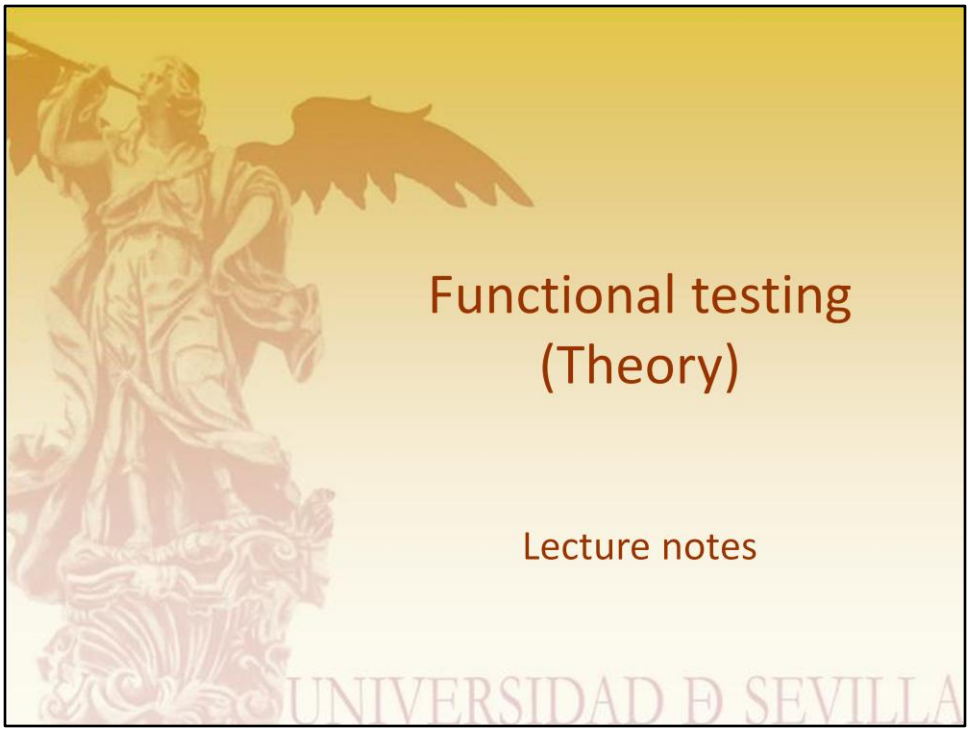


Welcome to this lesson! Today, our goal's to present the theory that you need to produce and run good functional tests.

--

What's functional testing about?

UNIVERSIDAD Ð SEVILLA

2

As usual, we start this theory lecture with a key question: what's functional testing about? As usual, please, make a point of producing your own answer before casting a glance at the following slides.

This is a good definition

It's about devising tests cases that help find defects in a piece of software

UNIVERSIDAD Ð SEVILLA

3

Functional testing's about designing and implementing test cases that help find defects in a piece of software. The utopian goal is to produce software that is defect-free, but this is impossible in practice because typical software systems are far too complex nowadays. The ultimate real goal is to keep their impact on our customers' businesses to a minimum.

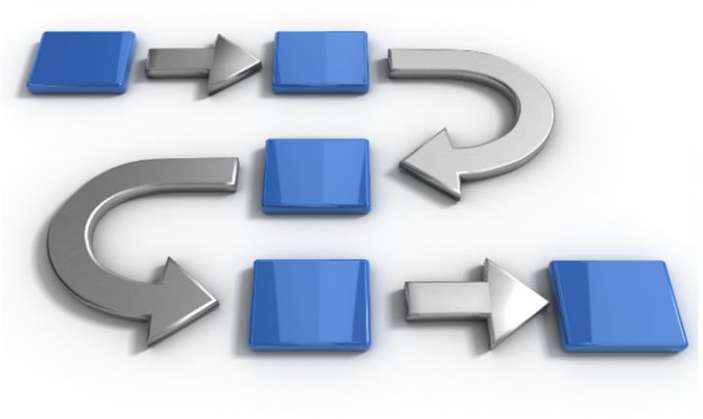
What do I have to know?

UNIVERSIDAD Ð SEVILLA

4

Let’s now report on what you have to know regarding functional testing.

Step 1: a methodology



UNIVERSIDAD Ð SEVILLA

5

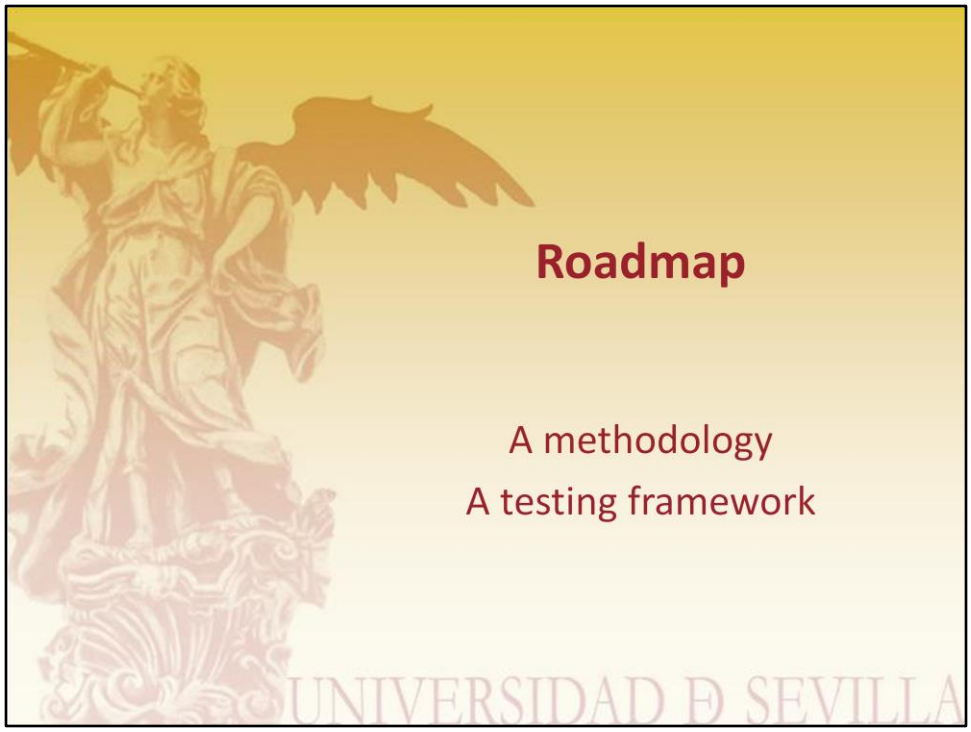
First and foremost important: you need to command a methodology that helps you write good test cases as systematically as possible.

Step 2: a testing framework

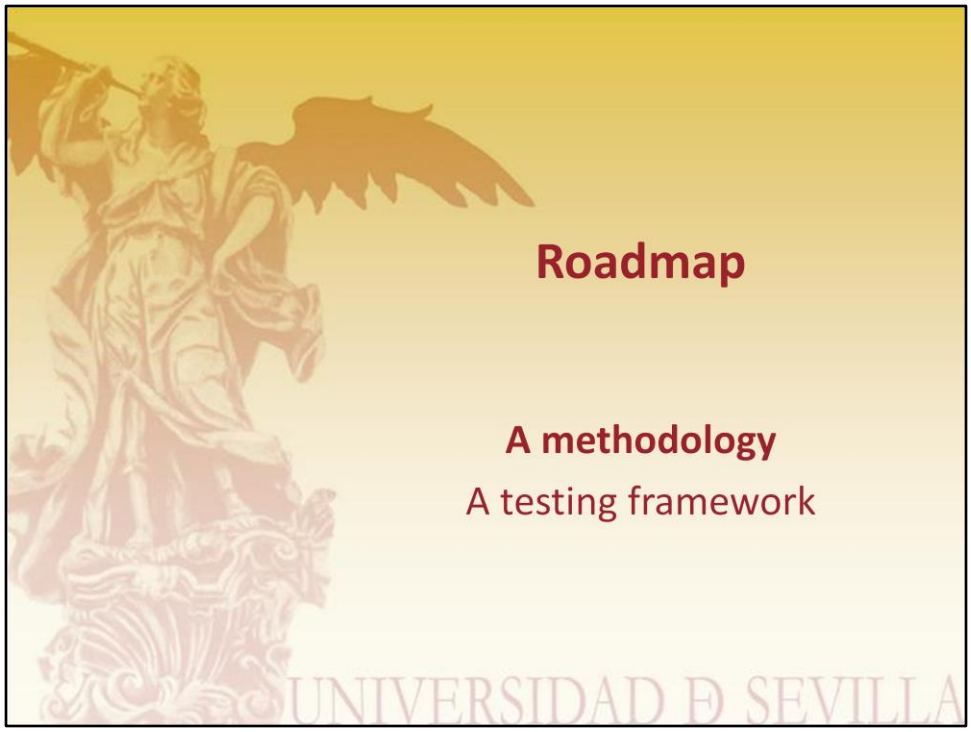
UNIVERSIDAD Ð SEVILLA

6

Then, you need to command a testing framework, that is, a software framework that helps you write and run your test cases.

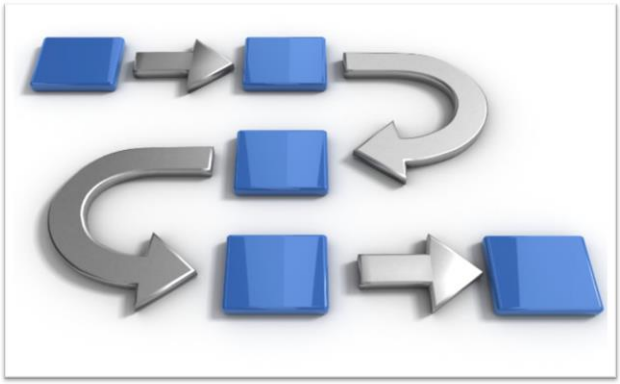


Today's roadmap shouldn't be surprising at all, then: we'll first provide an insight into the methodology and then will report on the testing framework. Pretty simple.



Let’s start with the methodology.

What is it about?

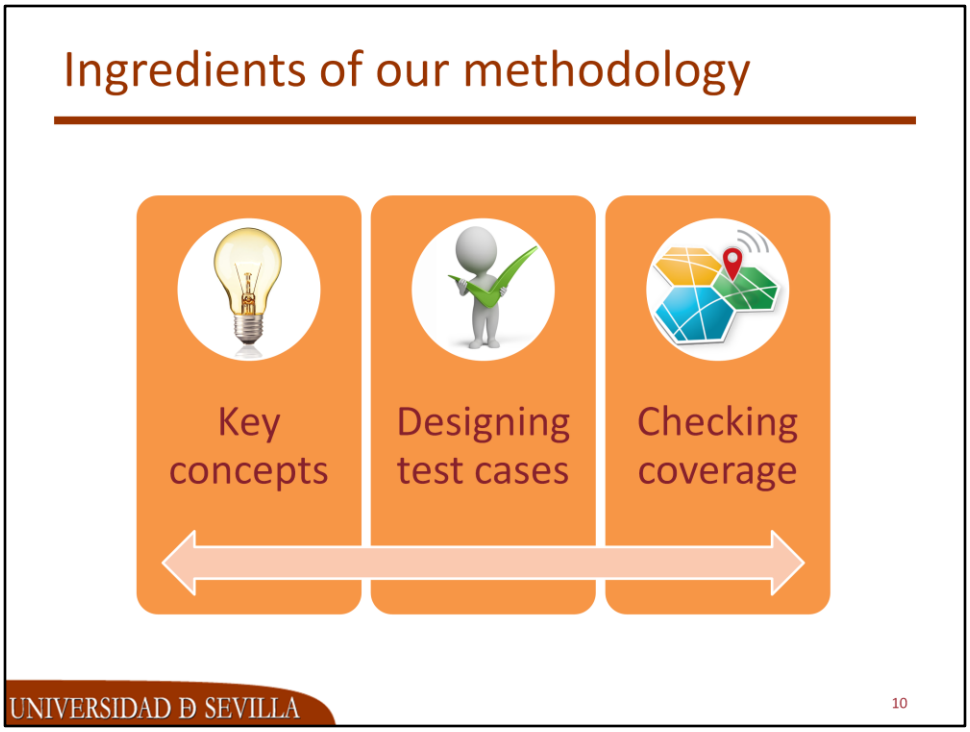


It's a systematic approach to *start* writing good test cases building on use cases

UNIVERSIDAD Ð SEVILLA

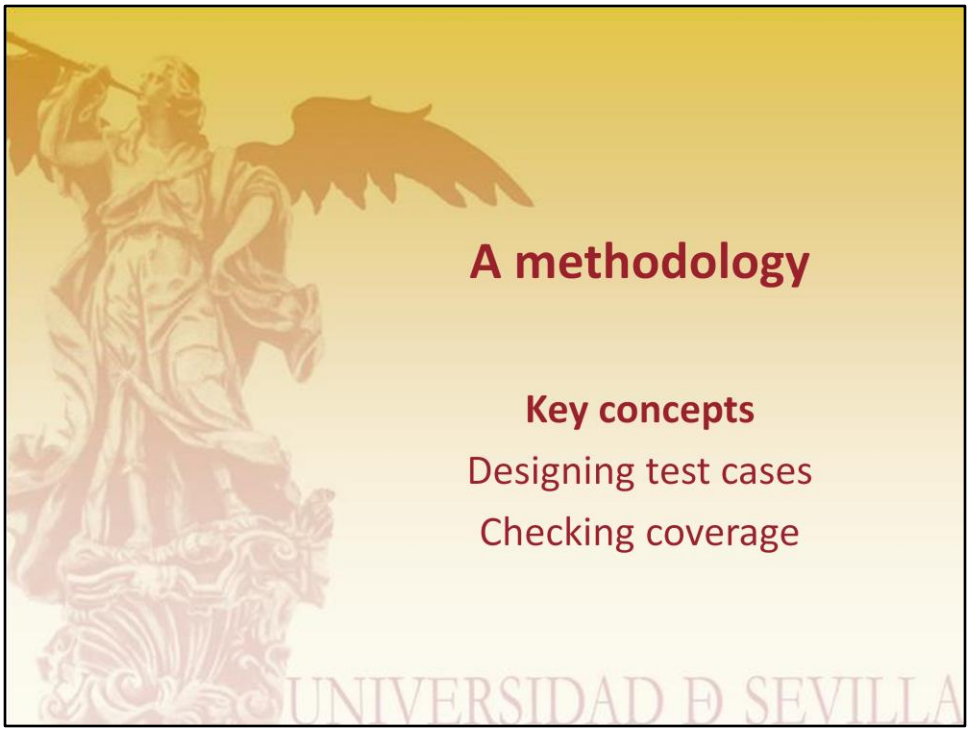
9

Our methodology's a systematic approach to *start* writing good test cases building on use cases. Please, note that we emphasise "start" because it's actually a systematic starting point, but requires you to command your requirements and be smart regarding finding defects in their implementation. Simply put: it's a guideline that helps you, not an algorithm that can be implemented by a computer.

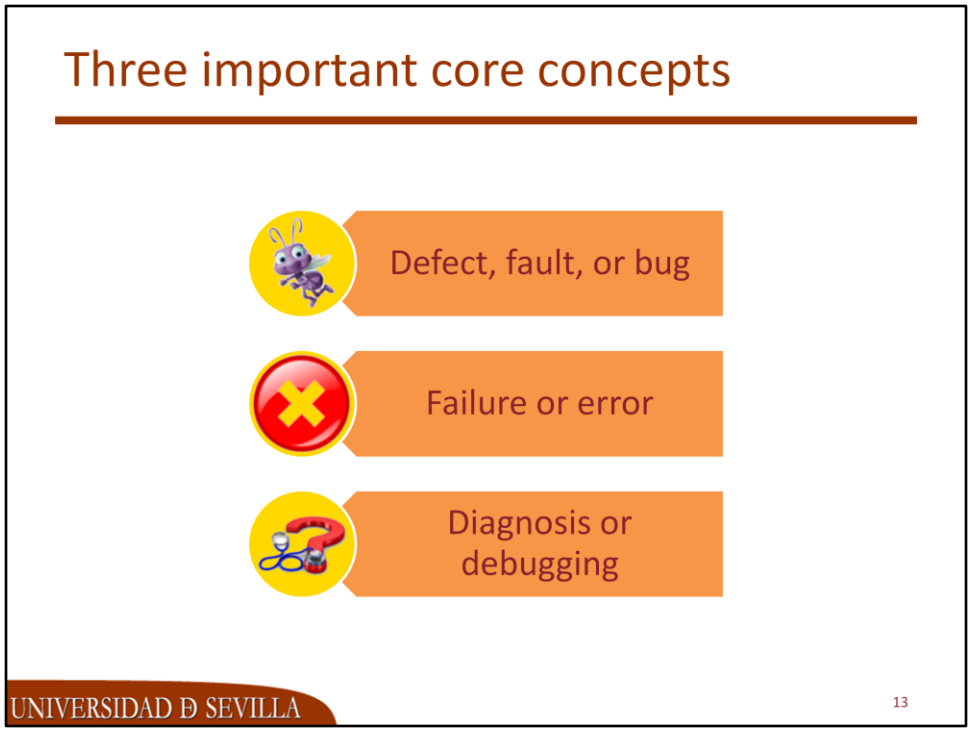


To understand our methodology, we need to command the following ingredients: some key concepts, designing test cases, and checking their coverage.

So, these are the topics with which we're going to deal in this section.



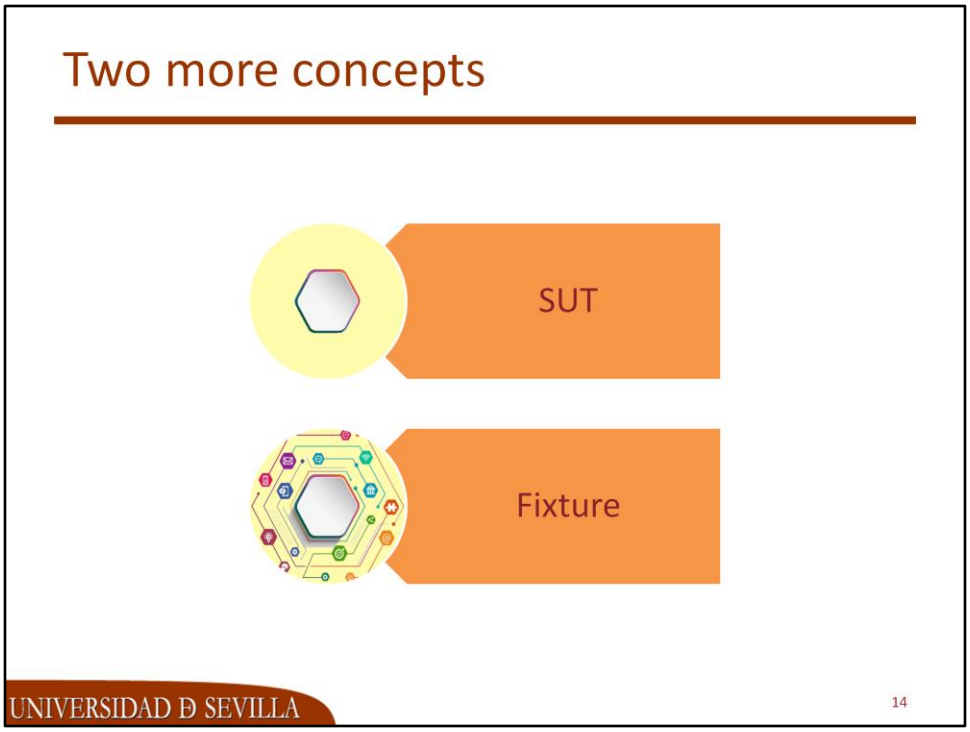
Let's start with some key concepts.



First of all, let's define three important core concepts:

- Defect, fault, or bug: these terms are synonyms and they all refer to something wrong in your code. For instance, you wrote "if x > 0 then…" instead of "if x >= 0 then…" then your software has a defect, a fault, a bug.
- Failure or error: it's the run-time effect of a defect. Failures are detected because something does not work as expected. For instance, "the system didn't transfer money well today at 17:00h".
- Diagnosis or debugging: it's a process by means of which a failure is traced back to the defect/s that caused it. For instance, "the system didn't transfer money well today at 17:00h because the code reads `if x> 0 then ...' instead of `if x >= 0 then ...'.

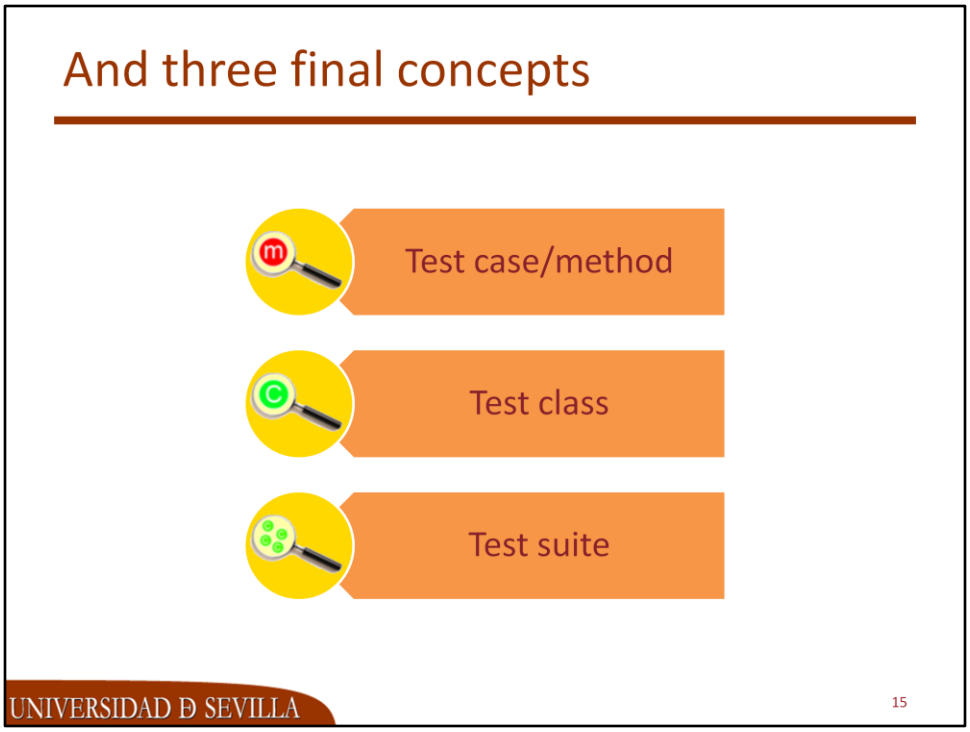
The whole thing typically works as follows: you realise that there must be a defect whenever you observe a failure in your system, that is, a behaviour that deviates from what is expected; typically, someone in your team will be assigned the task of diagnosing the failure in order to find the defect that caused it; once it's found, it'll be corrected as soon as possible and additional test will be performed to ensure that the corrections haven't introduced any further defects.



