

1. In the previous lesson, we discussed the fundamental concepts behind software verification in general, and software testing in particular. In this lesson, we will discuss one of the two main testing approaches. [Black box testing, also called functional testing.](#) We will cover the main characteristic of black box testing, its pros and cons, and discuss some commonly used black box testing techniques. We will conclude the lesson with a practical exercise in which we will apply a specific black box testing technique to a real program. We will derive test cases for the program and assess how the use of a systematic approach, in contrast to a brute force approach, can help.

SOFTWARE TESTING

BLACK-BOX TESTING

2. As we said at the end of the previous lesson, black-box testing is the testing of the software when we look at it as a black box, as a closed box, without looking at it inside, without looking at the code. And there are several advantages in using black-box testing. So let me recap those advantages, some of which we already mentioned, and let me also expand on that a little bit. The first advantage of when I mentioned is that black box focuses on the domain, on the input domain of the software. And as such, we can use it to make sure that we are actually covering this domain, that we are actually covering the important behaviors of the software. [A second advantage is that black box testing does not need the code. What that means is that you can perform early test design. So you can start designing and writing your test cases, even before writing your code,](#) so that when the code is ready, we can test it right away. And that helps prevent a problem that is very typical in real life software development, which is getting an idea of the project and having no time to create the tests. In this way, we already have the tests, so we just have to run them. Another advantage is that black-box testing can catch logic defects, because it focuses on the description of what the software should do, and therefore on its logic. If we derive test cases from such description, then we can catch these kind of problems. And finally, black-box testing is applicable at all granularity levels, which means that we can use black-box testing in unit testing, integration testing, system testing, and so on. We can use it at all levels.

BLACK-BOX TESTING

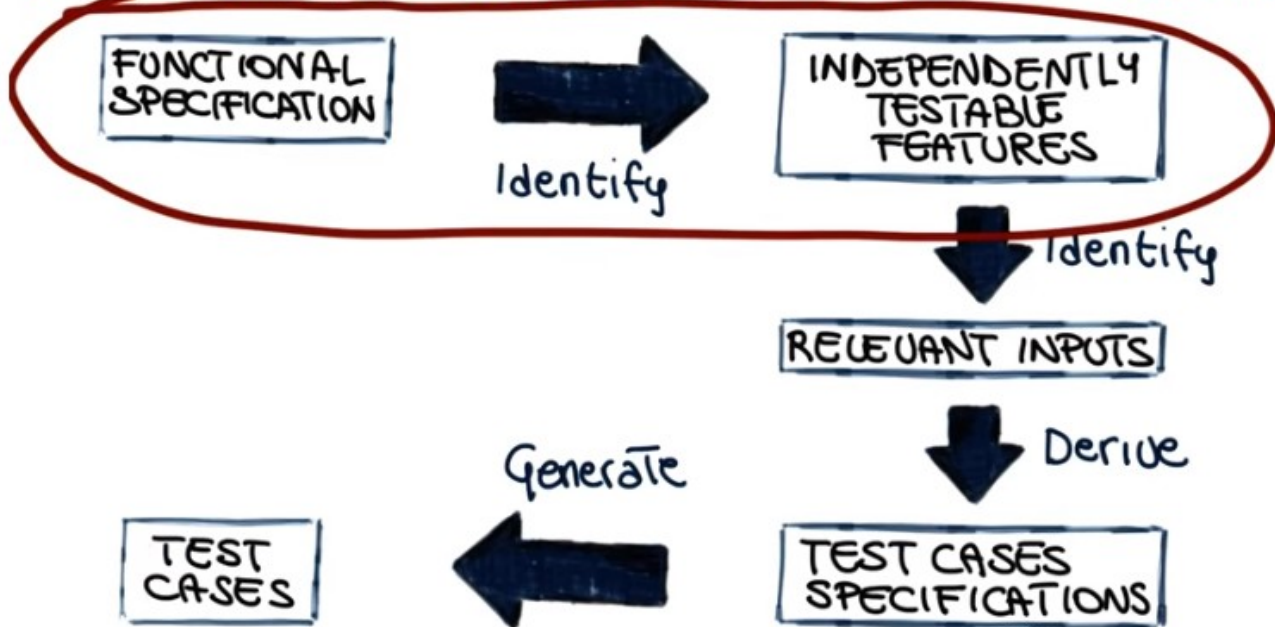


Advantages

- focus on the domain
- No need for the code
⇒ early test design
- Catches logic defects
- Applicable at all granularity levels.

3. So what is the starting point of black box testing? Black box testing start from a description of the software or as we call it, a functional specification. And the final result of black box testing is a set of test cases, a set of actual inputs and corresponding outputs that we can use to exercise our code and to try to find defects in our code. So the question is, how do we get from functional specification to test cases? Doing these derivations, so going from this description to a concrete set of tests, is a very complex analytical process. And normally brute force generation is not a good idea because it's inefficient and ineffective. What we want to do instead is to have a systematic approach to derive test cases from a functional specification. What a systematic approach does is to simplify the overall problem by dividing the process into elementary steps. In particular, in this case, we will perform three main steps. The first step is to identify independently testable features. Individual features in the software that we can test. And we're going to expand on each one of these steps in the next part of the lesson. The following step is once we have these independently testable features to identify what are the relevant inputs. So what are the inputs or the behavior that is worth testing for these features. Next once we have these inputs, we're going to derive test specifications. And test case specifications are description of the test cases that we can then use to generate actual test cases. And proceeding in this way, by this steps, has many advantages. It allows for the coupling different activities. It allows for dividing brain intensive steps from steps that can be automated, which is a great advantage. And also we will see, it allows you for monitoring the testing process. So to figure out whether your testing process is going as expected, for example, if you're generating too many test cases. Or you're generating the number of test cases that your amount of resources available allows you to run. So let's start by looking at the first step of this process in which our goal is to go from a Functional Specification to a set of features that we can test in the software. So what we want to do is to identify all of the feature of the software. And why do we want to do this? Well you know, in the spirit of breaking down the complexity of the problem, it does not make sense to just try to devise test cases for all the features of the software at once. For any non-trivial software, that's a humongous problem, and something that we cannot really handle effectively. A much better way is to identify independently testable features and consider one of them at a time when generating tests.

A SYSTEMATIC FUNCTIONAL-TESTING APPROACH



4. So, now I want to do a little quiz about identifying testable features. Let's consider this simple problem called printSum. We won't see the implementation because we are doing black-box testing. And all we need to know is that printSum takes two integers, a and b, and prints the sum of these two numbers. So what I want to ask, is how many independently testable features do we have here? Do we have one, two, three features? Or more than three features?



IDENTIFYING TESTABLE FEATURES

```
printSum(int a, int b)
```

How many independently testable features do we have here?

- ☒ 1
- ☐ 2
- ☐ 3
- ☐ > 3

5. Sum is a very simple program, that only does one thing, summing two number, adding two numbers. So the answer in this case, it's one. There's only one feature that we can test in PrintSum. So let's look at the slightly more interesting example.

6. Let's look at this spreadsheet. I'm pretty sure most of you are familiar with what a spreadsheet is and have used them before. So now I'm going to ask the same question, which is I'd like you to identify three possible independently testable features for a spreadsheet.

7. In this case there's not really a right answer because there's many, many features that you could identify in a piece of software as complex as a, a [spreadsheet](#). So I'm just going to give you three examples. So one could be the cell merging operation. So the operation in which we merge two cells in the spreadsheet. Another example could be chart creation, so I might want to test the feature that allows you to create charts in your spreadsheets. Yet another example could be the test of statistical functions, so the function that allows you to do various statistical calculations on the numbers in your cells. And as I said there's many, many more example that we could use. But [the key thing I want to convey here is the fact that there is no way you can look at a spreadsheet with all the functionality that it provides and just go and test it. The first step, what you need to do first, is to identify which ones are the pieces of functionality that I can test individually. So that's why this is the first step in black-box testing.](#)



IDENTIFYING TESTABLE FEATURES

Identify three possible independently testable features for a spreadsheet



[cell merging]

[chart creation]

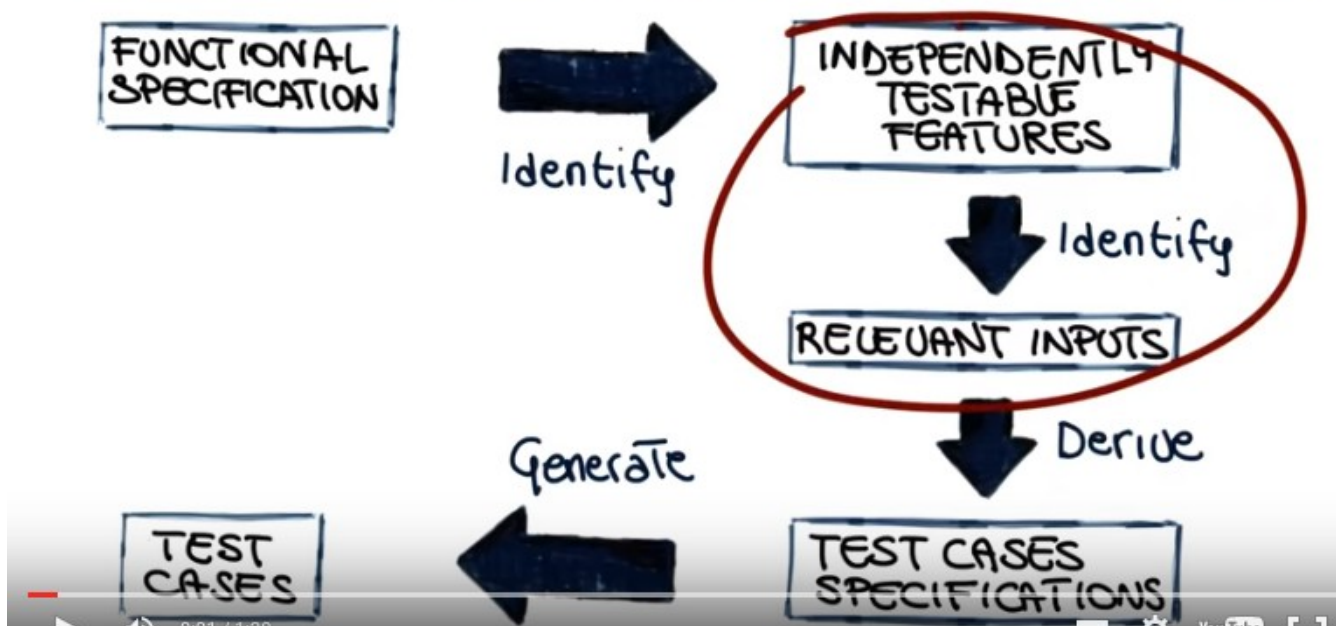
[statistical func.]

