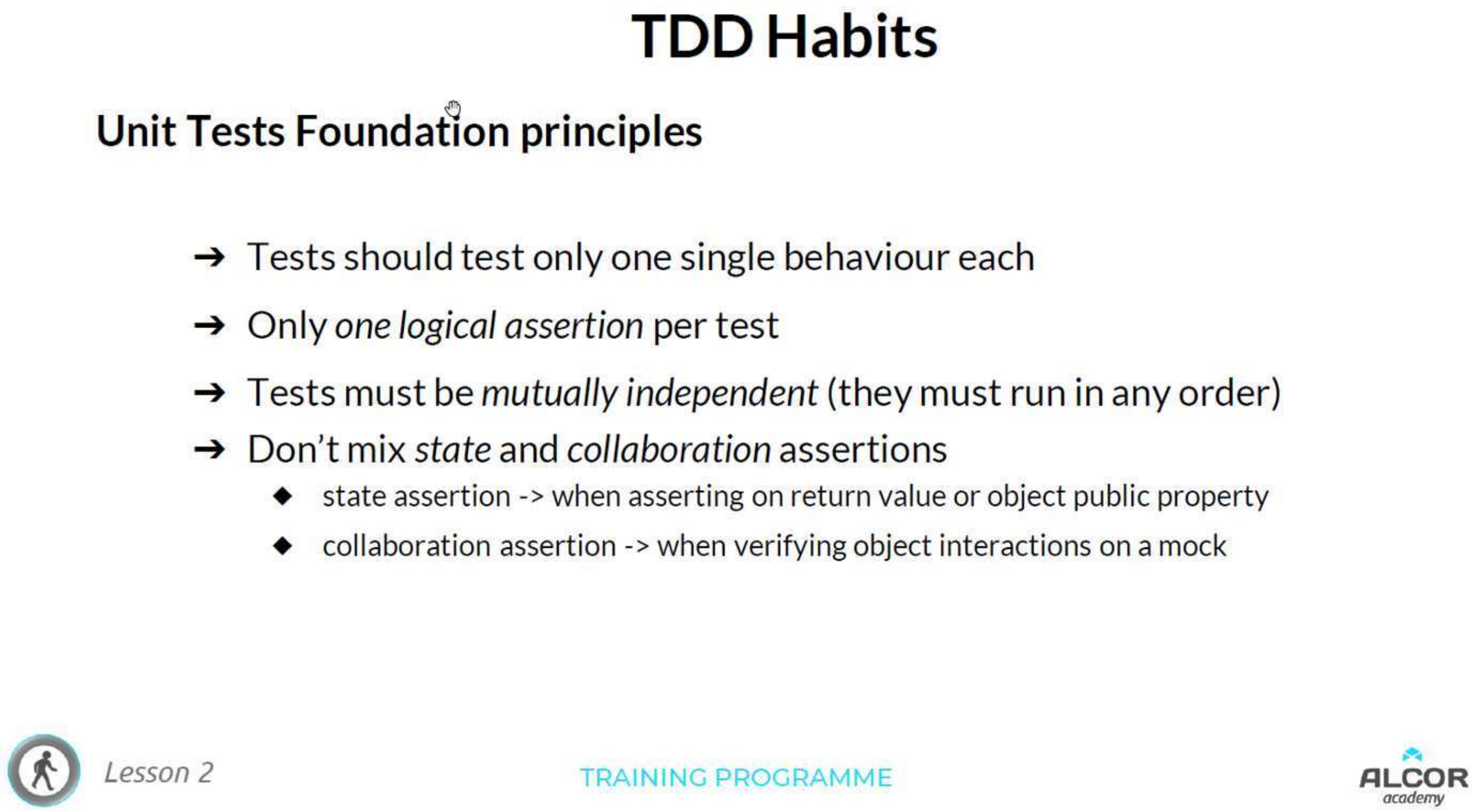
Transformation Priority Premise



Transformation Priority Premise - What is "Obvious implementation" ?

#	TRANSFORMATION	STARTING CODE	FINAL CODE
1	{} => **nil**		`return nil`
2	nil => **constant**	`return nil`	`return "1"`
3	constant => **constant+**	`return "1"`	`return "1" + "2"`
4	constant => **scalar**	`return "1" + "2"`	`return argument`
5	statement => **statements**	`return argument`	`return arguments`
6	unconditional => **conditional**	`return arguments`	`if(condition)return arguments`
7	scalar => **array**	`dog`	`[dog, cat]`
8	array => **container**	`[dog, cat]`	`{dog = "DOG", cat = "CAT"}`
9	statement => **tail recursion**	`a + b`	`a + recursion`
10	conditional => **loop**	`if(condition)`	`while(condition)`
11	tail recursion => **full recursion**	`a + recursion`	`recursion`
12	expression => **function**	`today - birthday`	`CalculateAge()`
13	variable => **mutation**	`day`	`var day = 10; day = 11;`
14	**switch case**		

TDD Habits



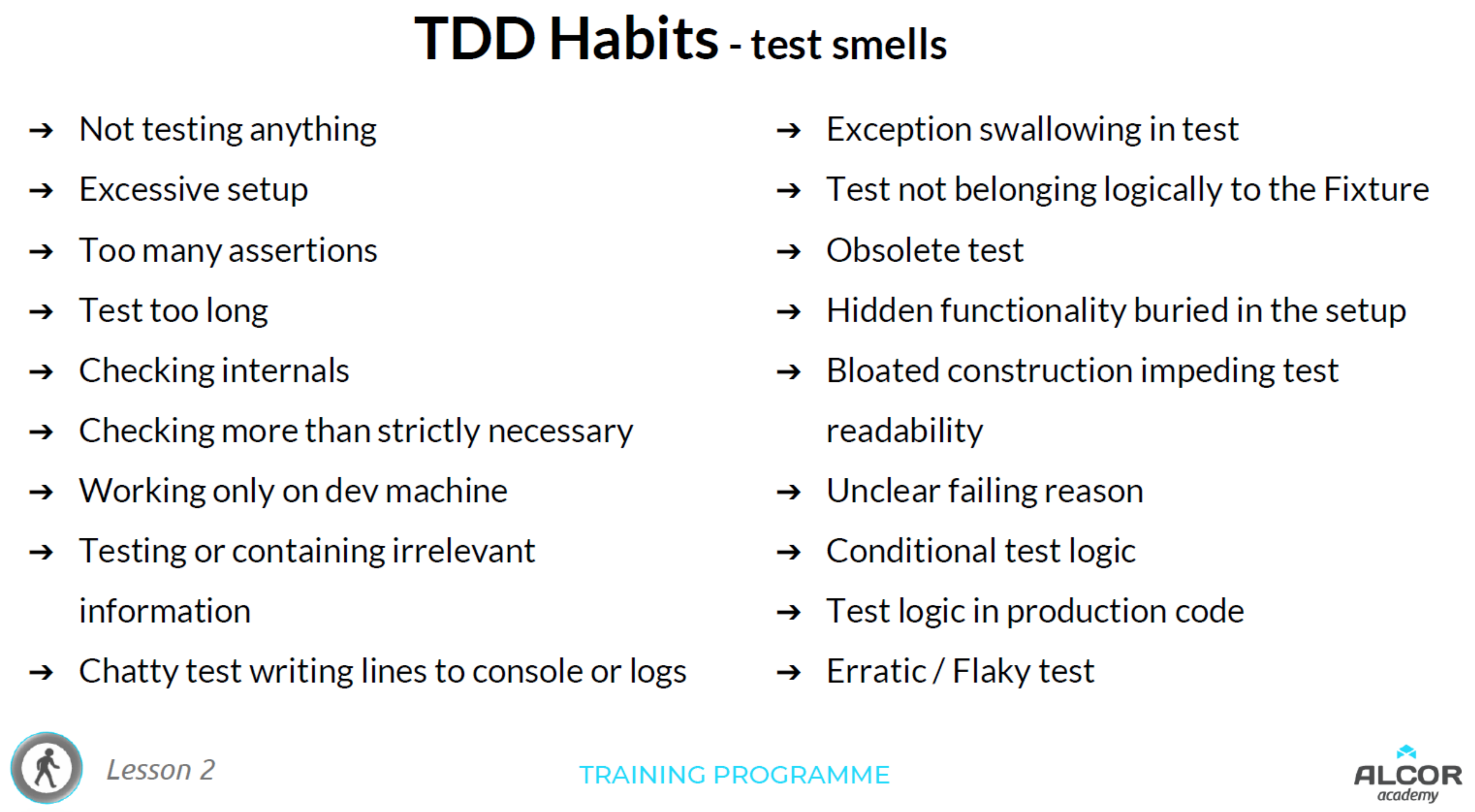
TDD Habits

Unit Tests Foundation principles

- ➔ Tests should test only one single behaviour each
- ➔ Only *one logical assertion* per test
- ➔ Tests must be *mutually independent* (they must run in any order)
- ➔ Don't mix *state* and *collaboration* assertions
  - ◆ state assertion -> when asserting on return value or object public property
  - ◆ collaboration assertion -> when verifying object interactions on a mock

Lesson 2 TRAINING PROGRAMME ALCOR academy

TDD Habits

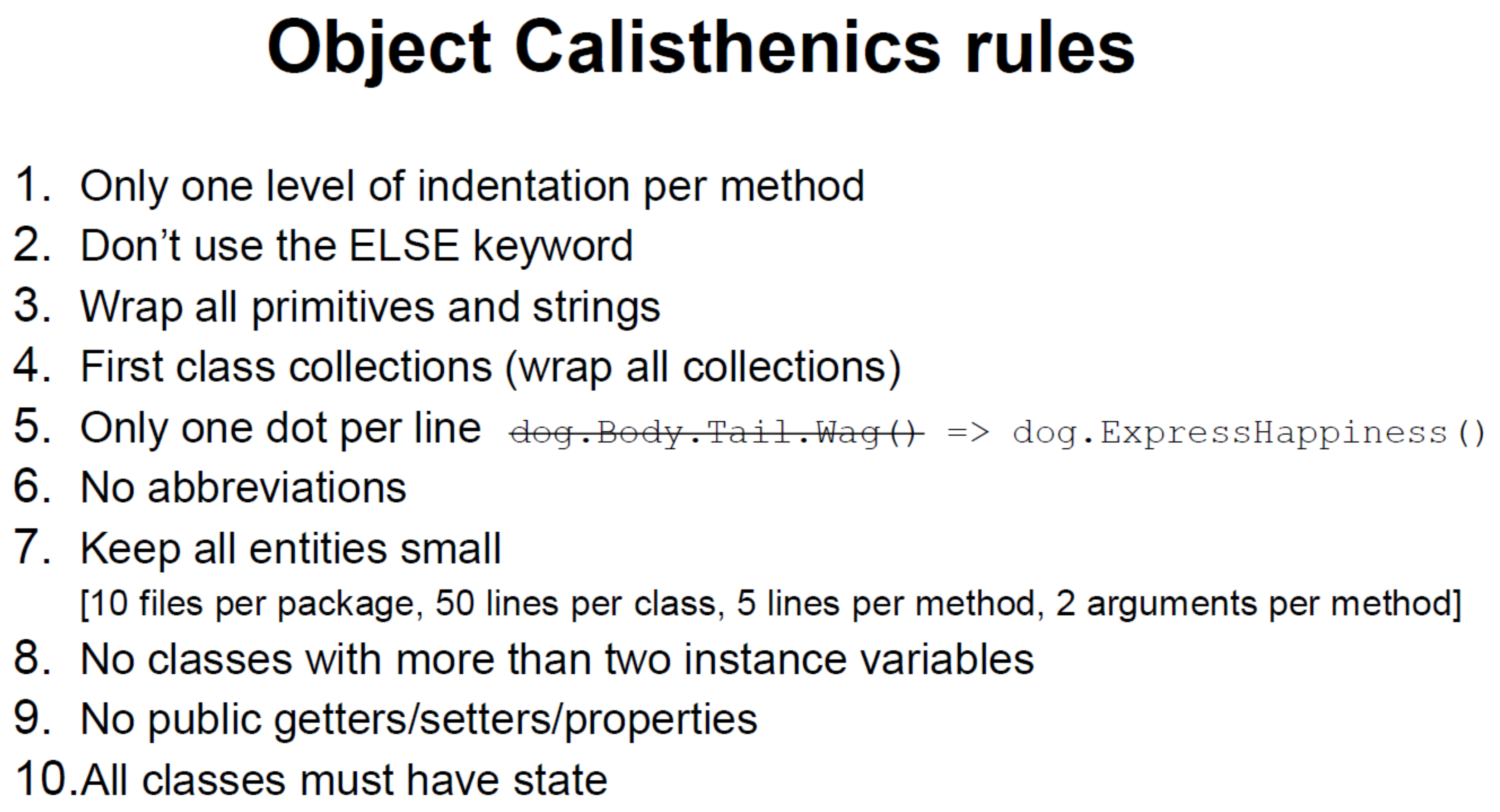


TDD Habits - test smells

- ➔ Not testing anything
- ➔ Excessive setup
- ➔ Too many assertions
- ➔ Test too long
- ➔ Checking internals
- ➔ Checking more than strictly necessary
- ➔ Working only on dev machine
- ➔ Testing or containing irrelevant information
- ➔ Chatty test writing lines to console or logs
- ➔ Exception swallowing in test
- ➔ Test not belonging logically to the Fixture
- ➔ Obsolete test
- ➔ Hidden functionality buried in the setup
- ➔ Bloated construction impeding test readability
- ➔ Unclear failing reason
- ➔ Conditional test logic
- ➔ Test logic in production code
- ➔ Erratic / Flaky test