There are two more concepts that are related to your system.

- SUT: it's an acronym for "System Under Test". Although "system" is typically used to refer to something big and complex, in the context of functional testing, "system" refers to a variety of artefacts that range from an object, to a collection of objects, a component, or a whole web information system. In our projects, the SUT will typically be a service or a repository. Unfortunately, we can't provide you with a tool to test controllers. We'll defer testing controllers to the lesson on acceptance testing.
- Fixture: this term refers to your SUT and the web of ancillary objects that it requires to work. For instance, assume that you're testing a repository; that's your SUT, but it won't work alone: it needs a data source, a connection pool, a Spring context… that's your fixture. Please, note that the SUT is a part of your fixture.

When working with our Spring-based framework, you typically can focus on the SUT and forget about the fixture; it's the framework that cares of most details to set the fixture up and get it running. You typically only have to write some configuration files, little else.



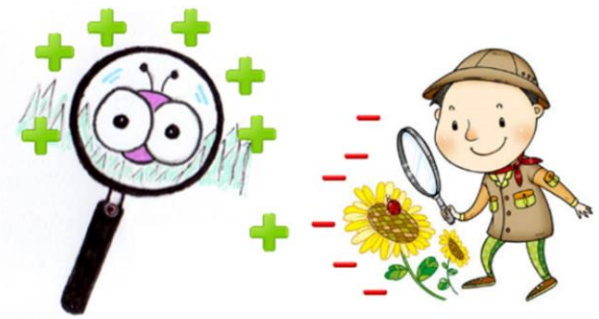
We have three more concepts, namely:

- Test case/method: it's a method whose goal is to make failures shows up; in such cases, a developer will have to diagnose them to find the corresponding defects.
- Test class: it's a class with a collection of test cases.
- Test suite: it's a collection of test classes.

They all are written in Java within the context of a testing framework and they are a part of your projects.

Let’s now delve into designing test cases.

Kinds of test cases

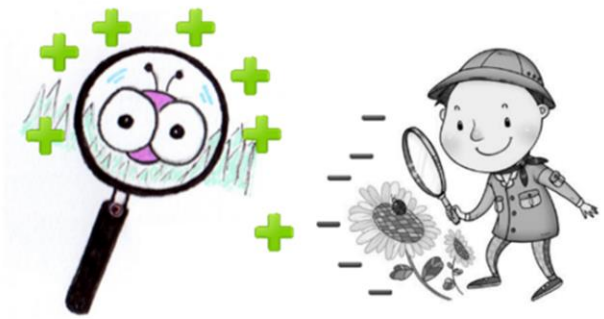


UNIVERSIDAD Ð SEVILLA

17

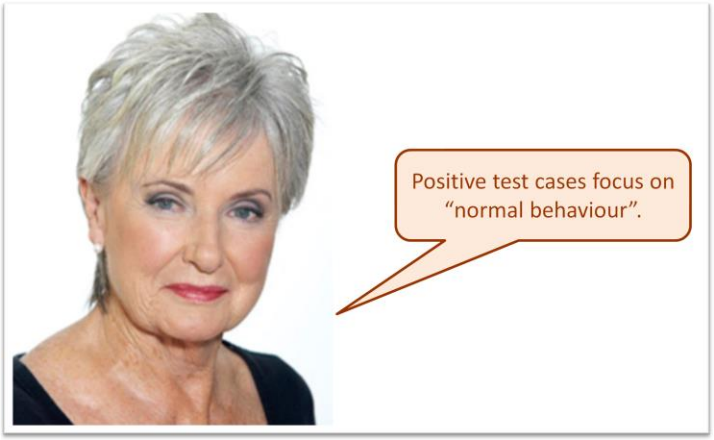
There are two kinds of test cases: positive and negative ones. In order to assemble a good test suite, we must create both of them since they focus on different defects.

Kinds of test cases

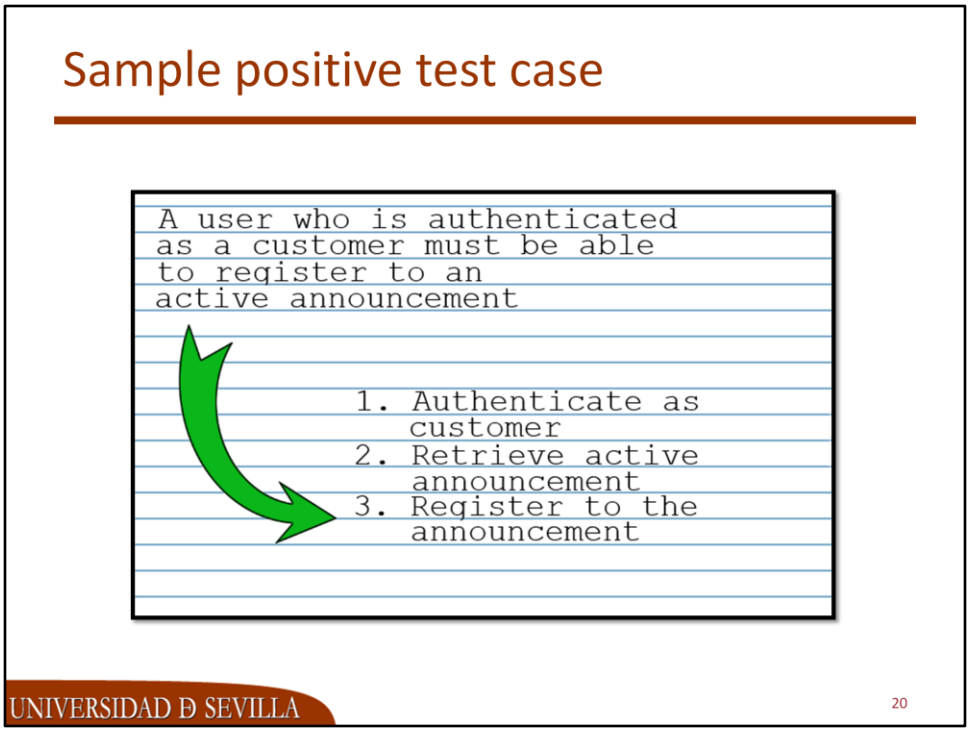


Let’s first analyse positive test cases.

A good definition

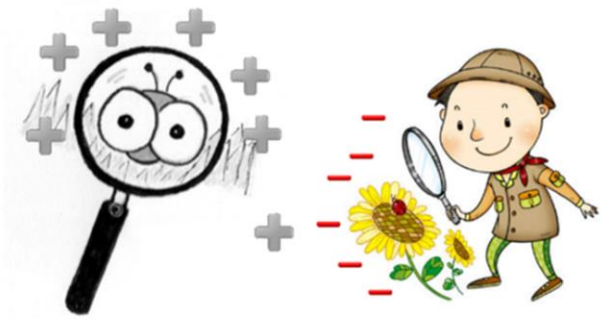


Simply put: positive test cases focus on “normal behaviour”. They’re intended to simulate use cases in situations in which nothing should fail.



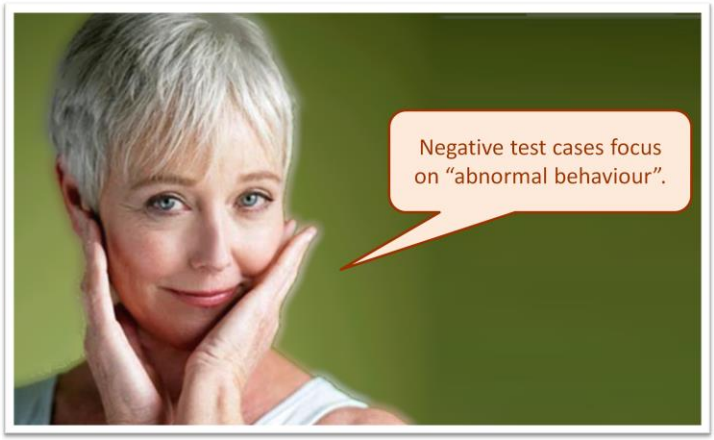
For instance, in project Acme Certifications, there is a functional requirement that reads as follows: "a user who is authenticated as a customer must be able to register to an active announcement." The corresponding use case might be as shown in this slide: a user authenticates as a customer, then retrieves the active announcements, and finally registers to one of them. No exception is expected when running this test case; if an exception's caught, then there's a failure, and we then have to diagnose it in order to find the corresponding defects.

Kinds of test cases



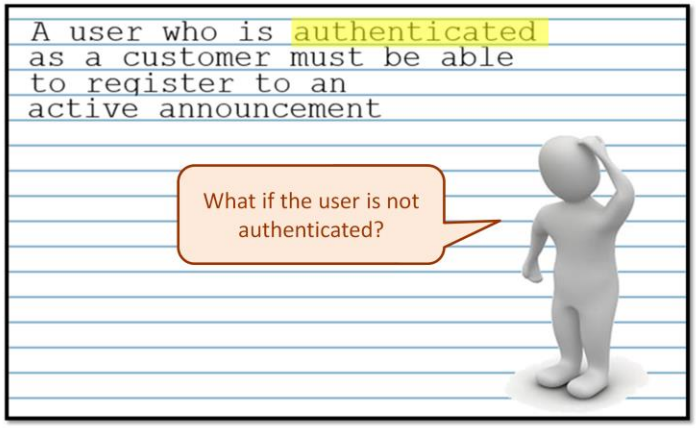
Let’s now analyse negative test cases.

A good definition



Simply put: negative test cases focus on “abnormal behaviour”. They’re intended to find breaches in the implementation of business rules.

Sample negative test case (I)

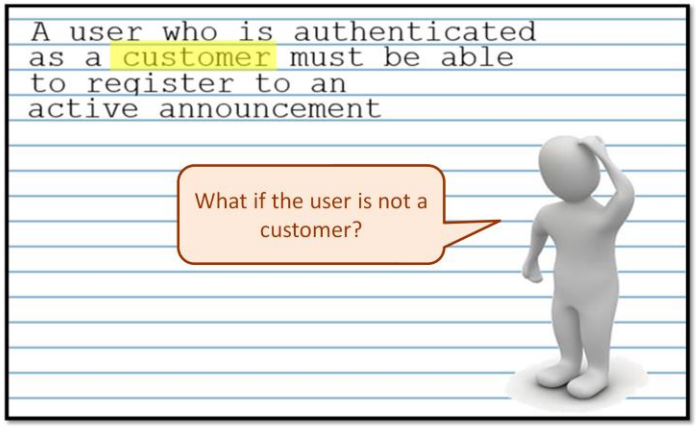


UNIVERSIDAD D SEVILLA

23

For instance, our running example requires the customer to be authenticated, so we should write a test case in which the user who attempts to register is unauthenticated.

Sample negative test case (II)

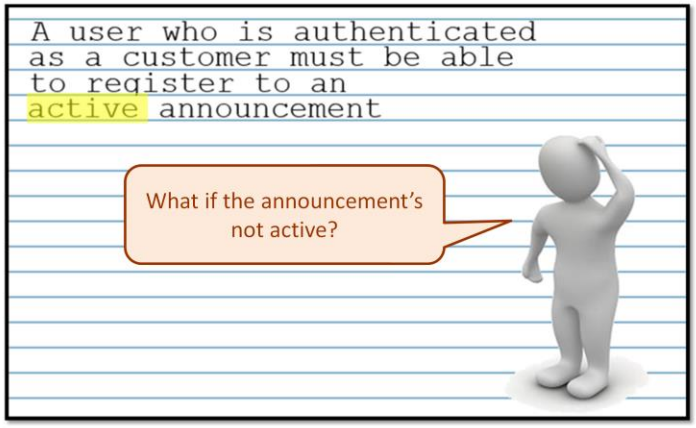


UNIVERSIDAD D SEVILLA

24

Let's try to break another business rule: the user who registers to an announcement must be authenticated, right. What if he or she's not authenticated as a customer?

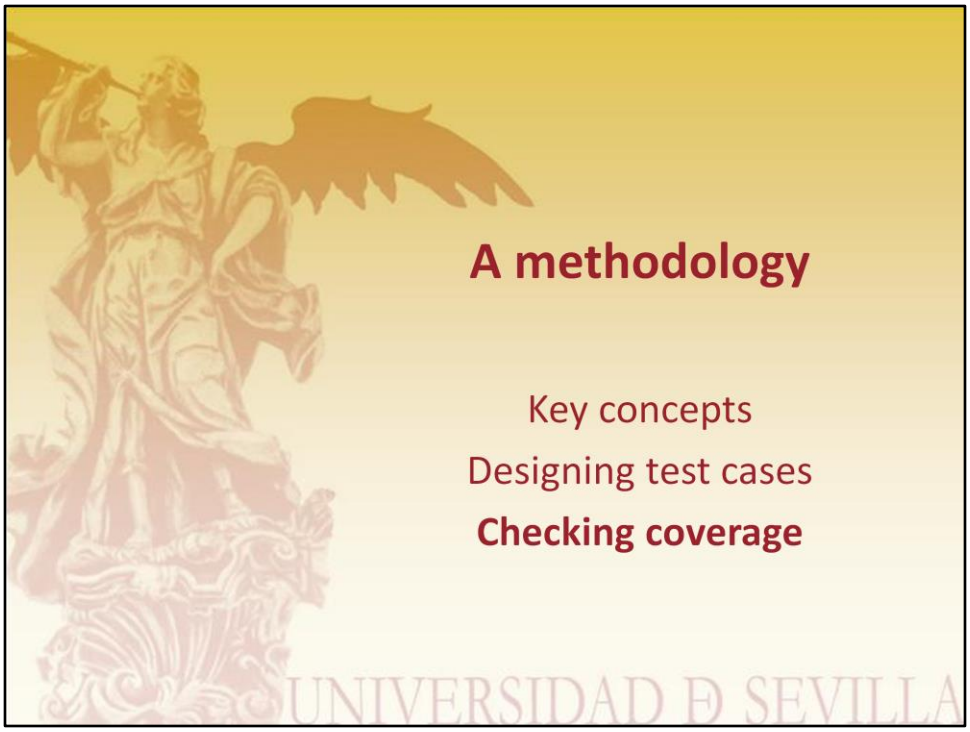
Sample negative test case (III)



UNIVERSIDAD D SEVILLA

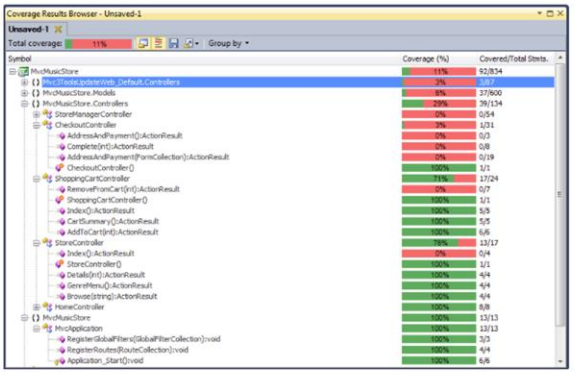
25

And finally, the announcement must be active. What if it is not?



Let's now delve into another important problem: checking coverage.

What's coverage?



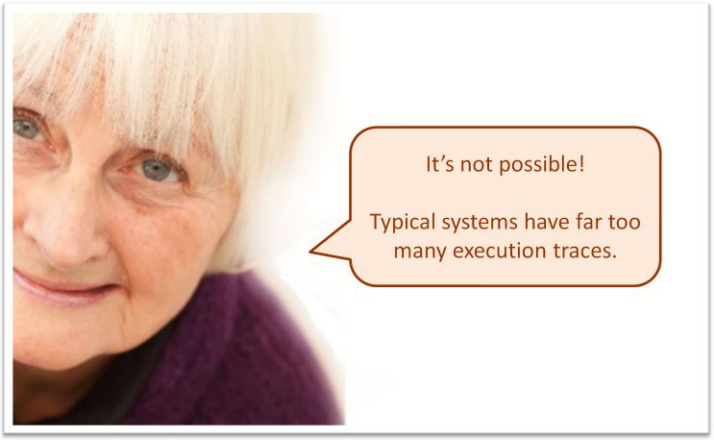
It's the percentage of execution traces that your test suite checks. The higher the coverage, the more you can trust your system

UNIVERSIDAD Ð SEVILLA

27

Coverage is defined as the percentage of execution traces that your test suite checks, that is, the percentage of execution traces that behave as expected, not anomalously. In other words: it's the percentage of execution traces that are known to work well regarding the total number of traces of a system. So the idea is to design a test suite that has as a high coverage as possible, so that we can be confident that existing defects are not evident at all. In other words, the higher the coverage, the less chances that a defect is unnoticed.

Can I achieve 100% coverage?



UNIVERSIDAD Ð SEVILLA

28

Is it generally possible to achieve 100% coverage? Unfortunately, the answer's no. Typical systems are far too complex and enumerating all of their execution traces is not generally possible. We can just make our best effort to achieve as a high coverage as possible.

How to achieve a high coverage?

UNIVERSIDAD Đ SEVILLA

29

We have a number of strategies regarding how to maximise the coverage: the goal's that your test suites must ensure that every statement's executed at least once and that every possible data variant is fed into your system. Please, note that this doesn't guarantee that the coverage is, say, 80%, 90% o 95%; this just guarantees that it is high enough for practical purposes.

Covering strategies

Statements

- Sequences
- Conditionals
- Loops

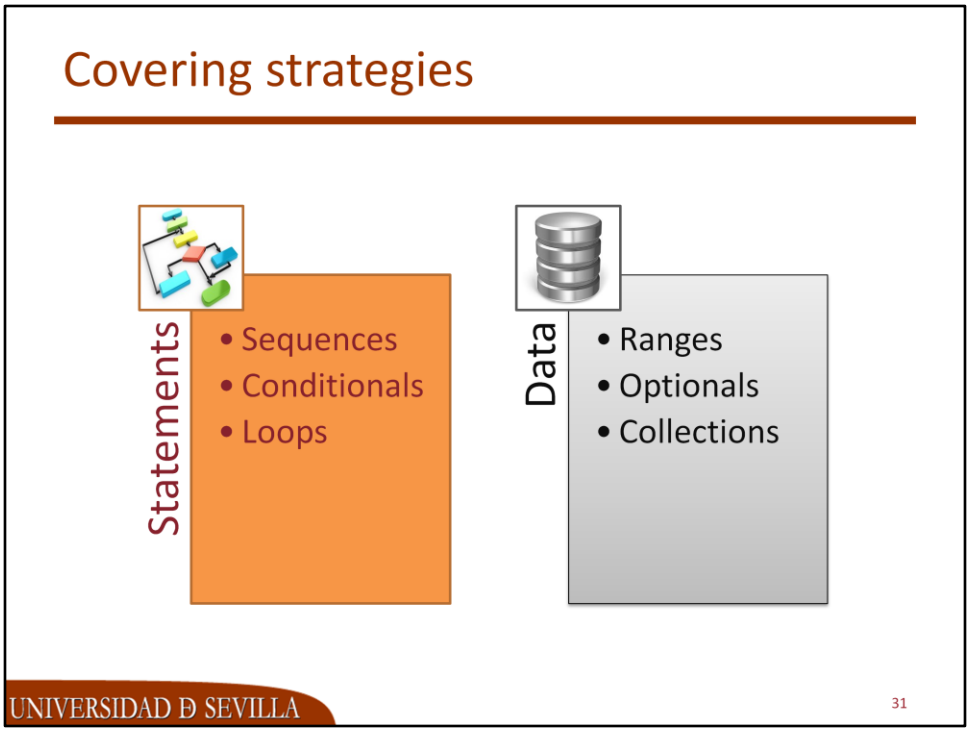
Data

- Ranges
- Optionals
- Collections

UNIVERSIDAD D SEVILLA

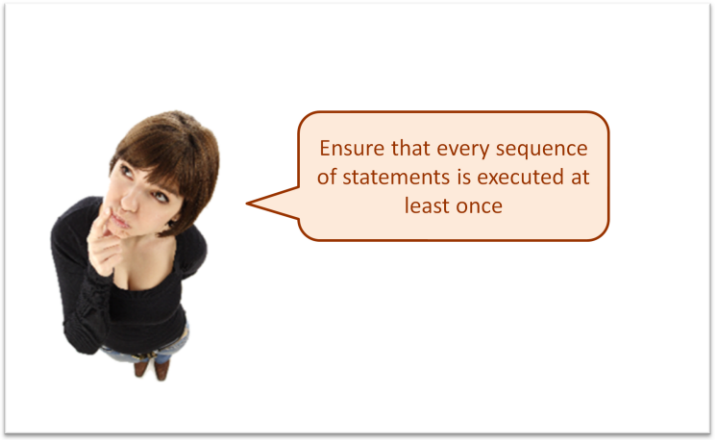
30

Let's now explore two covering strategies.



Let's start with the statements, which can be either sequences, conditionals, or loops.