...

8. Once we have identified Independently Testable Features, [the next step is to identify the Relevant Inputs for each one of these features](#). And there are many ways to do that. So, what we're going to do, instead of looking at them all, is that we're just going to focus on two different ways of doing it. And they are fairly general ways. So, they are applicable to a number of situations. And in addition, what I will do, I will point you to other sources in which you can look at different ways of doing this in the class notes. [The problem of identifying relevant inputs for some Software or some feature of it is called Test Data Selection](#) and can be expressed as followed. Let's consider our software as usual we have our

Input Domain, which is the set of inputs for all the software. And again as usual, we have our Output Domain, which is the set of corresponding outlets for these inputs. So the question here is, [how can we select a meaningful set of inputs in my domain?](#) And of course corresponding outputs because we know that test cases are an input, plus an expected output. So how can we select interesting inputs for our software? So a set of inputs that, after we run them on the software, if the software behaves correctly, we'll have enough confidence that the software is correctly implemented. [So one possible idea is, hey, why don't we just test them all? We just do exhaustive testing.](#) We do all the inputs, nowadays we have powerful machines, we have a lot of computational power in the cloud. Why not just doing it? So to answer that question, what I'm going to do? I'm going to use another quiz.

A SYSTEMATIC FUNCTIONAL-TESTING APPROACH



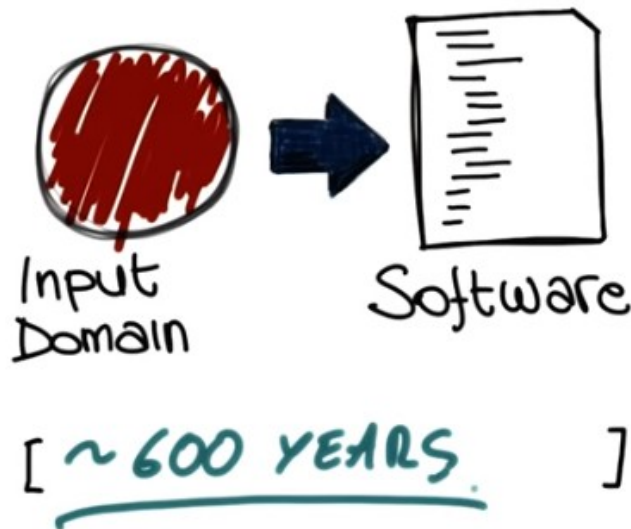
STRAW-MAN IDEA : EXHAUSTIVE TESTING !



9. So I'm going to ask you something. Which is if we consider again our function print sum, the one that takes two integers and prints the sum. How long would it take to exhaustively test this function? And this is a very simple one. There's just two inputs, right? So we can just enumerate them all. And put them throw them at the computer form, and wait for the results. How long would that take?



STRAW-MAN IDEA : EXHAUSTIVE TESTING !



How long would it
take to exhaustively
test the function

`print Sum(int a, int b) ?`

$$2^{32} \times 2^{32} = 2^{64} \approx 10^{19} \text{ TESTS}$$

1 TEST PER NANOSECOND
(10^9 TESTS / SEC)

$\Rightarrow 10^{10}$ SECONDS OVERALL

10. Okay, so now we're going to answer the question. So if we want to consider all these inputs, and run them all on the software, let's see how it will work. Let's assume that these are, 32 bit integers. So at this point what we will have is, a number of combination, which is 2 to the 32, times 2 to the 32. They're two integers. This is equal to 2 to the 64, which in turn, is more or less equal, to 10 to the 19. So 10 to the 19 is the number of tests that we need to run to cover the whole domain. Now let's assume that we can run one test per nanosecond. So what that means is that we can run 10 to the 9 tests per second, and that's a lot. If we do the math, that results in 10 to the 10 seconds over all, because we have 10 to the 19 tests, we could run 10 to the 9 tests per second so, we do the math, and we can run all these tests in 10 to the 10 seconds. And what that corresponds to, it's about 600 years, so a lot of time. So even for such a simple problem, a problem that takes two integers and adds them, it will take more than 500 years to test it exhaustively. So the bottom line here is that we just can't do exhaustive testing.

11. So then maybe what we can do is just to pick our test inputs randomly so to do what is called random testing. And what that means is that we pick the inputs to test just as we pick a number by rolling a set of dice randomly. And this will have several advantages. First, we will pick inputs uniformly. So if we use a uniform distribution as the basis for our random testing, we will make no preferences. In other words, all inputs will be considered equal, of equal value. And what that means in turn, is that random testing eliminates designer bias. So what does designer bias mean? Designer bias is the problem of making the same assumption, when we read the specification and we interpret it and when we develop test cases. Which means that the developer might develop code, assuming a given behavior of the user. And we may write tests, making the same assumptions. And the problem, of course, is even worse if it's the same person that develops the code and writes the test cases. With

random testing, the problem is gone, because we just pick randomly what our inputs will be. So why not do in random? The problem is that when testing, we are looking for a needle in a haystack. Actually, multiple needles in multiple haystacks, if we want to be precise. So, random approaches are not necessarily the best way to go about it, because we might just be looking in all the wrong places. So let me show you this, using a different representation for the haystack. What I'm showing here is a grid, and imagine this grid just expanding indefinitely outside the screen, and this grid represents the domain for the program, so each box in the grid, each square in the grid, it's a possible input. So what happens with bugs is that bugs are very scarce in this grid. Maybe there is a bug here, so that means that there is a bug, than an input, in this point we'll reveal. And maybe there is another bug that will be triggered by an input over here. So imagine this spread out over this infinite grid. Its very unlikely that just by picking randomly that we will be able to get to these two points. Fortunately not all is lost, there is a silver lining. So we need to look a little more in depth into this grid.