Lesson 2 TRAINING PROGRAMME ALCOR academy

Object Calisthenics

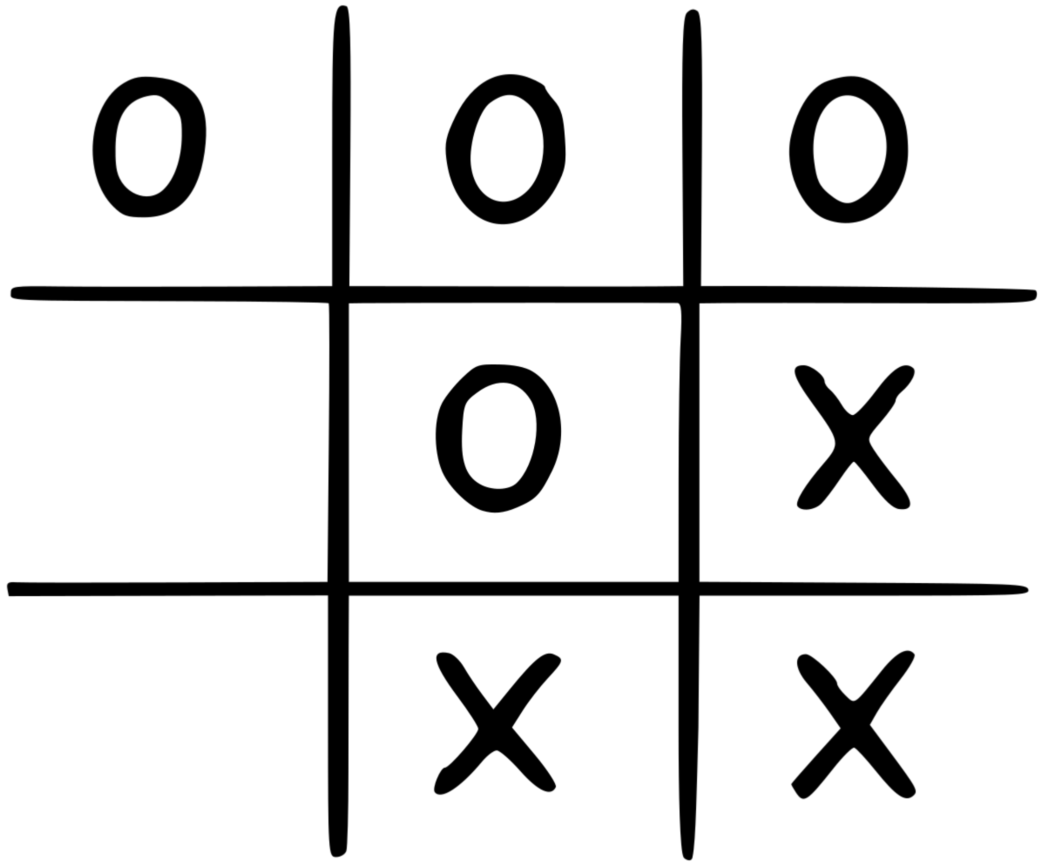


Object Calisthenics rules

1. Only one level of indentation per method
2. Don’t use the ELSE keyword
3. Wrap all primitives and strings
4. First class collections (wrap all collections)
5. Only one dot per line `~~dog.Body.Tail.Wag()~~ => dog.ExpressHappiness()`
6. No abbreviations
7. Keep all entities small
   [10 files per package, 50 lines per class, 5 lines per method, 2 arguments per method]
8. No classes with more than two instance variables
9. No public getters/setters/properties
10. All classes must have state

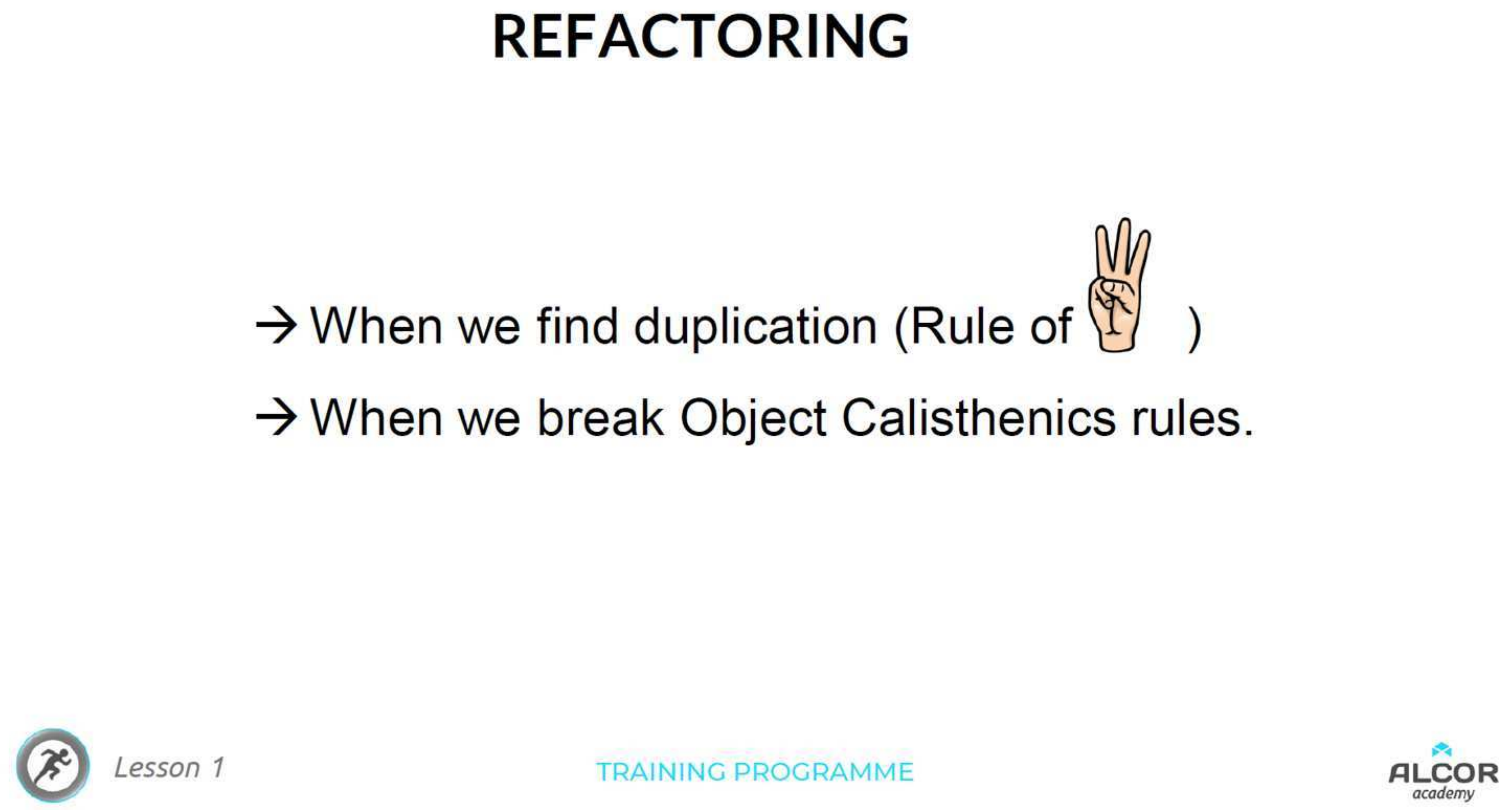
CSS Versicherung

First KATA in mob



Running

Refactoring



REFACTORING

→ When we find duplication (Rule of)

→ When we break Object Calisthenics rules.

Lesson 1

TRAINING PROGRAMME

ALCOR academy

Refactoring

Refactoring guidelines

STAY IN GREEN WHILE REFACTORING

There is no reason why we should break any tests during refactoring.
It may be that we have tests, but they are coupled with implementation.
In this case, we start refactoring by the tests, decoupling the test from the implementation.

Be strict about staying on green. We learned that in refactoring, it is more effective to let go of something as soon as tests break, *rather than stubbornly trying to fix things* to make tests pass.

Refactoring

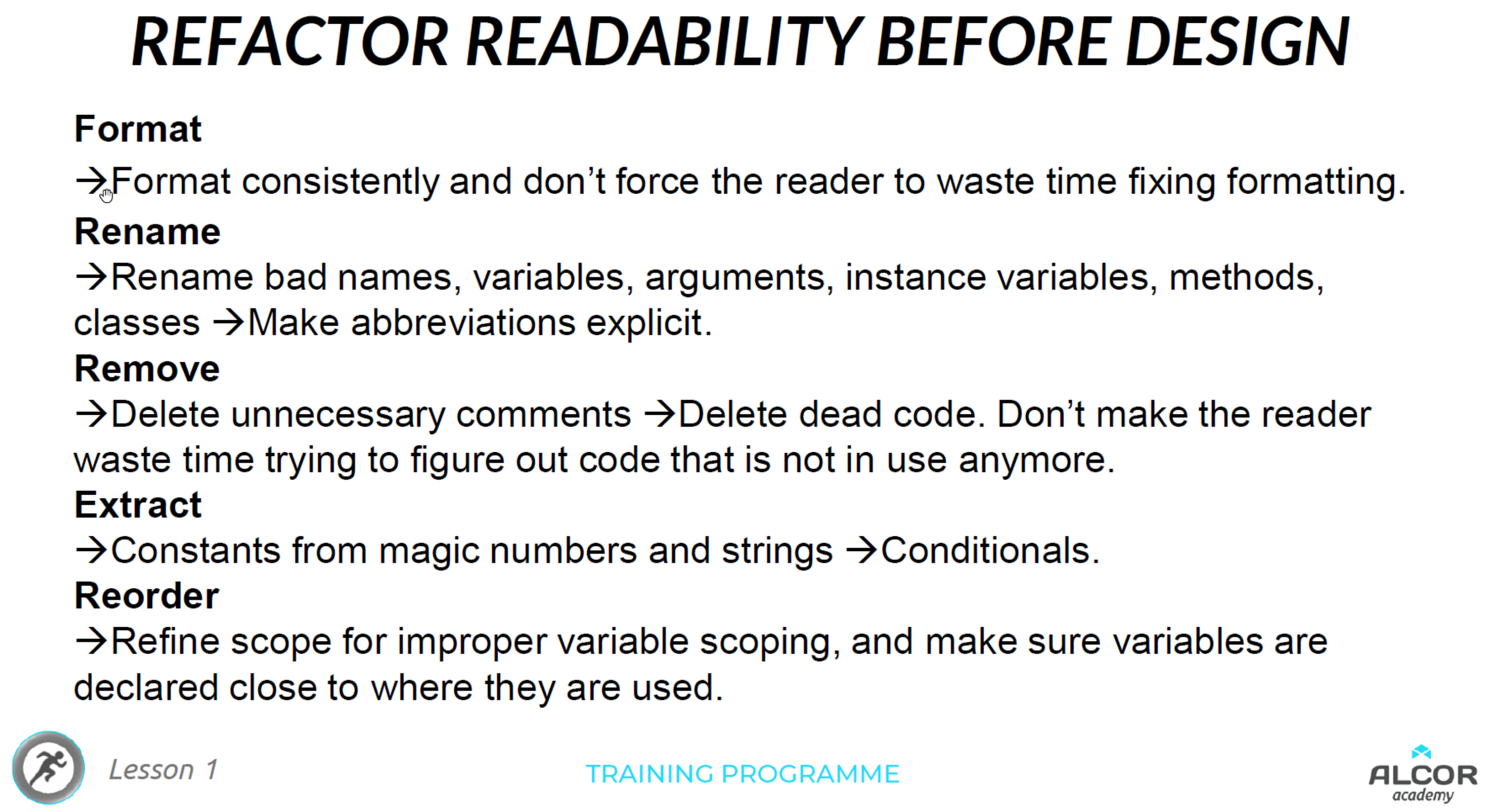
Refactoring guidelines

REFACTOR READABILITY BEFORE DESIGN

Small improvements in *code readability* can drastically improve *code understandability*.
Start with *better names* for variables, methods and classes.
The idea is to **express intent** rather than implementation details. We recommend Arlo Belshee's approach for naming.

https://www.digdeeproots.com/articles/naming-as-a-process/

Refactoring



REFACTOR READABILITY BEFORE DESIGN

Format

→Format consistently and don’t force the reader to waste time fixing formatting.

Rename

→Rename bad names, variables, arguments, instance variables, methods, classes →Make abbreviations explicit.

Remove

→Delete unnecessary comments →Delete dead code. Don’t make the reader waste time trying to figure out code that is not in use anymore.