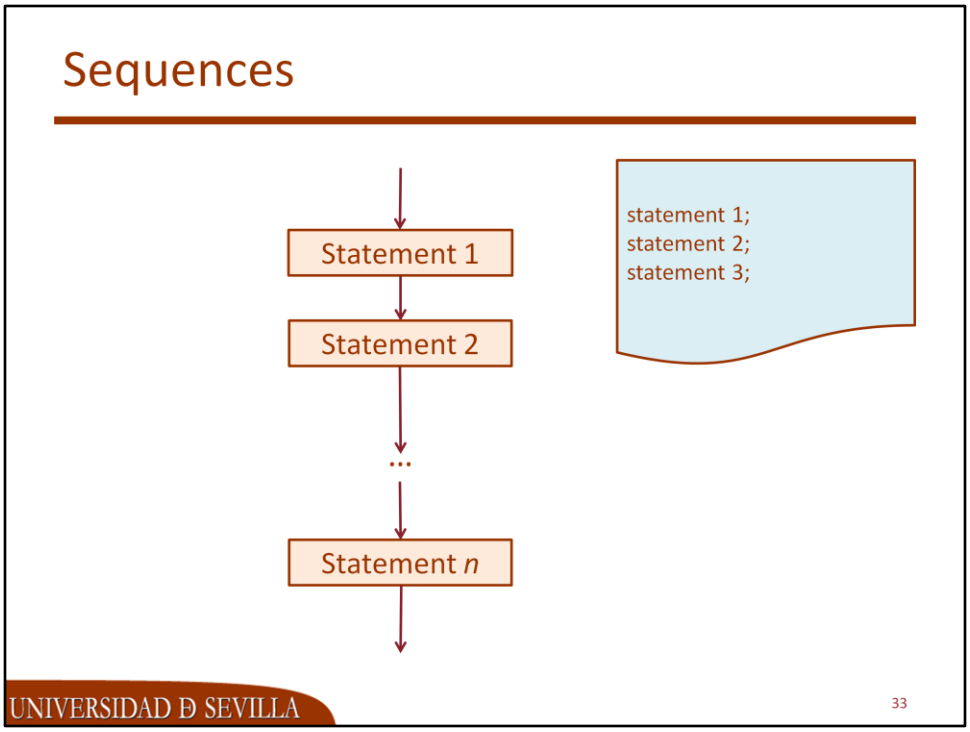
Regarding sequences



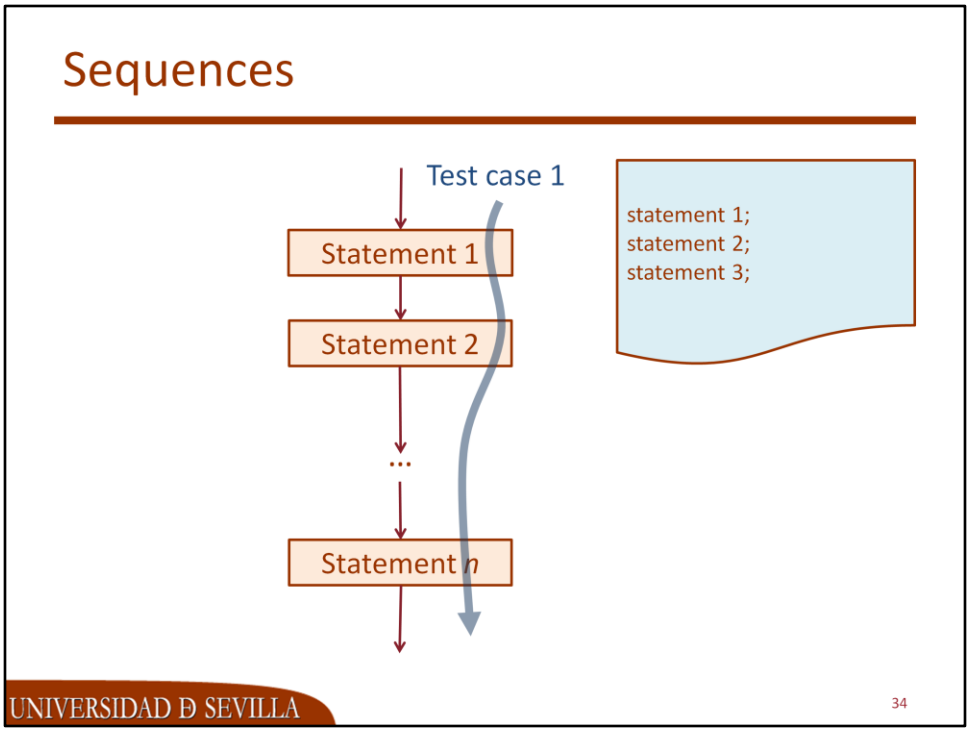
UNIVERSIDAD Ð SEVILLA

32

Sequences are the simplest case: your test cases must ensure that every sequence of statements is executed at least once.

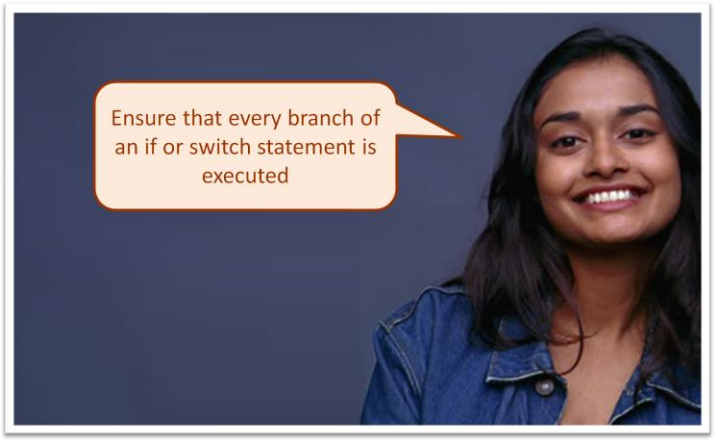


This slide shows how a typical sequence of statements looks like.



Simply put: you must design a test case in which the testing data used allows every statement in the sequence to execute one after the other.

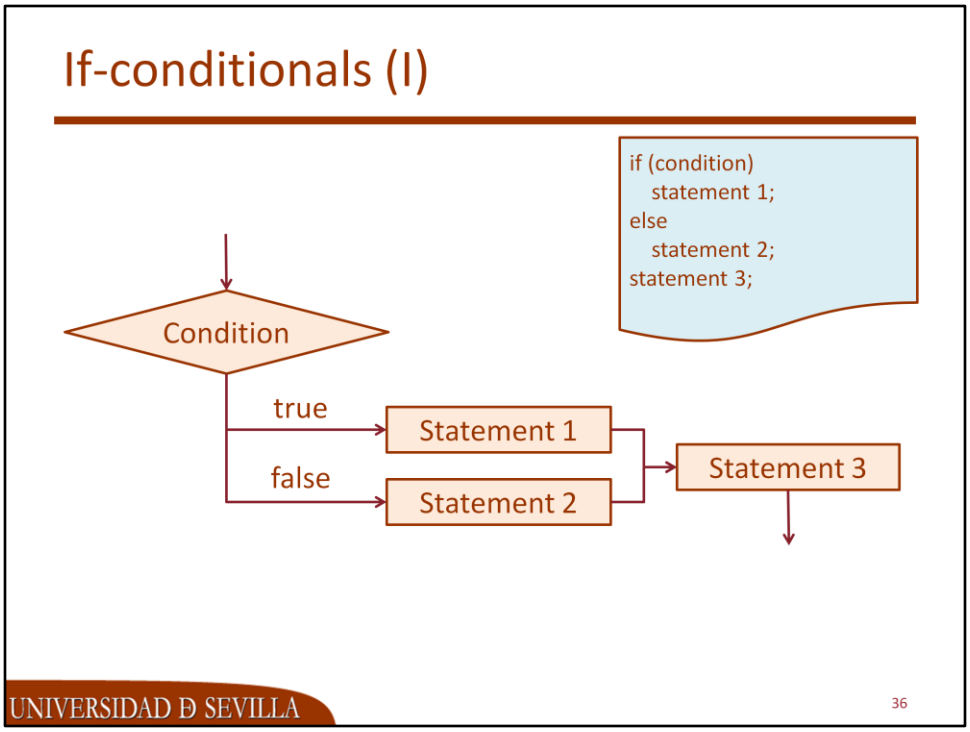
Regarding conditionals



UNIVERSIDAD Ð SEVILLA

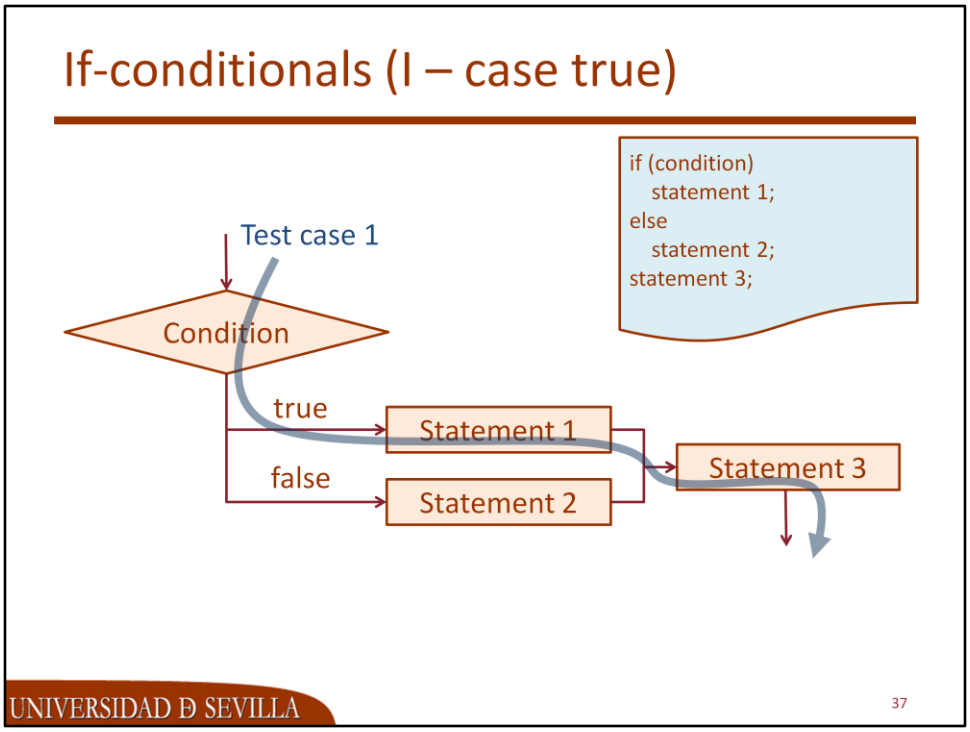
35

Conditionals are a little more involved. Your test cases must ensure that every branch of an if or a switch statement is executed least once.

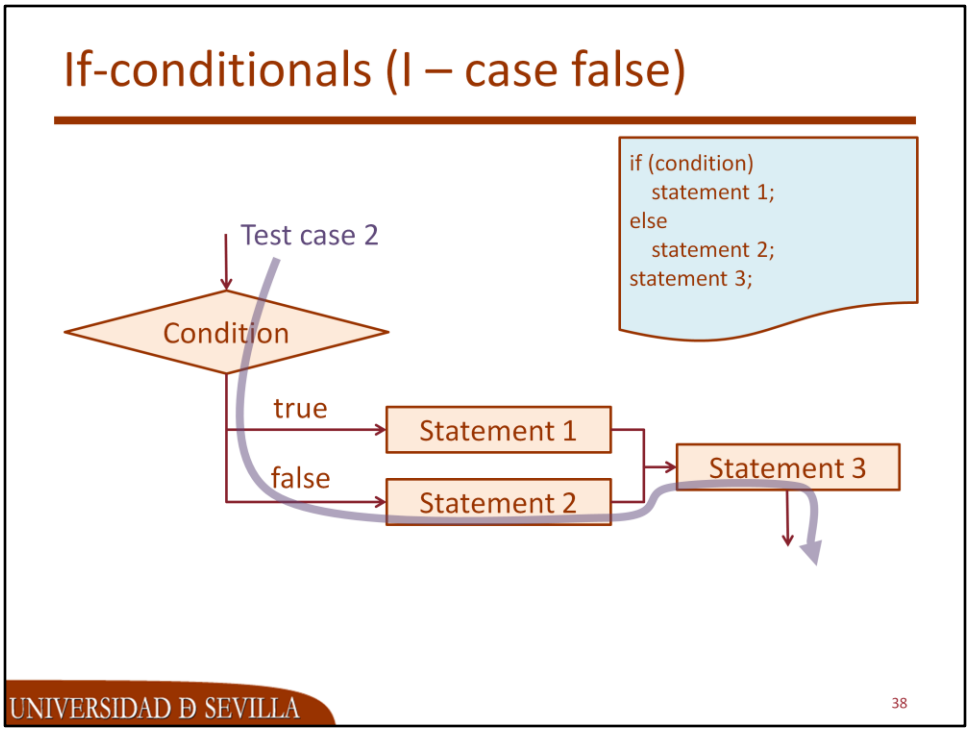


Let's start with the if-conditionals. This slide shows a template of a typical such statement, namely:

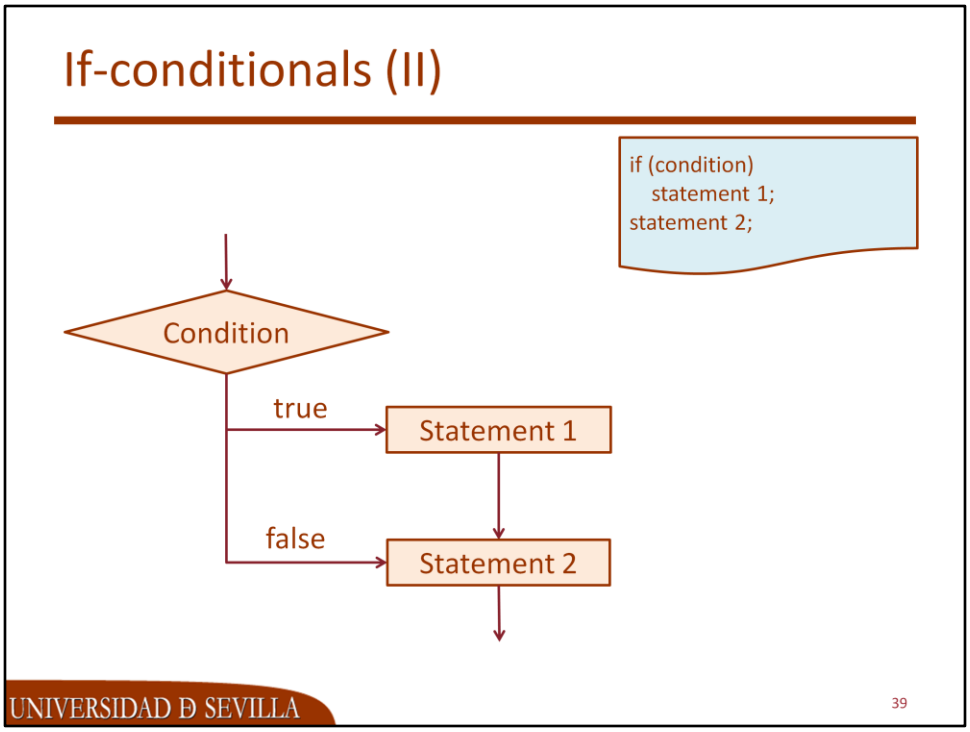
```
if (condition)
   statement 1;
else
   statement 2;
statement 3;
```



We have to devise a test case in which the condition is satisfied and then statement 1 is executed.

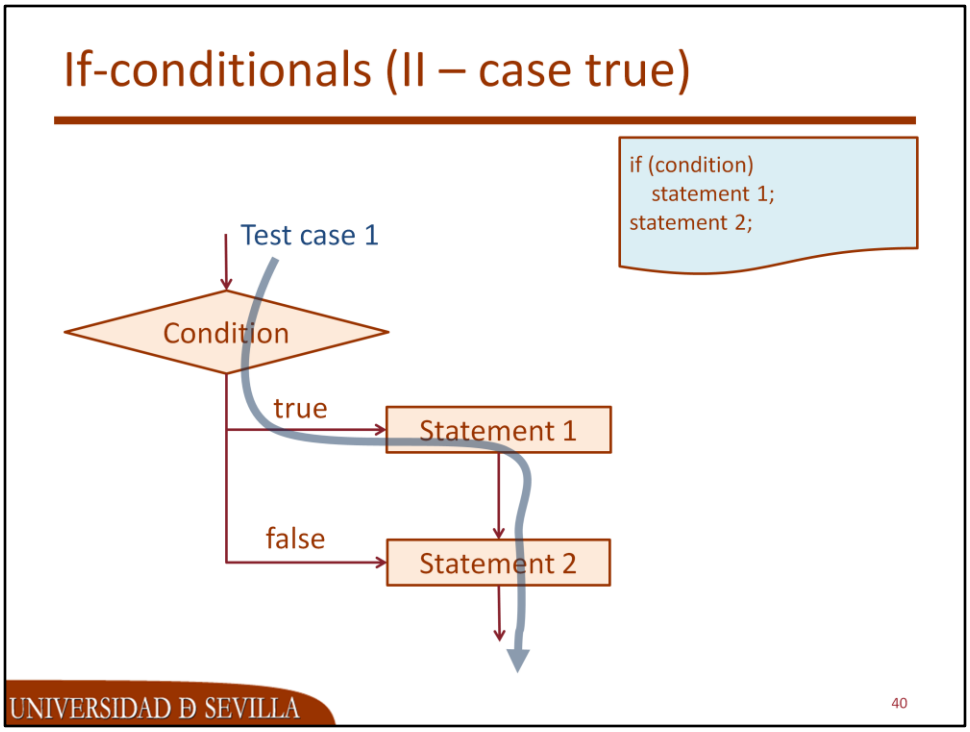


And we also have to devise a test case in which the condition isn't satisfied and then statement 2 is executed.

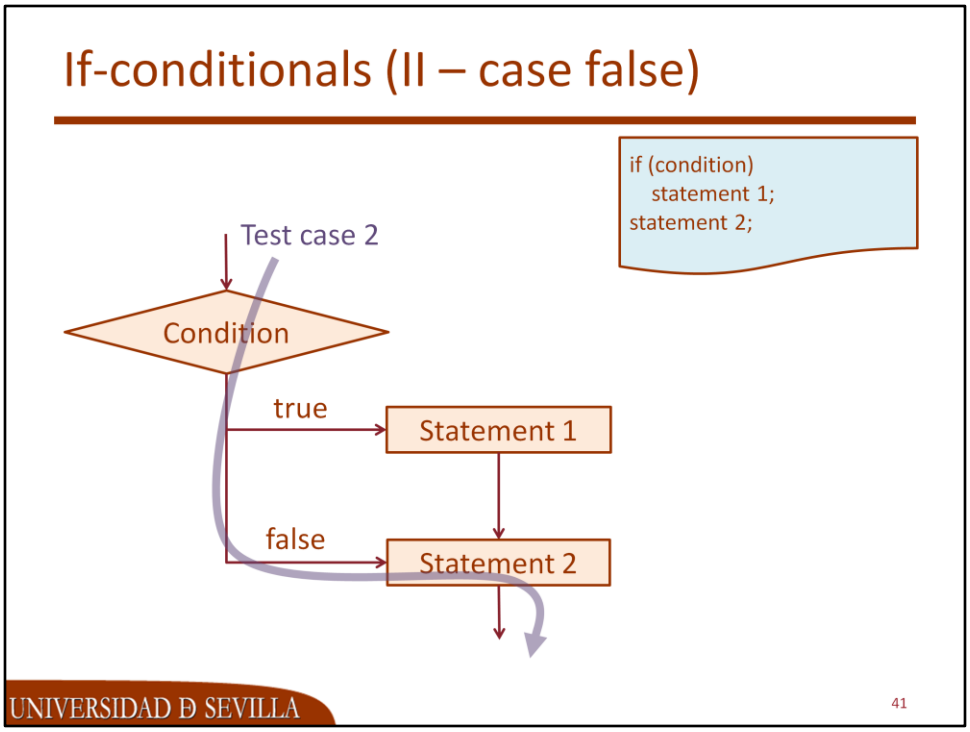


This slide shows another kind of if-conditional in which there's no else branch, namely:

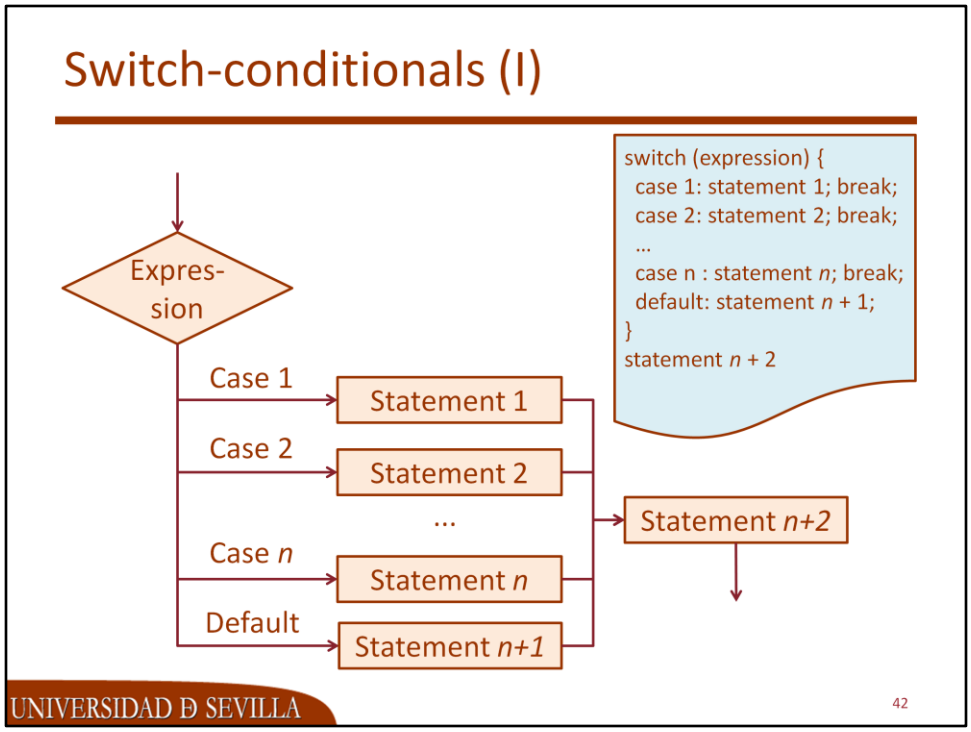
```
if (condition)
   statement 1;
statement 2;
```



In this case, you have to design a test case in which the condition holds and then statement 1 is executed.

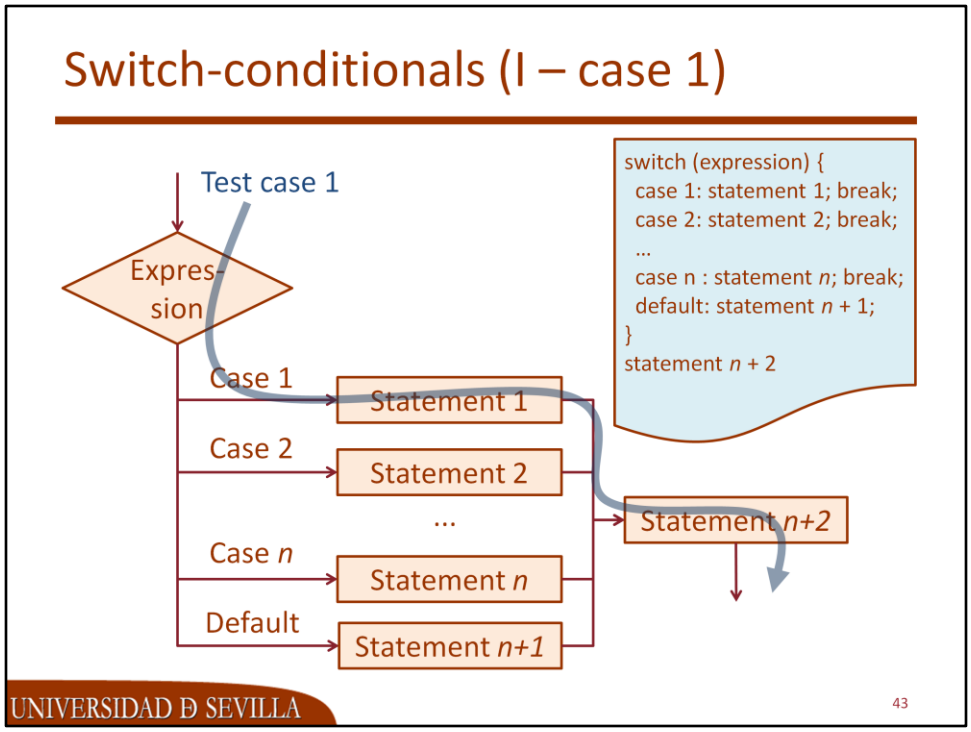


And another test case in which the condition doesn't hold and then statement 1 is not executed.