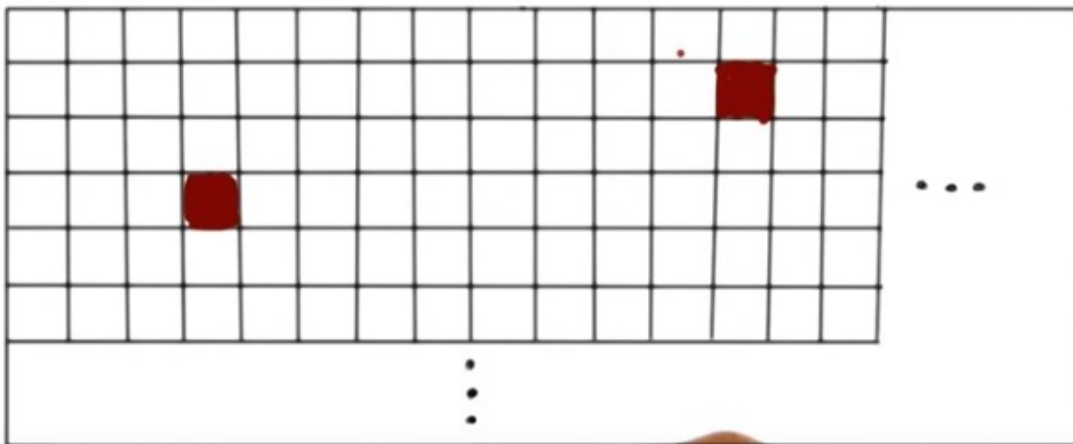
RANDOM TESTING



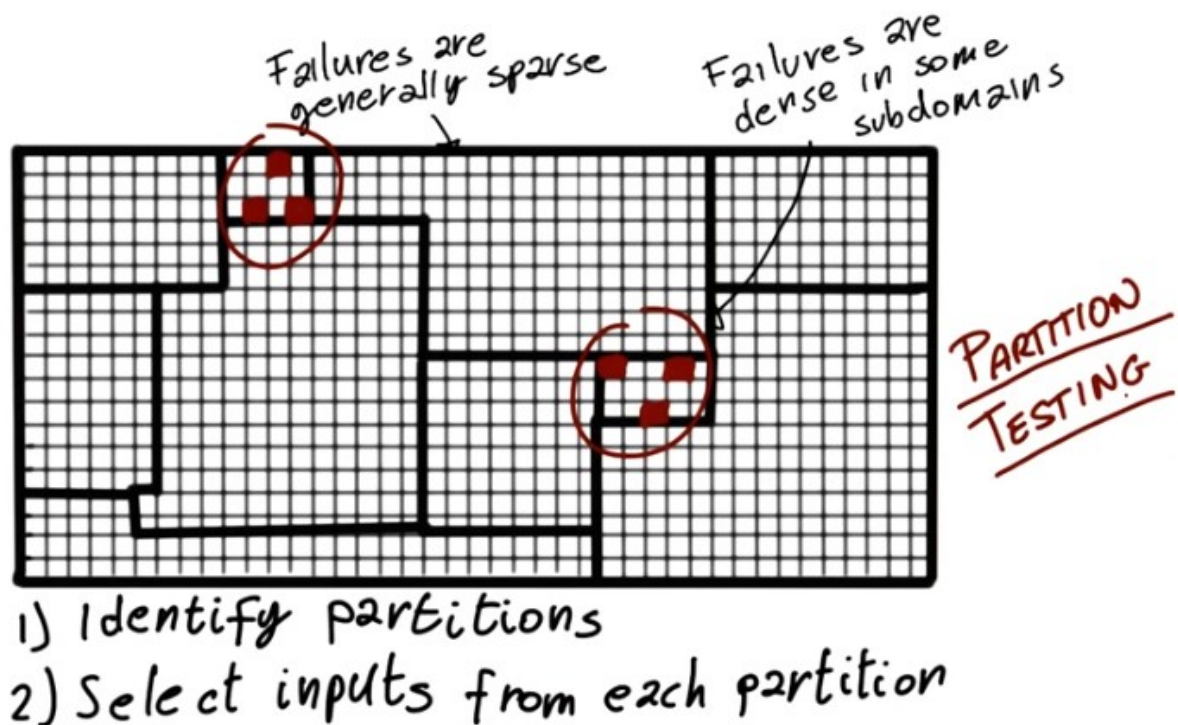
- pick inputs uniformly
- all inputs considered equal
- no designer bias



SO WHY NOT RANDOM ?



12. So let me use a slightly expanded version of this grid. Although we're indeed looking at a needle in a haystack. And failing inputs are generally sparse, very sparse, in the input domain. However, they tend to be dense in some parts of the domain. Like here or here. So how can we leverage this? The fact that the failures are dense in some subdomains? As it turns out, [the domain is naturally split into partitions](#). Where partitions are areas of the domain that are treated homogeneously by the software. And this is what happens, that [normally, failures tend to be dense in this partitions](#). So the way to leverage this characteristic of failures, is that we don't know want to pick inputs randomly, in the input domain. Just here and there. Rather we want to do two things. First we want to identify partitions of our domain. And second we want to select inputs from each partition. And by doing so, we can dramatically increase our chances to reveal faults in the code. So the name that is normally used for this process, is partition testing.



13. So let's look at how this will work with [an example](#). I'm going to use this simple program that takes two inputs. The first input is a string, `str`, and the second one is an integer, `size`. And the problem is called `split`. And as the name says what it does is to take this string, `str`, and split it into sub string, into chunks of `size` characters each. So how do we identify some possible partitions for this program? If [we consider the input size](#), we can identify three neutral partitions which are `size` less than 0. For example, we want to test how the program behaves. But if we pass an incorrect `size`, `size` equal to 0, which is also a partition. In this case, a partition with a single element. And the third case is `size` greater than 0, which I will consider to be kind of the standard case. And actually let me do a, you know, slight aggression so when I was talking about designer bias. So [this is a case in which designer bias might not make you think of using `size` less than 0 because you read the spec. And you sort of assume that the `size` will be positive](#). Whereas the right thing to do when we test is to consider the complete domain rather than just parts of it. [So now let's look at string, `str`, and let's see what kind of sub domains we could identify for this parameter. And notice another important aspect here is that we treat each different part of the input independently](#), which also helps breaking down the problem. One interesting sub domain is the domain that includes all the strings whose length is less than `size`. So all the strings

that will not be displayed. Another subdomain is all the strings with length which is between the value of size and twice the value of size. A third subdomain is the one including all the strings whose length is greater than twice the value of size. And we can continue and identify more and more subdomain. The key thing here is that we have to do that based on the domain. So we need to adapt what we just did here based on, on the specific domain involved and on the type of data in this domain. So at this point we said that there were two steps. One was to identify the subdomains and the second one was to pick values in this subdomain. The values that we'll actually use for the testing. In this case, we do not want to just pick any value. Rather we want to pick values that are particularly interesting, particularly representative. So what does that mean? Well, we're going to do that based on an intuitive idea.

EXAMPLE

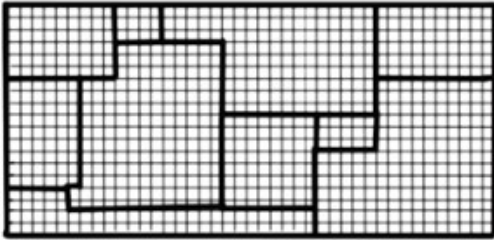
Split (string str, int size)

Some possible partitions:

- $size < 0$
- $size = 0$
- $size > 0$
- str with length $< size$
- str with length in $[size, size \times 2]$
- str with length $> size \times 2$
- ...

14. So let's go back again to our domain, with all the sub-domains identified. And the basic idea, or the intuitive idea I was talking about, is that **errors tend to occur at the boundary of a domain, or a sub-domain. Like in this case. And why? Because these are the cases that are less understood by the developers. Like for example, the last iteration of a loop, or a special value like zero for integers.** So if this is true, what we want to do is to select inputs at these boundaries. And this is complementary to partition testing, in the sense that partition testing will identify the partitions in which we want to select inputs, and boundary testing. So the selection of boundary values will help select inputs in these partitions.

BOUNDARY VALUES



Basic idea

Errors tend to occur at the boundary of a (sub) domain

⇒ Select inputs at these boundaries

15. So now let's go back to our split example. Let me rearrange things a little bit to make more room. So now I'm going to put the domains for size and for strength one next to the other. So let's look at what some possible inputs will be for the sub domains that we identified when we use the idea of selecting input of the boundary. If we look at the first subdomain, size less than zero, one reasonable input is, size equals to -1, because -1 is the boundary value for the domain of the integers less than zero. If we look at the third subdomain, possibly interesting case is the one of size of equal to 1, for the same reasoning that we used for the previous subdomain, for size less than zero. And, let's try to select another one for this subdomain, for the integers greater than zero. If there is a concept of maximal integer, we can select that one as our boundary value. And of course we could select much more, but this is just to give you an idea. Other possible inputs. One interesting example for the first one, string with length less than size will be a string with length size minus one. Again this is the boundary value for this domain. And we could continue in this way like for example selecting a string who's length is exactly size as a boundary value for this other domain. Instant one, and we look back actually to this example and look at it in a more extensive way when we actually talk about a specific method for doing this kind of process.

EXAMPLE

`split(string str, int size)`

Some possible partitions:

- | | |
|------------|---------------------------------------|
| - size < 0 | - str with length < size |
| - size = 0 | - str with length in [size, size x 2] |
| - size > 0 | - str with length > size x 2 |

Some possible inputs:

- | | |
|-----------------|-------------------------------|
| - size = -1 | - string with length size - 1 |
| - size = 1 | - string with length size |
| - size = MAXINT | |

16. Now, let's go back to our systematic functional testing approach and all the steps in this process. So far we've seen the first step and the second step. Now we're going to look at this step in which, once we have identified the values of interest, we derive test case specifications for these values, or using these values. And the test case specification defines how the values should be put together when actually testing the system. And test case specification describe how these values should be put together when testing the system. So let me go back one more time to our split program, so that we can use the information that we already computed. At this point what we have is some possible inputs for "string," our first parameter, and for "size," our second parameter. And we want to put them together, to generate the description of what the test case should be. So let me once more rearrange this a little bit. I first remove the description of the subdomains, because we won't use them in this step. And I moved out the set of all our possible inputs, that we're going to combine to create the test case specification. And one possible way of doing that is simply to combine the values for the first parameter, and the values for the second parameter. So the Cartesian product. So if we do that, what we will obtain is, for example, if we consider the first possible input, size is equal to minus 1, we can combine it with these two possible inputs for string, and we will get size is equal to minus 1 string with length minus 2, or size is equal to minus 1 string with length minus 1. And we'll go back in a second to see what this means. Now if we consider the second possible value for size, size is equal to one, we also have two cases so the first one in this case that will be considered a string with length zero. So the antistring. And we can continue combining this value, but one thing I want to point out is that if we just go in this straight forward and brute force sort of way, we will obtain many combinations that don't make any sense, like for example, this combination which doesn't make any sense because we can not create the string with length -2 (因為 size=-1, string with length size-1 即-2). Similar for this combination, because then by the same token, we cannot raise things with length minus 1. And so there's a lot of cases that we will have to eliminate afterwards. So what we're going to see in a few minutes is a possible way in which we can avoid producing these meaningless cases. And at the same time, keep under control, the number of test cases that we generate. So let's go back for the last time to our steps for systematic functional testing. What we just did was to derive test case specification from a set of relevant inputs. The following step is to use these test case specifications to generate actual test cases. And this is

normally a fairly mechanical step in the sense that we just have to instantiate what is in the test case specification as actual test cases. And it's really dependent on the specific type of partitions and values identified on the specific context. So instead of looking at that here in the, in the abstract, I'm going to show you with an example later on, in the lesson.

