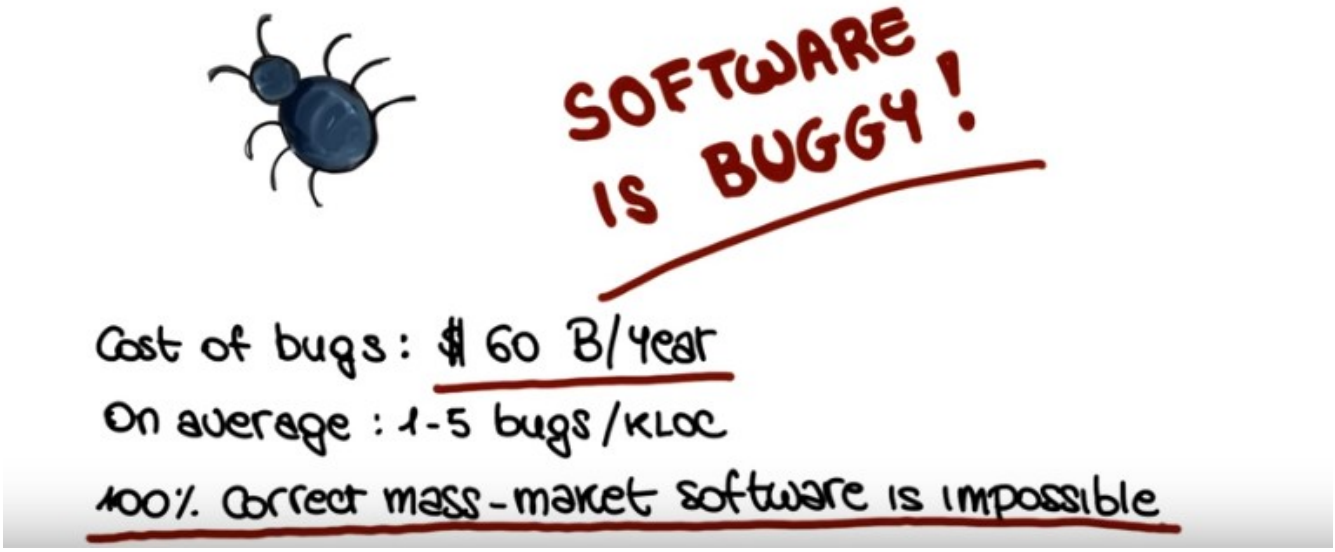
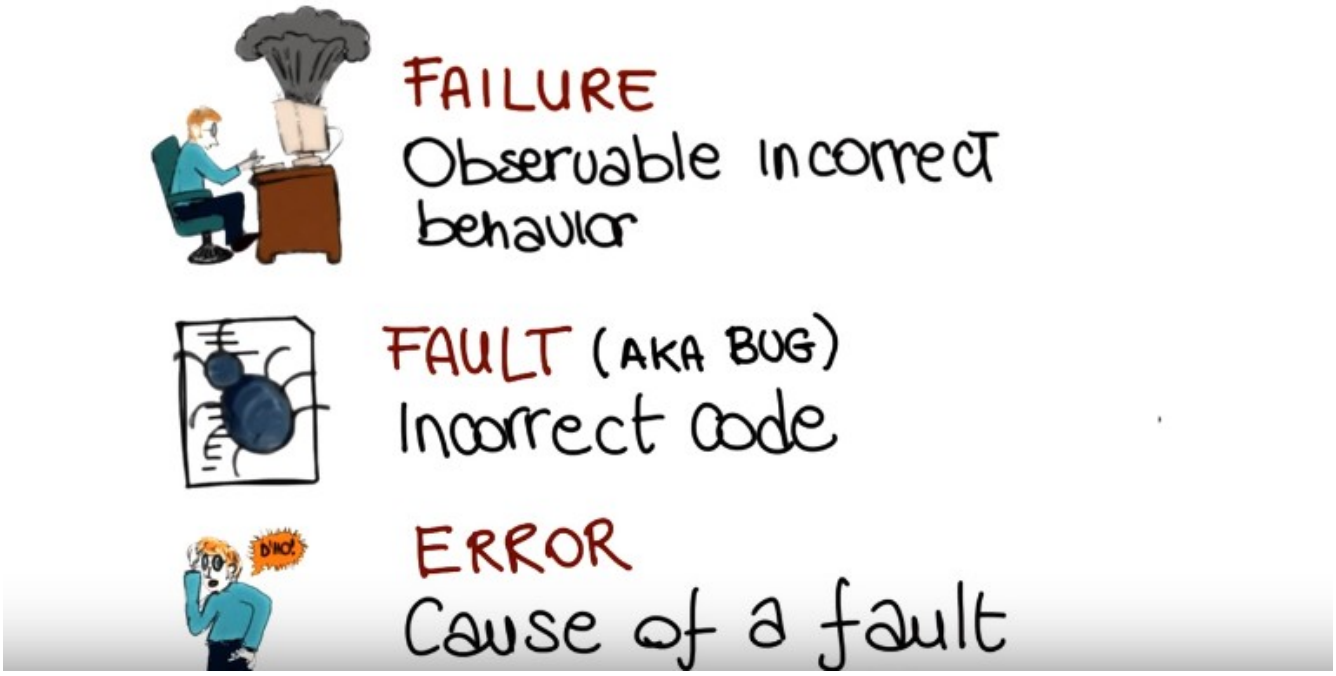
1. Hello, and welcome back. In the previous mini course we covered software design, discussed the UML and the unified software process, and worked on a complex project in which we developed a distributed software system. In this mini course, which is the last one for this class, we will cover my very favorite topic, software testing or more generally, software verification and validation. So why do I love software testing? Well, I love it because it is extremely important. It is very challenging and it is fun but only if you do it in the right way. In the upcoming lessons we will discuss why software verification is important, why software testing, which is a specific type of verification, is important, and what are the main techniques for performing software testing. We will also discuss test-driven development and agile methods, in which we'll lose some of the rigidity of the area processes and turn things around by writing tests before we write the code and then writing code that makes the test pass. Finally, we will perform a project, in which you get to apply most of the principles and practices of agile development in a realistic scenario. So let's jump right in.