Extract

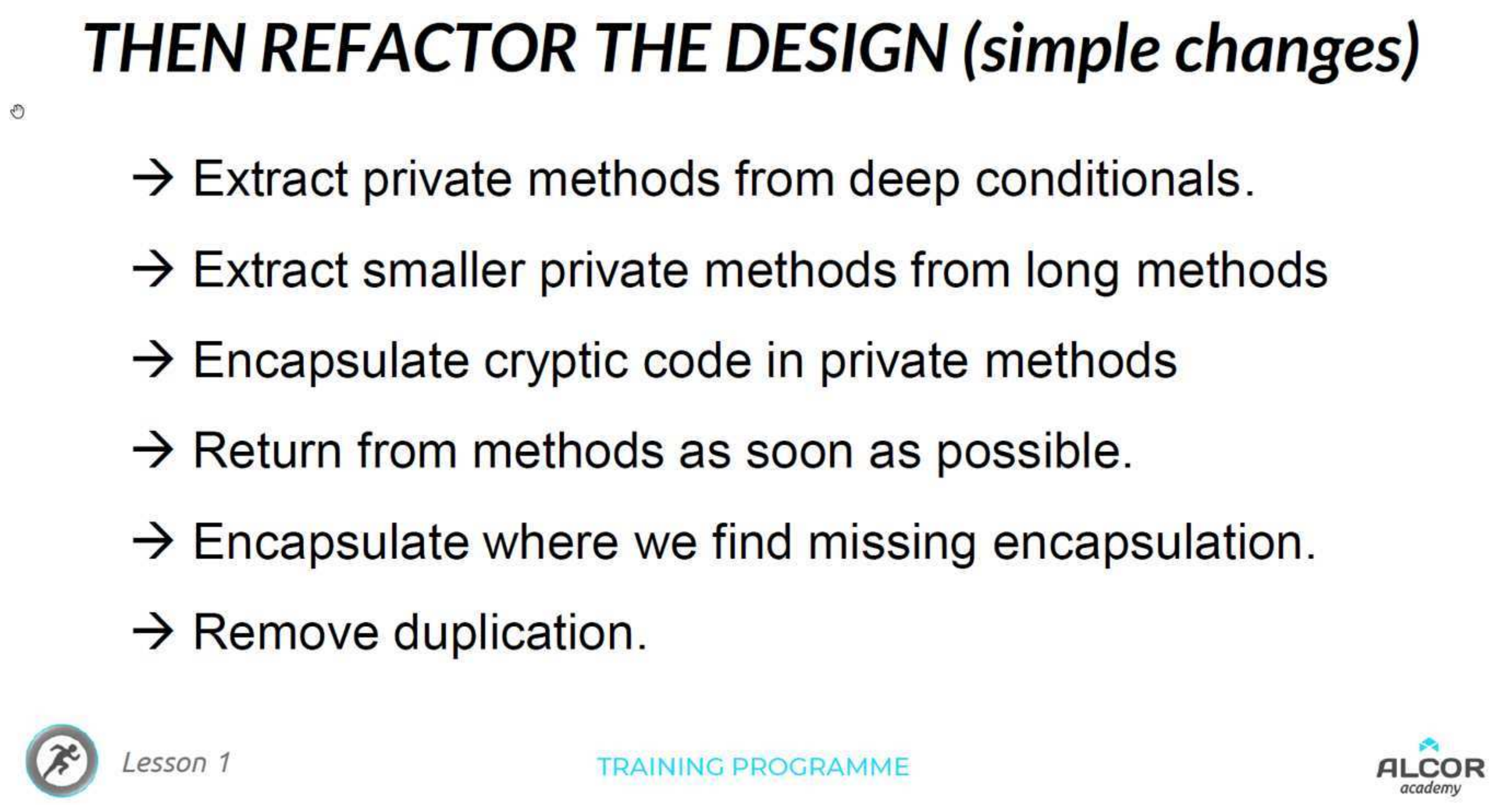
→Constants from magic numbers and strings →Conditionals.

Reorder

→Refine scope for improper variable scoping, and make sure variables are declared close to where they are used.

Lesson 1　TRAINING PROGRAMME　ALCOR academy

Refactoring



THEN REFACTOR THE DESIGN *(simple changes)*

→ Extract private methods from deep conditionals.

→ Extract smaller private methods from long methods

→ Encapsulate cryptic code in private methods

→ Return from methods as soon as possible.

→ Encapsulate where we find missing encapsulation.

→ Remove duplication.

Lesson 1

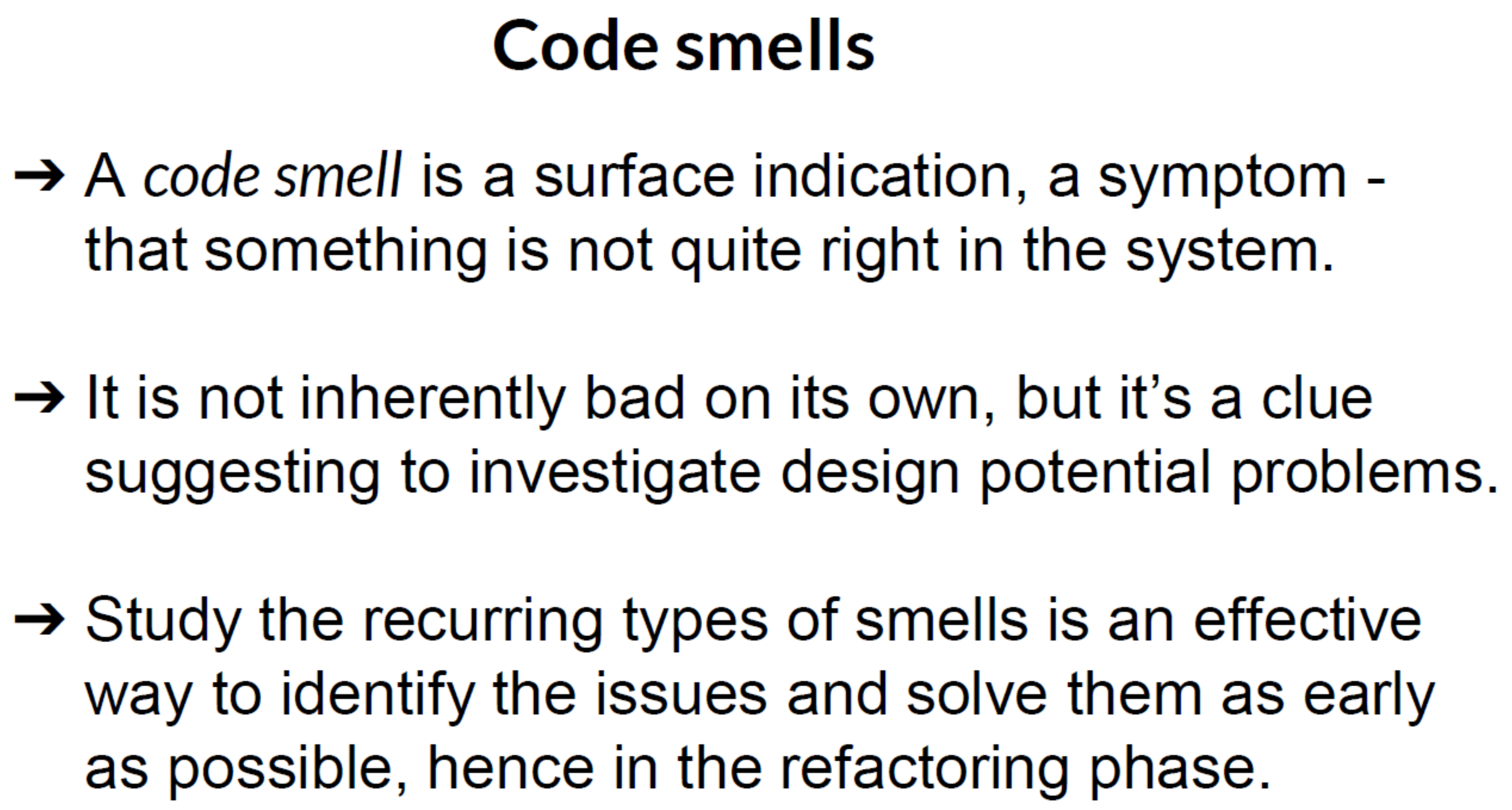
TRAINING PROGRAMME

ALCOR academy

Code smells

https://sourcemaking.com/refactoring/smells

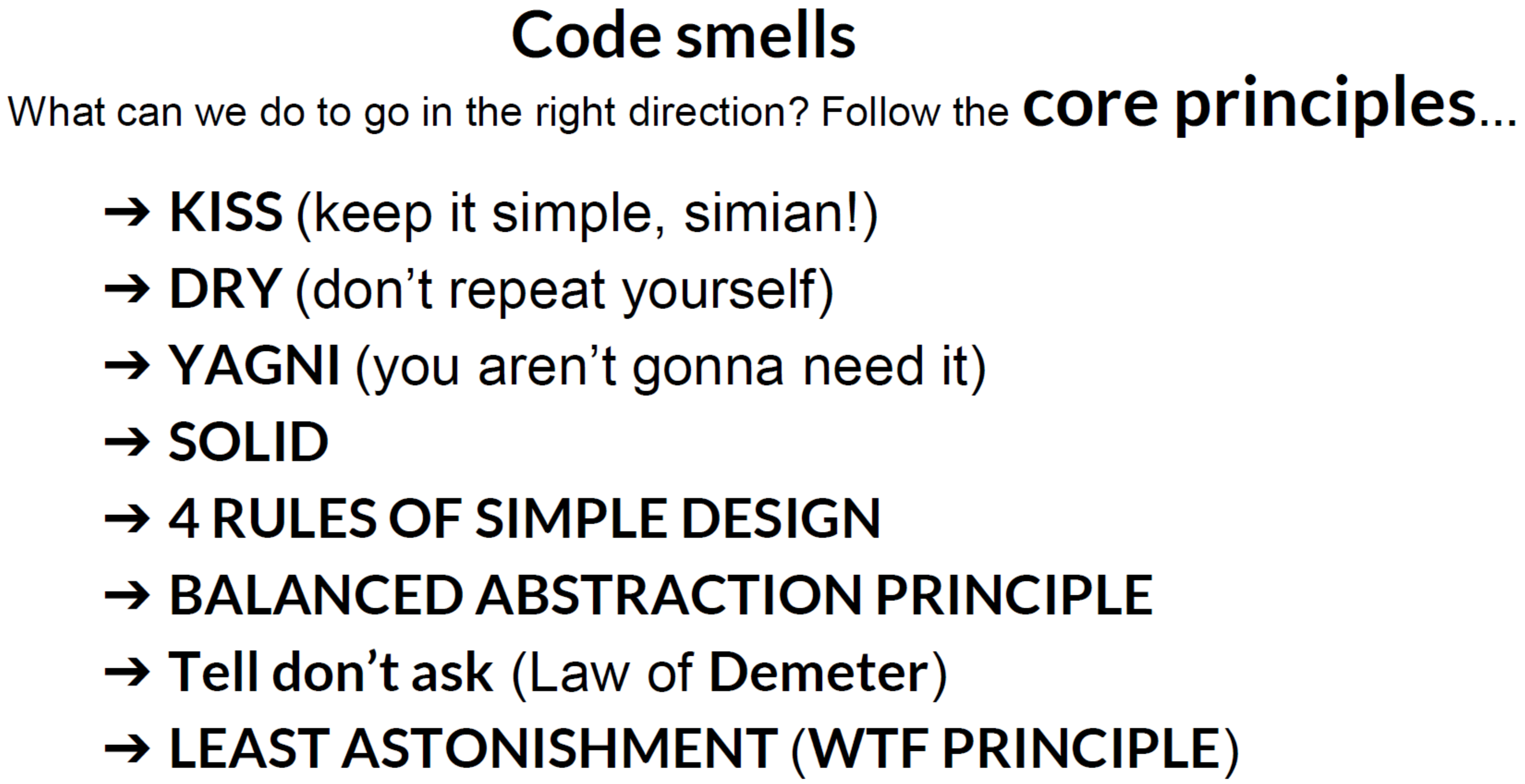
Code smells



Code smells

- ➔ A *code smell* is a surface indication, a symptom - that something is not quite right in the system.
- ➔ It is not inherently bad on its own, but it's a clue suggesting to investigate design potential problems.
- ➔ Study the recurring types of smells is an effective way to identify the issues and solve them as early as possible, hence in the refactoring phase.