EXAMPLE

Split (string str, int size)

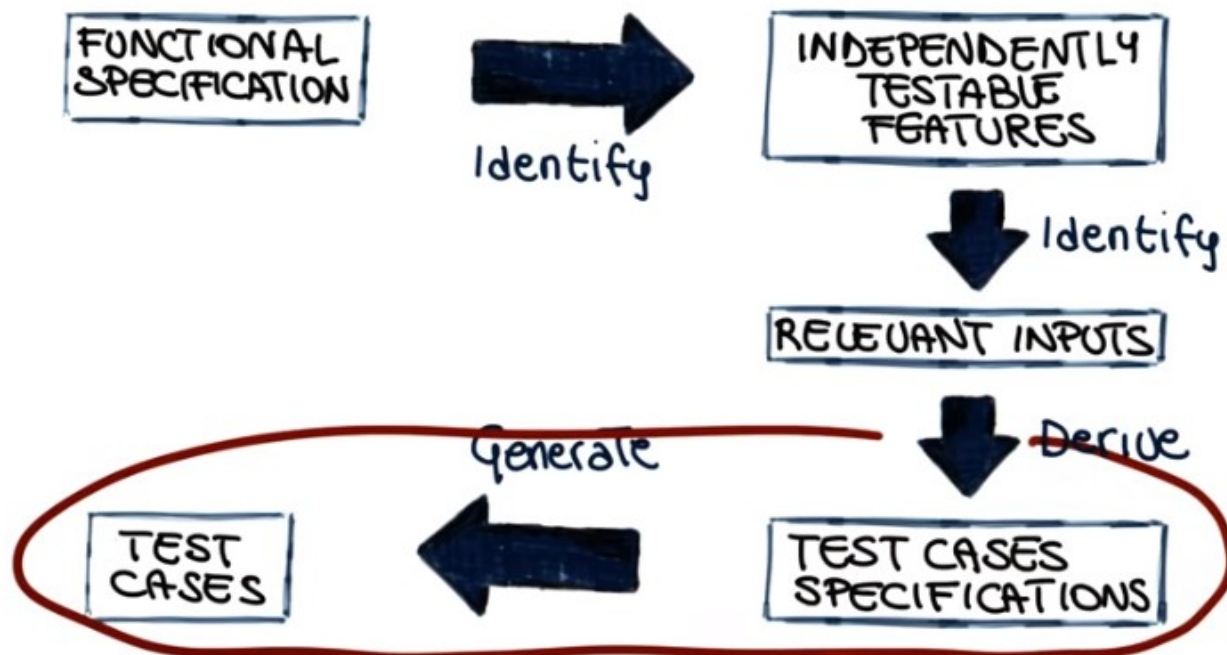
Some possible inputs

- size = -1
- size = 1
- size = MAXINT
- string with length size - 1
- string with length size
- ...

Test case specifications

- ~~- size = 1 str with length 2~~
- ~~- size = 1 str with length -1~~
- ~~- size = 1 str with length 0~~
- ...

A SYSTEMATIC FUNCTIONAL-TESTING APPROACH



17. What we will discuss next is a specific black-box testing approach. So a specific instance of the general approach that we just saw. And this approach is the category-partition method (category: 種類), and was defined by Ostrand & Balcer in 1988 in an article to the peer [UNKNOWN] communication of the ACM. So this is a method for 「going from a specification (指 functional specification, 不是 test cases specification), a description of the system, to a set of test cases」 like any other black-box testing approach by following six steps. So let's look at what these steps are. The first step is to identify independently testable features and this is a step that we are familiar with because its exactly the same step that we performed in the generic black box testing approach that we just discussed. The second step is to identify categories. Then the next step is to partition categories into choices. Identify constraints among choices. Produce and evaluate test case specifications. And finally, the sixth step is to generate test cases from test case specifications. So two of the key elements in these six steps are the two that give the name to the technique so the identification of the categories and the partition of these categories into choices. What we're going to do next is to go and look at each one of the steps independently, except for the first one. Because we're already familiar with that step, and this method doesn't really add much to it.

A SPECIFIC BLACK-BOX TESTING APPROACH THE CATEGORY-PARTITION METHOD

[Ostrand & Balcer, CACM, June 1988]





1. Identify independently testable features
2. Identify categories
3. Partition categories into choices
4. Identify constraints among choices
5. Produce/Evaluate test case specifications
6. Generate test cases from test case specifications



(以上的六步只有 2,3,4 跟之前不同)

18. In the second step of the category partition technique, the goal is to Identify Categories. Where categories are characteristics of each input element. So let me illustrate what that means using an example. And to do that I'm going to use again the example of the split program, as we are already familiar with it and we kind of already played with it. When we were talking about the generic black box approach. So let me bring back the program, and let me remind you that what the program does is to take two inputs, a string and the size, and it breaks down the string into chunks, whose length is size. If we look at the split program there are two input elements, str and size so we going to identify categories for these two. So starting from str, what are the interesting characteristics of the string? In, in this step you're going to use your domain knowledge, your understanding of what a string is, and for example we might identify the length of the string and the content of the string as the two main characteristics that we want to focus on. If we now move our focus to size, the only characteristic I can really think of for an integer is its value. So that's what I'm going to mark here. So at the end of the step what we have is that we have two categories. So two interesting characteristics for the string input str, which are the length and the content. And one category for the integer input size which is its value. And notice that there's not only one solution. So there's not only one possibility. So that the specific characteristics that you will identify are somehow subjective. But the important point is that you identify characteristics that are meaningful and they sort of cover the main aspects of the inputs, which is the case for the categories that we've identified in this example.

IDENTIFY CATEGORIES

Characteristics of each input element

Example: `split(string str, int size)`

Input str
- length

Input size
- value

- content

19. Now we move to the next step, which involves partitioning the categories that we just identified into choices. And these choices are the interesting cases for each category. So the interesting subdomains for each one of these categories. So once more, let's look at that using our example, the split program. So let's start by considering length. What are the interesting cases when we think about the length of a string? Some of those we already saw, one interesting case is the case of the length of size zero, so a string with no characters. Another interesting case is the one in which the length of the string is size minus one, so the string is just one character short of the size at which it will be cut by the split program. And we can continue along these lines, so we will select size, size plus one, size twice the value of size minus one, and so on and so forth. But even without listing all of those, I'm sure you get the idea of what it means to identify these interesting cases. Let's see the movements that are considering the content. So without the interesting cases when we think about the content of the string. So possible interesting case is the string that contains only spaces. Why? Well maybe because a split is written spaces in a special way. Similarly a string that contains special characters, like non printable characters, like tabulation characters, new line might also be an interesting case, something that we want to test. Also in this case we could continue and go on and on. So basically here you just want to put all the interesting cases that you can think of when you consider the content of a string. Now let's move to the value as the next category. So the value of the input size. And here we might want to consider a size zero, special case, a normal situation, like size greater than zero, another special case, size less than zero or maxint. And these are, if you remember, I accepted the cases that we consider when we look at this example, before. And also here we can continue and go on and on. So, at the end of the step, what we have is a set of interesting cases for each one of the categories, and now we can start to think about how we want to combine them.

PARTITION CATEGORIES INTO CHOICES

Interesting cases (subdomains)

Example: `split(string str, int size)`

Input str

- length

- 0

- size - 1

- ...

- content

- spaces

- special characters

- ...

Input size

- value

- 0

- > 0

- < 0

- MAXINT

- ...

20. Something that we saw when we were looking at the split program before is that if we just combine the interesting values that we identify, we might end up with a lot of cases. And I mentioned that we, we're going to look at some way of addressing that problem. And this is exactly what happens in the next step of the category partition method, in which we identify constraints among choices. And why do we identify these constraints? We do that to eliminate meaningless combinations of inputs. If you remember, for example, we had the case in which we were trying to create a string with a size less than 0, which doesn't make any sense. And very importantly, we also do that to reduce the number of test cases. Because every time we constrain one of the possible choices, we eliminate possible test cases, so we can use it to keep under control the number of tests that we generate. There are three types of properties. The pair property...if, error properties, and properties of type single. So we're going to look at what these properties mean, using, once more, our example of the split program. In particular, we're going to use some of the choices that we identified earlier. So let's look, for example, at choice 0, for category length of the string. All we can say is that, if the length is 0, this defines a special property of the string. And that was specified in this way by saying that this identifies property zerovalue. So every time that we use this choice, zerovalue is defined. At this point, we can use this to exclude some meaningless combinations. For instance, consider special characters. If we have a string of length 0, which means a string with no characters. Obviously, there cannot be special characters. So, considering this combination will just be a waste of time. So what we do is that we specify next to this choice, that we will only consider this if length is not 0. And we do this by saying that we consider this only if not zerovalue. So, if zerovalue is not defined. So this pair is an example of a property...if case. Define a property and use that property. Now let's look at a case in which we might want to use an error property. For instance, when we look at the category value for the input size, the choice value less than 0 is an erroneous choice. So it's a choice that we selected to test a possibly erroneous situation, so we can mark this as an error property. And what that means is that when generating a combination of choices, we will consider this only once because we assume that we just want to test this error

condition once. Finally, the single property is a property that we use when we want to limit the number of test cases. And it's similar as an effect to error. It just has a different meaning. So what we do when we use the single property is that we're saying that this choice, we want to use in only one combination. So don't combine it multiple times. And that, of course, given the combinatorial nature of the problem, cuts down dramatically the number of test cases. And we might use, for instance, the single property for maxint, which means that we will only have one test case in which the size is equal to maxint. We're actually going to see a demo on this topic so we'll have more chances of seeing how properties work in practice and how good they are at eliminating meaningless combinations and at reducing the number of test cases.