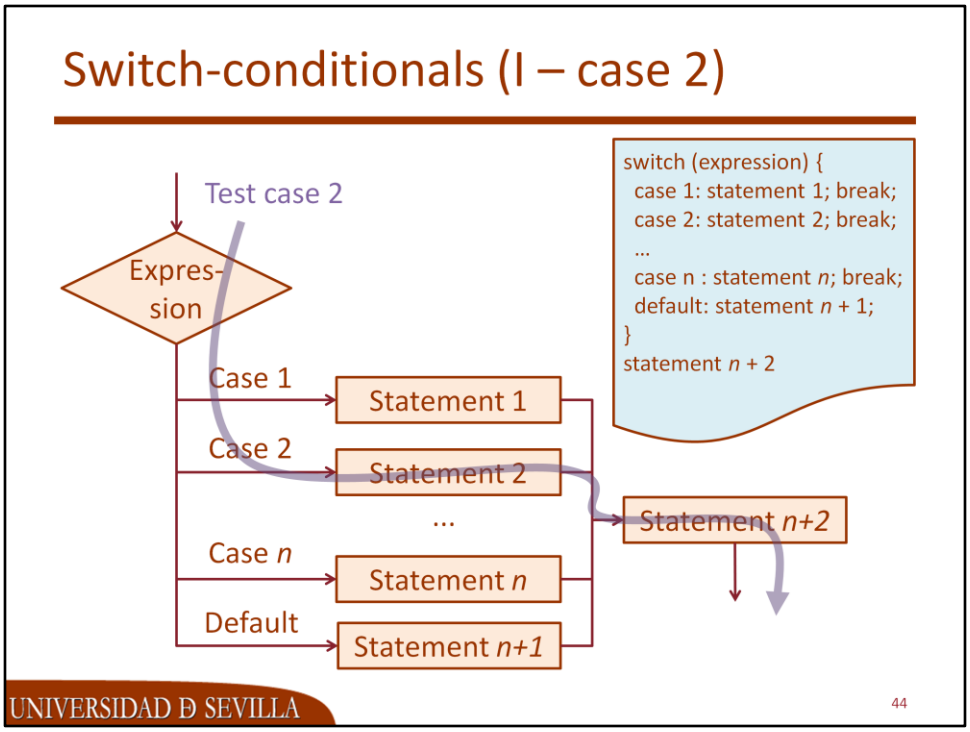


Switch conditionals are similar, namely:

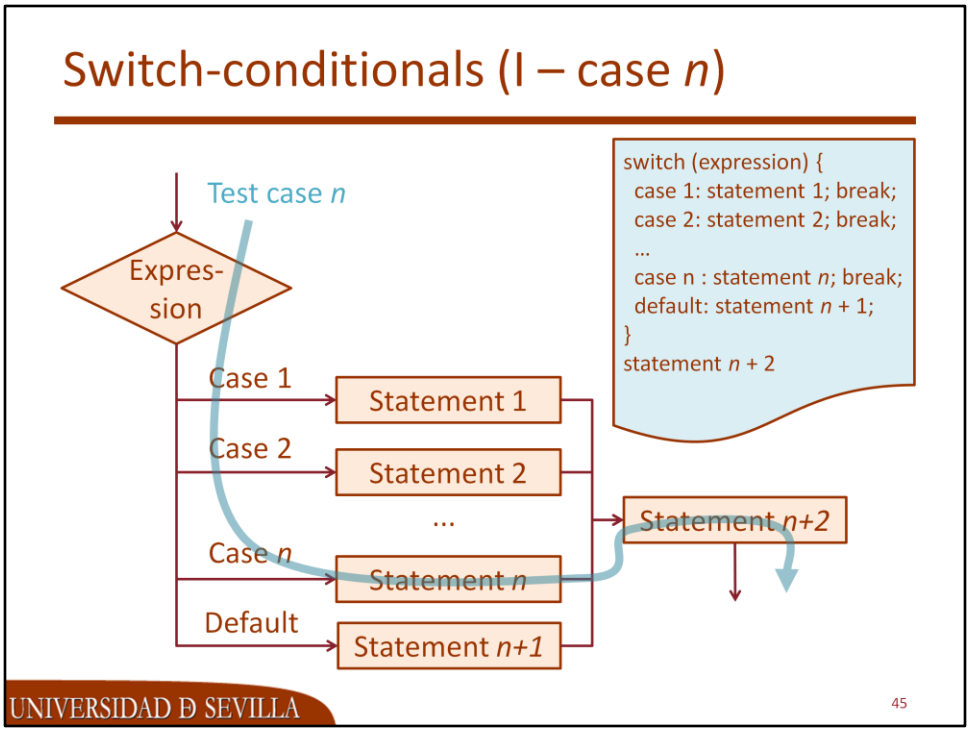
```
switch (expression) {
  case 1: statement 1; break;
  case 2: statement 2; break;
  …
  case n : statement n; break;
  default: statement n + 1;
}
statement n + 2;
```



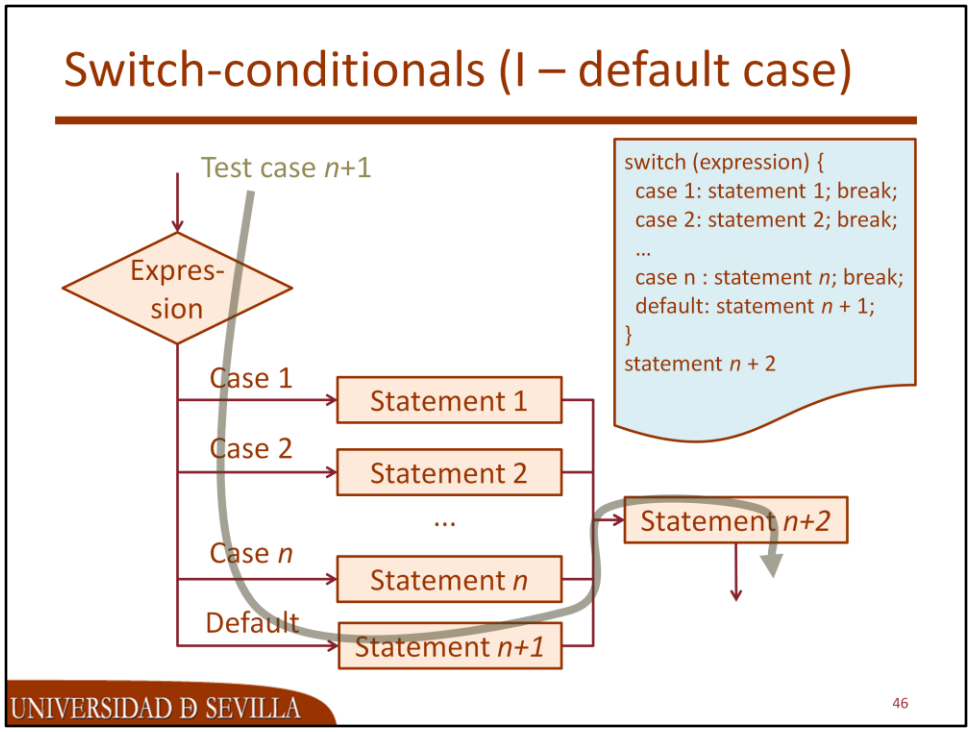
In this case, you must first design a test case in which the statement that corresponds to the first case is executed.



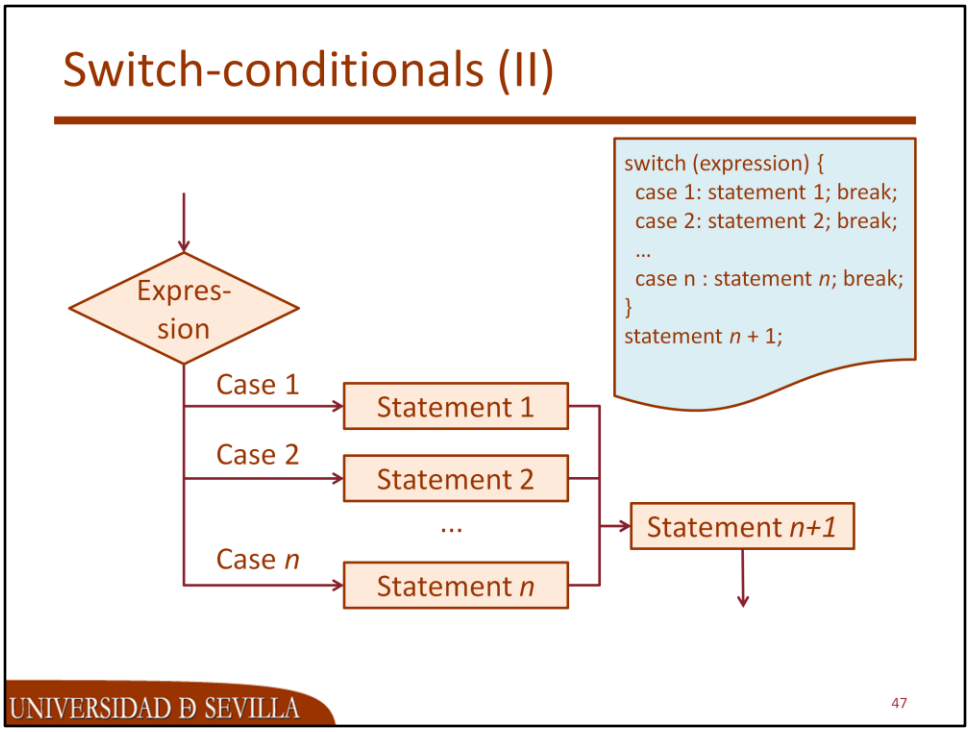
Then another in which the second case is executed.



And so on, till the *n*-th case is executed.

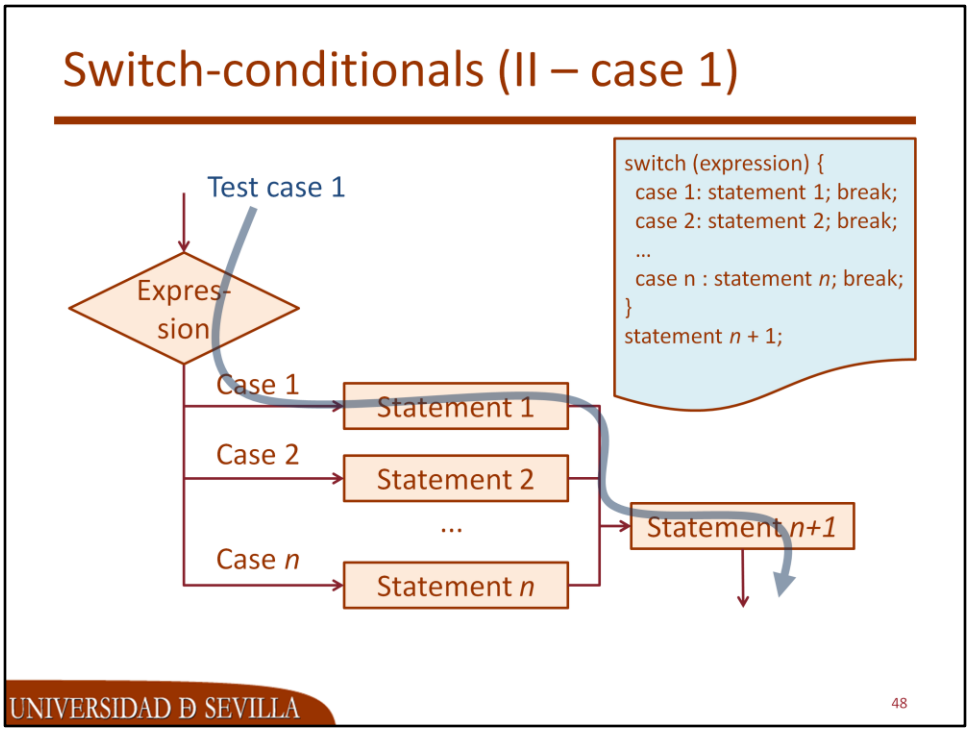


Don’t forget an additional test case in which the expression does not match any of the cases, and then the default case’s executed.

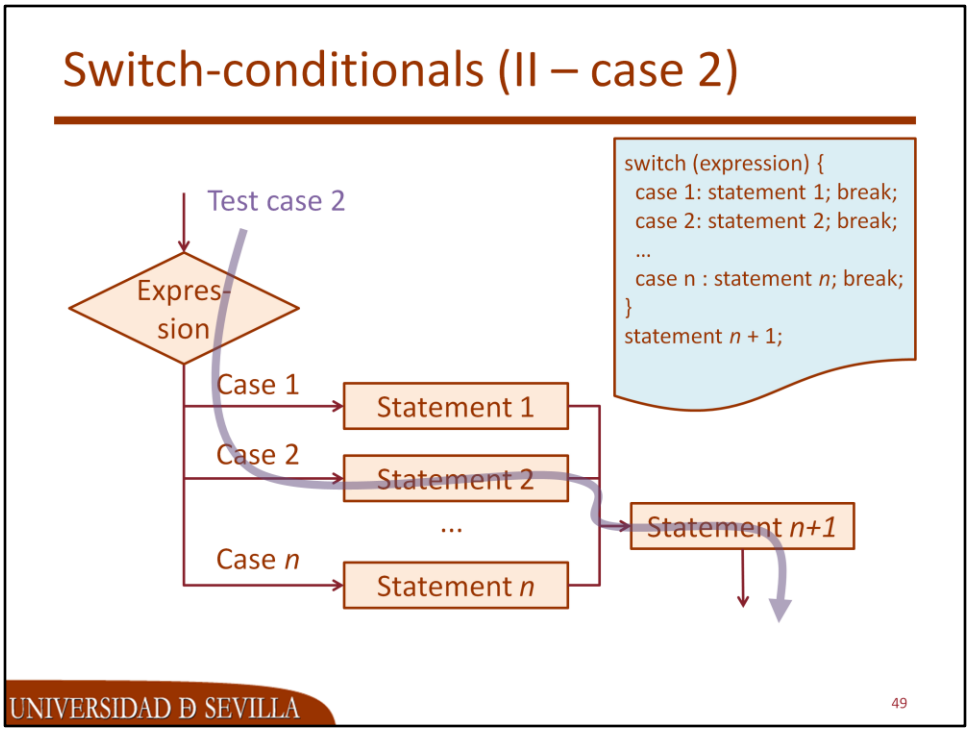


Note that there's an additional kind of switch statement in which there's no default case, namely:

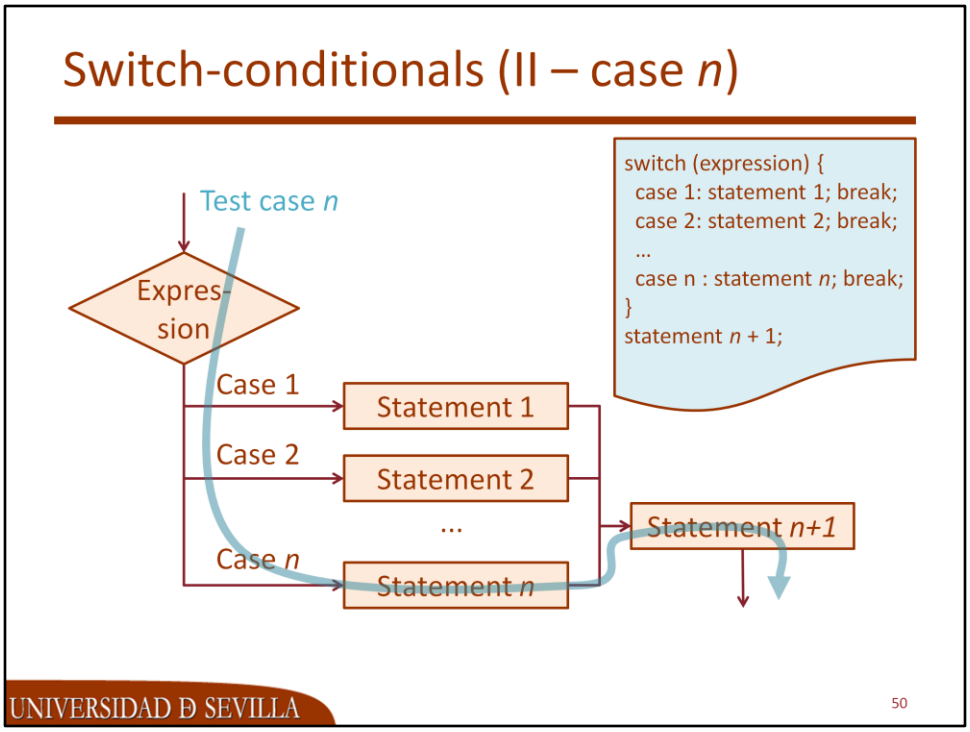
```
switch (expression) {
  case 1: statement 1; break;
  case 2: statement 2; break;
  …
  case n : statement n;
}
```



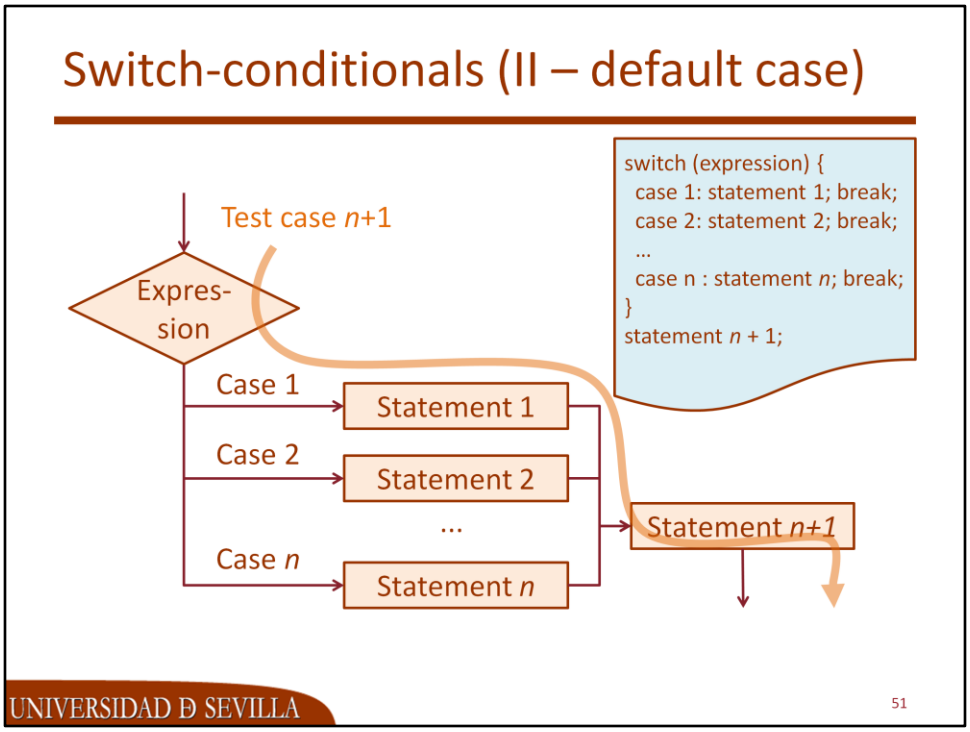
In this case, you have to design test cases in which the first, …


Switch-conditionals (II – case 2)
Test case 2
Expres-
sion
Case 1
Case 2
Case n
Statement 1
Statement 2
...
Statement n
Statement n+1
switch (expression) {
case 1: statement 1; break;
case 2: statement 2; break;
...
case n : statement n; break;
}
statement n + 1;
UNIVERSIDAD D SEVILLA
49

… the second …



… and the *n*-th cases are executed.



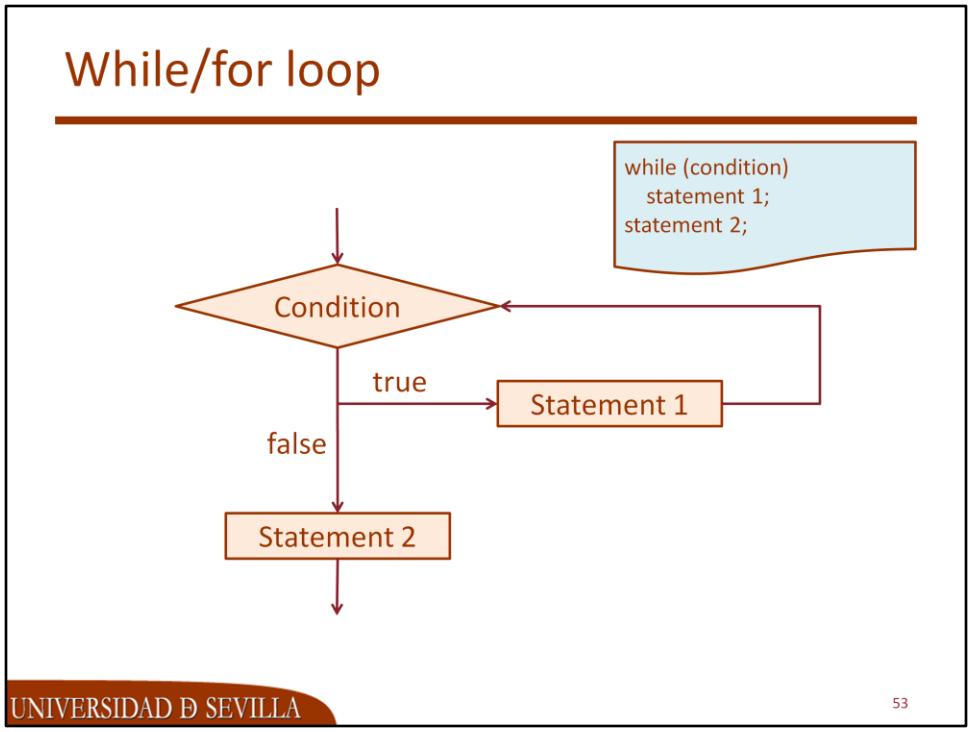
But you must not forget to design a test case in which the expression does not match any of the cases, and then the execution jumps to the statement right after the switch statement.

Regarding loops

UNIVERSIDAD Ð SEVILLA

52

The theory behind loops is similar in spirit: your test cases must ensure that every loop executes zero, one, and several times.