Code smells

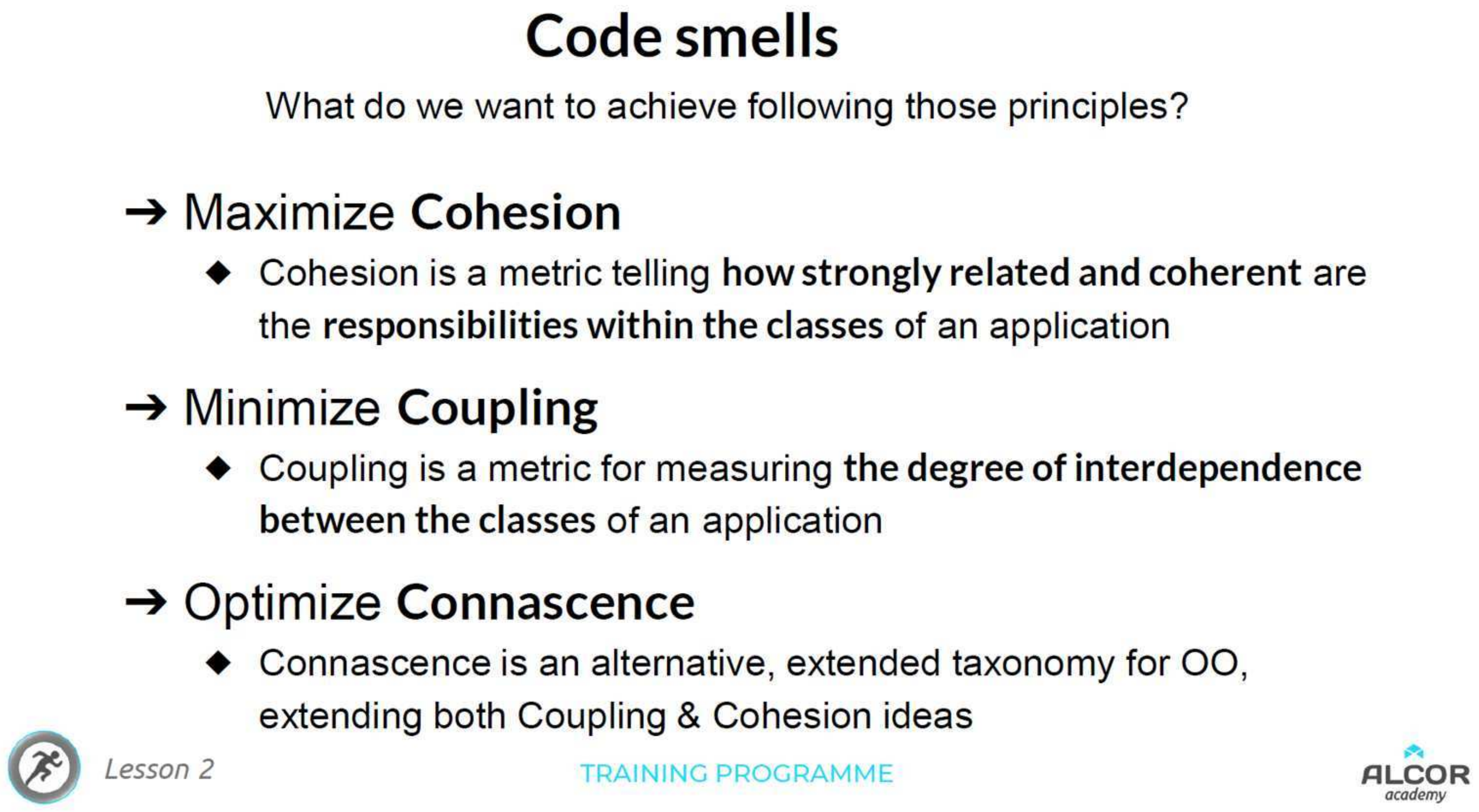


Code smells

What can we do to go in the right direction? Follow the **core principles...**

- ➔ **KISS** (keep it simple, simian!)
- ➔ **DRY** (don't repeat yourself)
- ➔ **YAGNI** (you aren't gonna need it)
- ➔ **SOLID**
- ➔ **4 RULES OF SIMPLE DESIGN**
- ➔ **BALANCED ABSTRACTION PRINCIPLE**
- ➔ **Tell don't ask** (Law of **Demeter**)
- ➔ **LEAST ASTONISHMENT (WTF PRINCIPLE)**

Code smells



Code smells

What do we want to achieve following those principles?

→ Maximize **Cohesion**

- Cohesion is a metric telling **how strongly related and coherent** are the **responsibilities within the classes** of an application

→ Minimize **Coupling**

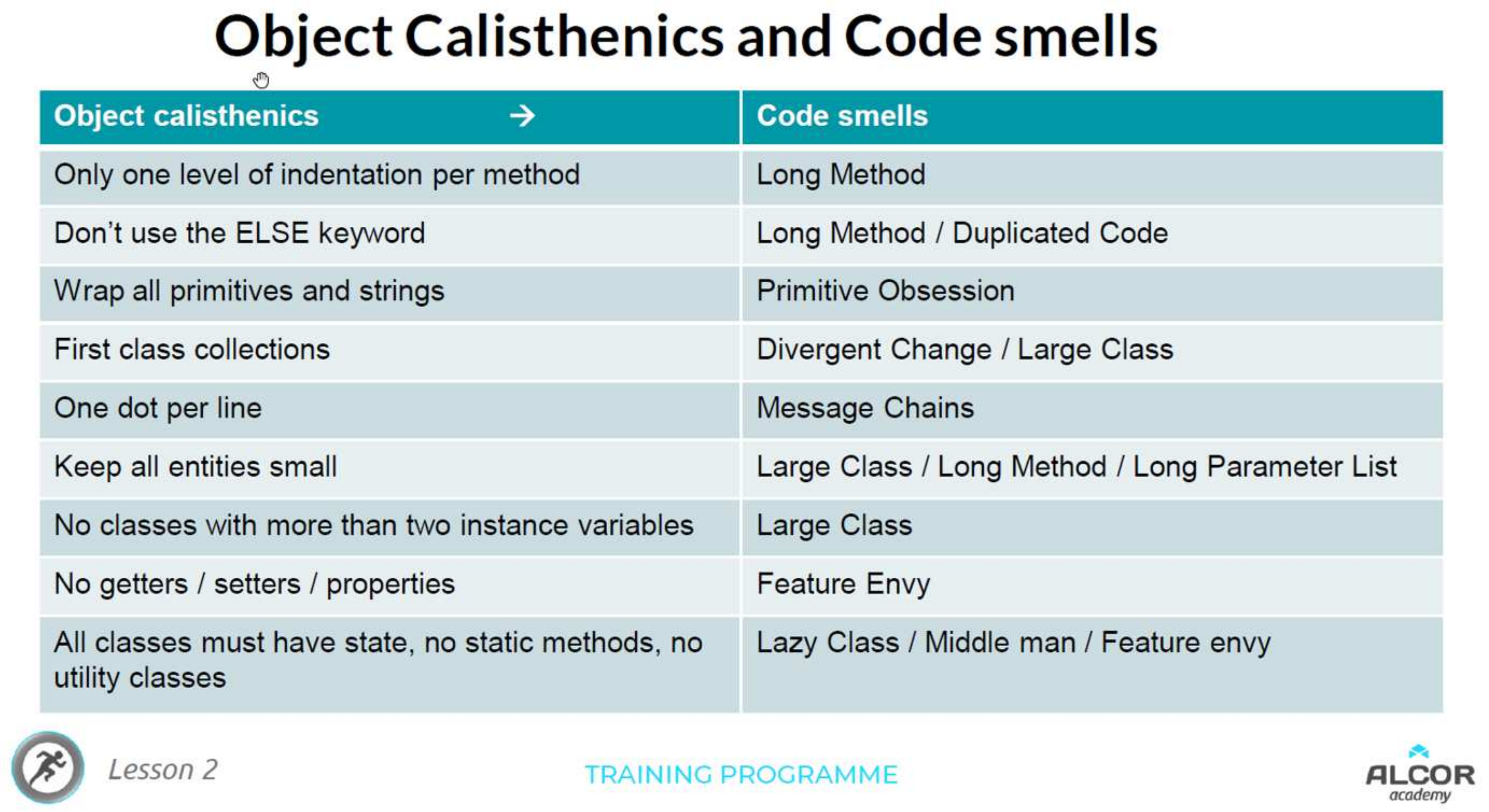
- Coupling is a metric for measuring **the degree of interdependence between the classes** of an application

→ Optimize **Connascence**

- Connascence is an alternative, extended taxonomy for OO, extending both Coupling & Cohesion ideas

Lesson 2 TRAINING PROGRAMME ALCOR academy

Object Calisthenics => Code smells



Object Calisthenics and Code smells

Object calisthenics →	Code smells
Only one level of indentation per method	Long Method
Don’t use the ELSE keyword	Long Method / Duplicated Code
Wrap all primitives and strings	Primitive Obsession
First class collections	Divergent Change / Large Class
One dot per line	Message Chains
Keep all entities small	Large Class / Long Method / Long Parameter List
No classes with more than two instance variables	Large Class
No getters / setters / properties	Feature Envy
All classes must have state, no static methods, no utility classes	Lazy Class / Middle man / Feature envy

Lesson 2

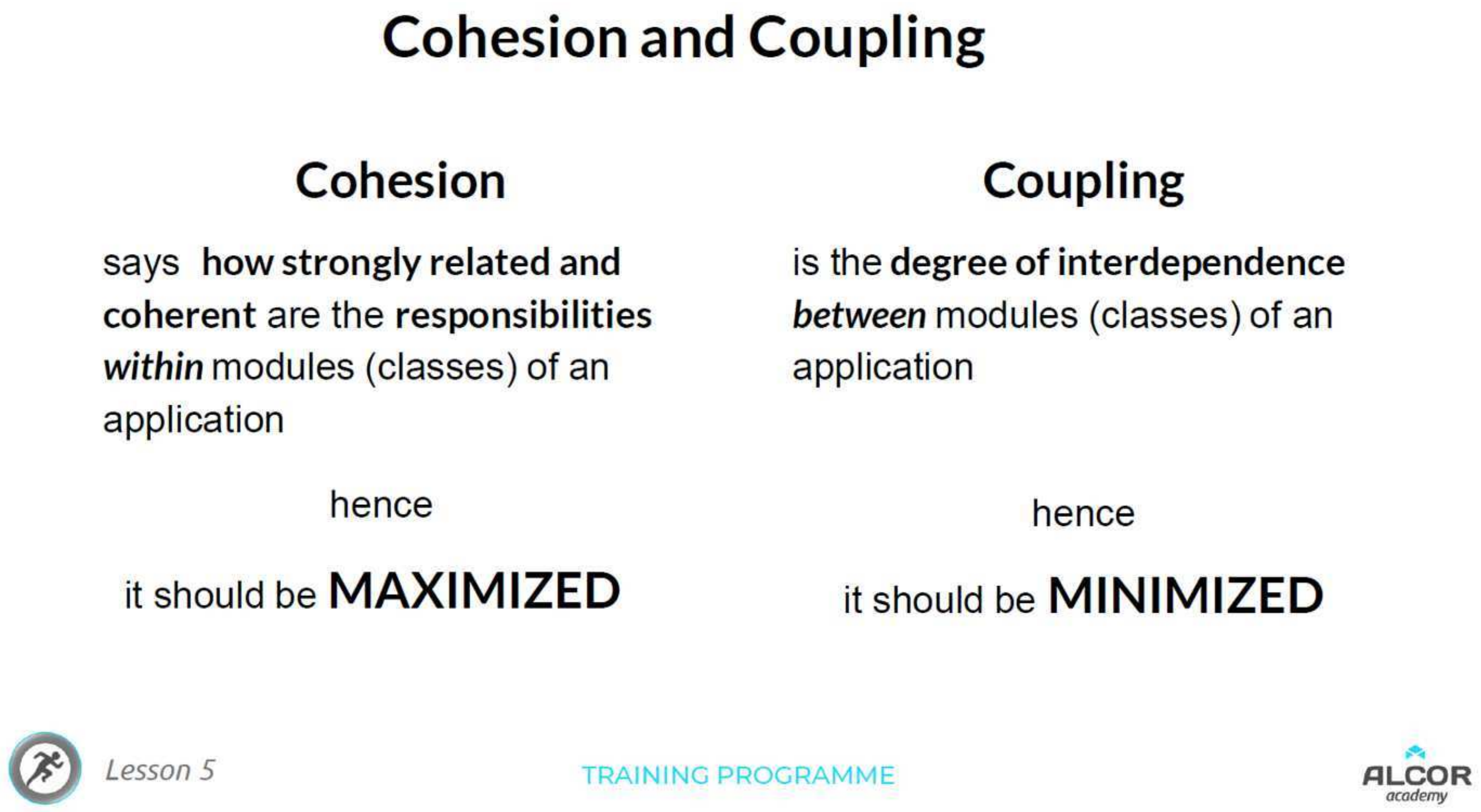
TRAINING PROGRAMME

ALCOR academy

SOLID++ principle

- **S**ingle Responsibility
- **O**pen/Closed (open for extension/closed for modification)
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion
- **+**:Balanced Abstraction
- +**+**: Least Astonishment (a.k.a. WTF)

Cohesion & Coupling



Cohesion and Coupling

Cohesion

says **how strongly related and coherent** are the **responsibilities** ***within*** modules (classes) of an application

hence

it should be **MAXIMIZED**

Coupling

is the **degree of interdependence** ***between*** modules (classes) of an application