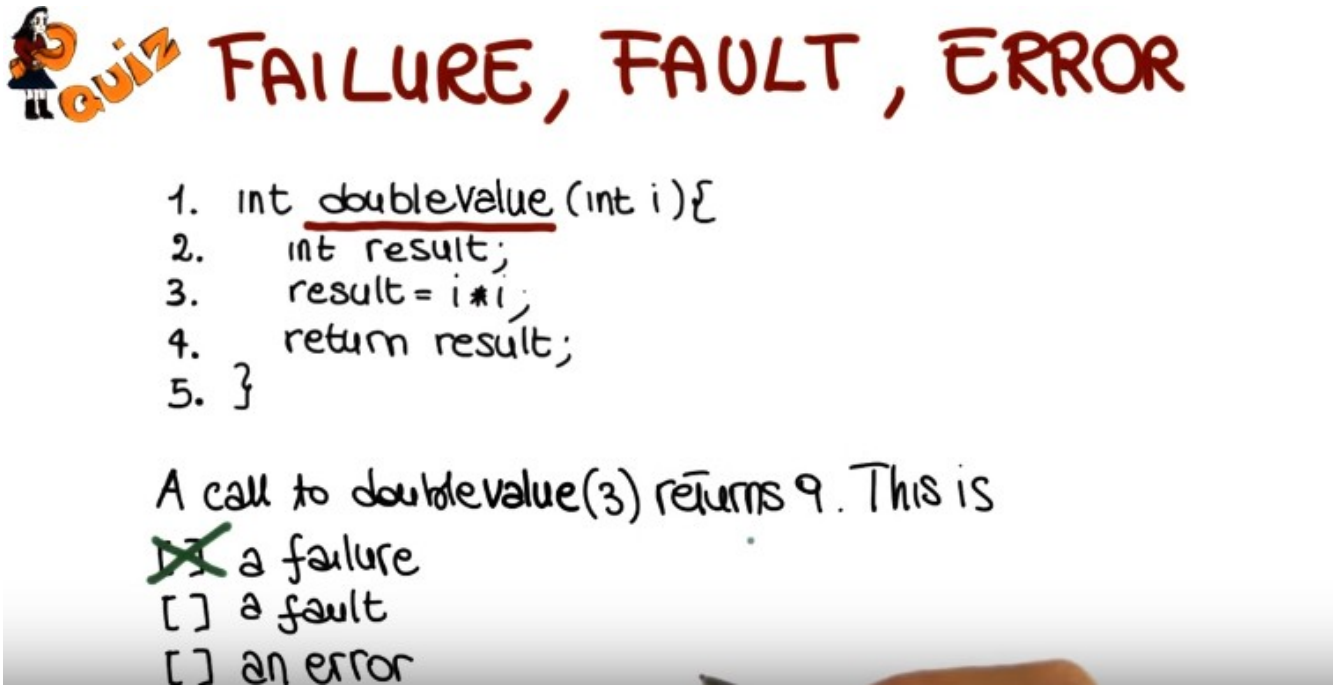
2. So let me start with some examples that motivate the need for very fine software. The first example I want to use is the famous Arian five. And if you remember that's a rocket that exploded not too long after departure. Because of a software error. And even without going to such dramatic examples. I'm sure you're all familiar with this kind of situation. Or this one, or again in this one. And here I'm not really picking on any specific organization, operating system or software. The point I want to make is that software is buggy. In fact, a federal report from a few years ago assessed that software bugs are costing the US economy, $60 billion every year. In addition, studies have shown that software contains on average one to five bugs every 1000 lines of code. Building 100% correct mass-market software is just impossible. And if this is the case, what can we do? What we need to do is to verify software as much as possible. In this part of the course, we will discuss how we can do this. We will discuss different alternative ways of very fine software systems. With particular attention to the most common type of verification. Which is software testing. Before doing that however, let me go over some basic terms that are commonly used. And I have to say, often misused in the context of software verification.



3. The first term I want to define, is failure. A failure is an observable incorrect behavior of the software. It is conceptually related to the behavior of the program, rather than its code. The second term I want to introduce is fault, which is also called bug. And the fault or bug, is an incorrect piece of code. In other words, a fault is related to the code. And is a necessary, but not sufficient condition for the occurrence of a failure. The final term I want to introduce is, error. Where an error is the cause of a fault. It is usually a human error, which can be conceptual. A typo or something along those lines. And know that this terminology, failure, fault and error, is the official [UNKNOWN] terminology. So you cannot go wrong if you use it. Now, let me illustrate the difference between failure, fault and error. Using a small example. What I'm showing here is a small function that, as you can see from its name, takes an integer parameter i. And is supposed to double the value of i, and return it. As we can clearly see, this is not what the function does.

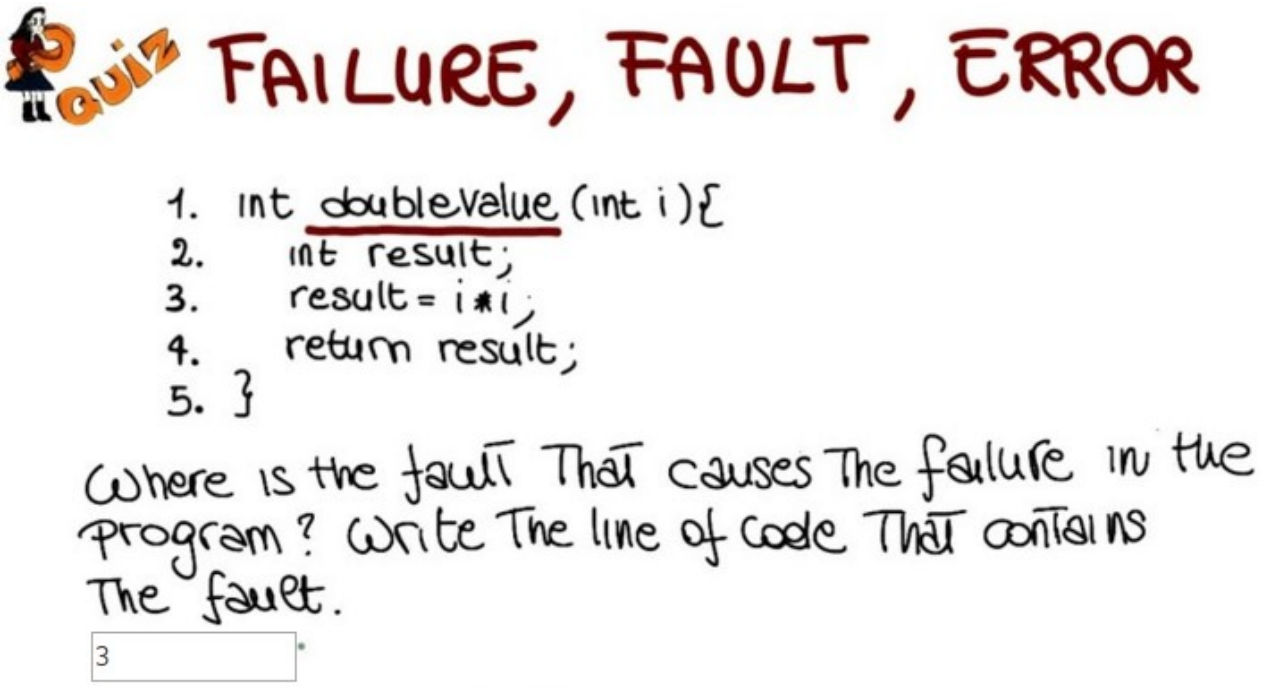


4. So now I have a few questions for you. And so we use, as usual, our developer Janet to introduce a quiz. And the first question I want to ask you is the following. A call to double passing three as a parameter returns the value nine. What is this? Is that a failure, a fault, or an error?



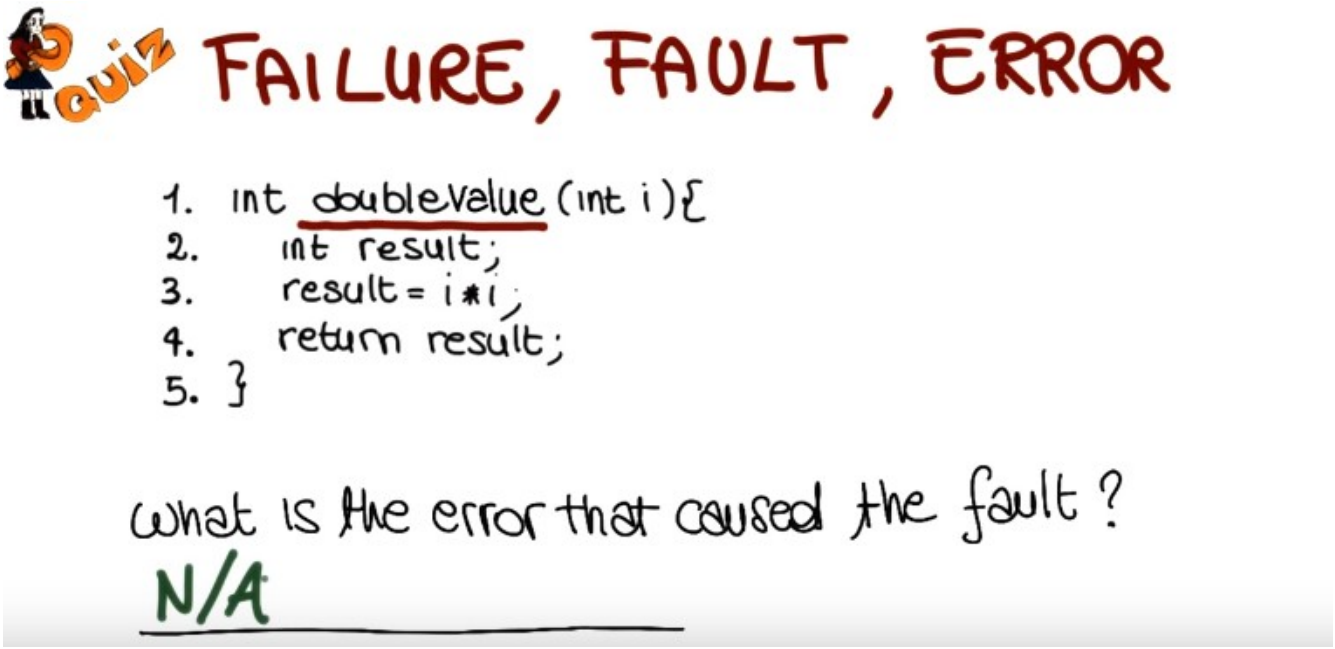
5. The fact that the call to double three returns nine instead of six is clearly a failure because it is an observable incorrect behavior of the program. So let me remind you that the failure is conceptually related to the behavior of the program, to the way the program acts and not its code.