IDENTIFY CONSTRAINTS AMONG CHOICES

To eliminate meaningless combinations

To reduce the number of test cases

Three types: PROPERTY ... IF, ERROR, SINGLE

Examples

Input str

- length

- 0

- content

- special characters if ! zero value

PROPERTY zero value



Input size

- value

- < 0 ERROR

- MAXINT SINGLE

上圖中的 PROPERTY...IF 是一個組合。

上圖中的 PROPERTY zero value 是 define 了 zero value 這個 property, 然後在 if !zero value 中用到

21. Before getting to our demo, we still have two steps to consider. The first step corresponds to the identification of the test case specifications in our general systematic approach. And in fact, it's called produce and evaluate test case specifications. This is a step than can be completely automated given the results of the previous steps. And the final result of this step is the production of a set of test frames. Where a test frame is the specification of a test. Let me show you an example of this. What we are looking at here is a test frame for the program split. Test frames are normally identified by a sequence number. But in this case we are looking at the 30th six test frame. And what they do is simply to specify the characteristic of the inputs for that test. In this case, since we have two inputs, we have two entries, the first one for string str tells us that the length of the string has to be size minus 1, and that the string has to contain special characters. And for size, it tells us that the value of size has to be greater than zero. As the title says, this step is meant to produce but also evaluate the case specification. What does it mean to evaluate? One of the advantages of this approach is that we can easily use it to assess how many test frames and therefore how many test cases we will generate with the current least of categories, choices and constraints. And the beauty of this is that if the number is too large we can just add additional constraints and reduce it. And given then the step is automated we just add constraints push a button and we get our new set of test frames. And again we can have a wait it either go hat or add more constraints if we need to further reduce it and this is something else that we will see in our

demo.

PRODUCE AND EVALUATE TEST CASE SPECIFICATIONS

Can be automated

Produces test frames

Example

Test frame #36

input str

length : size - 1

content : special characters

input size

value : > 0

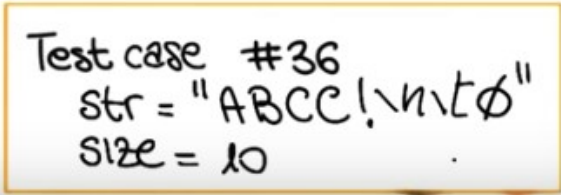
22. So we get to the last step of the technique in which once we have generated test case specifications. We create test cases starting from this specifications. This step mainly consists in a simple instantiation of frames and it's final result is a set of concrete tests. For our example, test frame number 36 that we just saw, this will be the resulting test case, which has the same ID, so that we can track it and will specify to concrete values, not just the specification for the input elements. So string STR will have this value. And the integer size will have this value. And these two values satisfy what this test case specification was. Which was, having a string contain special characters. Here, we have two special characters, like the new line and the tab. And, we have a size which is greater than zero, in particular, okay? And this is a test case that we can actually run on our code. That we can run on the split program. So, to summarize, we perform six steps in which we went from a high level description of what the program does, to a set of concrete test cases. And this is one of those test cases. So what, what we're going to do next, we're going to do a mini-demo, in which we do this for real. We take the program, we identify categories, choices, constraints, and we actually generate test frames and then test cases.

GENERATE TEST CASES FROM TEST CASE SPECIFICATIONS

Simple instantiation of frames

Final result : set of concrete tests

Example



```
Test case #36  
str = "ABCC!\n\tø"  
size = 10
```

23. In this demo, we're going to do exactly what we did just now in the lesson. We're going to use the category partition method to go from a high-level description of a piece of software or a program to a set of test cases for that program. To do that, we're going to use a simple tool. So I'm going to show you here the tool that is called a tsl generator right here. This tool is available to you, so you can look in the class notes to see information on how to download it. And together with the tool, we are also going to provide a manual for the tool, and a set of files that I'm going to use in this demo. So you should be able to do exactly what I'm doing. So again, all of those are available from the class notes. So specifically, today we're going to write test cases for the grep program. So in case you're familiar with the grep utility, this is a simplified version of that utility. So basically the grep utility allows you to search a file for the occurrences of a given pattern. So you can invoke it, as it's shown here in the synopsis, by executing grep, the pattern that you're looking for, and the filename in which you want to look for the pattern. And let me read the description of the grep utility. The grep utility searches files for a pattern and brings all lines that contain that pattern on the standard output. A line that contains multiple occurrences of the pattern is printed only once. The pattern is any sequence of characters. To include a blank in the pattern, the entire pattern must be enclosed in single quotes. To include a quote sign in the pattern, the quote sign must be escaped, which means that we have to put a slash in front of the quote sign. And in general, it is safest to enclose the entire pattern in single quotes. So this is our high level description for the program, for the softer system, that we need to test. So now let me show you what a possible set of categories and partitions could be for this program. So what I have here is a file, a textual file, which contains all the categories and partitions for the elements that are relevant for my program. In particular, when we look at the file, we can see that the file can be characterized by its size. And in this case, I've got two choices. The file can be empty or not empty. The second characteristic of the file that I'm considering is the number of occurrences of the pattern in the file. And I'm considering that the pattern might not occur in the file or it might occur once, or multiple times. I'm not going to go through the rest of the file because we already covered how to apply the category

partition method in the lesson. So if you had doubts about that, about the method and how to apply, you might want to go back and watch again the lesson. What I want to show you here is how you can go from this information that you have here, that we have derived by applying the, the first steps of the method, to a set of test frames, and then, a set of test packs. So to do that we're going to use the tool that I just mentioned. So let me bring back my terminal. So first of all, let's see how we can run the tool. So you have a manual that will explain all the details on how to build the file that we're going to feed the tool. So what is the format and so on. Here I'm just going to see how I can run the tool. So first of all, let me point out that this was developed together by professors from the University of California Irvine and Oregon State University. And as you can see, we can run TSL generator and specify that we want to see the main page. So in this case if we run it this, this way, you'll have some basic information on how to run the tool. And from the main page you can see that you can specify the minus c flag and in this case the TSL generator will report the number of test frames generated without writing them to output (即將有多少個 test frame 只顯示到屏幕上). For example, you might want to use this as we will do to see how many tests that you will generate with a current set of category partitions and choices. The minus s option will bring the result of the TSL generator on the standard output. And finally, you can use minus o to specify an output file, where to put the output of the program. So let's at first run our TSL generator by specifying the minus c option and by bypassing our current set of category partitions and choices. Okay, so let me remind you that what the, the tool will do is what we will do manually. Otherwise, which is to combine all these choices so as to have one test case for each combination. So if we do that, you can see that the tool tells us that we will generate 7776 test frames in this case. And this seems to be a little too much for a program as small as the one that we are testing. And assume for instance that we don't have the resources to run this many test cases for, for the grep program. In addition, consider that in this case, we're computing all possible combinations of choices. And there's going to be some combination that do not make sense as we discussed in the lesson. So what we might want to do in this case is to go back to our spec and start adding constraints to eliminate this meaningless combination. So I'm going to show you the result of doing that. And I'm going to show you a few examples. For example here, when the file is empty, I'm going to define this property empty file. And how am I going to use this property? Well for example here, it doesn't make sense to consider the case in which we have one or many occurrences of the pattern in the file if the file is empty. Therefore I'm going to tell the tool that it should consider this specific choice only if the file is not empty, only if empty file is not defined. And that will skip, for example, all of the combinations in which the file is empty. And I'm trying to generate the test case that has one occurrence of the pattern in the file, which is simply not possible. For another example, in case I have an empty pattern, I define the property empty pattern. And then I avoid the choices that involve the pattern in case the pattern is empty. because, for example, I cannot have quotes in a pattern that is empty. For example, it doesn't make sense to have blanks. So, one or more blanks if the pattern is empty. So I'm going to specify again that this choice should be considered only if we don't have an empty pattern. And so on and so forth. So now after I edit these constraints, I can go back and compute again the number of test frames and therefore the test cases that will be generated with these constraints. So let me go again to my terminal. Okay, so now I'm going to run my TSL generator again, and I'm going to run it on the second version of this file. And you can see that I reduced the, the number of test frames from about 7800 to about 1700. So it's quite a, quite a big reduction by eliminating all these combinations that do not make sense. But let's assume again that we want to reduce this further so that we don't want to generate those many test frames and therefore test cases. So what can we do? We go back to our spec. And in this case, we start adding error constraints. So if you remember what we said in the lesson, error constraints are constraints that indicate a choice that has to do with an erroneous behaviour. For example, an erroneous input provided to the problem. So here for instance, we're indicating the presence of incorrectly enclosing quotes as an error choice. Same thing if there's no file corresponding to the name that we provide to the tool, we say that this corresponds to an error. So how is the tool going to use this

information? It uses this information by producing only one combination that involves error choices, instead of combining them with other choices. So let's see what happens after we added this error constraints. So we go back to our console once more. And in this case, we want to run the TSL generator with the version of the, of my file that contains the area of constraints. And again, I reduce quite a bit the number of test frames. So now I have only 562 test frames that will be generated by using the file that I provided. So for the last time, let's assume that we really want to cut down the number of test frames or the number of test cases. So once more, we go back to our file, and at this point what we can add is the final type of constraints that we have, which are single constraints. And single constraints are basically indicated choices that we don't want to combine with other choices. So they have the same effect of the error constraints, but they have a different meaning, so they do not indicate choices that corresponds to an error. In other words, I can use a single constraints to identify choices that I want to test only once. So for example in this case, I might decide that I want to have only one test frame that tests my program with a file being empty and I can do the same for other choices. So basically I can continue adding this single constraint until I get down to the number of test frames and therefore the number of test cases that I want. So now let's go back once more to our console. And so now if we run using this file as input, you can see that we have 35 test frames generated. So this is a fairly low number of test cases, so we might decide that we want to go ahead and write these test frames to a file. So now let's open this file that we just generated. And as you can see here, I have exactly 35 test frames, as expected. Some of those correspond to the single and error cases. So in this case, the only choice that I have indicated is the one that corresponds to the single or error constraint. What is for the other ones? I actually have the whole test spec. So let's pick one just to give you an example. In this case, that's frame number 15 that will correspond to test case number 15. And here you can see that we have all the information. So this is a test specification. All the information that we need to generate the corresponding test. We know that we need a file that is not empty. That we need to have one occurrence of the pattern in the file. One occurrence of the pattern in one line. The position of the pattern in the file can be any position. The length of the pattern must be more than one character. The pattern should not be enclosed in quotes. There should be one white space, one quote within the pattern, and finally the file that would pass through the program should exist. So the file should be present. So I can easily transform all of this into an actual test case. And notice that even though we're not, we're not going to do it here. In cases like this, it might even be possible to automatically generate the test cases from the test specifications because, here for example, here it should be relatively straight forward to parse these test specifications and generate test cases accordingly. So, just to summarize, what we have done is to go from one high-level description of a program to a set of categories, partitions, and choices for that program. Then we have combined them in different ways, adding more and more constraints to reduce the number of combinations until we ended up with the right number of test cases, so the number of test cases that we were fine generating. We generated the corresponding test specifications. And at that point, we could just go ahead, generate the test case, and test our application. So, and you can see how this can result in a much more thorough testing of your application. Because instead of reading this description and just trying to come up with test cases for it, we can break down the process in steps that are easy to perform individually. They can be automated as much as possible. And they will end up with a set of test cases that will test all the interests and aspects of your application.