hence

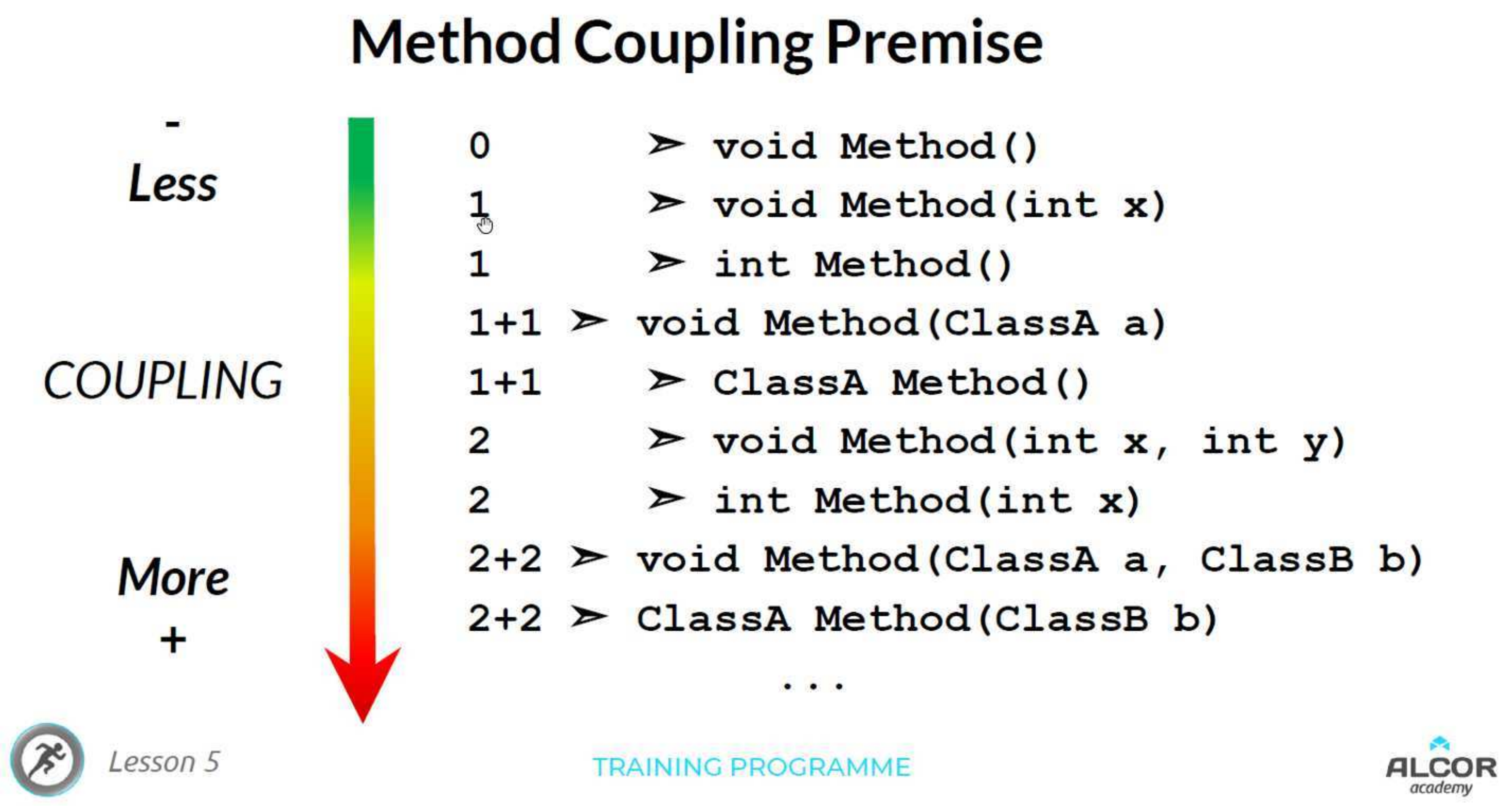
it should be **MINIMIZED**

Lesson 5

TRAINING PROGRAMME

ALCOR academy

Coupling



Method Coupling Premise

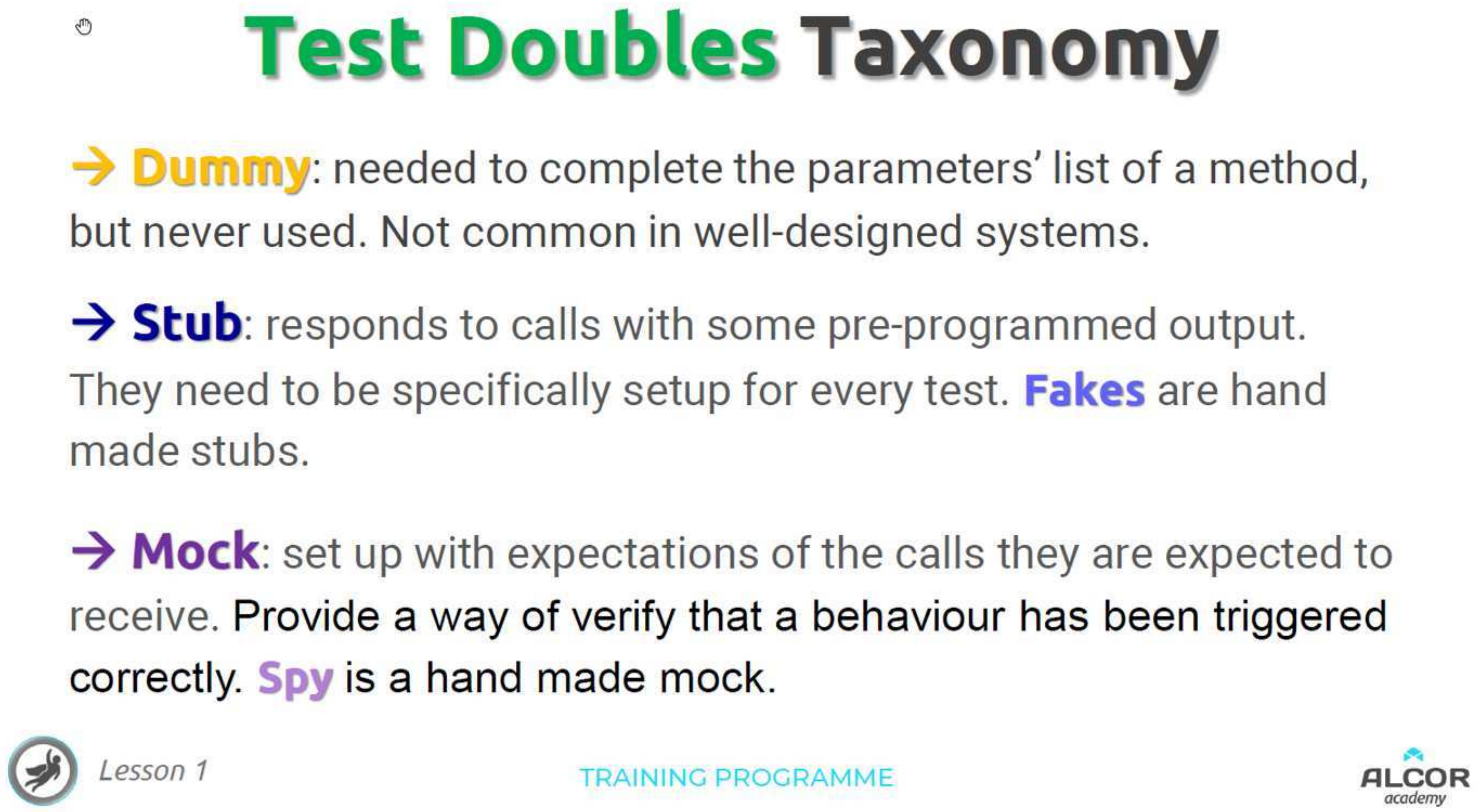
-
Less
COUPLING
More
+

```
0        ➢ void Method()
1        ➢ void Method(int x)
1        ➢ int Method()
1+1 ➢ void Method(ClassA a)
1+1      ➢ ClassA Method()
2        ➢ void Method(int x, int y)
2        ➢ int Method(int x)
2+2 ➢ void Method(ClassA a, ClassB b)
2+2 ➢ ClassA Method(ClassB b)
               ...
```

Lesson 5
TRAINING PROGRAMME
ALCOR academy

FLYING

Test doubles



Test Doubles Taxonomy

→ **Dummy**: needed to complete the parameters' list of a method, but never used. Not common in well-designed systems.

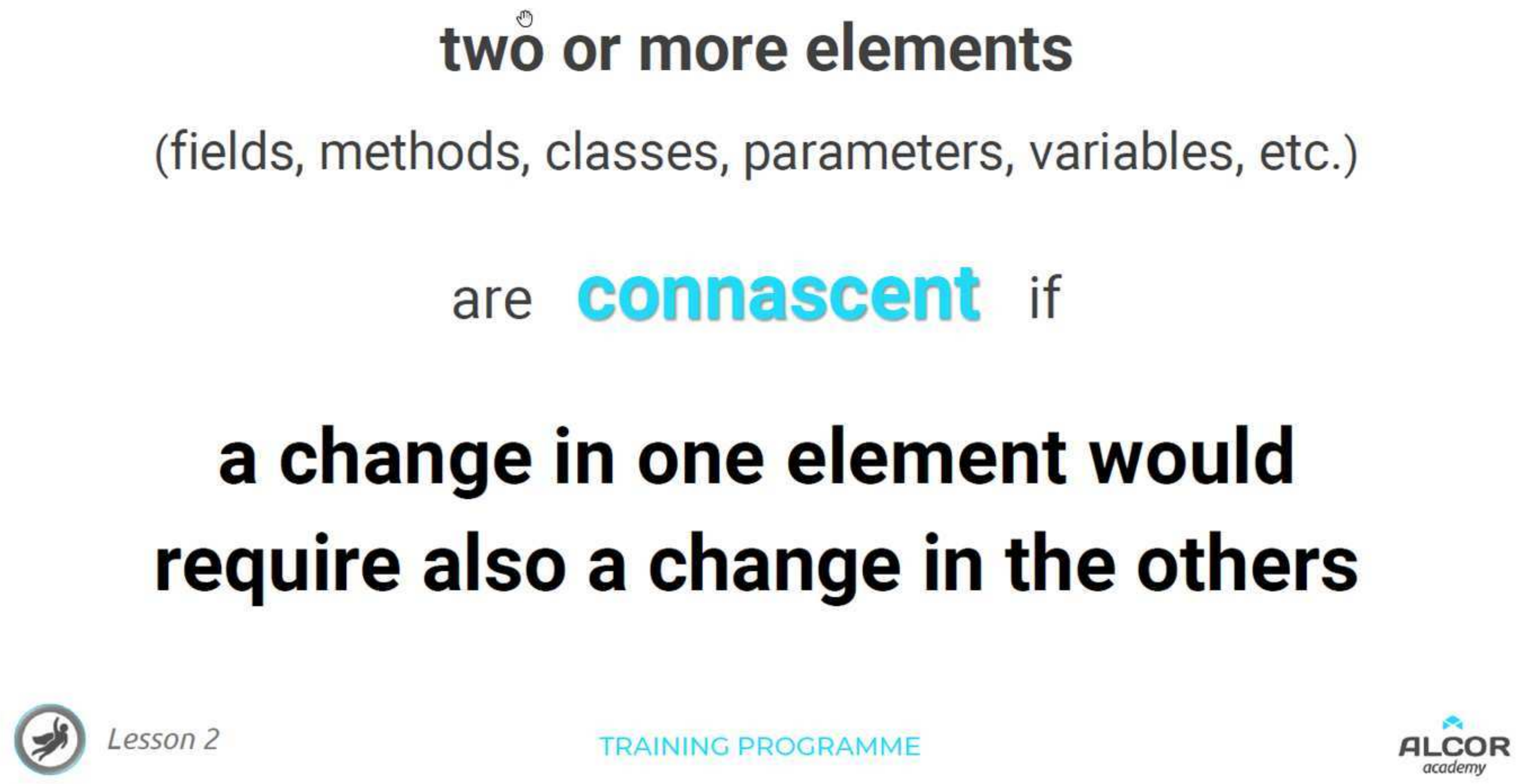
→ **Stub**: responds to calls with some pre-programmed output. They need to be specifically setup for every test. **Fakes** are hand made stubs.

→ **Mock**: set up with expectations of the calls they are expected to receive. Provide a way of verify that a behaviour has been triggered correctly. **Spy** is a hand made mock.

Lesson 1 TRAINING PROGRAMME ALCOR academy

Emelie was born

Connascence



two or more elements

(fields, methods, classes, parameters, variables, etc.)

are **connascent** if

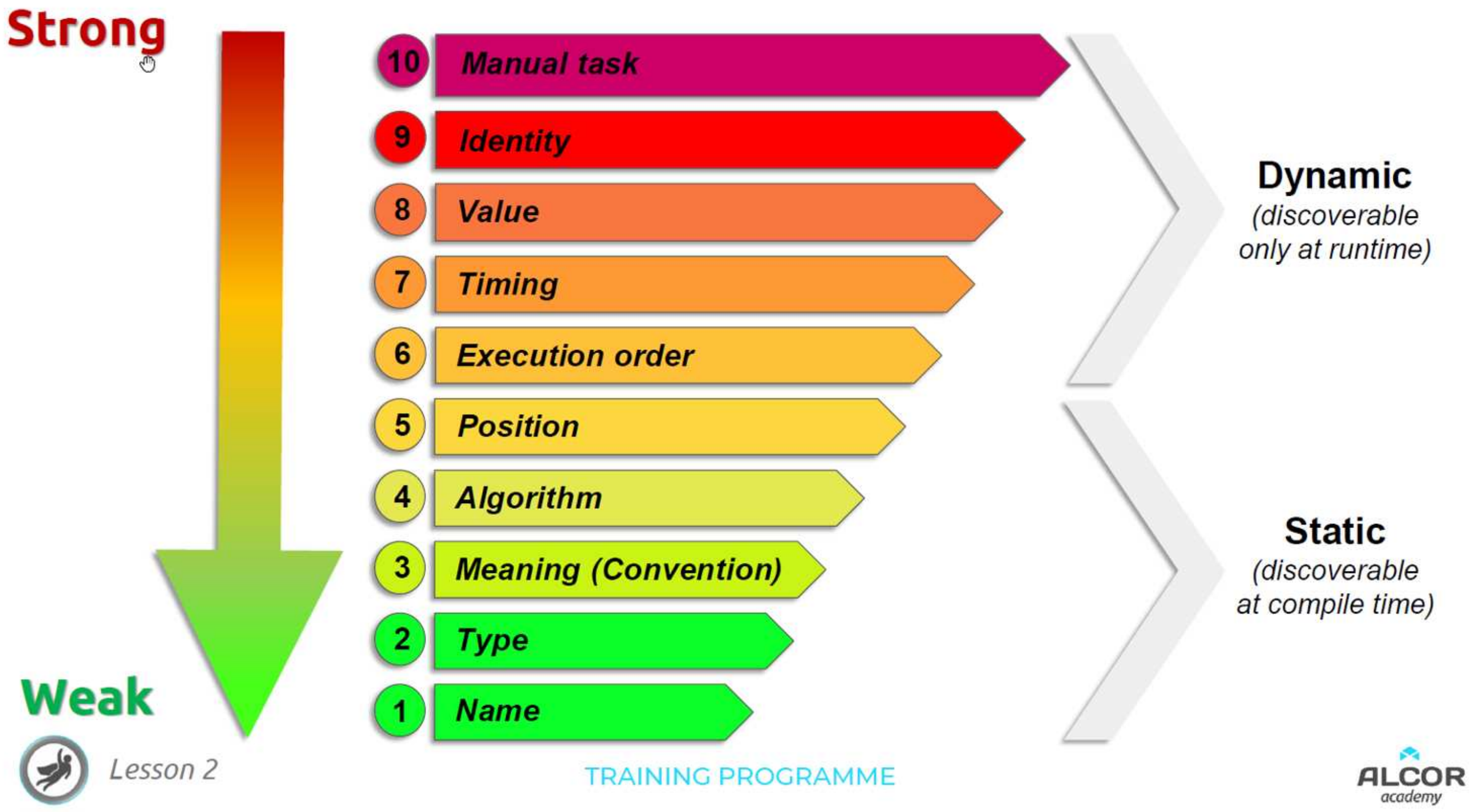
a change in one element would require also a change in the others