6. So now, let me ask you a second question. We just saw that we can, reveal a failure in the program by calling double with parameter three. Where is the fault that causes such failure in the program? So I want you to write the number of the line of code that, that contains the fault.



7. Before telling you what the right answer is, let me remind you that the fault is related to the code, and is a necessary, but not sufficient condition for the occurrence of a failure. So in this case, a single faulty line is responsible for the failure, which is line three. So the correct answer, is three. At line three, the program computes i times i, instead of i times 2, as it should do.

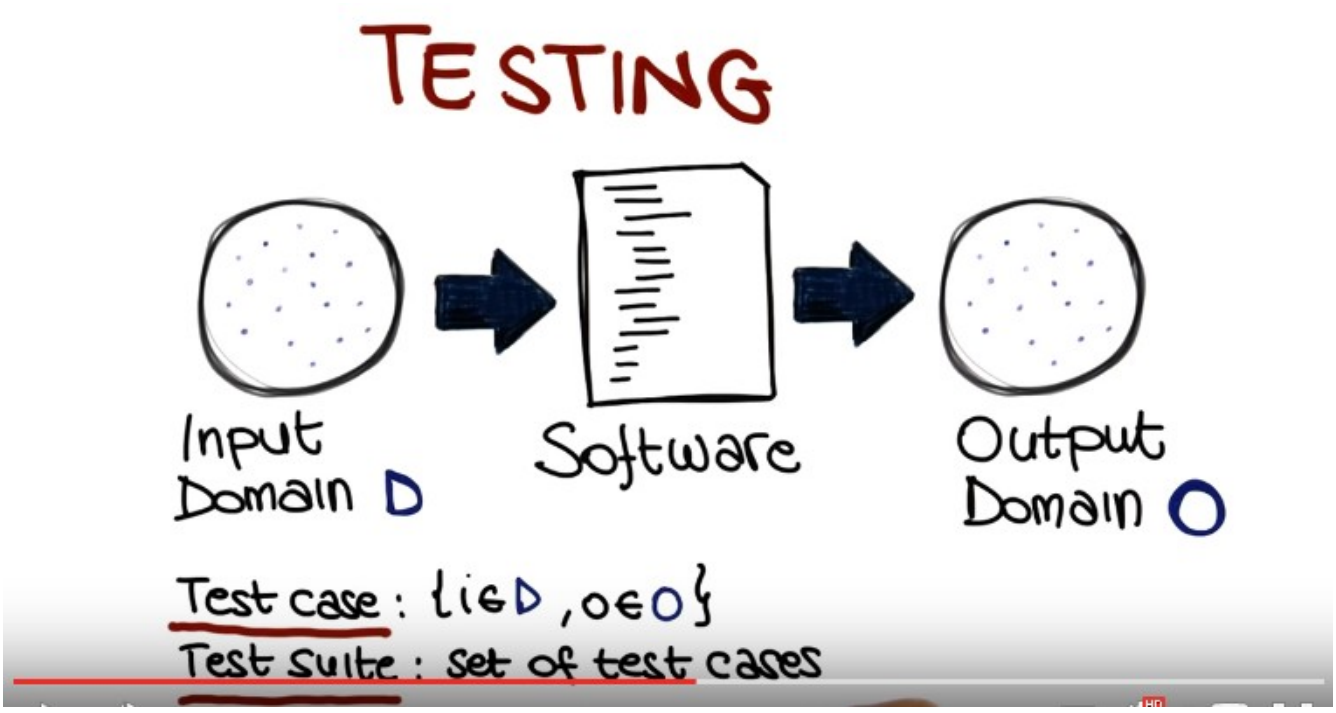
8. So, I want to ask you one last question about this problem. What is the error that cause the fault at line three? Remember that an error is the cause of a fault and it's usually a human error.



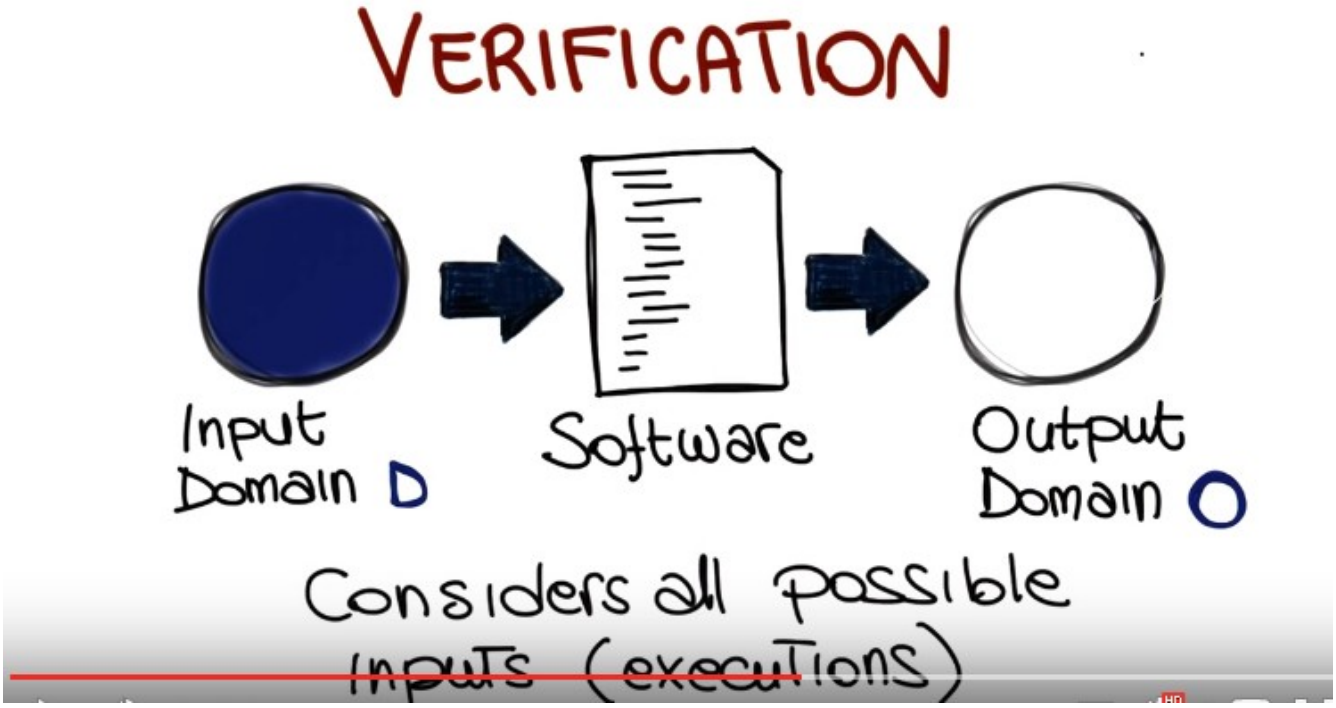
9. And I apologize, but this was a tricky question that we cannot really answer. We really have no way to know what the error was. It could have been a typo, an erroneous copy and paste operation, or even worse a conceptual error. In case a developer did not know what it means to double a number. Unfortunately though, the developer is the only one who could actually answer this question. So even though we cannot really answer this question, having it help us think about how the error is in fact related to human behavior.