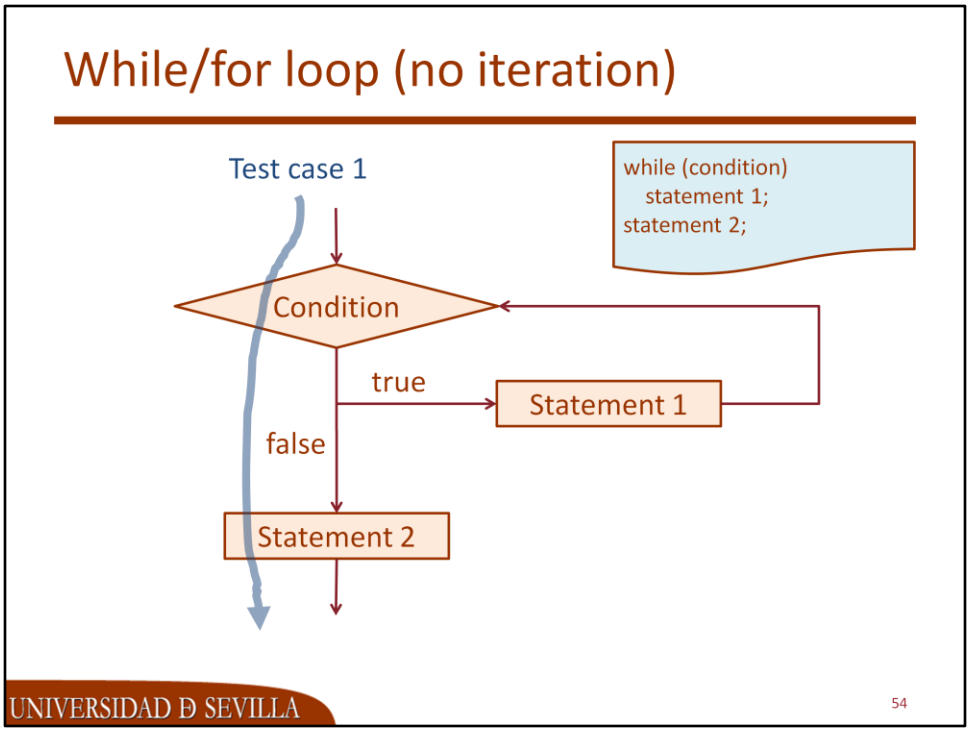


Let's start with a typical while or for statement, namely:

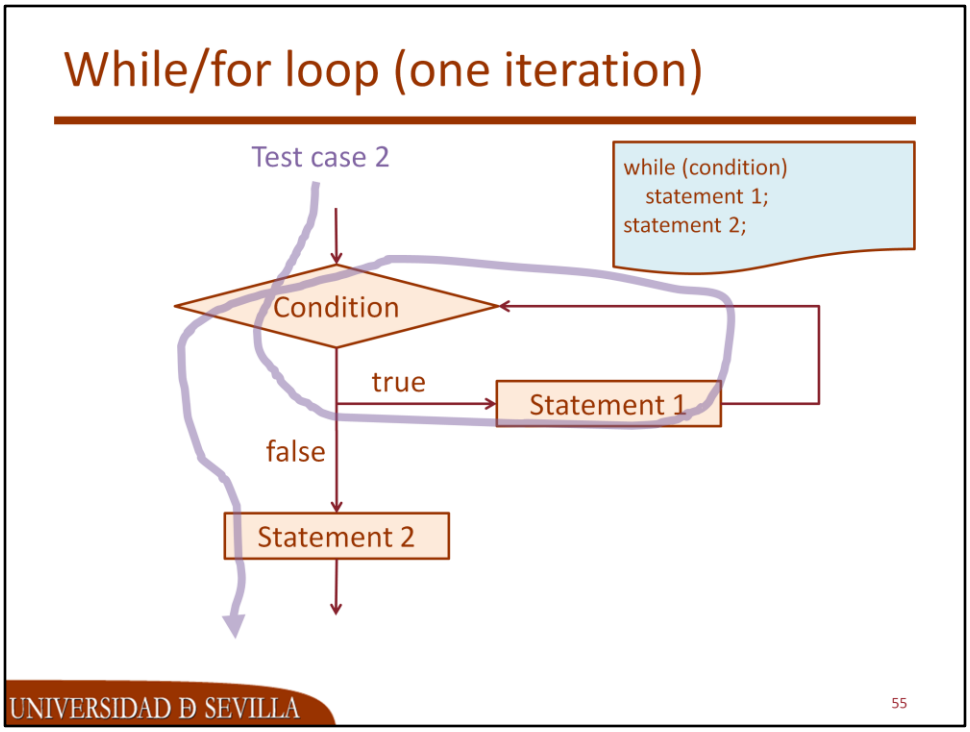
```
while (condition)
   statement 1;
statement 2;
```

or

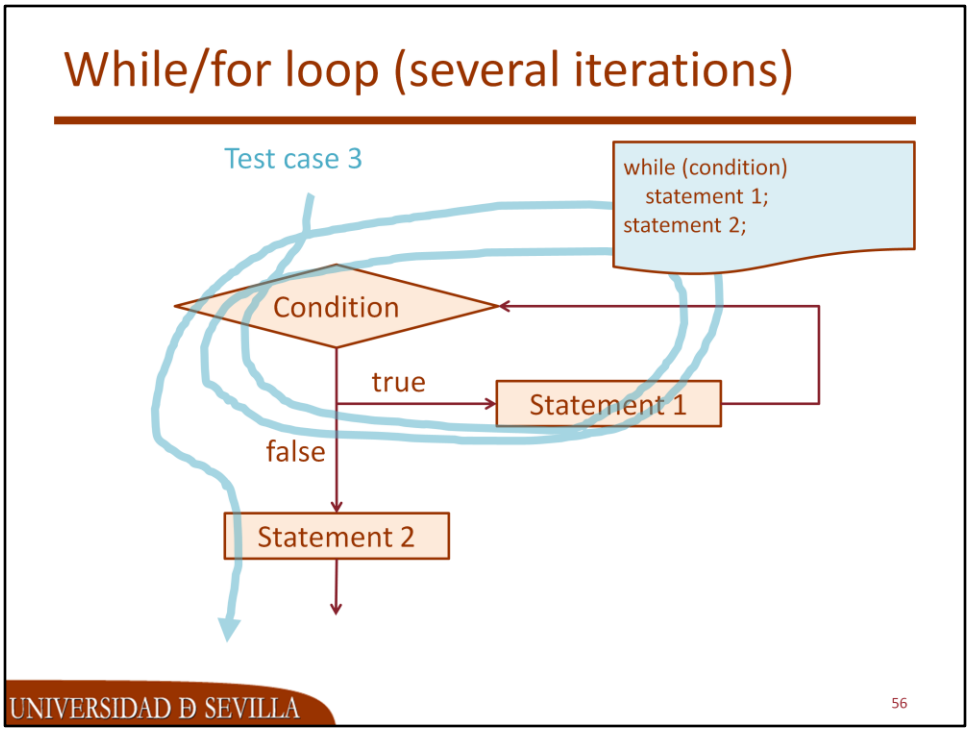
```
for (initialisation; condition; increment)
   statement 1;
statement 2;
```



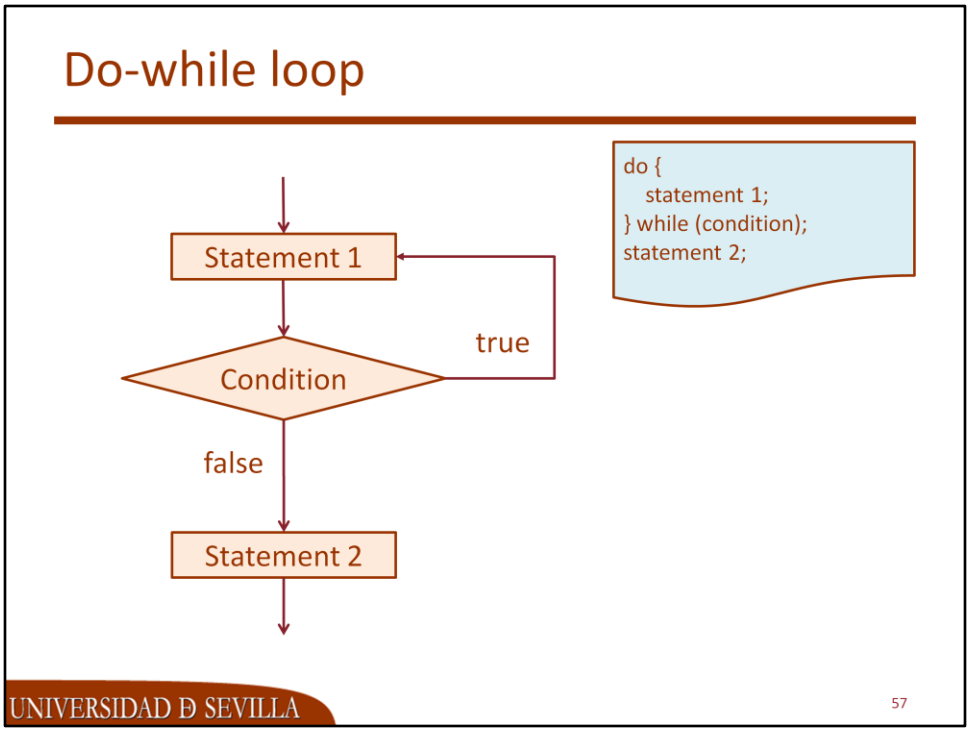
You must design a test case in which the condition doesn't hold and, consequently, the statement within the loop's never executed.



Design another test case in which the condition's satisfied once so that the statement executes only once.

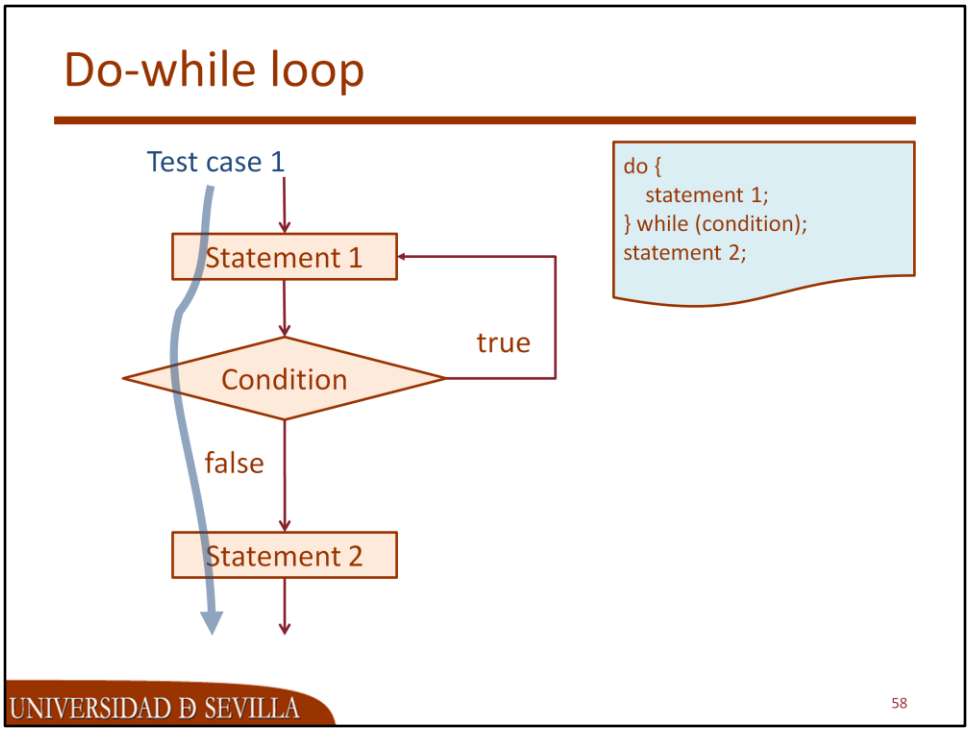


And, finally, design an additional test case in which the condition's satisfied several times and then the statement within the loop's executed several times. How many times is several times? Three, ten, twenty… as many as possible as long as they actually make sense.

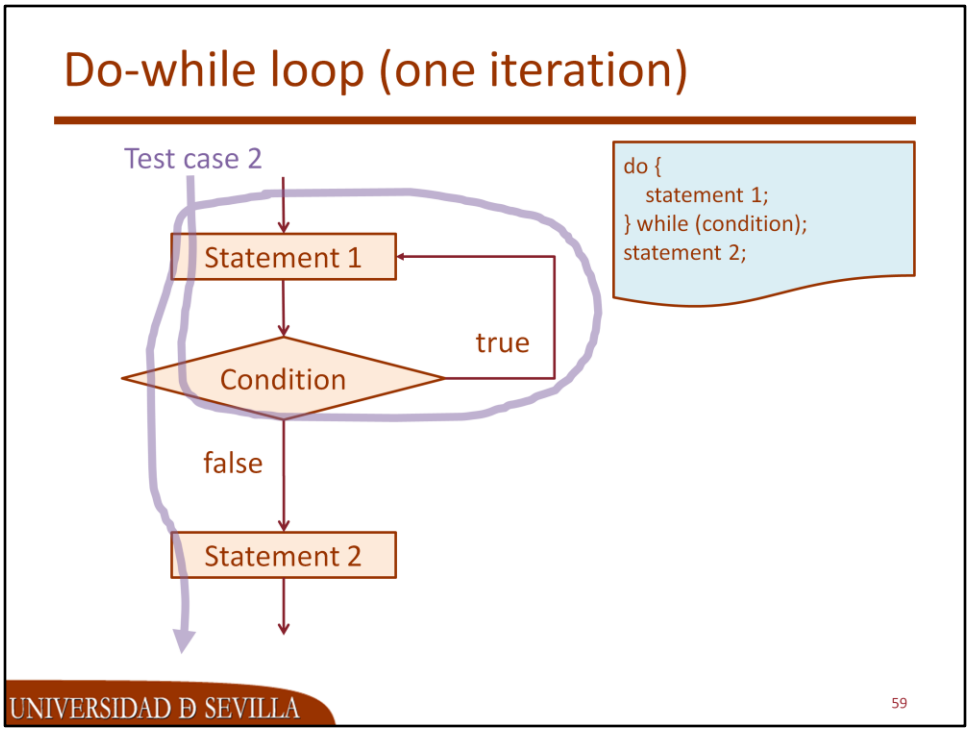


The do-while loop's very similar:

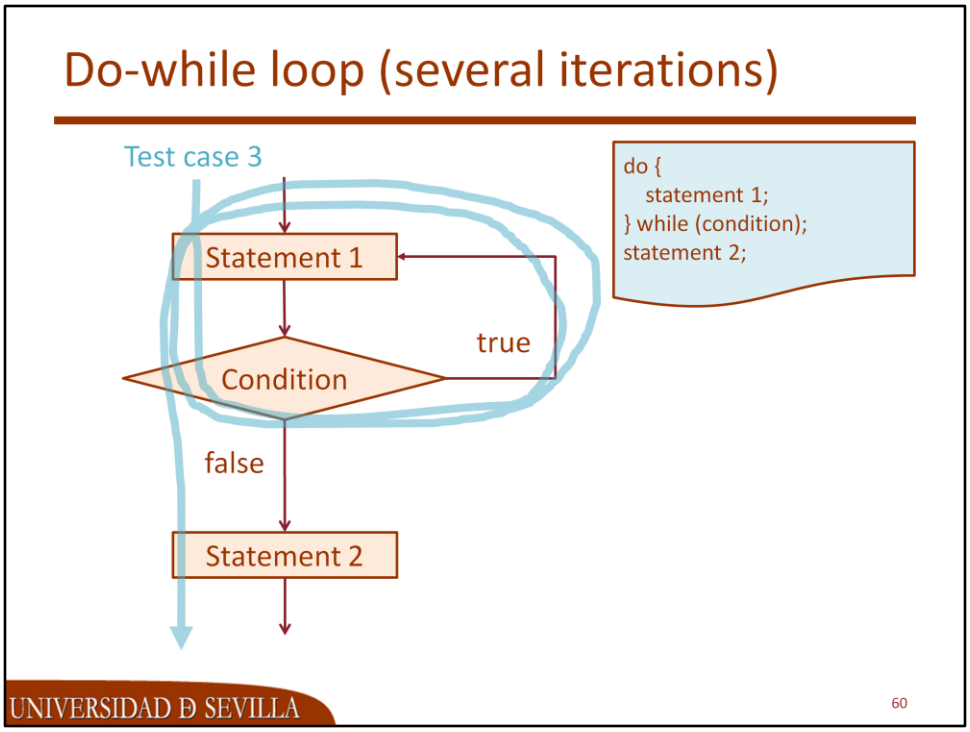
```
do {
   statement 1;
} while (condition);
statement 2;
```



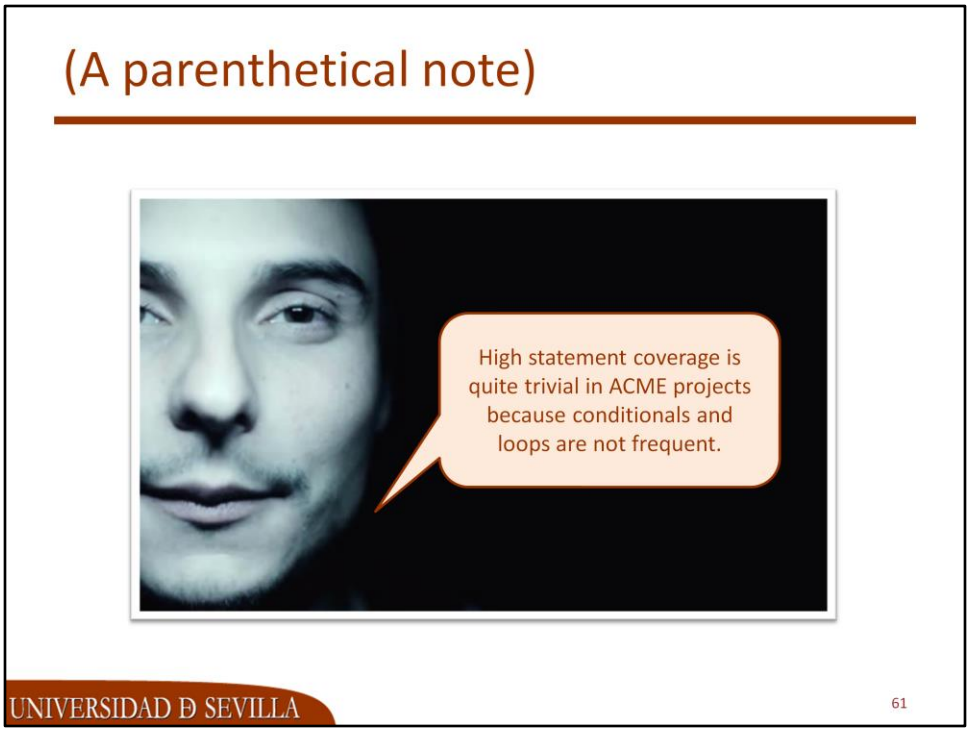
You must design a test case in which the condition does not hold, so that statement 1 is executed once and then the control's passed onto the next statement.



Design another test case in which the condition holds only once, so that statement 1 is actually executed twice.

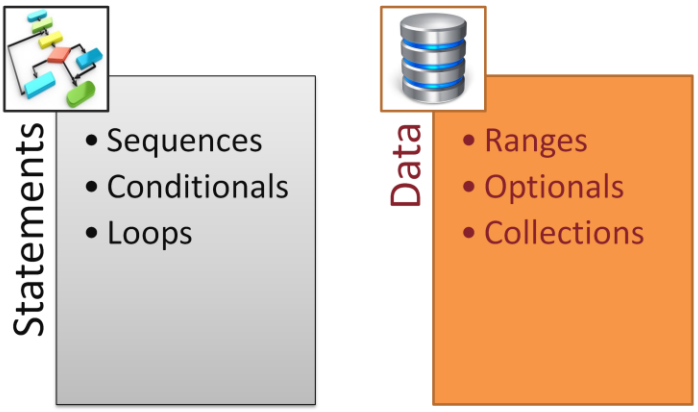


And, finally, design a test case in which the condition's satisfied several times and statement 1 is then executed several times.



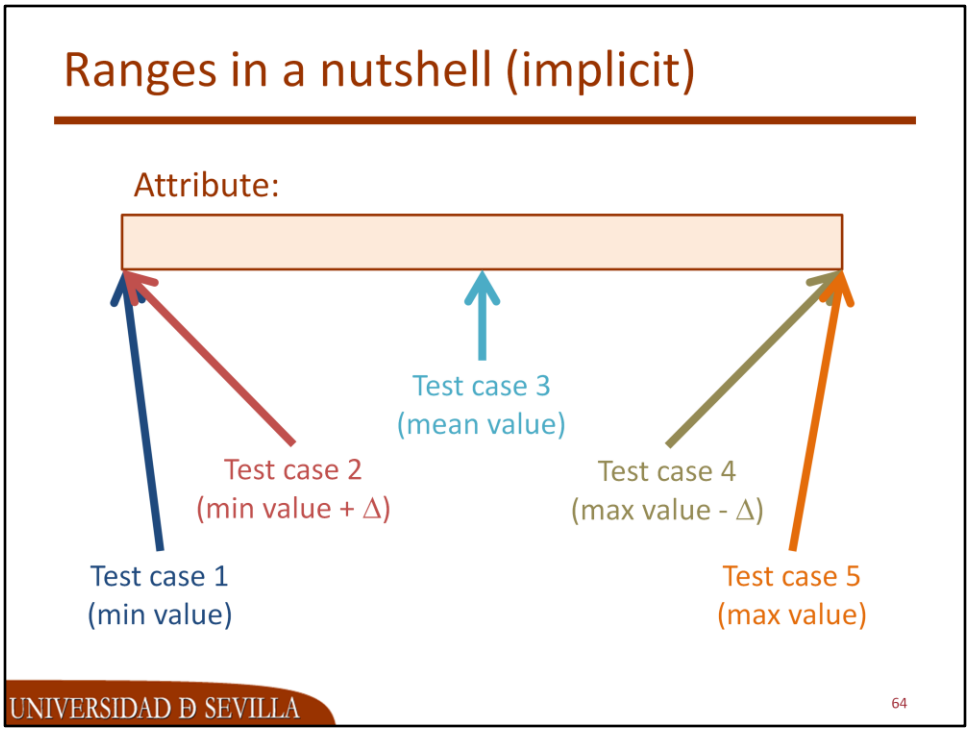
Before concluding the section on the statement strategy to maximise coverage, we'd like to make a parenthetical note. You might well argue that you haven't used a single conditional or loop in your services thus far. That's ok. The reason is that typical Acme projects have many functional requirements that help you get as trained as possible, but they are very simple and can usually be implemented using a sequence of statements. Coverage's trivial in such cases, but real-world projects have functional requirements that are typically a lot more complex and then require more complex algorithms. In such cases, following the strategy that we've described in this section is a must in order to maximise the coverage of your test suites.