Lesson 2

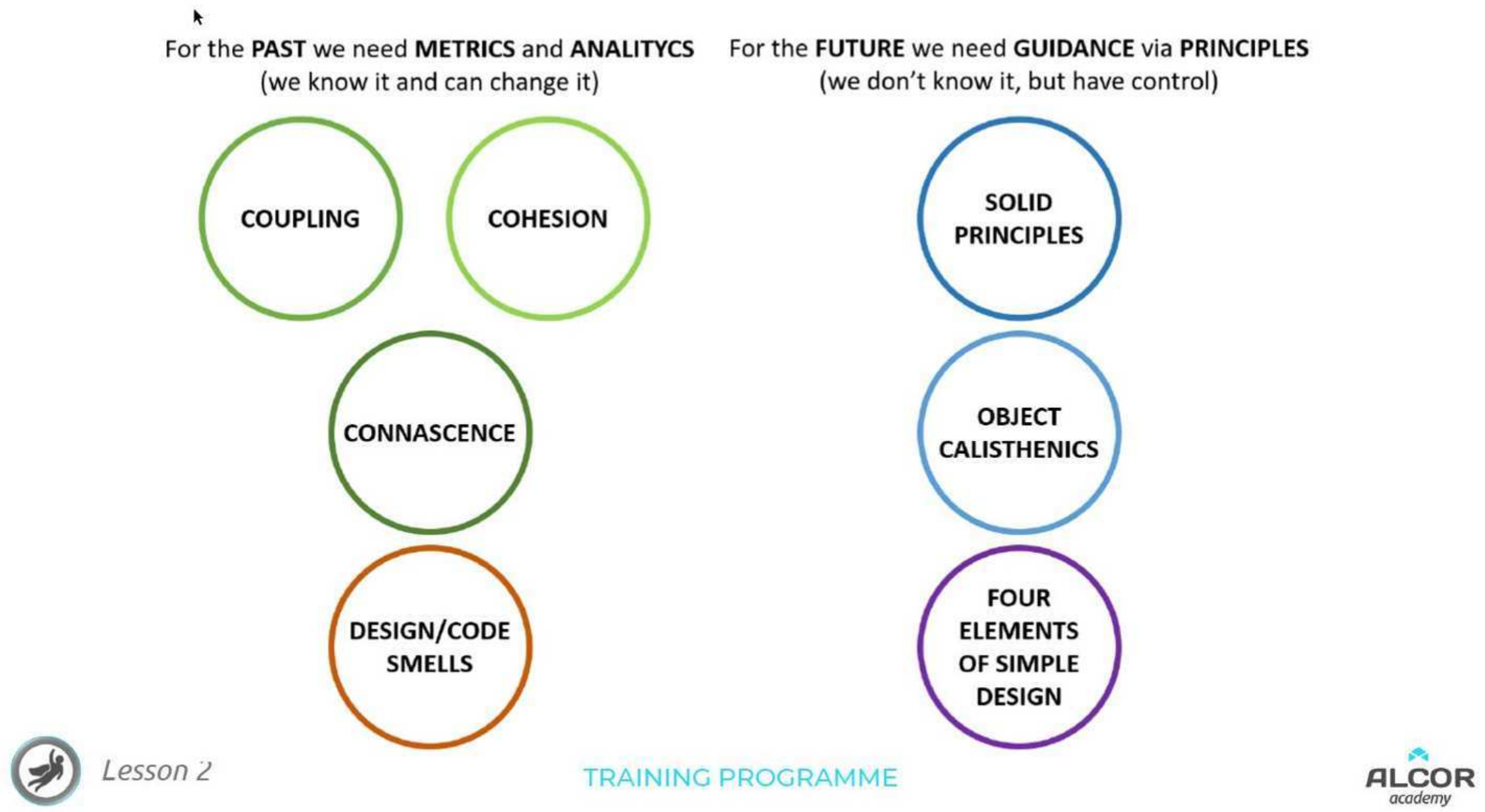
TRAINING PROGRAMME

ALCOR academy

Connascence



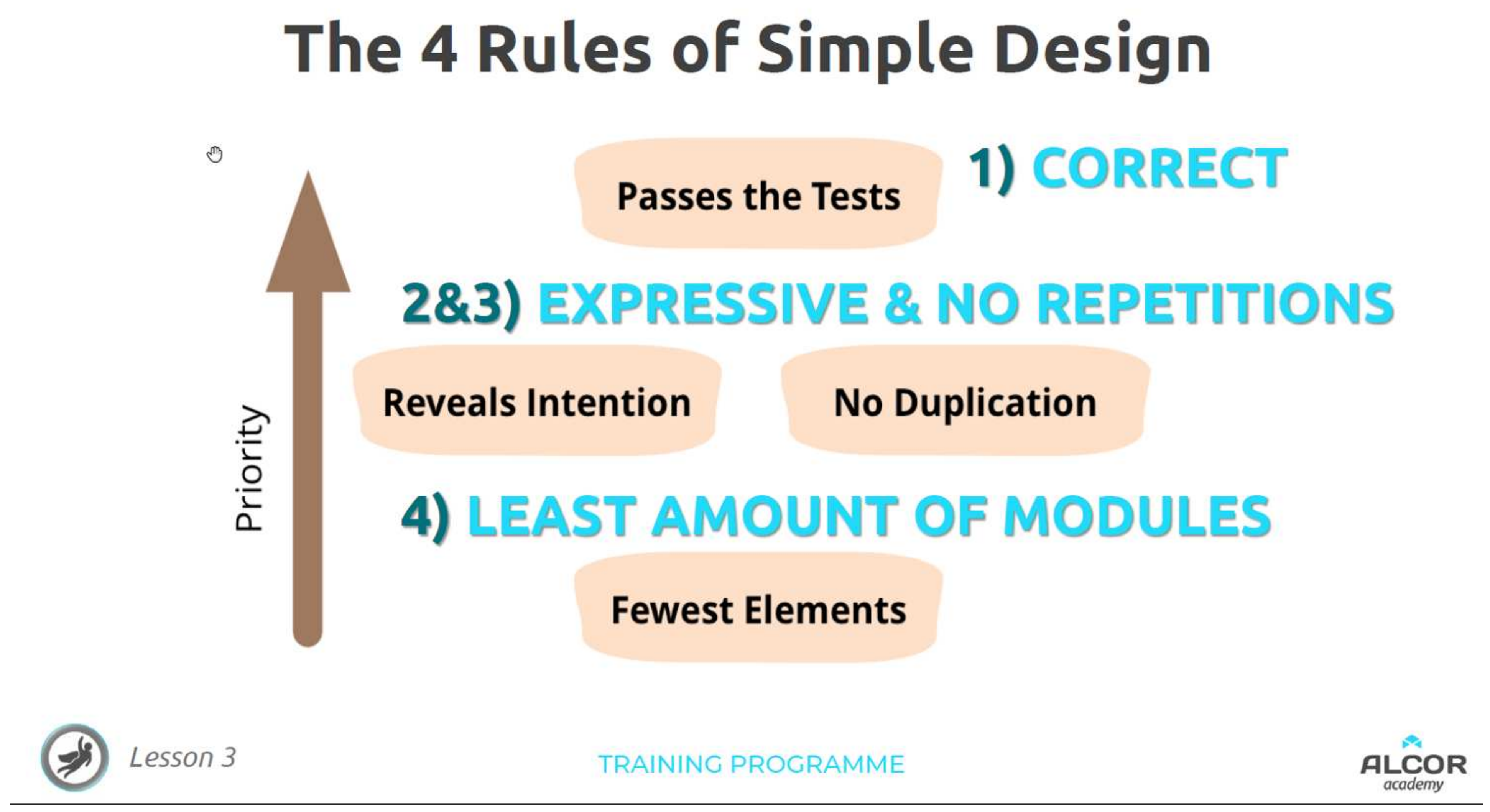
Past vs Future



For the **PAST** we need **METRICS** and **ANALITYCS**
(we know it and can change it)

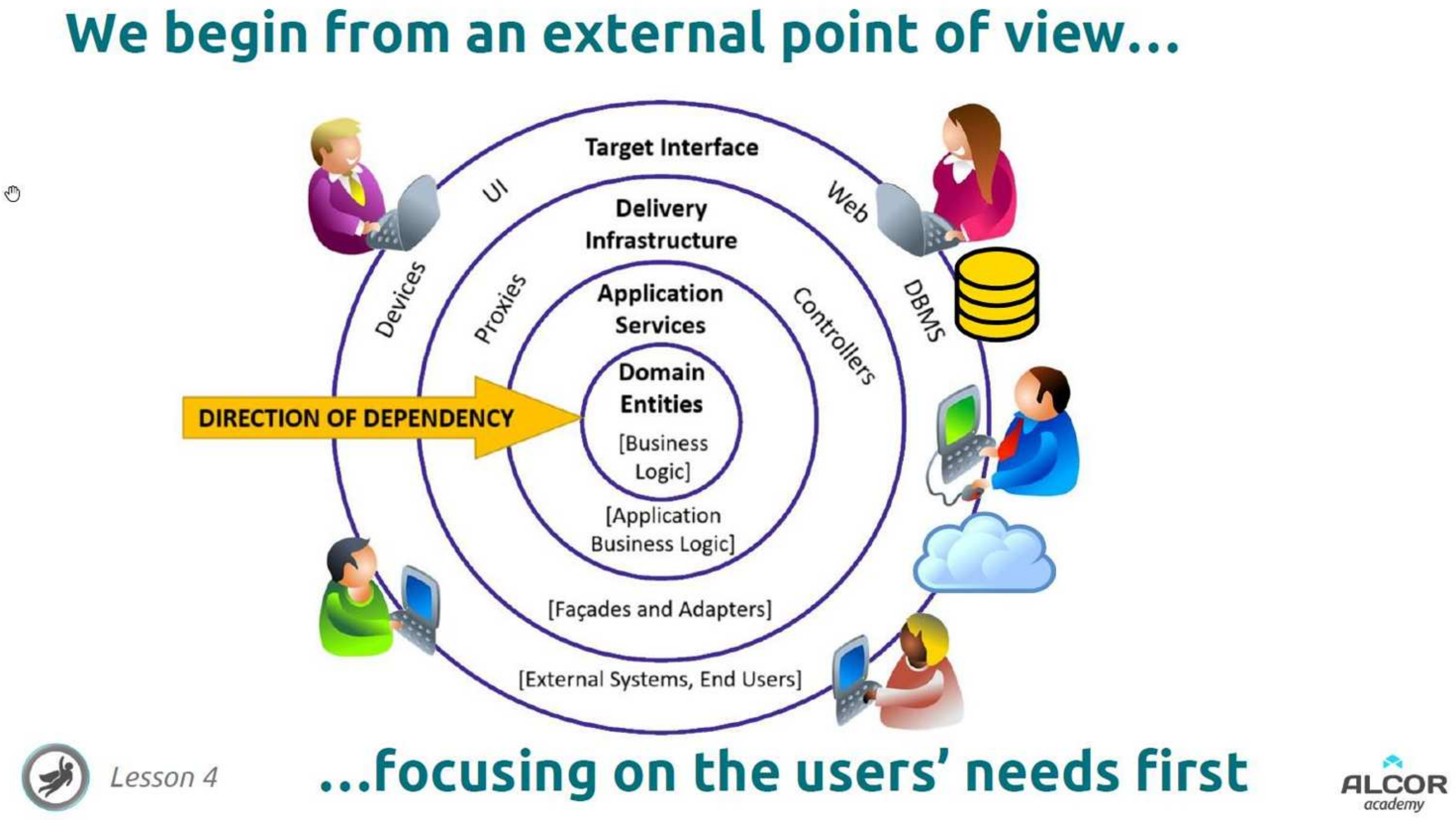
For the **FUTURE** we need **GUIDANCE** via **PRINCIPLES**
(we don't know it, but have control)