Lab: Category-Partition Method

NAME

grep - search a file for a pattern

SYNOPSIS

grep <pattern> <filename>

DESCRIPTION

The **grep** utility searches files for a pattern and prints all lines that contain that pattern on the standard output. A line that contains multiple occurrences of the pattern is printed only once.

The pattern is any sequence of characters. To include a blank in the pattern, the entire pattern must be enclosed in single quotes ('). To include a quote sign in the pattern, the quote sign must be escaped ('\'). In general, it is safest to enclose the entire pattern in single quotes '...'.

```
# File
Size:
  Empty.
  Not empty.
Number of occurrences of the pattern in the file:
  None.
  One.
  Many.
Number of occurrences of the pattern in one line:
  One.
  Many.
Position of the pattern in the file:
  First line.
  Last line.
  Any.

# Pattern
Length of the pattern:
  Empty.
  One.
  More than one.
  Longer than the file.
Presence of enclosing quotes:
  Not enclosed.
  Enclosed.
  Incorrect.
Presence of blanks:
  None.
  One.
  Many.
Presence of quotes within the pattern:
  None.
  One.
  Many.

# Filename
Presence of a file corresponding to the name:
  Not present.
  Present.
```

The above figure: what I have here is a file, a textual file, which contains all the categories(跟 File 相對應的 categories 即 Size, Number of occurrence of...) and partitions (如跟 Size 相對應的 partitions 即 Empty, Not empty) for the elements (即 File, Pattern, Filename, 它們可以理解為 grap 的參數) that are relevant for my program.

```

1.spec.plain<2> 1 2.spec.const 2 3.spec.consterror 3 4.spec.constsingle 4
# File
Size:
  Empty. [single][property emptyfile]
  Not empty.
Number of occurrences of the pattern in the file:
  None. [single][if !emptyfile] [property noOccurrences]
  One. [if !emptyfile]
  Many. [if !emptyfile]
Number of occurrences of the pattern in one line:
  One. [if !noOccurrences && !emptyfile]
  Many. [single][if !noOccurrences && !emptyfile]
Position of the pattern in the file:
  First line. [single][if !emptyfile]
  Last line. [single][if !emptyfile]
  Any. [if !emptyfile]

# Pattern
Length of the pattern:
  Empty. [single][property emptypattern]
  One. [single]
  More than one. [property patternlengthgt1]
  Longer than the file. [single]
Presence of enclosing quotes:
  Not enclosed. [if !emptypattern]
  Enclosed.
  Incorrect. [error]
Presence of blanks:
  None.
  One. [if !emptypattern]
  Many. [if !emptypattern && patternlengthgt1]
Presence of quotes within the pattern:
  None.
  One. [if !emptypattern]
  Many. [single][if !emptypattern && patternlengthgt1]

# Filename
Presence of a file corresponding to the name:
  Not present. [error]
  Present.

```

```

1.spec.plain<2> 1 2.spec.const 2 3.spec.consterror 3 4.spec.constsingle 4 4.spec.constsingletsl 5
Test Case 1 <single>
  Size : Empty

Test Case 2 <single>
  Number of occurrences of the pattern in the file : None

Test Case 3 <single>
  Number of occurrences of the pattern in one line : Many

Test Case 4 <single>
  Position of the pattern in the file : First line

Test Case 5 <single>
  Position of the pattern in the file : Last line

Test Case 6 <single>
  Length of the pattern : Empty

Test Case 7 <single>
  Length of the pattern : One

Test Case 8 <single>
  Length of the pattern : Longer than the file

Test Case 9 <error>
  Presence of enclosing quotes : Incorrect

Test Case 10 <single>
  Presence of quotes within the pattern : Many

```

```
1.spec.plain<2> 2.spec.const 3.spec.const.error 4.spec.const.single 5.spec.const.single.ts1
Number of occurrences of the pattern in one line : One
Position of the pattern in the file : Any
Length of the pattern : More than one
Presence of enclosing quotes : Not enclosed
Presence of blanks : One
Presence of quotes within the pattern : None
Presence of a file corresponding to the name : Present

Test Case 15 (Key = 2.2.1.3.3.1.2.2.2.)
Size : Not empty
Number of occurrences of the pattern in the file : One
Number of occurrences of the pattern in one line : One
Position of the pattern in the file : Any
Length of the pattern : More than one
Presence of enclosing quotes : Not enclosed
Presence of blanks : One
Presence of quotes within the pattern : One
Presence of a file corresponding to the name : Present

Test Case 16 (Key = 2.2.1.3.3.1.3.1.2.)
Size : Not empty
Number of occurrences of the pattern in the file : One
Number of occurrences of the pattern in one line : One
Position of the pattern in the file : Any
Length of the pattern : More than one
Presence of enclosing quotes : Not enclosed
Presence of blanks : Many
Presence of quotes within the pattern : None
Presence of a file corresponding to the name : Present

Test Case 17 (Key = 2.2.1.3.3.1.3.2.2.)
Size : Not empty
Number of occurrences of the pattern in the file : One
Number of occurrences of the pattern in one line : One
Position of the pattern in the file : Any
Length of the pattern : More than one
Presence of enclosing quotes : Not enclosed
```

Steps:

Go to the folder CategoryPartition

(./TSLgenerator-Linux)

(./TSLgenerator-Linux --manpage 可看一些 quick manual.)

./TSLgenerator-Linux -c Specs/1.spec.plain

顯示: 7776 test frames generated

./TSLgenerator-Linux -c Specs/2.spec.const

顯示: 1696 test frames generated

./TSLgenerator-Linux -c Specs/3.spec.const.error

顯示: 562 test frames generated

./TSLgenerator-Linux -c Specs/4.spec.const.single

顯示: 35 test frames generated

Write test frames to Specs/4.spec.const.single.ts1 (y/n)? y

該文件中寫的是 Test Case XX, 但實際上它們不是 Test Case, 而是 Test Specification.

From Piazza:

TSLGenerator errata

Just as a note, it appears that the manual and the implementation differ with respect to boolean operators.

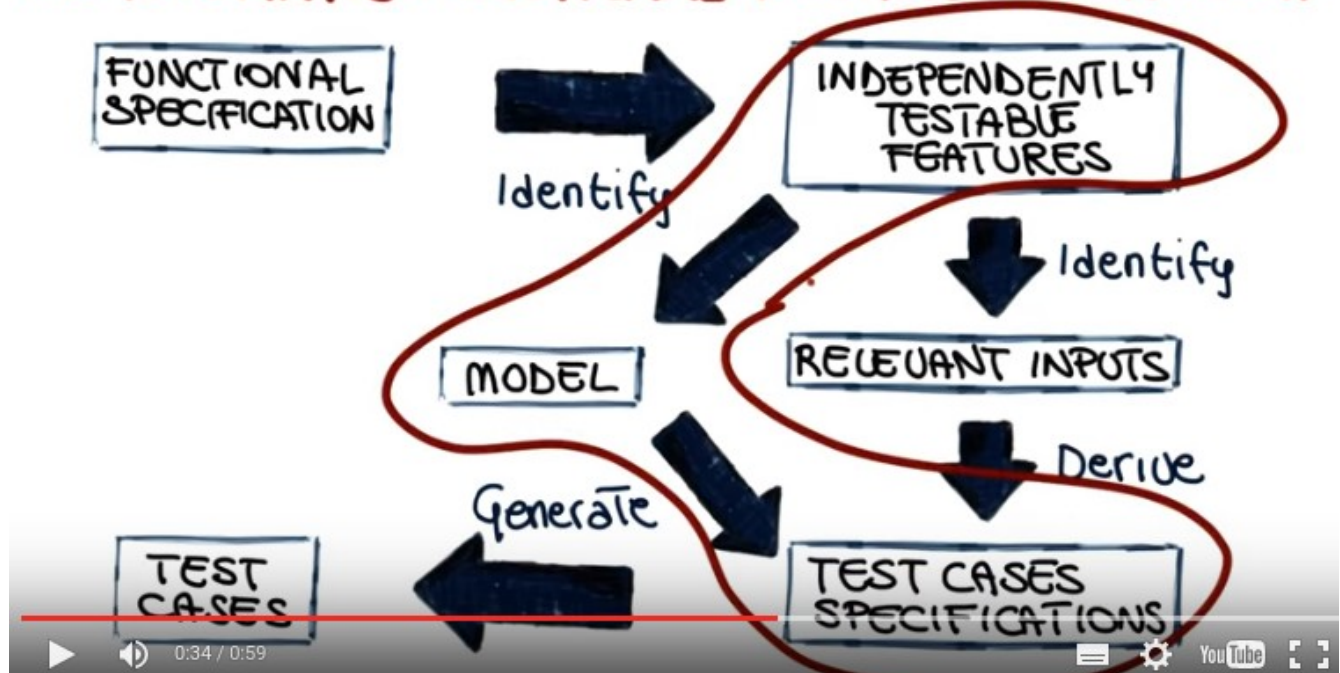
The docs say to use "or" and "and" whereas the sample files and the actual executable use "||" and "&&", respectively. When you try to use "or" or "and" you get a very unhelpful error message about undefined properties.