Covering strategies



Let's now explore how to make testing data as assorted as possible. We'll explore ranges, optionals, and collections.

Let's start with ranges. Note that testing data are entities with attributes. In turn, attributes have implicit or explicit ranges that set boundaries to their values. Your test cases must ensure that their boundary values are covered, which means that there must be at least a test case in which every attribute gets a boundary value, if possible.

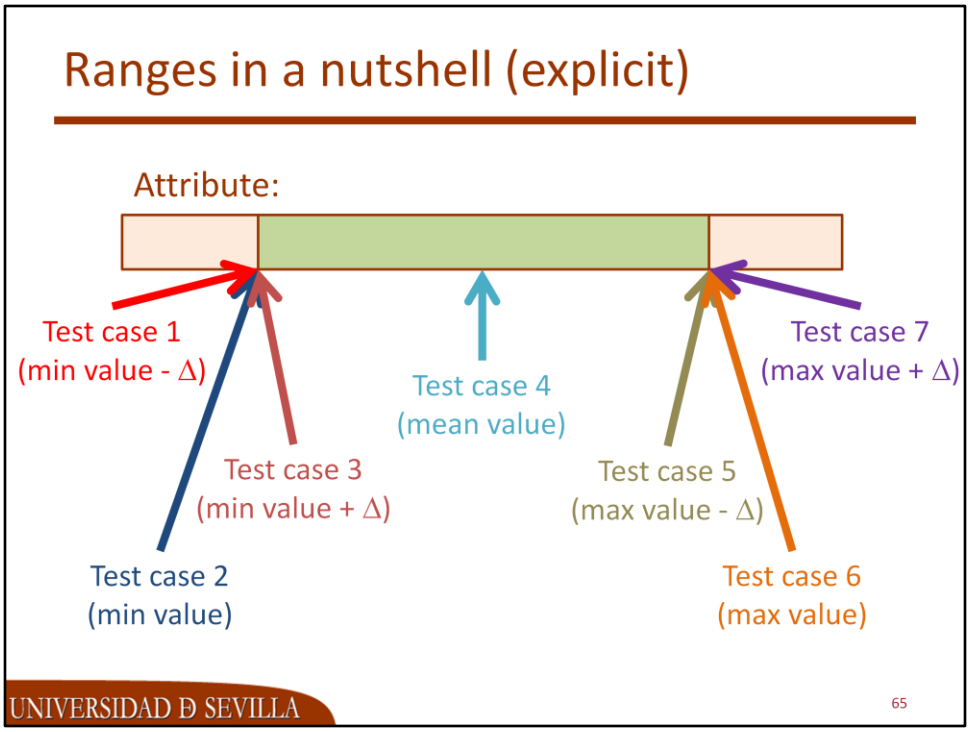


Let's start with the implicit ranges. Every numeric attribute has an implicit range that depends on its type. To maximise coverage, you must design the following test cases:

- A test case in which the attribute has the minimum value;
- Another in which it has the value that is immediately after minimum value;
- Another on in which it's assigned the mean value;
- One more test case in which it gets the value immediately before the maximum value;
- And a final test case in which it gets the maximum value.

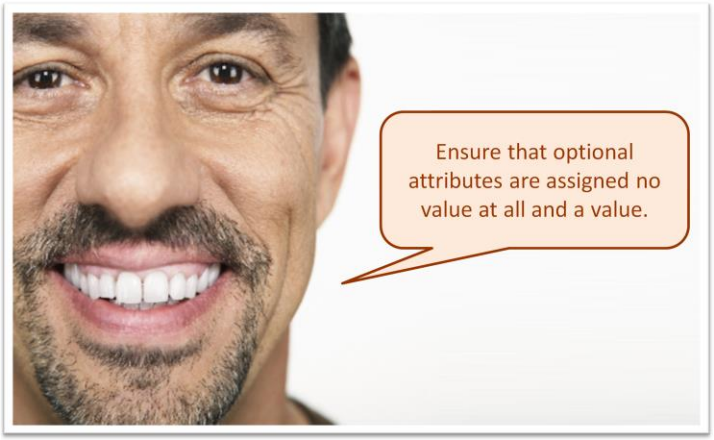
The minimum value, the maximum value, and the values immediately before or after are referred to as boundary values and they must be covered because developers typically make mistakes around them. In the slide, we've used "Δ" to mean the minimum value that can be added to a number to get the next one or the previous one.

NOTE: Visit https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html to learn more about the ranges of primitive types in Java.



Some numeric attributes have explicit range constraints. For instance an attribute like "expirationYear" is likely to have a range constraint of the form "@Range(2000, 2100)", so that credit cards whose expiration year's before the year 2000 or after the year 2100 are considered invalid. This constraints the implicit range and changes the implicit boundary values. Note that the test cases include the previous ones plus two additional cases to check the values out of the ranges.

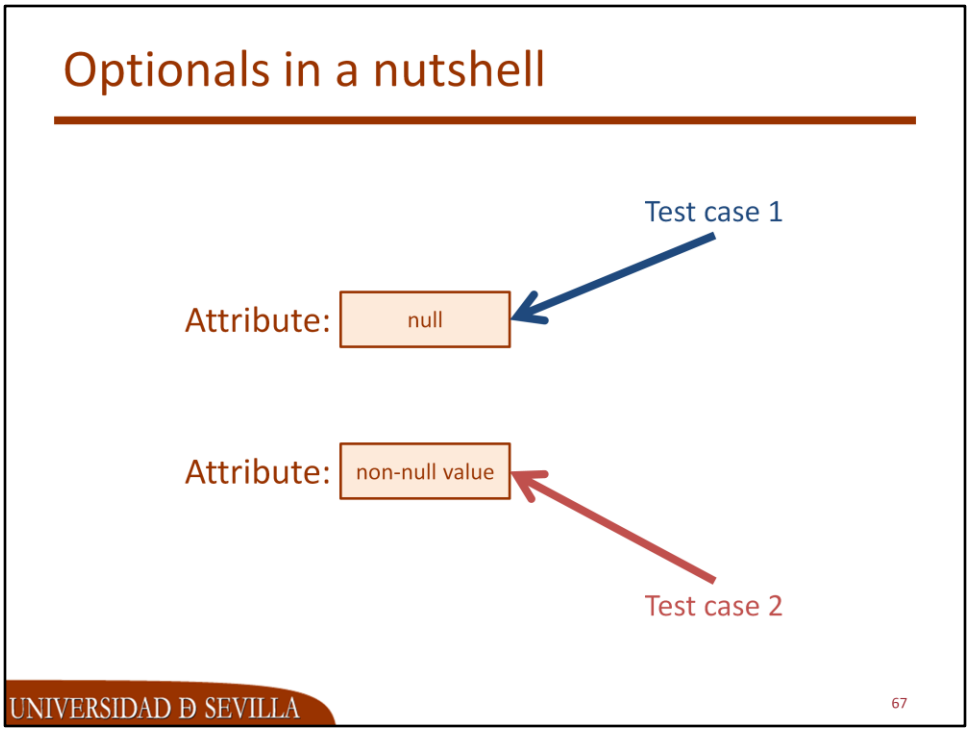
Regarding optionals



UNIVERSIDAD Đ SEVILLA

66

Another issue's regarding attributes that are optional. There must be a test case in which they are assigned no value at all and a test case in which they're assigned a value.



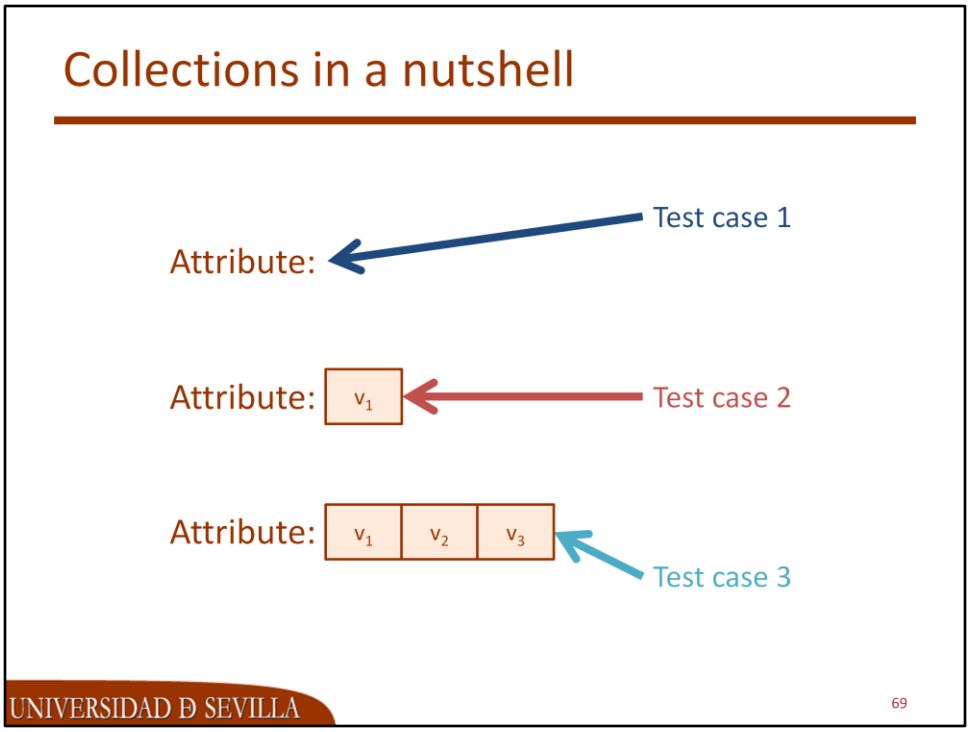
It's very simple: design a test case in which your optional attribute gets value "null" and another one in which it gets a non-null value.

Regarding collections

UNIVERSIDAD Ð SEVILLA

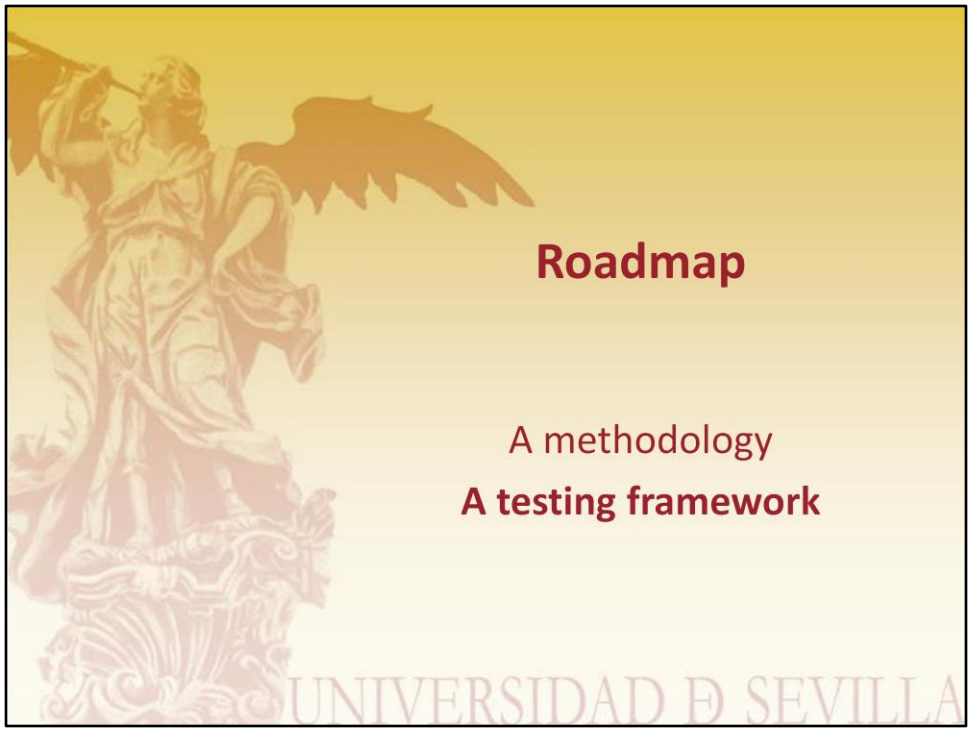
68

Finally, there are attributes that hold collections of values. Your test cases must ensure that your testing data includes cases in which they are empty collections, cases in which they have one element, and cases in which they have several elements.



Simple, right? As usual, there's a problem regarding how many elements are several elements. As usual, there's not a simple answer. There are cases in which several amounts to three and cases in which several amounts to 100; just try to guess what the average number of elements is and use that guessed size.

Before concluding this section, we'd like to comment a little on something that is very important: note that the previous ideas may very easily lead to too many combinations. For instance, think of a domain object with attribute "a1" of type "Integer" with constraint "@Range(0, 999)", attribute "a2" of type "Collection<Stuff>" with constraint "@NotEmpty". Let's try to enumerate the test cases: (a1 = null, a2 = []), (a1 = -1, a2 = []), (a1 = 0, a2 = []), (a1 = 1, a2 = []), (a1 = 499, a2 = []), (a1 = 998, a2 = []), (a1 = 999, a2 = []), (a1 = 1000, a2 = []), (a1 = null, a2 = [a]), (a1 = -1, a2 = [a]), (a1 = 0, a2 = [a]), (a1 = 1, a2 = [a]), (a1 = 499, a2 = [a]), (a1 = 998, a2 = [a]), (a1 = 999, a2 = [a]), (a1 = 1000, a2 = [a]), (a1 = null, a2 = [a, b, c]), (a1 = -1, a2 = [a, b, c]), (a1 = 0, a2 = [a, b, c]), (a1 = 1, a2 = [a, b, c]), (a1 = 499, a2 = [a, b, c]), (a1 = 998, a2 = [a, b, c]), (a1 = 999, a2 = [a, b, c]), and (a1 = 1000, a2 = [a, b, c]). Ok, that's not a myriad test cases, but they are many; and the number of test cases increases as the number of attributes and constraints increases, which makes it difficult or impossible in practice to enumerate them all. You must implement as many as possible in order to maximise your data coverage, that's all.



That's all regarding our methodology. It's now time to report on the testing framework that we're using to implement it.

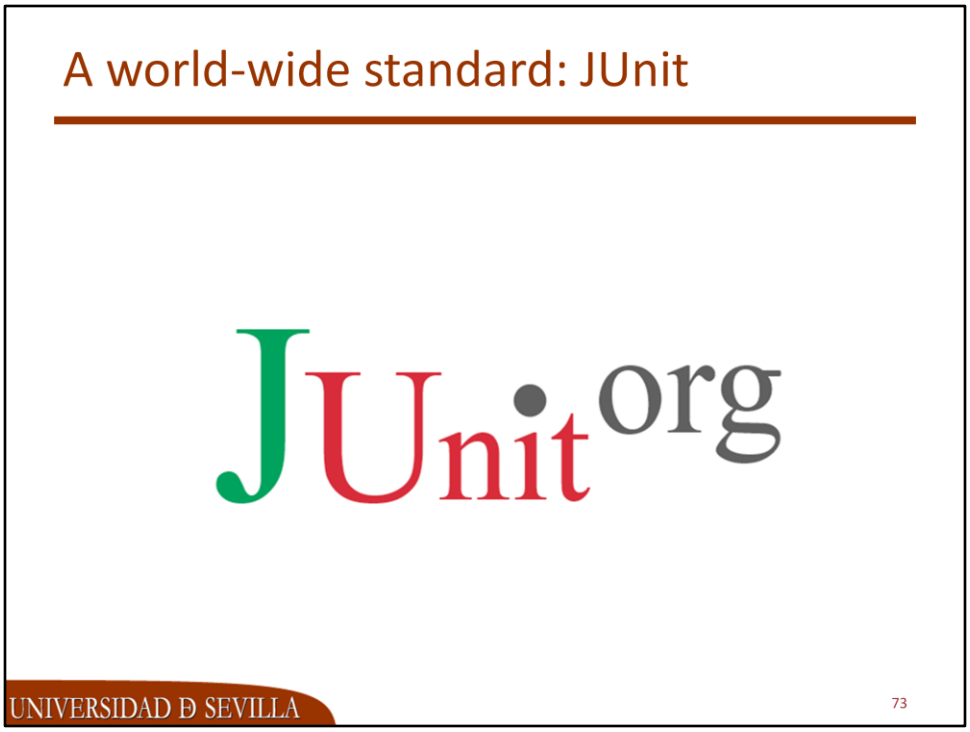
What's a testing framework?

It's a framework that helps developers write and run their test cases

UNIVERSIDAD Đ SEVILLA

72

As usual, we start with a definition. In the first lecture, we introduced the idea of framework: it's a collection of classes that constitute the common core of a number of systems and you only have to provide the classes that are specific to a specific system. A testing framework's then a framework that helps developers write and run their test cases; you only have to provide them, and the framework provides a variety of utilities to help you.

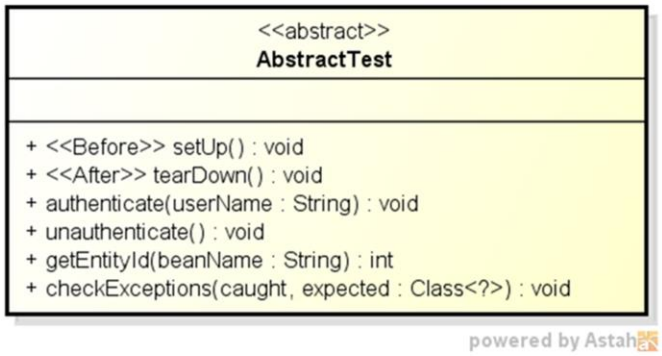


In this subject, we're going to use a framework that is based on JUnit, which is an industrial standard. It's quite a simple framework, which means that it's really easy to command, but it's very helpful.

This is what we're going to learn about the testing framework: first, we'll report on a class called "AbstractTest"; then we'll teach on how to write a test class and a test case; finally, we'll delve into the details of running a test class or a test suite.

Let's start with the "AbstractTest" class.

The `AbstractTest` class



UNIVERSIDAD Ð SEVILLA

This class was introduced a few lessons ago, when we started to work with JUnit as a means to implement simple checks for your repositories and services. We assume that you're familiar with it, so we'll just scratch its surface.

The fixture setup

Build your fixture by overriding

```
@Before
void setup();
```

Override it only if you need something that Spring does not provide by default.

UNIVERSIDAD Ð SEVILLA

77

The “setUp” method’s introduced by means of the “@Before” annotation. Override it in your test cases only if you actually need a special fixture to set your SUT up. Typically, this method needs not be overridden since Spring cares of setting up the fixtures required to get your repositories, services, controllers, converters, and many other components working.

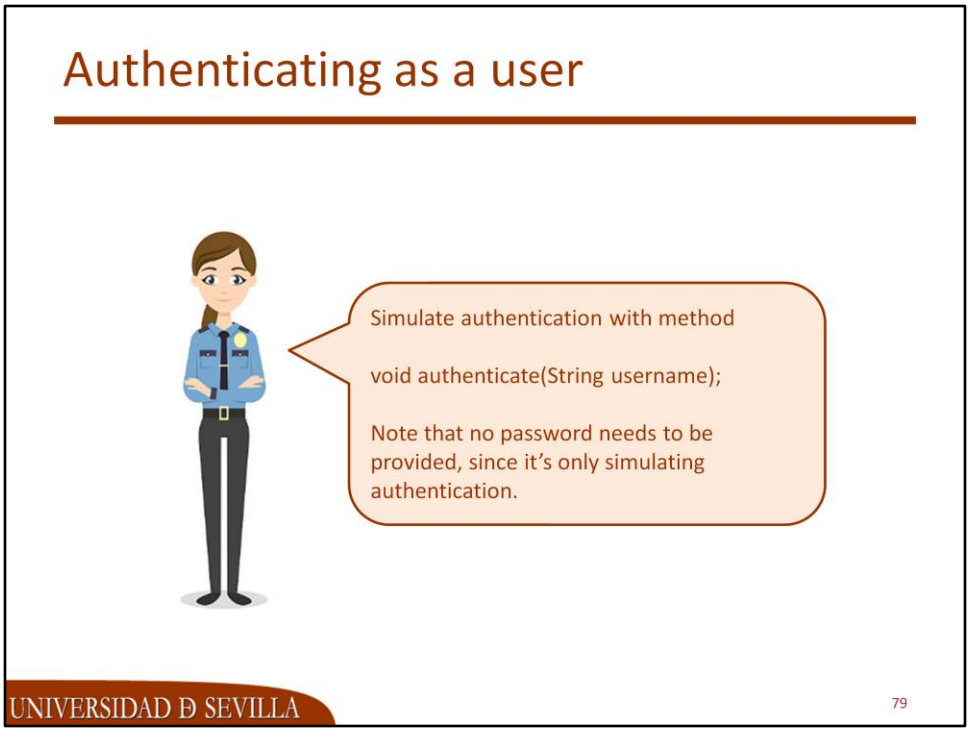
The fixture teardown

Tear your fixture down by overriding

@After
void tearDown();

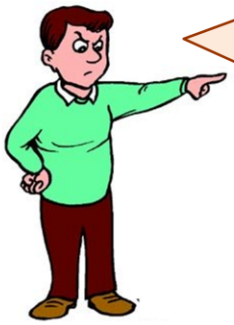
Override it only if you need something that Spring does not provide by default.

The “tearDown” method’s introduced by means of the “@After” annotation. By default, it does nothing at all but you may override it in your test classes in order to properly tear your own fixtures down. Note that it’s not necessary to override it unless you also override the “setUp” method.



By default, a test case is executed in a context in which the user who runs the code is an anonymous user. The "authenticate" method allows you to simulate that a given user has authenticated to the system, so that when your services request the principal they get the appropriate user. Please, note that no password is required, since the method just simulates that a user has authenticated.

Unauthenticating



Unauthenticate with either method

void unauthenticate();

or call authenticate with a “null” username.

This is the complementary method. It allows to log out from the application so that the current user is restored to an anonymous user. Note that invoking this method or the “authenticate” method using a “null” username is the same.

Getting entity ids

Find the id of an object with method

int getEntityId(String beanName);

This makes your tests cases more robust!

This is a very handy method: given the name of a bean in your "PopulateDatabase.xml" file, it returns its corresponding identifier in the database. It then helps make your tests more robust to changes to the "PopulateDatabase.xml" file.

Checking exceptions

Check exceptions in negative test cases with

```
void checkExceptions(
        Class<?> expected,
        Class<?> caught);
```

Both “expected” ad “caught” must be Throwable classes.