The 4 Rules of Simple Design

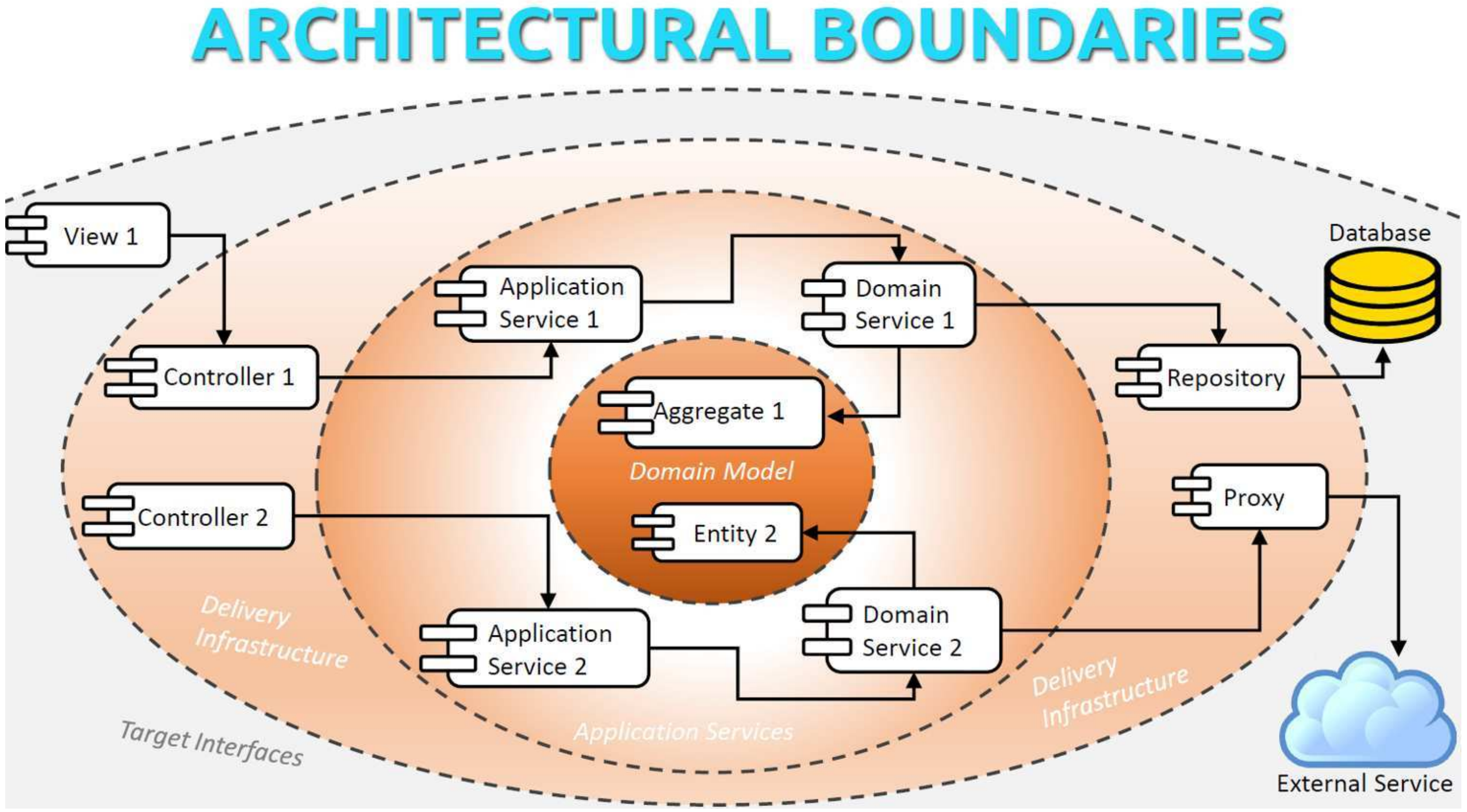


The 4 Rules of Simple Design

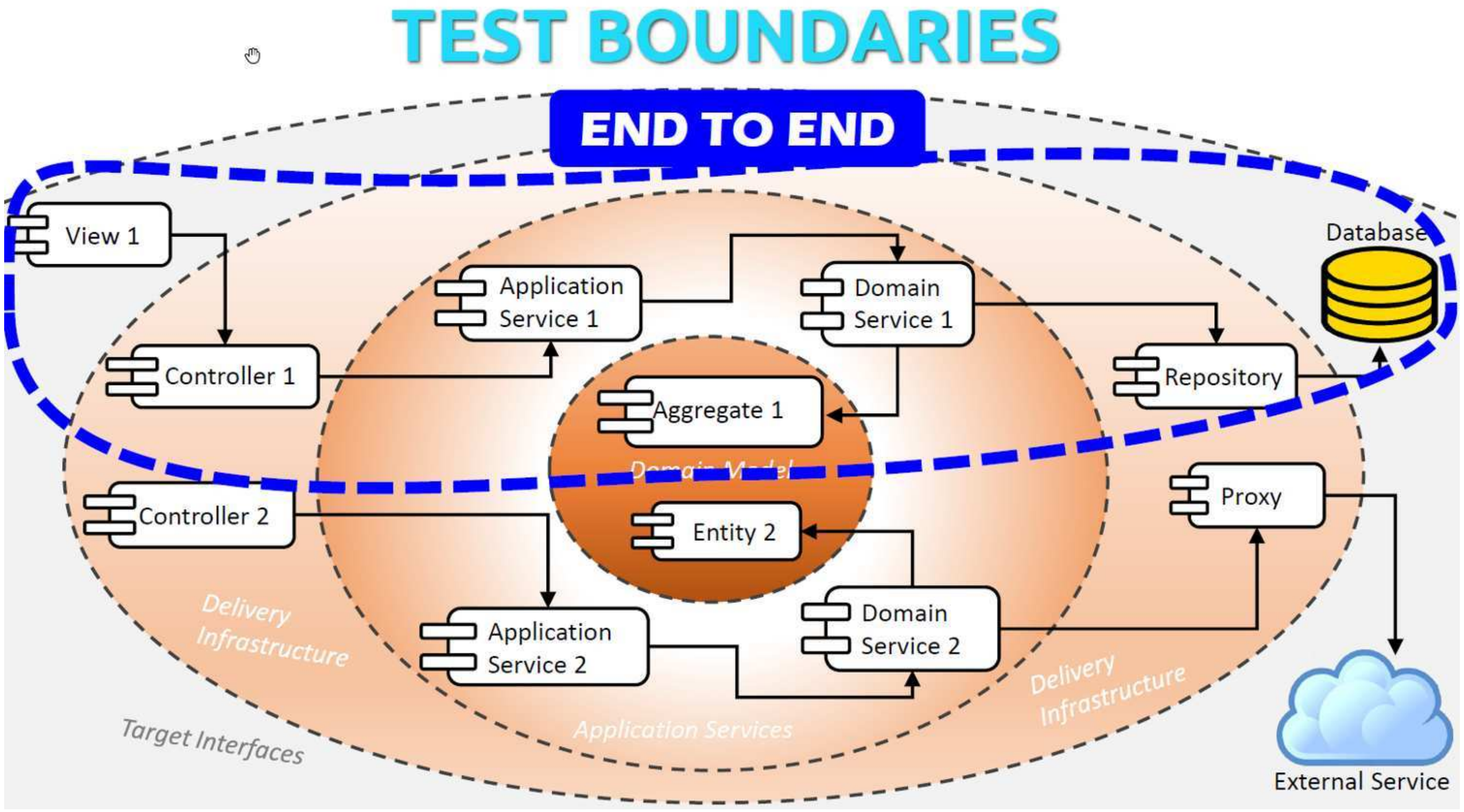
Clean/Onion/Hexagonal Architecture



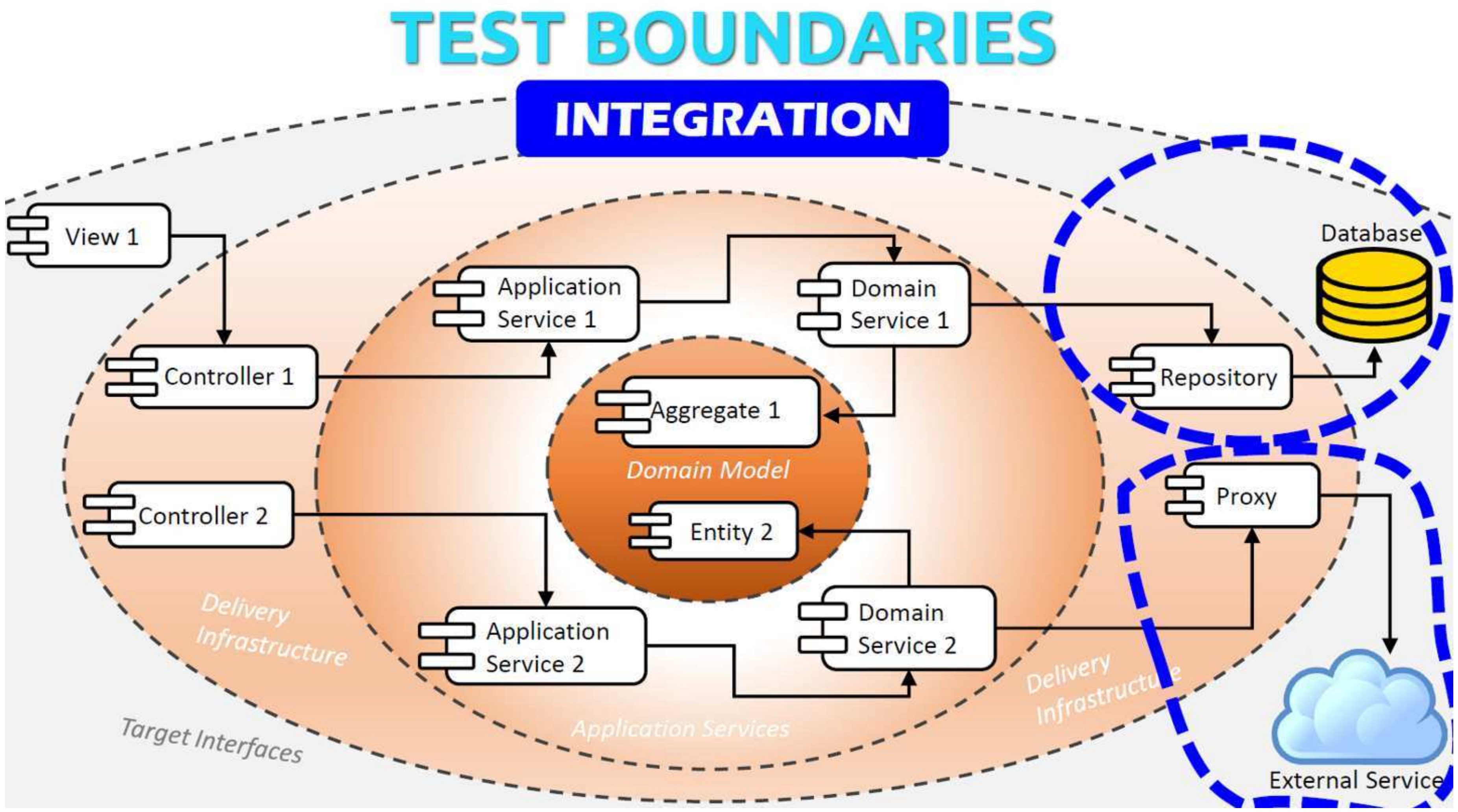
Architecture



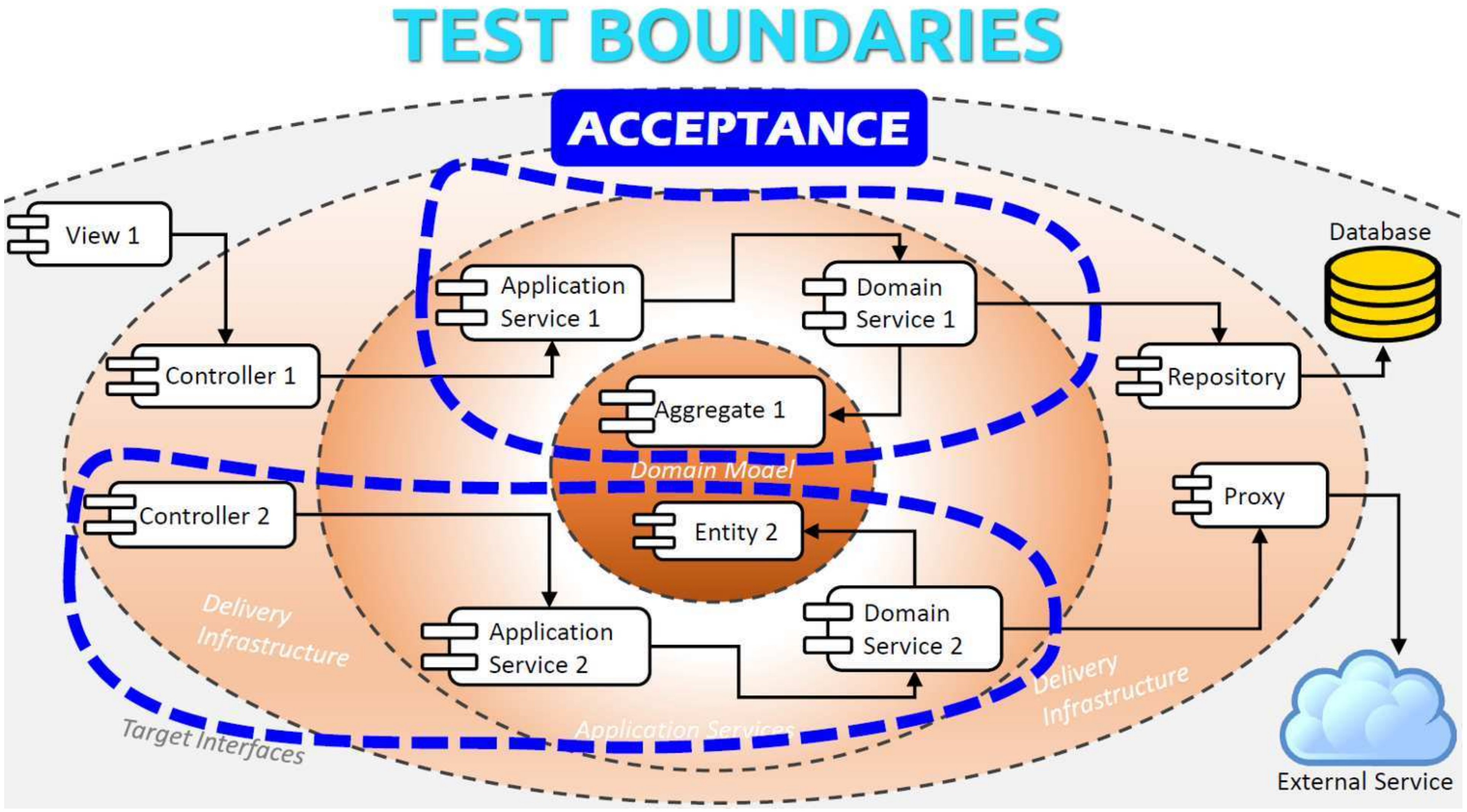
E2E Test


TEST BOUNDARIES
END TO END
View 1
Controller 1
Application Service 1
Domain Service 1
Database
Repository
Aggregate 1
Domain Model
Controller 2
Entity 2
Proxy
Application Service 2
Domain Service 2
Delivery Infrastructure
Delivery Infrastructure
Application Services
Target Interfaces
External Service