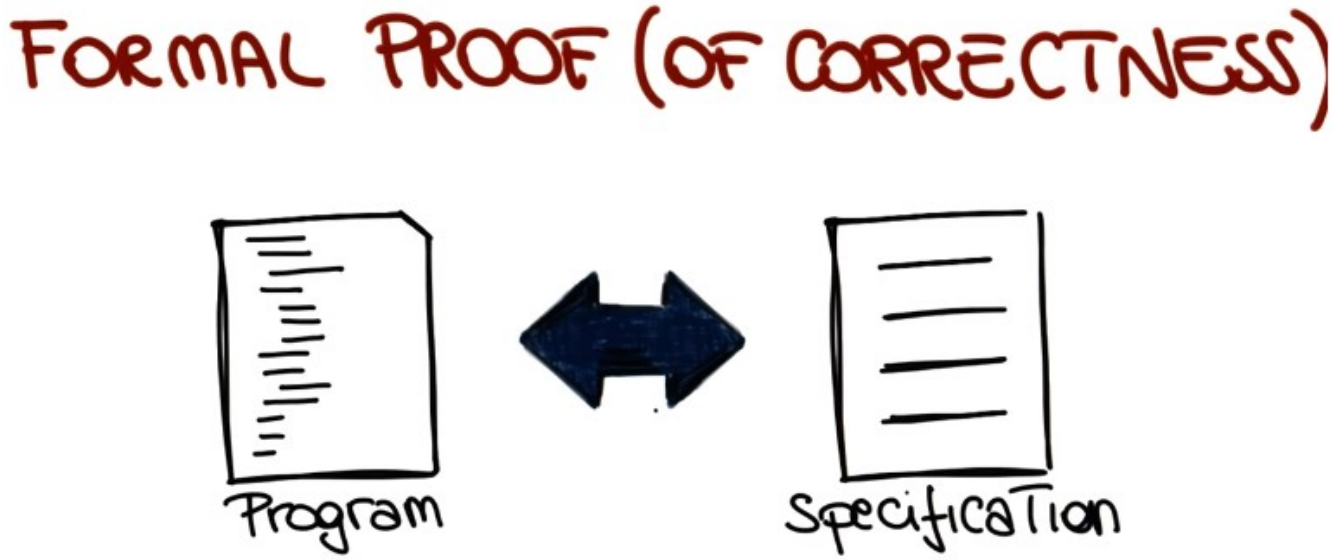
10. Now that we got out of the way this initial set of basic definitions. Let's go back to our main concept, which is software verification. We said that software is buggy, and because software is buggy, we need to verify the software as much as we can. But how can we verify software? There are several ways to verify a software system. Among those, we will discuss four mainstream approaches. The first one is testing, also called dynamic verification. The second approach is static verification. The third approach is inspections. And finally, we're going to consider a fourth approach which is formal proofs of correctness. So what I'm going to do next, I'm going to first provide an overview of these approaches and then discuss some of them in more depth and please note that although we will discuss all four approaches we will spend most of our time on software testing. As software testing is the most popular and most used approach in industry. So let's start with our overview and in particular with testing. Testing a software system means exercising the system to try to make it fail. More precisely, let's consider a program. Its input domain, which is the set of all the possible inputs for the program and, its output domain, which is a set of all the possible corresponding outputs. Given this context, we can define what a test case is. A test case is a pair that consists of a, an input from the input domain D, and then, expected output O from the output domain. And O is the element in the output domain that a correct software would produce when ran against I. We can also define the concept of test suite, which is a set of test cases, and we're going to use these two concepts of test case and test suite quite a bit in the rest of the lessons. Subject verification, tries to identify specific classes of problems in the program. Such as null pointer dereferences. And unlike testing, what it does is that it does not just consider individual inputs, it instead considers all possible inputs for the program. So it consider in a sense all possible executions of the program and all possible behaviors of the program, that's why we

save the verification unlike testing it's complete. The 3rd technique we are going to consider is inspections, and inspections are also called reviews or walkthroughs. And unlike the previous techniques, inspections are a human intensive activity, more precisely, they are a manual and group activity in which several people from the organization that developed the software, look at the code or other artifacts developed during the software production and try to identify defects in these artifacts. And interestingly inspections have been shown to be quite effective in practice and that's the reason why they're used quite widely in the industry. Finally, the last technique I want to mention is Formal Proof (of correctness). Given a software specification, and actually a formal specification, so a document that formally defines and specifies the behavior, the expected behavior of the program. A form of proof of correctness proves that the program being verified, actually implements the program specification and it does that through a sophisticated mathematical analysis of the specifications and of the code.



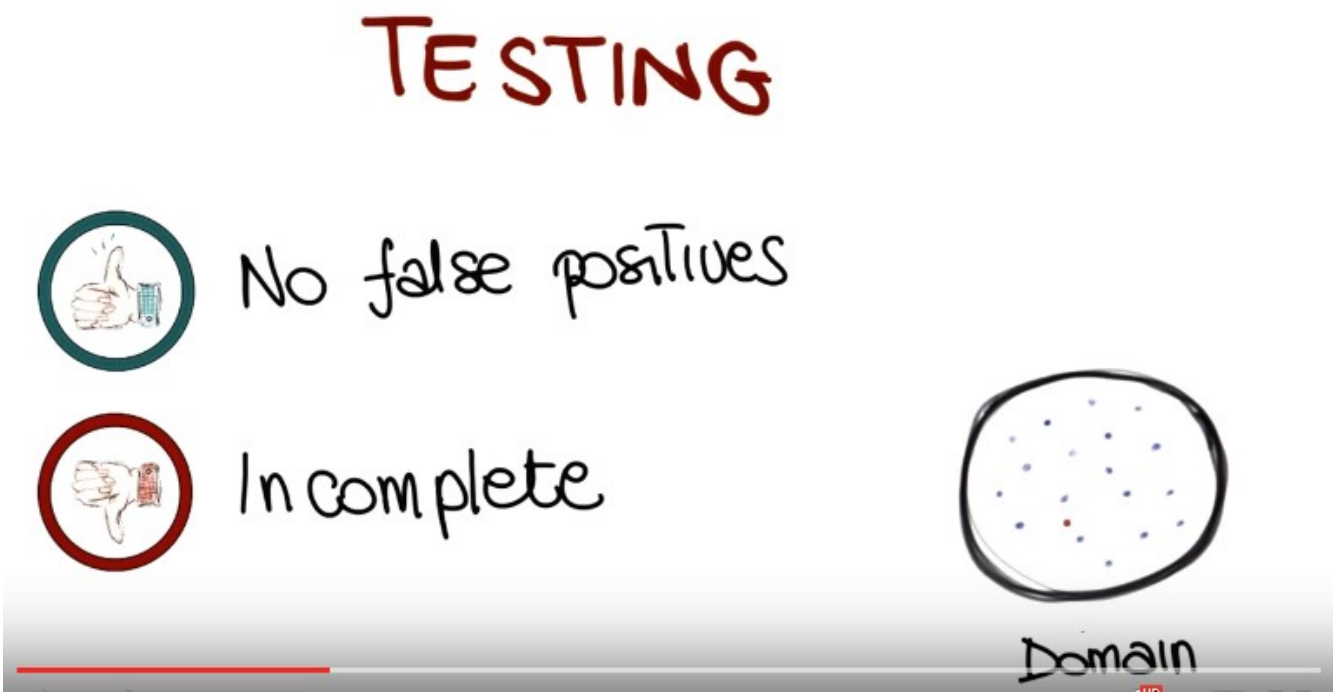
下圖的標題 Verification 應該是 Static verificaiton:





11. The four different techniques that we just discussed have a number of pros and cons. So next we are going to discuss the main pros and cons for these techniques, so as to be able to compare them. When testing is concerned the main positive about this technique is that it does not generate false alarms. In other words, it doesn't generate false positives. What that means, is that when testing generates a failure, that means that there is an actual problem in the code. The main limitation of testing, however, is that it is highly incomplete. Consider again the picture that we drew a little earlier. The one representing the input domain of the program being tested. Even in the best scenario, testing can consider only a tiny fraction of the problem domain, and therefor a tiny fraction of the program's behavior, and we'll say a lot more about that in the following lessons. Static verification, unlike testing, has the main advantage that it considers all program behaviors. If we look back at our diagram, whereas testing will select only a few of those inputs, static verification will consider them all. Unfortunately, however, this comes with a price. Due to limitation of this kind of analysis and due to infeasibility issues, static verifiation considers not only all the possible behaviours, but also some

impossible behaviors. And what that means is that static gratificaition can generate false positives. And this is, in fact, the main issue with static verification techniques. As we will further discuss later in the class, static verification can generate results that are not true. For example, it might report a possible no point of the refernce that cannot actually occur in practice. The strongest point about inspections is that, when they're done in a rigorous way, they're systematic and they result in a thorough analysis of the code. They are nevertheless a manual process, a human process. So they're not formal and their effectiveness may depend on the specific people performing the inspection. So its results can be subjective. Finally, the main pro about formal proofs of correctness is that they provide strong guarantees. They can guarantee that the program is correct, which is not something that any of the other approaches can do, including study verification. But the main limitation of formal proofs is that they need a form of specification, a complete mathematical description of the expected behavior of the whole program, and unfortunately such a specification is rarely available, and it is very complex to build one. In addition, it is also very complex, and possibly expensive, to prove that the program corresponds to a specification. That is a process that requires strong mathematical skills and, therefore, a very specialized personnel.



STATIC VERIFICATION

INSPECTIONS

Systematic, Thorough

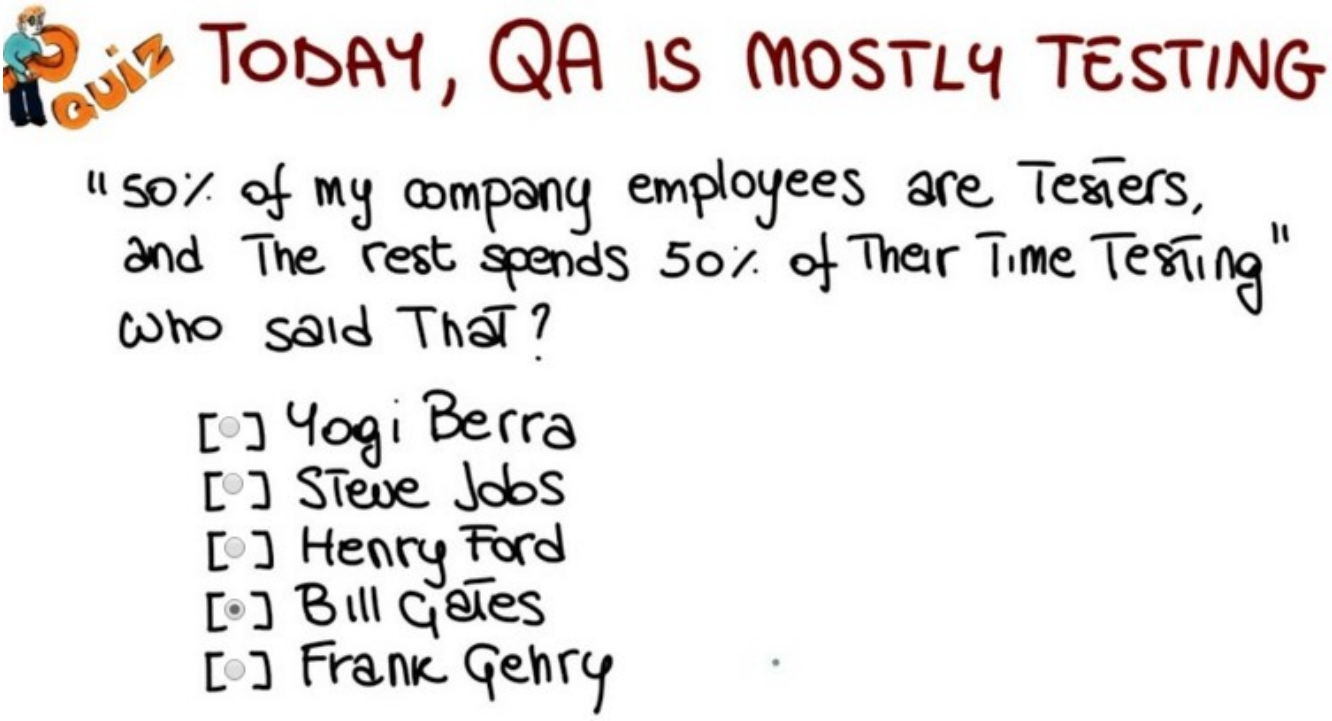
Informal, subjective

FORMAL PROOF (OF CORRECTNESS)

Strong guarantees

Complex, expensive

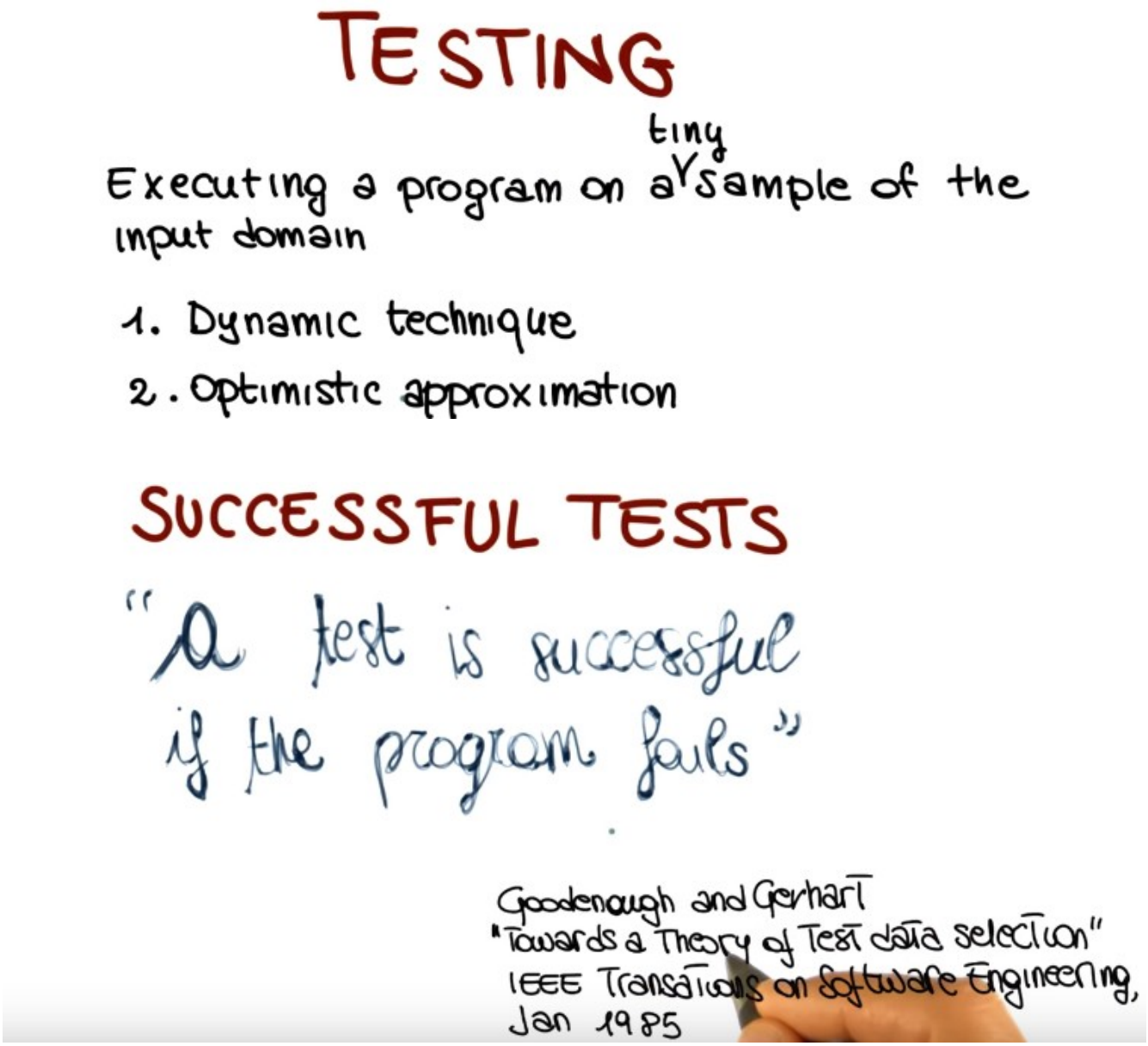
12. So let's have another simple quiz, and we're going to have our developer Brett introducing the quiz. And the starting point for this quiz is the fact that today, quality assurance, or verification, if you wish, is mostly testing. That is, testing is the most commonly used activity to perform software verification. So now, I'm going to show you a quote, 50% of my company employees are testers and the rest spends 50% of their time testing, so I want to ask you, who said that? I'm going to give you some possibilities. Was that Yogi Berra, Steve Jobs, Henry Ford, Bill Gates or Frank Gehry? Take your best guess.