This is a method that is intended to check if two classes, which are assumed to represent two exceptions, match or not. If the expected exception is the same as the exception caught, then nothing happens; otherwise, a new runtime exception is thrown to indicate that they do not match. This method is typically used in negative test cases to check that the exception that was thrown by a piece of code is the exact exception that was expected.

Let's now report on how to write a test class and test methods.

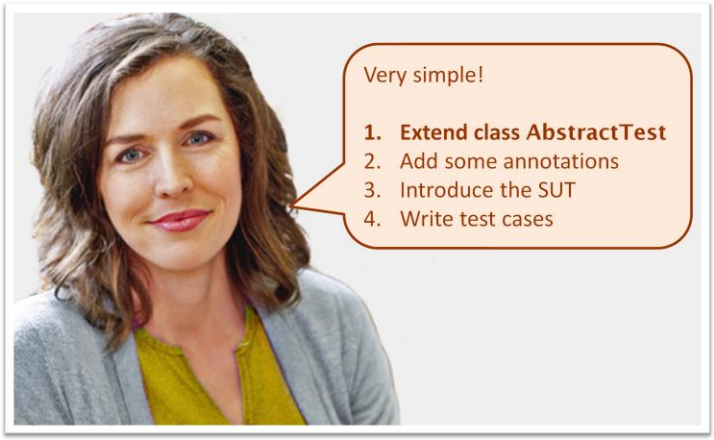
Follow this guideline

UNIVERSIDAD Ð SEVILLA

84

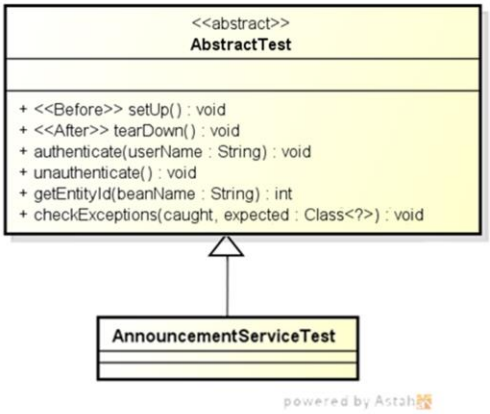
Writing test classes and method is relatively easy. Just follow this guideline.

Follow this guideline



First, extend class “AbstractClass”.

Extend the `AbstractTest` class



Your test classes must extend class “AbstractTest” so that they inherit its methods. This allows to setup and tear down a complex fixture, to authenticate and unauthenticate in your test cases, to get the identifiers of your entities in the database, and to check exceptions in negative test cases.

Your test class… thus far

```
...
public class AnnouncementServiceTest extends AbstractTest {
                ...
}
```


This is how your test class must look… thus far.

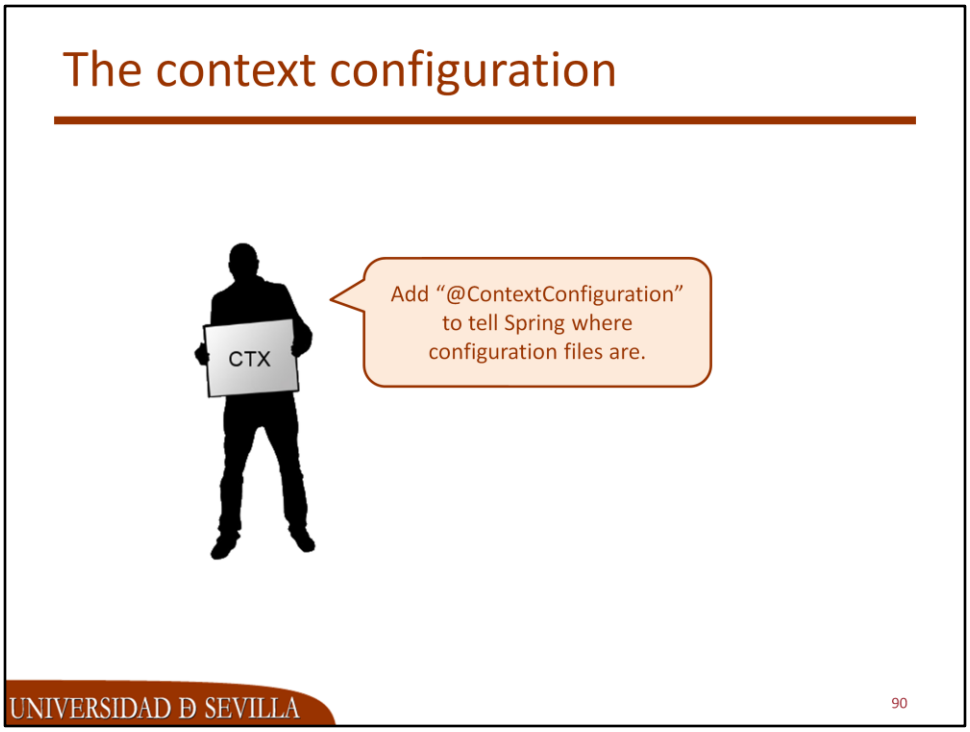
Follow this guideline

UNIVERSIDAD D SEVILLA

88

Now, add some annotations.

The first one is a "@RunWith" annotation, which allows to integrate JUnit with other technologies. Simply put, this annotation allows to specify the name of a class that is used to instrument your test class before JUnit runs the test cases that it provides. Please, recall that Spring provides many useful annotations, including "@Repository", "@Service", "@Controller", or "@Autowired"; for these annotations to work, Spring must instrument your classes, i.e., it must change the code of your classes so as to implement the annotations. Since our test cases use Spring, we need to specify Spring's JUnit runner, which is class "SpringJUnit4ClassRunner".



The next annotation is "@ContextConfiguration", which tells Spring where the configuration files that it needs are stored. In our framework, we provide a configuration file called "spring/junit.xml" that links all of the configuration files required. Typically, this annotation looks as follows:

```
@ContextConfiguration(locations={"classpath:spring/junit.xml"})
```

The transactional support

Add “@Transactional” to activate Spring’s transaction support.

UNIVERSIDAD D SEVILLA

91

The final annotation’s “@Transactional”, which instructs Spring to activate its transactional support.

Your test class… thus far

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:spring/junit.xml"})
@Transactional
public class AnnouncementServiceTest extends AbstractTest {
                ...
}
```


This is how your test class must look… thus far.

Follow this guideline

UNIVERSIDAD D SEVILLA

93

Now, it's time to introduce the SUT.

A familiar pattern, right?

UNIVERSIDAD D SEVILLA

94

Typically, the SUT is a service or a repository, which is not difficult to introduce thanks to the "@Autowired" annotation. Please, note that if you need to test the "AnnouncementService" class, for instance, you just need to introduce it as a regular attribute in your test class; just don't forget to prepend it with an "@Autowired" annotation so that Spring instantiates the service and injects its unique instance into that attribute.

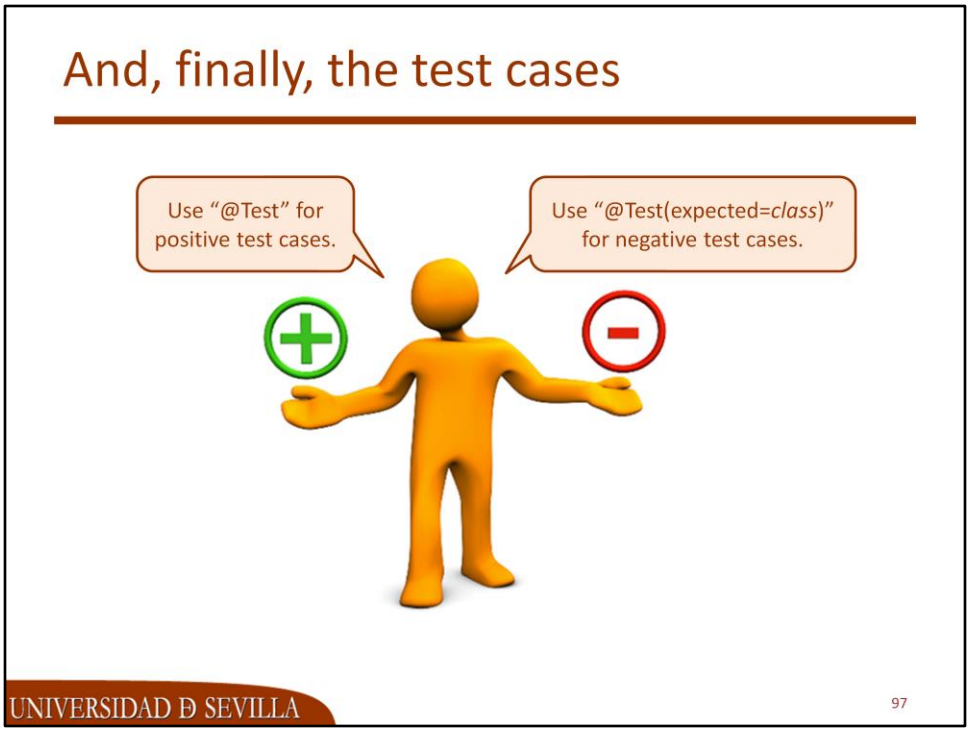
A typical test class… keep reading!

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:spring/junit.xml"})
@Transactional
public class AnnouncementServiceTest extends AbstractTest {
        // The SUT ------------------------------------
        @Autowired
        private AnnouncementService announcementService;
        ...
}
```


This is how your test class must look… thus far.

Follow this guideline

And, finally, it's about time to write your test cases.



Test cases are regular methods, but they must be prepended with the following annotations:

- @Test, which is used to implement positive test cases.
- @Test(expected=*exception-class*), which is used to implement negative test cases.

Let's take a look at several examples.

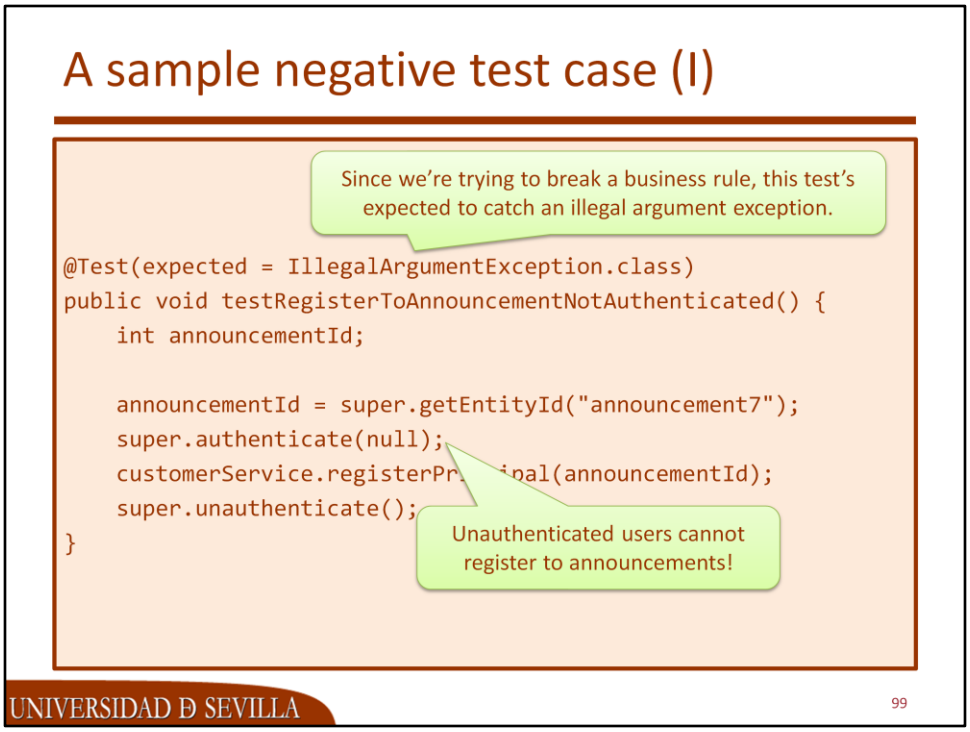
A sample positive test case

Nothing must fail in this positive test case.

```
@Test
public void testRegisterToAnnouncement() {
    int announcementId;

    announcementId = super.getEntityId("announcement7");
    super.authenticate("customer1");
    customerService.registerPrincipal(announcementId);
    super.unauthenticate();
}
```


This slide shows a sample positive test: it first fetches the identifier that corresponds to the announcement whose bean name is “announcement7” in your “PopulateDatabase.xml” file, then authenticates as user “customer1”, then registers to the previous announcement, and, finally, unauthenticates. Realise its simplicity: it just simulates a use case in which everything should work well. Later, we’ll explore the annotations and the methods that are provided by the testing framework; so far, it’s enough to know that annotation “@Test” allows it to recognise this method as a positive test case.



This is a sample implementation of a negative test case in which the user who attempts to register to an announcement is not authenticated. It first fetches the identifier of the announcement whose bean name is "announcement7", then authenticates as "null" to make sure that the principal's anonymous, then tries to register to the previous announcement, and, finally, unauthenticates (note that unauthenticating when no user is authenticated does not have any effects). Note that this use case should not work because we're breaking a business rule: the user who registers to an announcement must be authenticated. So the attempt to register is expected to throw an exception. Which one? In this case, it's "IllegalArgumentException", which is the typical exception that is thrown when a business rule is broken if you implement them using the "Assert" class; in other cases, you have to find out what the most appropriate exception is. Note that the "@Test" annotation accepts an optional parameter that informs the testing framework about the exception that is expected.

A sample negative test case (II)

Since we're trying to break a business rule, this test's expected to catch an illegal argument exception.

```
@Test(expected = IllegalArgumentException.class)
public void testRegisterToAnnouncementNonCustomer() {
    int announcementId;

    announcementId = super.getEntityId("announcement7");
    super.authenticate("reviewer1");
    customerService.registerPrincipal(    ouncementId);
    super.unauthenticate();
}
```

Reviewers cannot register to announcements!

In this case, we authenticate as "reviewer1", which should break another business rule that prevents users other than customers from registering to announcements. Again, the expected exception's "IllegalArgumentException" because we control that by means of a business rule that was implemented using the "Assert" class.

A sample negative test case (III)

Since we're trying to break a business rule, this test's expected to catch an illegal argument exception.

```
@Test(expected = IllegalArgumentException.class)
public void testRegisterToInactiveAnnouncement() {
    int announcementId;

    announcementId = super.getEntityId("announcement10");
    super.authenticate("customer1");
    customerService.registerPrincipal(announcem      d);
    super.unauthenticate();
}
```

It must not be possible to register to an inactive announcement!

In this case, we authenticate as "customer1", but try to register to the announcement whose bean name's "announcement10", which we know is not currently active. Again, the expected exception's "IllegalArgumentException" because we control that by means of a business rule that was implemented using the "Assert" class.

Let's now learn on a design pattern that will allow us to write many test cases very concisely.

This is a good question

UNIVERSIDAD Đ SEVILLA

103

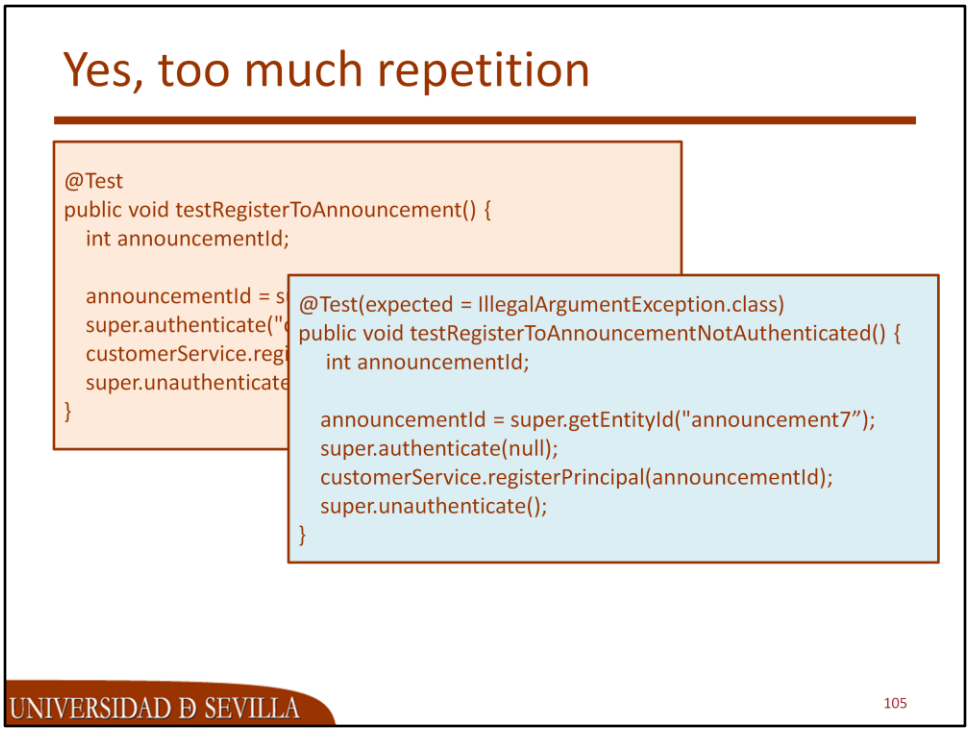
Clearly, a naive approach that builds solely on the @Test annotations and the few other gadgets that we've presented in the previous slides results in too much code repetition. See the next slides to realise what the problem is.

Yes, too much repetition

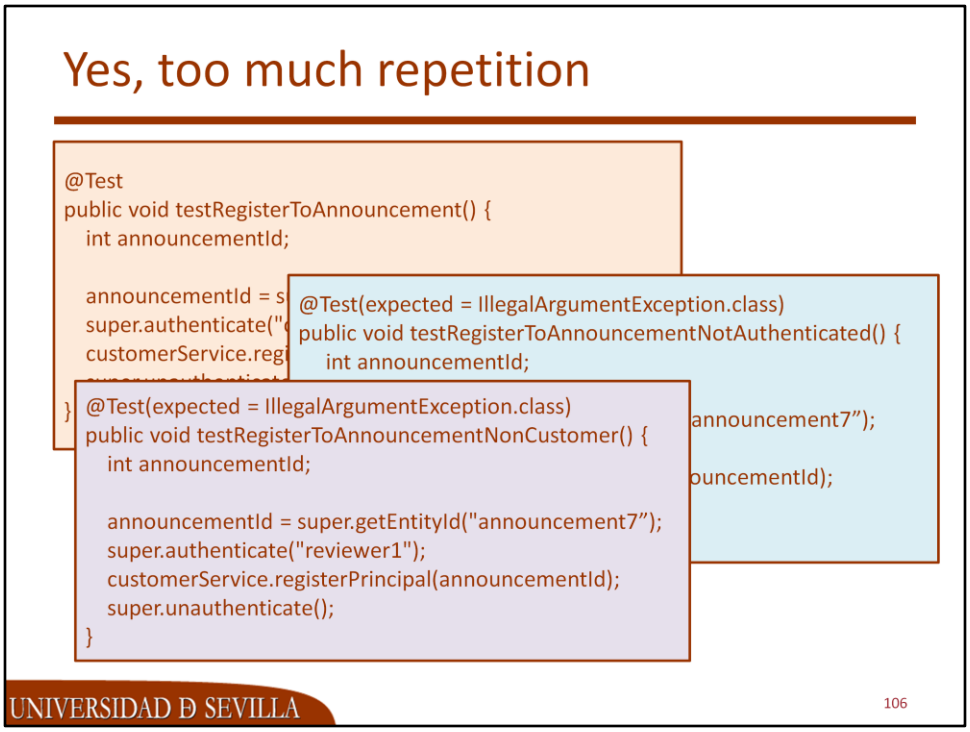
```
@Test
public void testRegisterToAnnouncement() {
   int announcementId;

   announcementId = super.getEntityId("announcement7");
   super.authenticate("customer1");
   customerService.registerPrincipal(announcementId);
   super.unauthenticate();
}
```

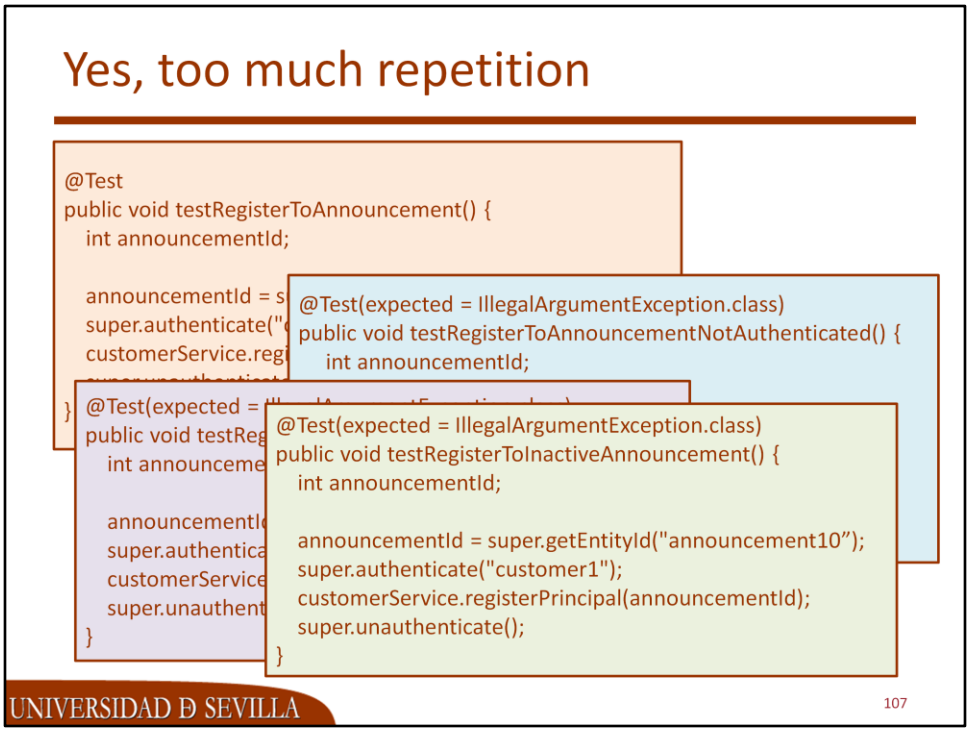

This is the first test that we presented.