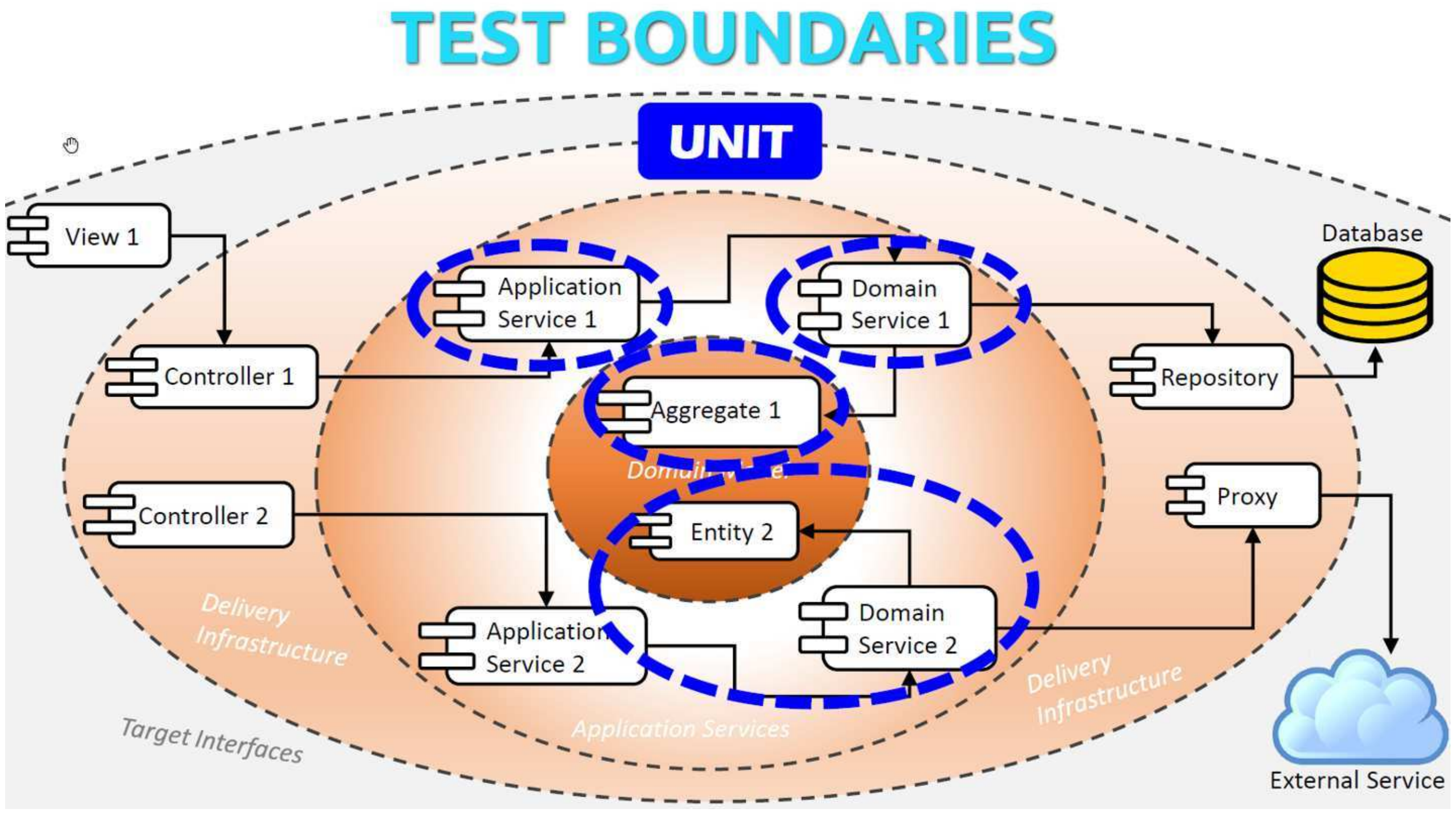
Integration Test



Acceptance Test

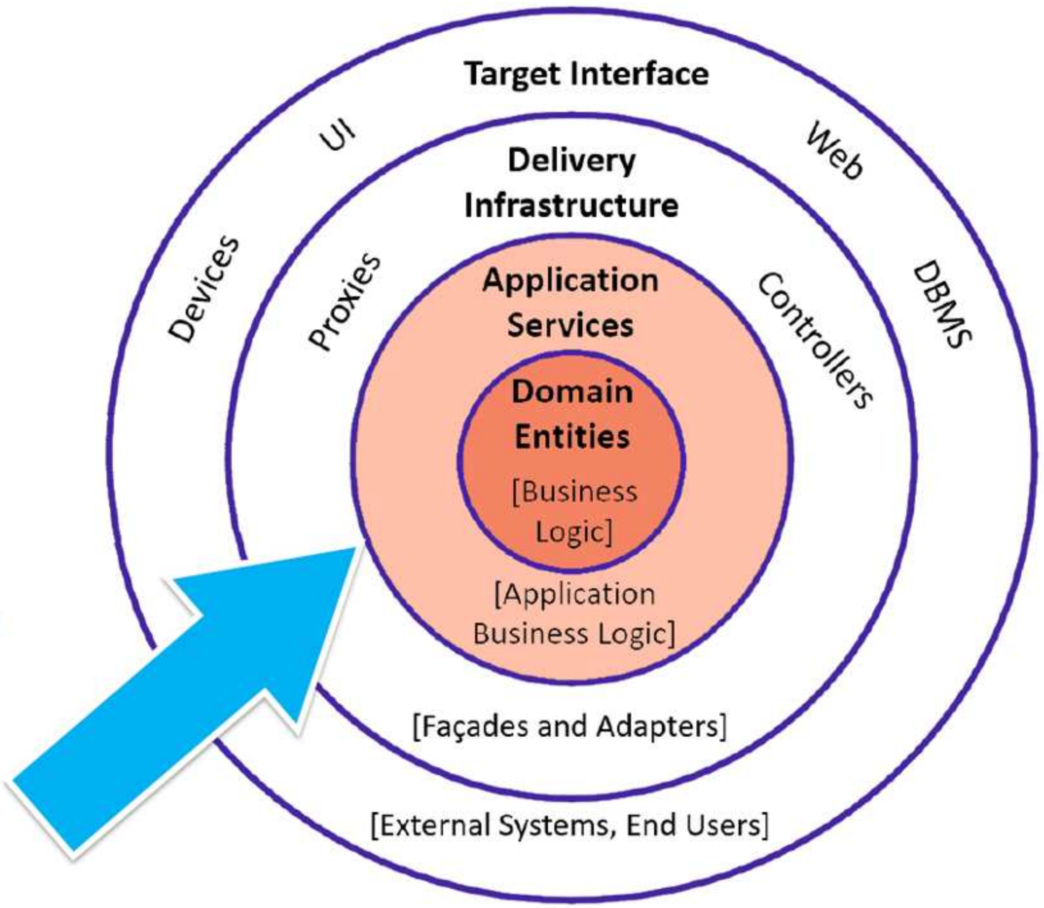


Unit Test


TEST BOUNDARIES
UNIT
View 1
Controller 1
Controller 2
Application Service 1
Application Service 2
Domain Service 1
Domain Service 2
Aggregate 1
Entity 2
Repository
Proxy
Database
External Service
Delivery Infrastructure
Application Services
Target Interfaces
Delivery Infrastructure

Outside-in mindset

We drive the design of our application's public interface from an Acceptance Test, simulating user interaction

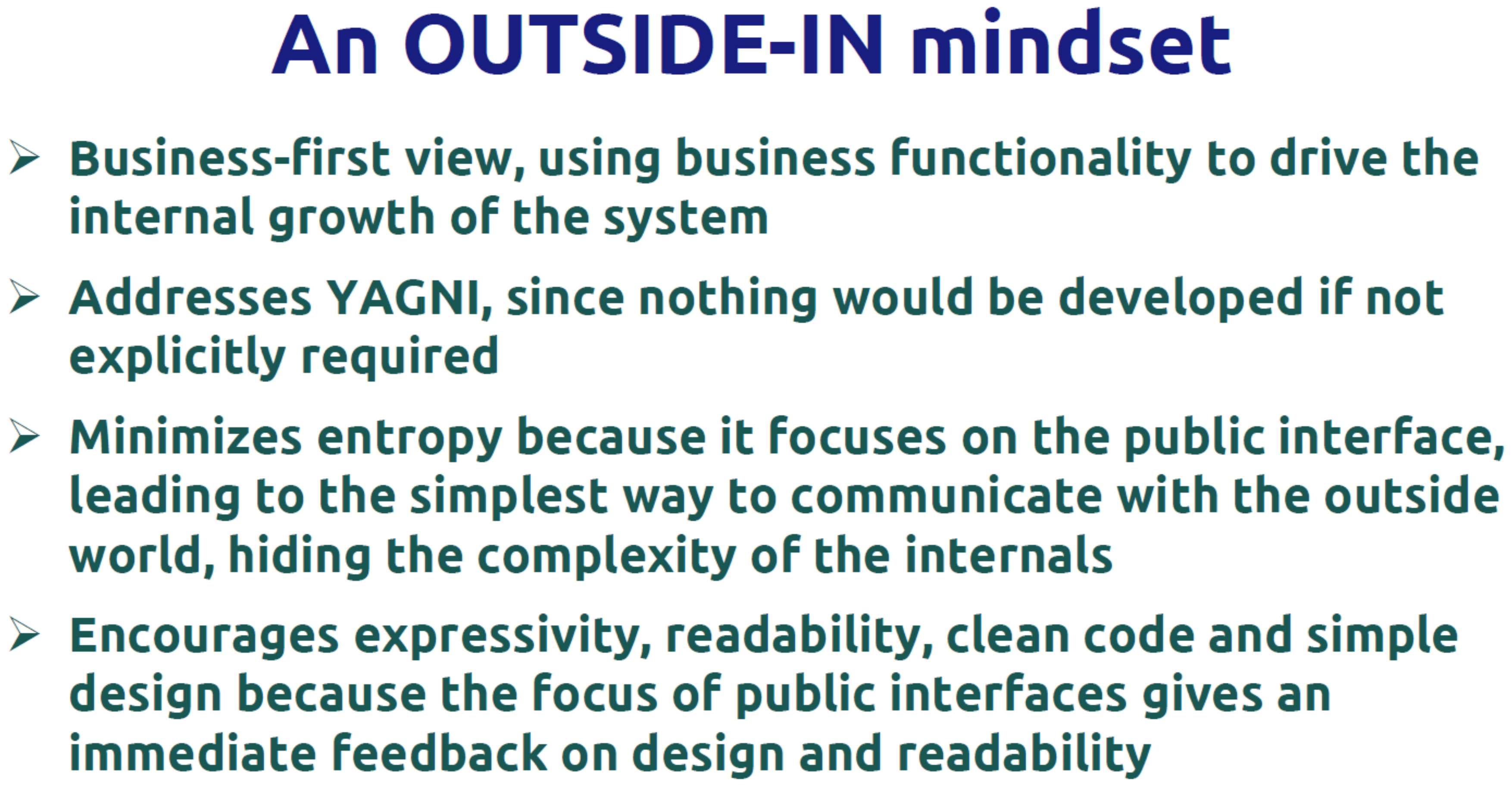


Lesson 4

TRAINING PROGRAMME

ALCOR academy

Outside-in mindset



An OUTSIDE-IN mindset

- ➢ Business-first view, using business functionality to drive the internal growth of the system
- ➢ Addresses YAGNI, since nothing would be developed if not explicitly required
- ➢ Minimizes entropy because it focuses on the public interface, leading to the simplest way to communicate with the outside world, hiding the complexity of the internals
- ➢ Encourages expressivity, readability, clean code and simple design because the focus of public interfaces gives an immediate feedback on design and readability

ALCOR academy

Conclusion

- I learned so much in the last 2.5 month !
  In so many different ways: mob programming, using DIE efficiently, how to write good code, good software design, tests, how to express my opinion,…

- I realized that you don’t have to be a genius to write good code.
  When you respect a few simple rules/aspects, you can write good code.

- I’m now much more confident to write my own code instead of just expanding/adapting/copying the existing code.

- I’m very excited to use my new knowledge more and more in my daily work.

Thank you

- I want to thank you guys from ALCOR Academy for this amazing journey.
  It was awesome, I learned so much and it was one of the best trainings I’ve ever had!