13. And the correct answer is Bill Gates. So this gives you an idea of how important testing is in Microsoft in particular, but in many other software companies in general.

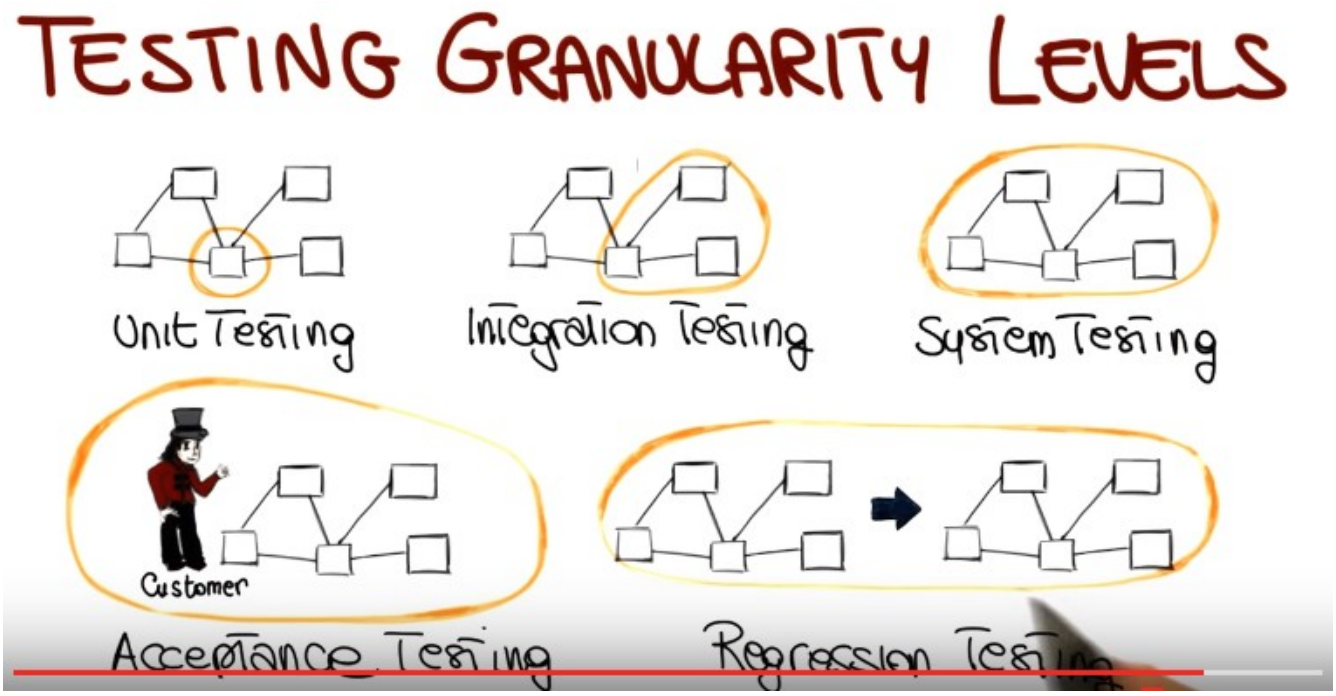
14. So let's talk more about testing, as we said a little earlier in the lesson, testing means executing the program on a sample of the input domain, that is of all the possible input data and I really want to stress that this sample is tiny sample of the input domain. There are two important aspects of testing that I'm want to mention here, there first one is that testing is a dynamic technique. And what that means is that the program must be executed in order to perform testing. The second important point is that testing is an optimistic approximation. And what does it mean to be optimistic? Well, it means that the program under test is exercised with a very small subset of all the possible inputs as we just said. And this is done under the assumption that the behavior with any other input is consistent with the behavior shown for the selected subset of input data, that is why it is an optimistic approach. Another concept that I want to mention explicitly, is the concept of successful test. And I'm going to do that, using another quote. This one from Goodenough and Gerhart in their paper Towards a Theory of Test Data Selection, and what the quote says is that a test is successful if the program fails. And this might sound counterintuitive, but the point here is that testing cannot prove the absence of errors, but only reveal their presence. If a set of tests does not produce any failure, we are either in the extremely unlikely case of a correct program, or in the very likely situation of a bad set of tests that are not able to reveal failures of the program. And that is why we say that the test is successful if you can show that there are

problems in the program.