Hope this helps someone.

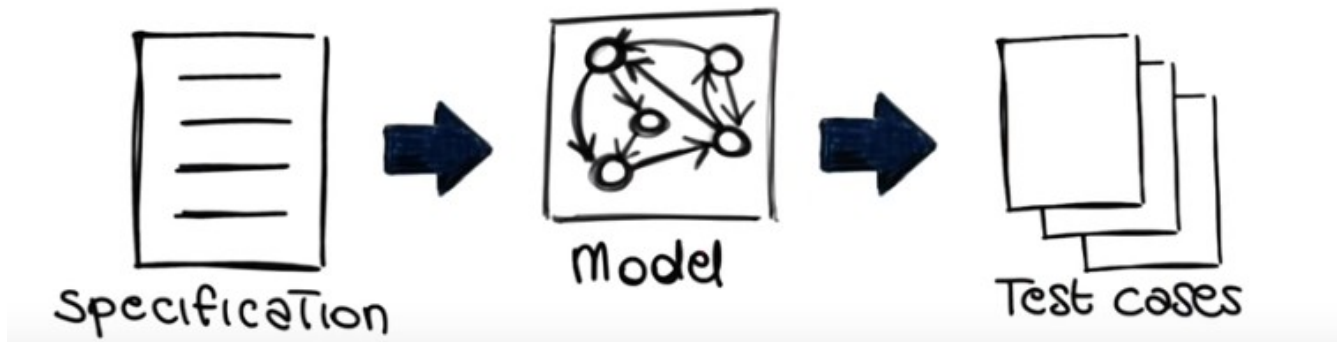
24. It's time to check your understanding of the Category Partition method. For this assignment, you'll be coming up with test case specifications for a simple application. See the link in the instructor notes for details.

25. What we just saw with the category-partition method, is a specific instance of this systematic functional testing approach. So specific instance of the steps that we represented here (弄 category partition 實際上就是在弄 input 啊). And, as I mentioned earlier on, this is not the only way in which you can generate test cases, starting from a functional specification. In particular, this step, in which we identified relevant inputs and then we combine them to generate test case specifications, can also be done in different ways. And, we're going to look at one of these ways. Which is through the construction of a model. And, the reason why I want to talk about models. Is because, model based testing is also, fairly popular in industry. And fairly used in practice. In model based testing, the way in which we go from specifications, to test cases, is through the construction of a model. Where a model is an abstract representation of the software under test. Also in this case there are many possible models, that we can use. And what we're going to do, we're going to focus on a specific kind of model. And I'll just point you to additional sources of information, in case you're interested in seeing other examples.

A SYSTEMATIC FUNCTIONAL-TESTING APPROACH



MODEL-BASED TESTING



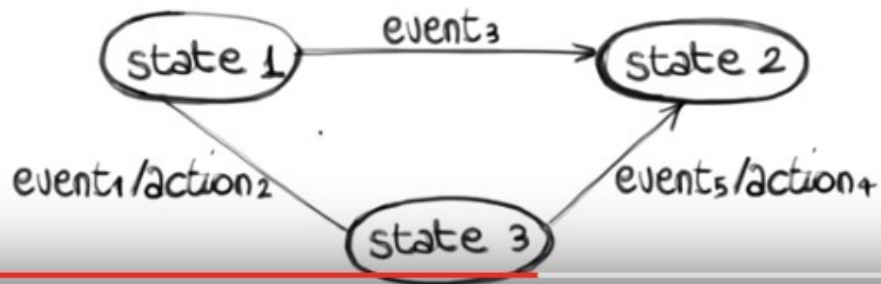
26. The model that we will consider, is a very well known one. Which is finite state machines. And you might have seen them before. At a high level, a state machine is a graph in which nodes represent states of the system. For example, in this case, state 1, state 2, and state 3. Edges represent transitions between states. For instance, in this case we have one edge from state 1, to state 2. That means that the system can go from state 1, to state 2. And finally, the labels on the edges represent events and actions. For example, what this label means is that the system goes from state three to state two when event five occurs. And when going from state three to state two, it generates action four. And does reacher model, sir reacher's kind of state machines, but we're just going to stick to this ones which are enough. For our purpose. So how do we build such a final state machine starting from a specification? The first thing we need to do is to identify the system's boundaries and the input and output to the system. Once we have done that, we can identify, within the boundaries of the system, the relevant states and transitions. So we split this single state We'll refine it into several states. And we also identify how the system can go from one state to another. Including which inputs cause which transition, and which result in outputs we can obtain. To better illustrate that, let's look at a concrete example.

FINITE STATE MACHINES (FSM)

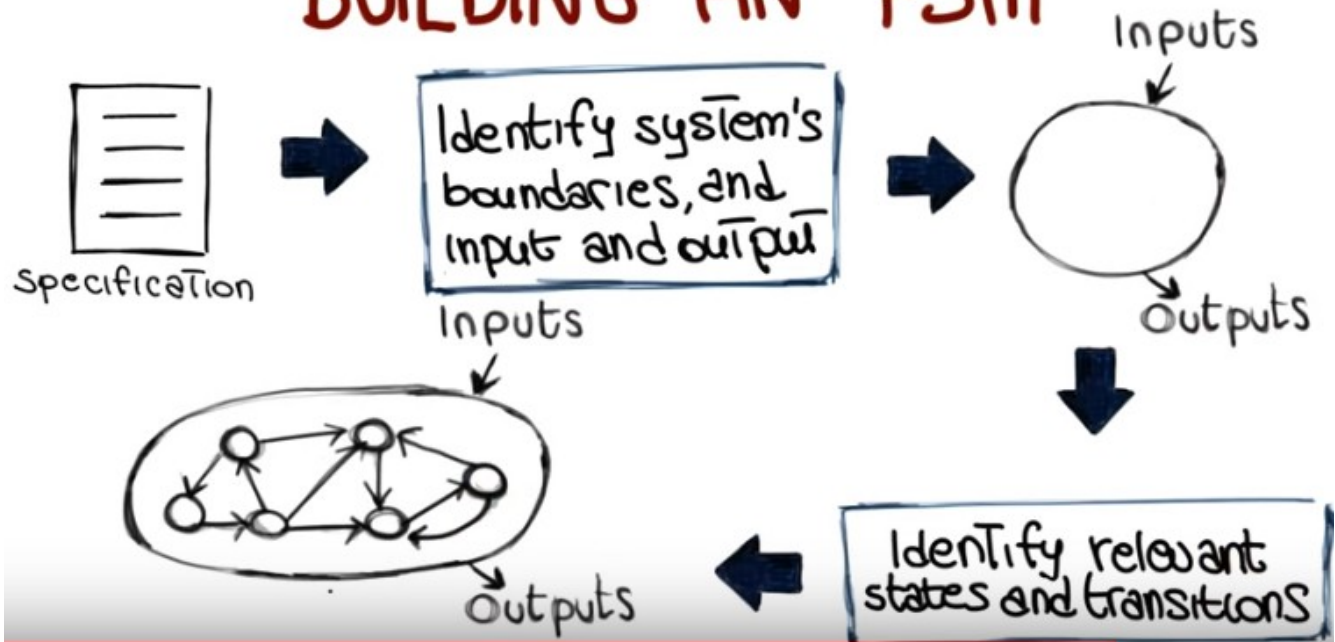
Nodes = states

Edges = transitions

Edge labels = events/actions



BUILDING AN FSM



27. In this example, we're going to start from an informal specification, and the specification is the one shown here in file spec.txt. This is the specification for the maintenance function in a specific system. So what we're doing is that we're taking the description of the functionality of a system, and we're building a model, in this case a final state machine for it. And there is no need to look at all the details for this specification, but I want to point out that if you look at the way the specification is written, we can identify specific cases that we need to take into account. Like here if something happens, something else will follow. Again, if something happens something else will follow. So we have multiple choices here. Here will determine the next steps and so on. So all we have to do is to go through this process, identify these cases and then build a machine that represents these cases. For the spec that we just consider this is the state machine that will result. Again there is no need to go through all the details, but what I want to point out is that we have a set of states. So for instance, we have state zero, which is no maintenance, and if a request comes in, the system will move, and the system wait for pickup. Then if the pickup actually occurs, the system will move to the repair state, and so on and so forth. So this is just a more systematic representation of what was in the former specification. And I will argue that this is much easier to understand at least for somebody who has to develop tests for this system. In fact what we're going to see next is how we can go from that representation to a set of test cases. And the way which we do it is by covering the behaviors represented by defining state machine. And we can decide how we want to cover them. For example we might want to cover all the states. So we might want to identify paths in the state machine that go through all the states in the machine. Like the one I just draw or this one, this one and this one. So if we consider these four test cases(一個 path 該就是一個 test case), we can see that all the states in my system or at least all the states that I have identified are covered. I might want to go a little further, and decide that I don't only want to cover all of the states, but I want to cover, all of the transitions, because, it makes sense to visit a state, when coming from different states. And, if I want to do that, and I look at the test cases that I generated so far, I can see that there is one transition, the one here, that is not covered. And, the same can be said for the two transitions here. So what I can decide to do is to generate another test case, that covers those or extend an existing one. For instance, I could extend this test case by adding a visit to the state, before going back to these two. Alternatively, I could also generate new test cases, such as this one. To cover