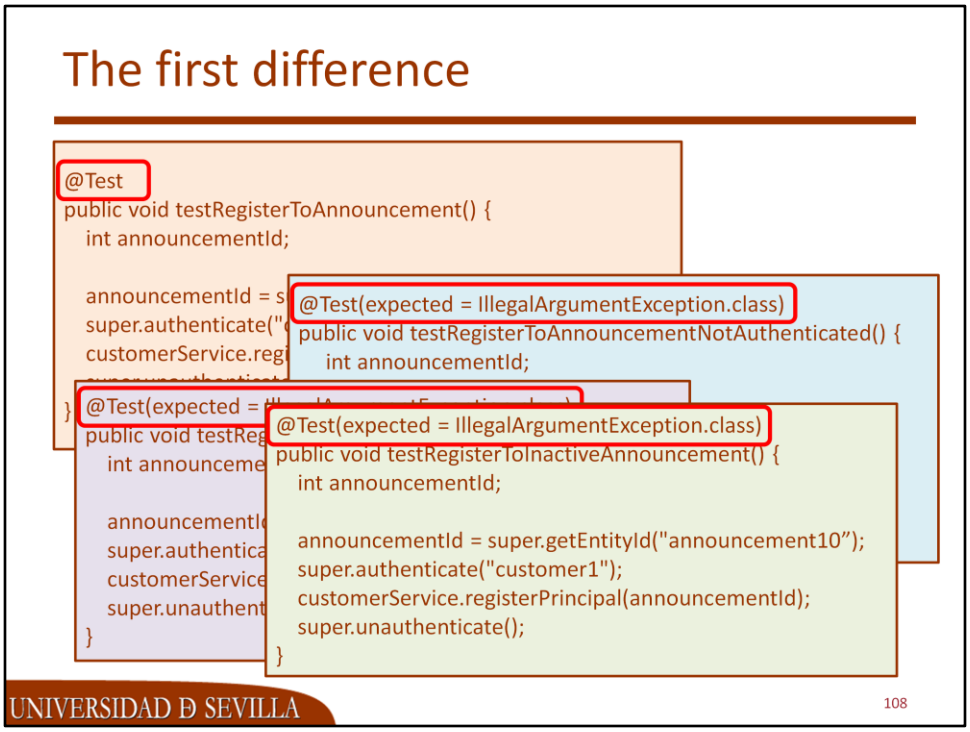
This is the second one. Can you see any commonalities?



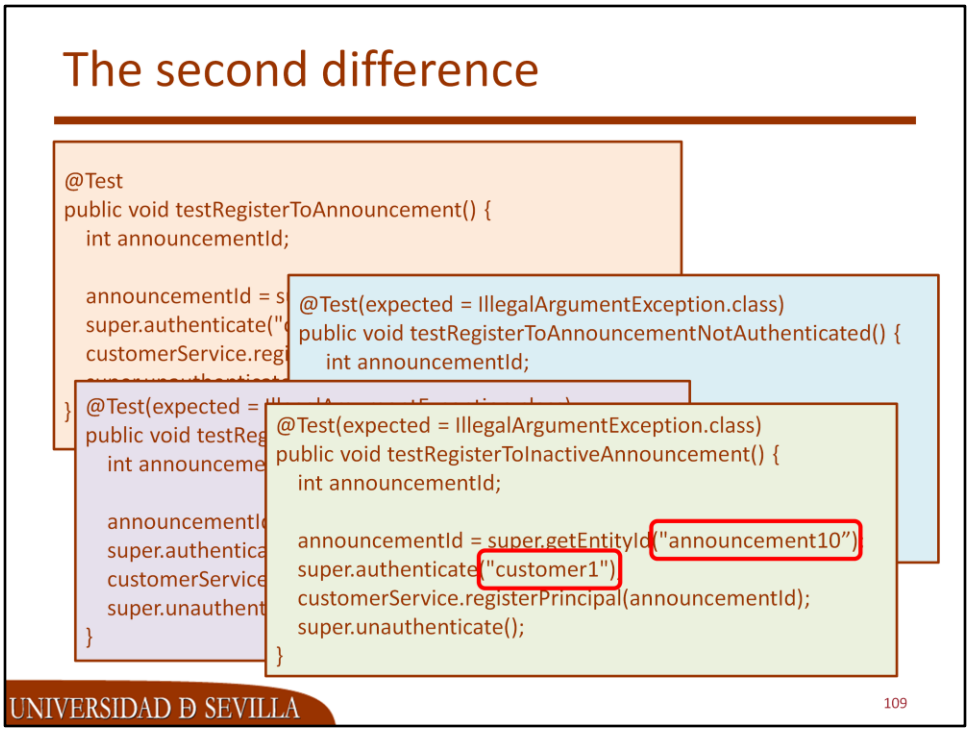
This is the third one. Can you see any other commonalities?



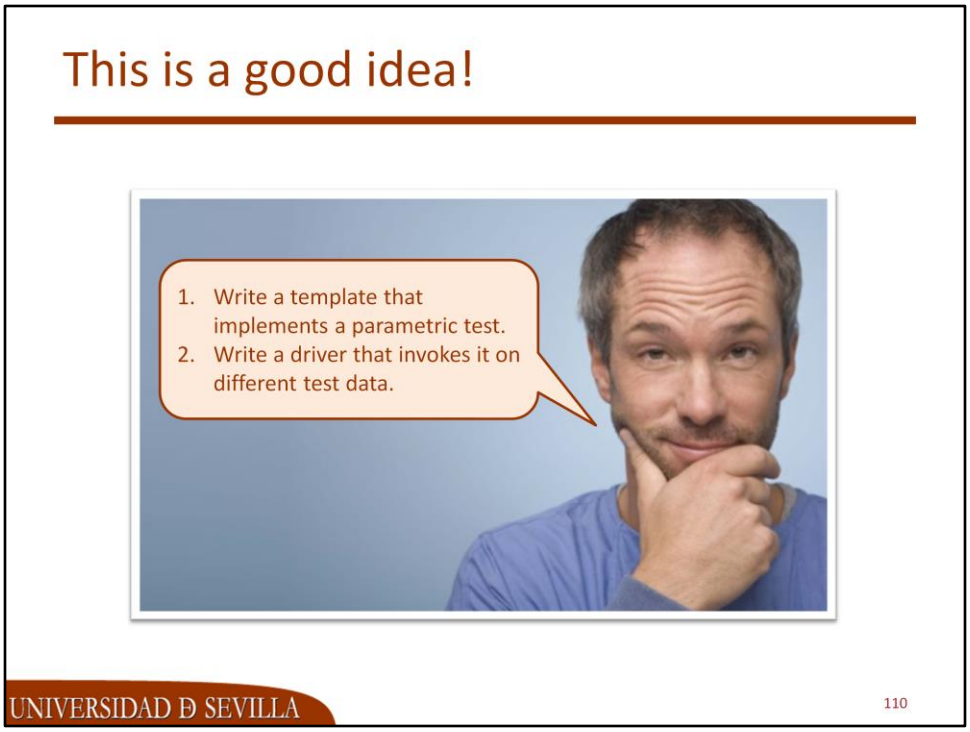
And this is the fourth one. The commonalities should be clear now, right?



Note that the first difference is that some tests are positive and some tests are negative; in the former case, everything's expected to work well, whereas an exception is expected in the latter case.



The second difference's regarding the data used to run the tests, namely: the announcement and the user.



This is a good idea to avoid repetition: let's create a template that implements a parametric test case and a driver that invokes it on different data. This is a simple pattern, but it may save a lot of repetitive work. It's a clean solution that typically scales well to a couple of dozens of test cases, which is exactly what we need.

An implementation of the driver

```
@Test
public void driver() {
    Object testingData[][] = {
    {"customer1", "announcement7", null},
    {null, "announcement7", IllegalArgumentException.class},
    {"reviewer1", "announcement7", IllegalArgumentException.class},
    {"customer1", "announcement10", IllegalArgumentException.class}
    };

    for (int i = 0; i < testingData.length; i++) {
        template((String) testingData[i][0],
                 (int) super.getEntityId(testingData[i][1]),
                 (Class<?>) testingData[i][2]);
    }
}
```


This is a sample implementation of the driver.

- It first declares an array to hold the testing data.
- And then loops over the array and invokes the template on each iteration.

Note that the driver's implemented as a positive test case using the "@Test" annotation; it'd great to have kind of a "@Driver" annotation, but it doesn't exist.

An implementation of the template

```
protected void template(String username,
                        int announcementId,
                        Class<?> expected) {
    Class<?> caught;

    caught = null;
    try {
        authenticate(username);
        customerService.registerPrincipal(announcementId);
        unauthenticate();
    } catch (Throwable oops) {
        caught = oops.getClass();
    }

    checkExceptions(expected, caught);
}
```


And this is a sample implementation of the template:

- It gets the testing data as parameters. Note that these data includes the username and the announcement identifier, plus the class of the exception that is expected to be thrown, if any.
- It then executes the test case within a try-catch block so as to capture any exception that can possibly be thrown.
- Finally, it calls the inherited “checkExceptions” method to check if the expected exception (if any) matches the exception that has been caught (if any); if they do not match, then this method throws a runtime exception that is propagated upwards and reported by JUnit.

Pretty simple, isn’t it?

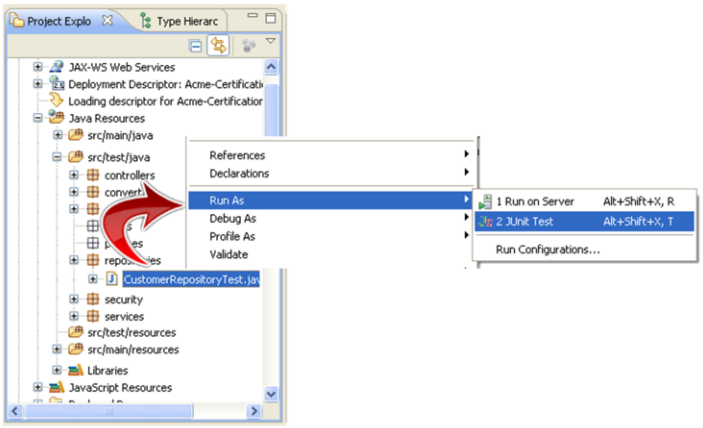
Let's now learn on how to run your tests.

The first question

The first question is: where should you put your test classes so that Spring can locate them?

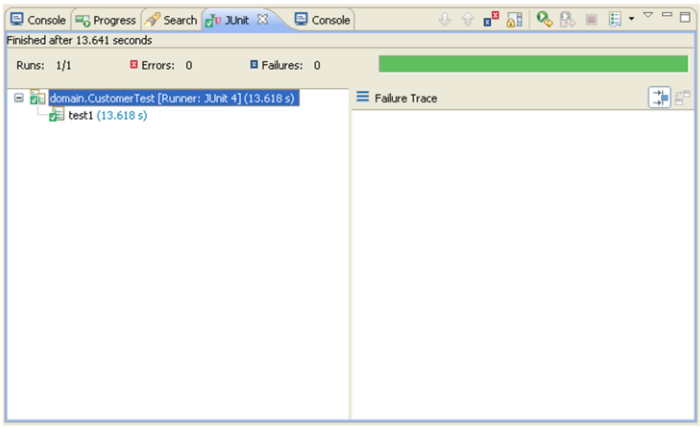
The JUnit classes are not intended to be used in the pre- or the production configurations, but in the development one. Therefore, it should not be placed in “src/main/java”, but in “src/test/java”. Note that we recommend that you should create a package per component kind, e.g., “services”, “repositories”, or “usecases”, then a test class per component, e.g., “AnnouncementServiceTest”, and, finally, a “utilities” package that holds class “AbstractTest”.

Running test classes or suites

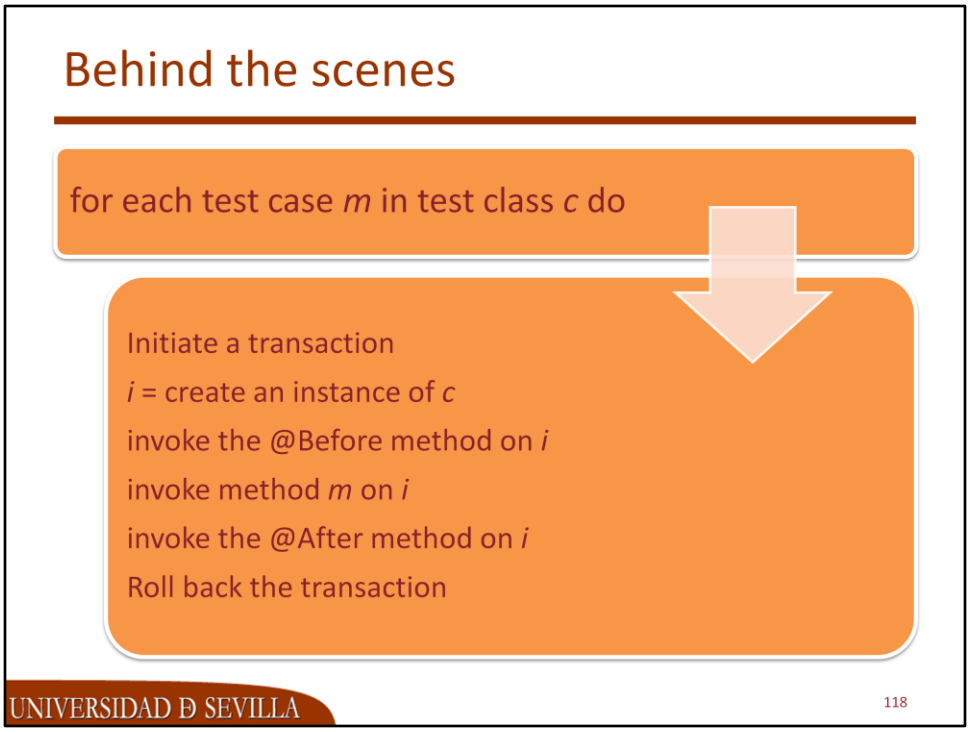


It's not difficult at all to run a test class: just right click it, select "Run as" and then "JUnit test". Please, note that you might even run a test case individually: click on the "+" symbol in the corresponding test class, right-click the corresponding method, and select "Run as > JUnit test". To run a whole test suite, just right click the folder "src/test/java" and JUnit will run every test class there.

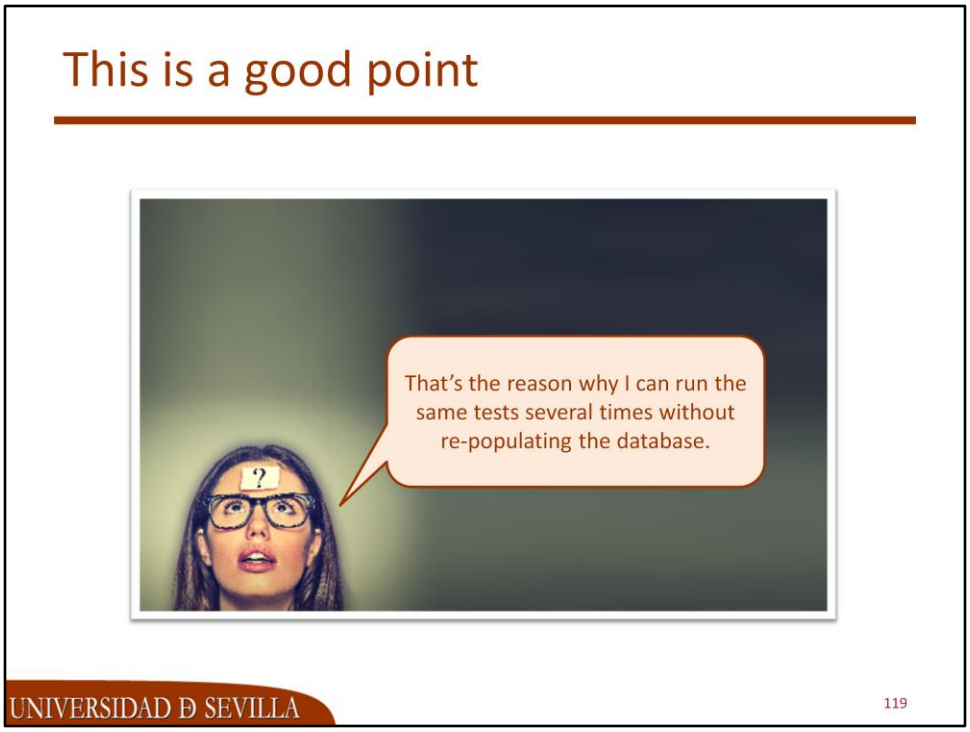
JUnit's view



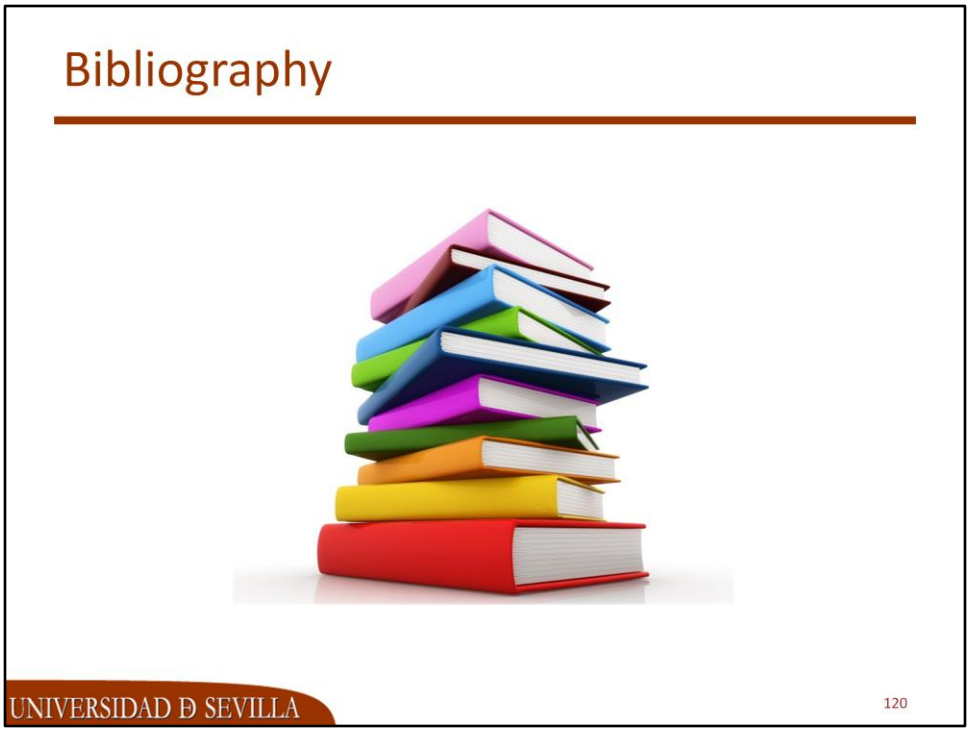
Take a look at JUnit's view when running your test suites. If every test succeeds, then a green bar is shown on the screen; if a test fails, then a red bar is shown. Whatever happens, you may navigate the tree of tests cases on the left pane to try to diagnose the problem.



Running a test class is very easy, isn't it? But you must understand perfectly what JUnit does with your test class. In this slide, we show a simple pseudo-code that illustrates the process. Note that JUnit iterates over every test method in your test class; for every such method, it first instantiates a transaction; then, it creates a new instance of your test class; then, it invokes the method that is annotated with "@Before" on that new instance (if it exists); it then invokes your test method; then, it invokes the "@After" method (if it exists); and, finally, it rolls back the transaction.



This is another good point: in cases in which a test case involves making changes to the database, it is important that the changes are rolled back before the test case concludes or, otherwise, we would have to re-populate the database every time we run our test suite, which would be very time-consuming. The magic behind the scenes is that every test case is executed within the context of a transaction that JUnit automatically rolls back when it finishes executing the test case. So, if you execute your test suite on a newly-populated database, no test will ever change its state; that is, you can execute the tests as many times as you wish and they all should work well.



Unfortunately, there's not a single book on testing methodologies that we can recommend. Please, use the material in this lesson as your primary source of information. Should you however need more information on the technology that we use to implement functional tests, please take a look at any of the following books:

JUnit recipes: practical methods for programmer testing
Joe B. Rainsberger
Manning, 2005

JUnit in action
Vincent Massol, Ted Husted
Manning, 2004

This bibliography is available in electronic format for our students at the USE's virtual library. If you don't know how to have access to the USE's virtual library, please, ask our librarians for help.

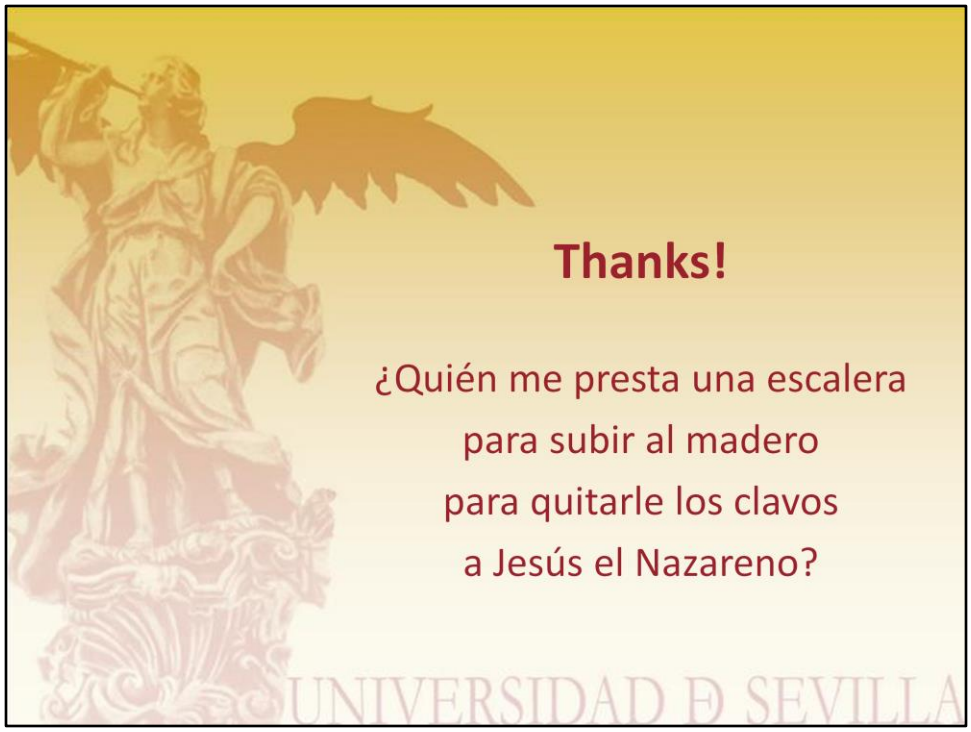
You should also consult Spring JUnit documentation, which is available at https://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/html/testing.html. It's also interesting to take a look at Prof. Thomas Huckle's home page, who is doing a good work at collecting and documenting famous software bugs that have led to very bad problems. His home page is available at http://www5.in.tum.de/~huckle/bugse.html and take a look.

Time for questions, please

UNIVERSIDAD Ð SEVILLA

121

Time for questions, please. Note that we won't update the slides with the questions that are posed in the lectures. Please, attend them and take notes.



Thanks for attending this lecture! See you next day!