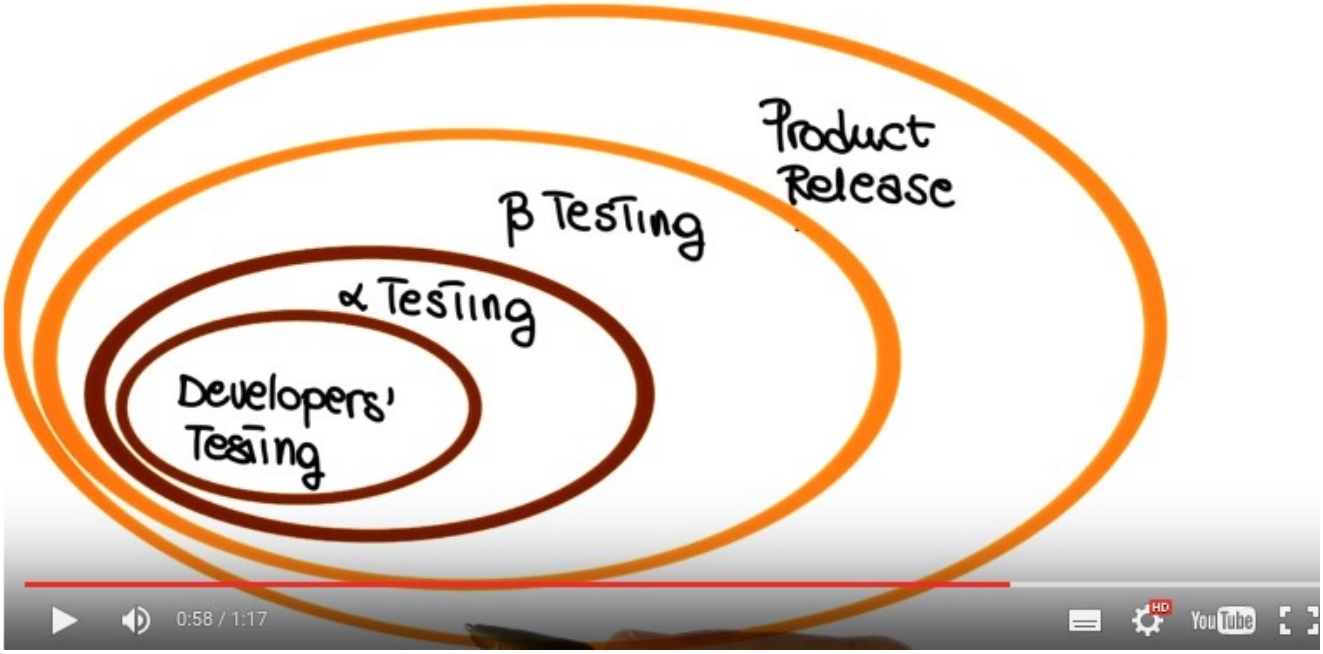


15. And before I start talking about specific testing techniques, there's something else that I want to discuss, which is Testing Granularity(粒度) Levels. So let's consider a software system, a system made out of components that interact with one another. So the first level that we consider in testing is called Unit Testing, which is the testing of the individual units or modules in isolation. The next step, is to see there are multiple modules and their interactions. And this is called Integration Testing. So, integration testing is the testing of the interactions among different modules. And it can be performed according to different strategies. Depending on the order in which the modules are integrated and on whether we integrate one module at a time or multiple modules together, all at once. And in this latter case, we call this kind of integration testing, the one that integrates all the modules at once, Big Bang integration testing. And after performing integration testing, the next step is to test the complete system as a whole. And this level of testing is normally called, System Testing. So system testing is the testing of the complete system and it includes both functional and non functional testing. We will discuss

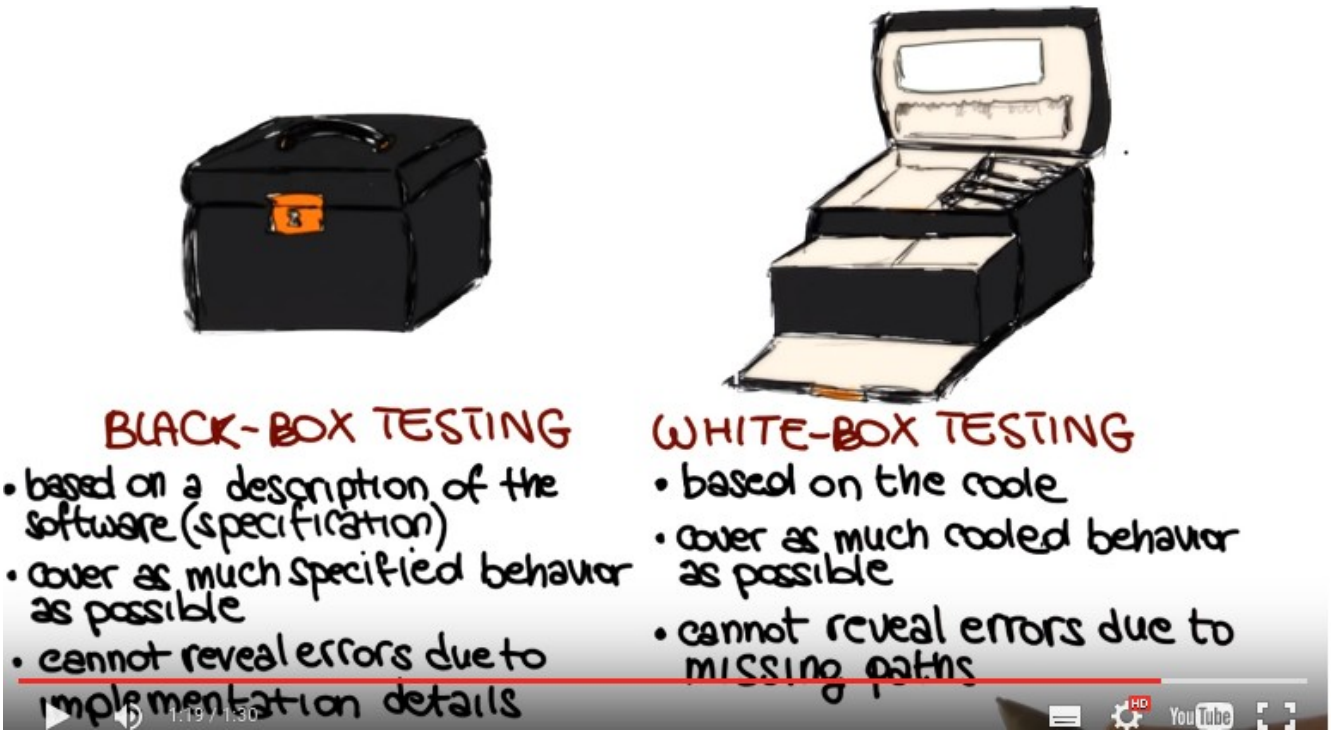
functional and non functional testing in details in the next two lessons. But I just want to give you an idea of what they are intuitively. Functional tests are the test that aim to verify the functionality provided by the system. For example if you consider the function doubleValue() that we saw earlier in the lesson, a functional test will try to assess that that function is producing the right value given a specific input. Conversely, no functional test are the one that target, as surprisingly, no functional properties of the system. For example, no functional test will include performance tests, load tests, robustness tests. In general, no functional tests will try to assess different qualities of the system, such as reliability, maintainability, usability, so basically, all the ilities that you can think about. In addition to these three basic testing levels, there are two more levels that I want to consider and that I want to discuss. And they both involve the whole system. And the first one is Acceptance Testing which is the validation of the software against the Customer requirements. So this is the testing that makes sure that the system does what the customer wants it to do. And the last type of testing that I want to mention is Regression Testing. And regression testing is the type of testing or retesting, that we perform every time that we change our system. And we need to make sure that the changes behave as intended and that the unchanged code is not negatively affected by the modification, by these changes. In fact, what can happen when you modify the code is that parts of the code that are related to the changes, are actually affected by the changes, and start misbehaving. And we call those regression errors. And regression errors, are very common. For example, you're probably familiar with the situation in which, one software update is released, and just a few days later, another software update is released. In many cases that happens because the first update was containing regression errors. So the changes in the code that broke some functionality, that resulted in failures on the user's machine and in bug reports and therefore that caused further maintenance, further bug fixes, and a release on a new version. Something else I'd like to mention about regression testing, is that regression testing is one of the main causes why software maintenance is so expensive. And that's also why researchers have invested a great deal of effort into refining regression testing techniques that can make regression testing more effective and more efficient. So let me leave you, with a little piece of advice which is try to automate as much as possible regression testing. For example use scripts, use tools, make sure to save your harness, make sure to save your input, and outputs for the test, because you want to be able to rerun your test, at a push of a button as much as possible every time you change your code, to avoid the presence of regression errors in the code you release.