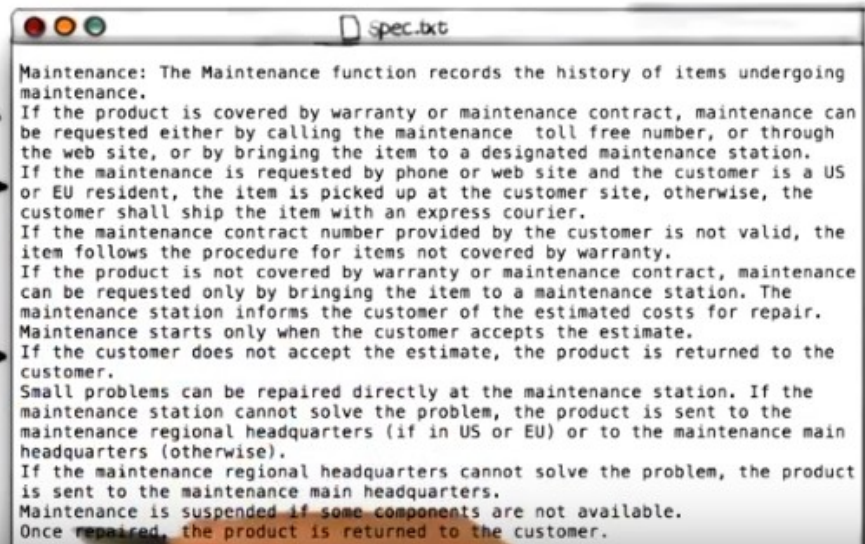
the missing transitions. And once I have these test cases, I can express them in a clearer way by simply specifying what are the states that they cover. I'm just going to give you a couple of examples. Say, if we look at the last one that I added, which will be test case number five, I just need to specify that it will go through state zero, which is this one, five, which is this one, six, and then back to zero. And I can do the same for the other test cases. So this will be my complete set of test cases. [So the bottom line here is that it is much harder to build a set of test cases that will cover the behavior of an informal description. But by going through a model](#), so by building in this case, a finite state machine for that description, [we can, in a much easier way](#), see what the behaviors of interest of the system are, and try [to cover them](#). And there is again in the spirit of breaking down a complex problem into smaller steps that we can better manage, which in the end, results in a more efficient and effective testing.

FROM AN INFORMAL SPECIFICATION...

Multiple
choices here →

Determine
the next step →

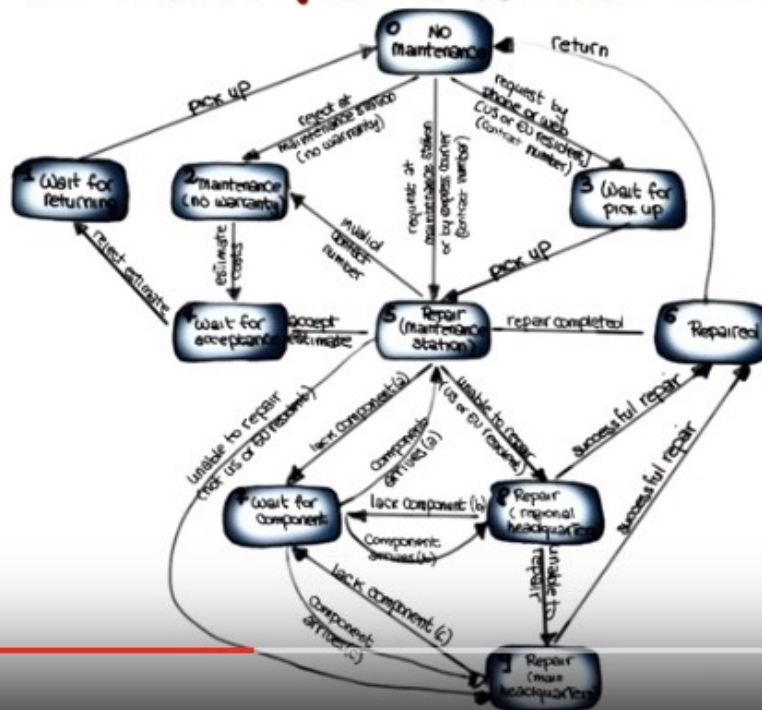
and so on →



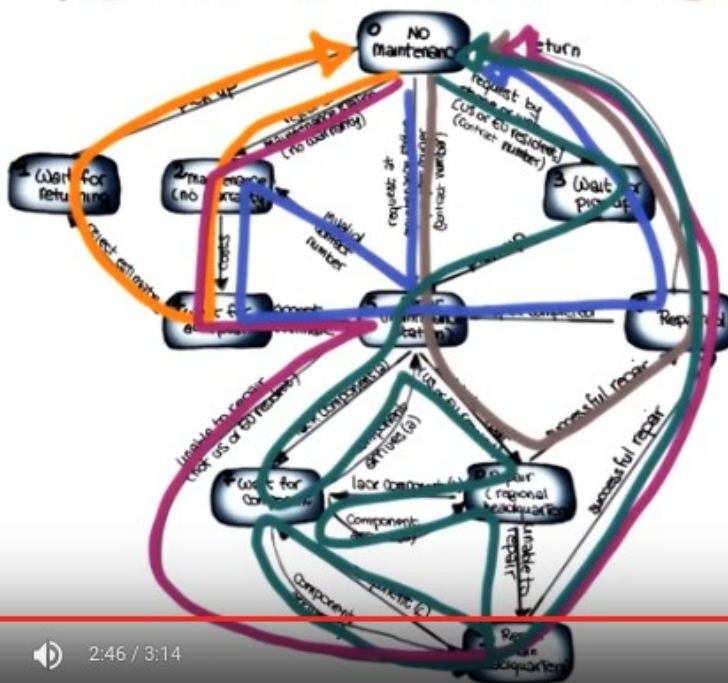
```
spec.txt

Maintenance: The Maintenance function records the history of items undergoing
maintenance.
If the product is covered by warranty or maintenance contract, maintenance can
be requested either by calling the maintenance toll free number, or through
the web site, or by bringing the item to a designated maintenance station.
If the maintenance is requested by phone or web site and the customer is a US
or EU resident, the item is picked up at the customer site, otherwise, the
customer shall ship the item with an express courier.
If the maintenance contract number provided by the customer is not valid, the
item follows the procedure for items not covered by warranty.
If the product is not covered by warranty or maintenance contract, maintenance
can be requested only by bringing the item to a maintenance station. The
maintenance station informs the customer of the estimated costs for repair.
Maintenance starts only when the customer accepts the estimate.
If the customer does not accept the estimate, the product is returned to the
customer.
Small problems can be repaired directly at the maintenance station. If the
maintenance station cannot solve the problem, the product is sent to the
maintenance regional headquarters (if in US or EU) or to the maintenance main
headquarters (otherwise).
If the maintenance regional headquarters cannot solve the problem, the product
is sent to the maintenance main headquarters.
Maintenance is suspended if some components are not available.
Once repaired, the product is returned to the customer.
```

... TO A FINITE STATE MACHINE



TO A SET OF TEST CASES



TC1: $\emptyset, 3, 5, 7, 5, 8$

$7, 8, 9, 7, 9, 6, \emptyset$

TC2: $\emptyset, 5, 2, 4, 5, 6, \emptyset$

TC3: $\emptyset, 2, 4, 1, \emptyset$

TC4: $\emptyset, 4, 5, 9, 6, \emptyset$

TC5: $\emptyset, 5, 6, \emptyset$

SOME CONSIDERATIONS

Applicability

- Very general approach
- In UML, state machine are readily available

Abstraction is key

Many other approaches

- decision tables
- flow graphs
- historical models

28. There are some important considerations I want to make on final state machines. And more in general, on model based testing. The first one is about applicability. Testing based on final state machines is a very general approach, that we can apply in a number of contexts. And in particular, if you are working with UML, you have state machines for free. Because state charts are nothing else but a special kind of state machine. So you can apply the technique that we just saw directly on state charts, and try to cover their states and their transitions. Another important point is that abstraction is key. You have to find the right level of abstraction. The bigger the system, the more you have to abstract if you want to represent it with a model, and in particular, with the final state machine. So it's like having a slider, and you have to decide where you want to move on that slider. The more you represent, the more complex your system is going to be and the more thorough your testing is going to be but also more expensive. The less you represent the less expensive testing is going to be, but also testing might not be as thorough as it would be otherwise. So you have to find the right balance between abstracting the weight too much and abstracting the weight too little. And finally there are many other approaches. So we just scratched the surface, and we just saw one possible approach. But for instance, other models that you can use are decision tables, flow graphs and even historical models. Models that can guide your testing based on problems that occurred in your system in the past. And also, in this case, I'm going to put pointers to additional materials in the class notes.

29. Now we are at the end of this lesson, and I just want to wrap it up by summarizing what we've seen. We talked about black-box testing, the testing of software based on a functional specification, a description of the software rather than its code. We saw a systematic way of doing that, that allows for breaking down the problem of testing software, so the problem of going from this functional specification to a set of test cases into smaller steps, more manageable steps. And we saw two main ways of doing this. One by identifying relevant inputs for the main features in the system and then deriving test case specifications and test cases from this set of inputs. And the second way by building a model for the main features of the system and then using this model to decide how to test the system. In

the next lesson, we are going to discuss, how to do testing by looking inside the box? So, how to do testing in a white-box fashion.