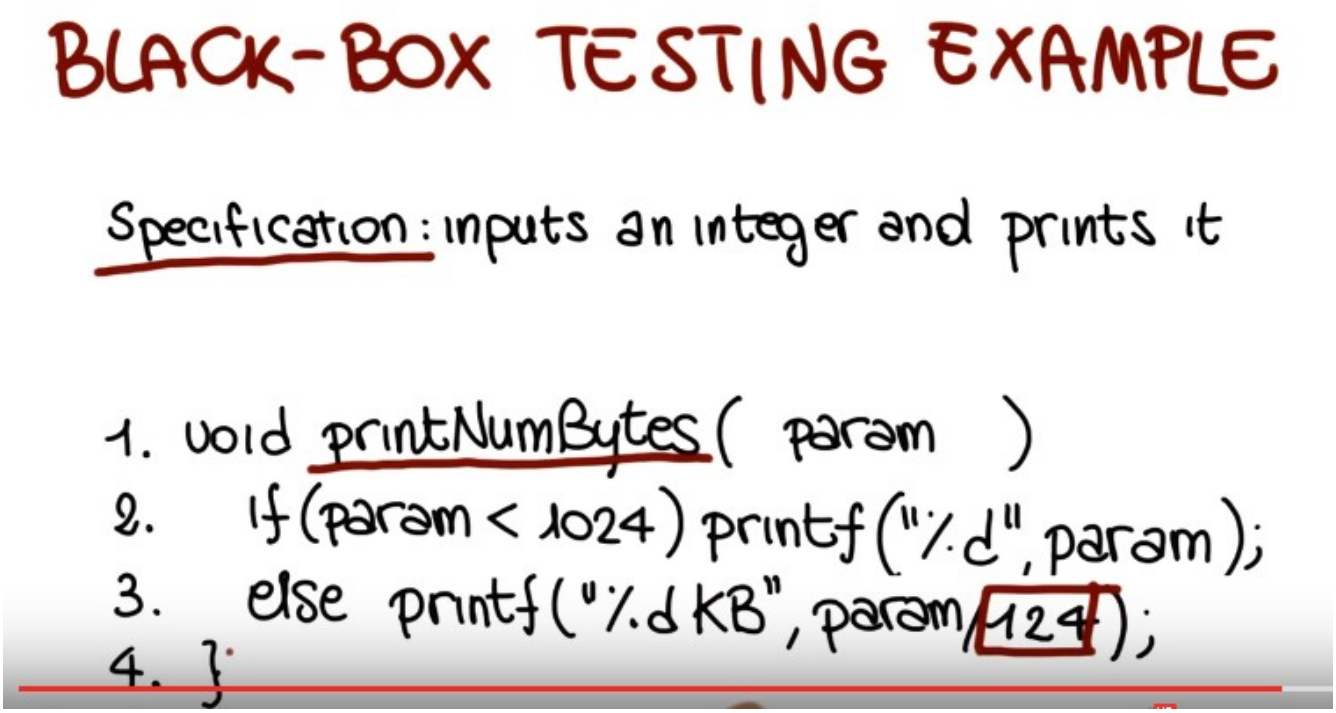
16. All the testing levels that we've seen so far is what we can call developer's testing. So that's testing that is performed either within the testing organization, or by somebody who's doing like third-party testers on behalf of the testing organization. But there are two other kinds of testing that are worth mentioning that are also related to testing phases and these are alpha and beta testing. Alpha testing is the testing performed by distributing a software system ready to be released to a set of users that are internal to the organization that developed the software. So you can consider these users as, if you pass me the term, guinea pigs that will use an early version of the code and will likely discover errors that escaped testing and will have made it to the field if not caught. Beta testing is the next step after alpha testing, in which the software is released to a selected subset of users, in this case, outside your organization. And also in this case, the users are likely to discover latent(潜伏性的) errors in the code before it is officially released to the broader user population, so before we have an actual product release. So you may wonder why do we need to do both alpha and beta testing. Why not just one of the two? The reason is that alpha testing is performed to iron out the very obvious issues that still escape testing, but we want to do that before involving people outside your organization. And the rationale is that alpha testers have a higher tolerance for problems than beta testers, who expect a mostly working system.



17. We're almost at the end of this lesson. In the next two lessons we're going to talk about two main families of testing techniques, black-box testing techniques, and white-box testing techniques. So, what I want to do before getting into the discussion of the specific techniques in this families. I want to give you an overview of what black-box testing and white-box testing are. Black box testing is the kind of testing in which we consider the software as a closed box. That's why it's called black box. So we don't look inside the software, we don't want to look at the code. We just going to look at the description of the software. So this is the testing that is based on a description of the software, which is what we normally call the specification for the software. And what black box testing tries to do is to cover as much specified behavior as possible, and the main limitation black box testing and the reason why this is complimentary to white-box testing is that it cannot reveal errors due to implementation details. Conversely, white-box testing is the kind of testing that looks inside the box. So looks at the code and how the code is written and uses this information to perform the testing. So white-box testing is based on the code, its goal is to cover as much coded behavior in this case, as possible, and its limitation is that unlike black-box testing, it can't reveal errors due to missing paths. Where missing paths are a part of a software specification that are not implemented and the reason why you can not reveal them is because it is focused on the code and not on the specification.



18. To give you a slightly better understanding of the differences between black-box testing and white-box testing, I am going to provide you a couple of simple examples that illustrate the, the strengths and limitations of these two techniques. So, in this case, let's start with black-box testing, so we're only working with this specification. So, let's say that our specification says that this is a program that inputs an integer value and prints it. And implementation, we don't know because we're working at the black box level. If we wanted to test this function according to its specification, what we will probably do is to select a positive integer, a negative integer, and the zero as test inputs and see how the program behaves for these inputs. So let me now show you a possible implementation for this specification. What I'm showing here is this function that we called print NumBytes, which takes the parameter and prints it. And one thing that we notice right away is that, although in the specification, numbers that are less than 1024 and numbers that are greater or equal to 1024 are exactly equivalent from the specification standpoint. They're however treated differently in the code, so the developer decided that the program was just going to print the value of the parameter if it's less than 1024. But it was actually divided by 1024 and printing it with a kilobyte mark after it if you are greater than 1024. And notice that here, there is a problem. The developer, just a number 124, instead of 1024. So there's probably a typo in this point in the code. So this is a case in which by simply doing black-box testing, so by simply looking at the specific issue, we might miss this problem. Because we have no reason to consider numbers that are less than 1024 or greater than 1024. However if we were to look at the code, so operating at white-box manner, we will right away see that we need to have a test case that checks the program when the parameter is greater than 1024. And we will find the problem right away. So now let me show you a dual example.



19. In this case we focus on white box testing. So consider now this other function, called fun. And let's assume that we want to test this function without having a specification. So without knowing exactly what it needs to do. But just by looking at the code. So we will try to do the problem in this case is to try to just execute all the statements. In the function. And notice I will talk extensively of what does it means to do white box testing later on in the next, two classes. So if that's our goal, if our goal is to cover all the statements, any input will really do. So any test case will excecute all statements in the code. And we'll a complete, you know, white-box testing coverage for the program. Imagine that I now give you a specification for this function. And what the specification says is that this function inputs an integer parameter, param, and returns half of its value, if param is even, and its value unchanged otherwise. That means if param is odd. So looking at this specification, we can clearly see that the function fun works correctly only for even integers, and it doesn't work for odd integers. Because it computes. Half of the value of the parameter and returns it every time, no matter what param is. So this is a case in which white box testing could easily miss the problem, because as we said any input will exercise the code. It's just by chance that we could reveal one that revealed the problem in the code. Conversely if we were to work, in a black box manner. Typically looking at the specification, we will select at least one odd, and one even input number to exercise all of the specified behavior. And we will find the problem right away. So these two examples are just very small examples, and they're kind of, you know, stretched. But these kind of issues occur on a much bigger scale and in much more subtle ways in real world software. And so what this examples do is to show you, how black box and white box tests are really complimentary techniques. So in the next two lessions we will explore these two types of techniques in detail. We will see different kinds of white box and black box testing. And we'll talk about their strengths and the mutations.

WHITE-BOX TESTING EXAMPLE

Specification: inputs an integer param and returns half of its value if even, its value otherwise

```
1. int fun(int param){
2.     int result;
3.     result = param/2;
4.     return result;
5. }
```