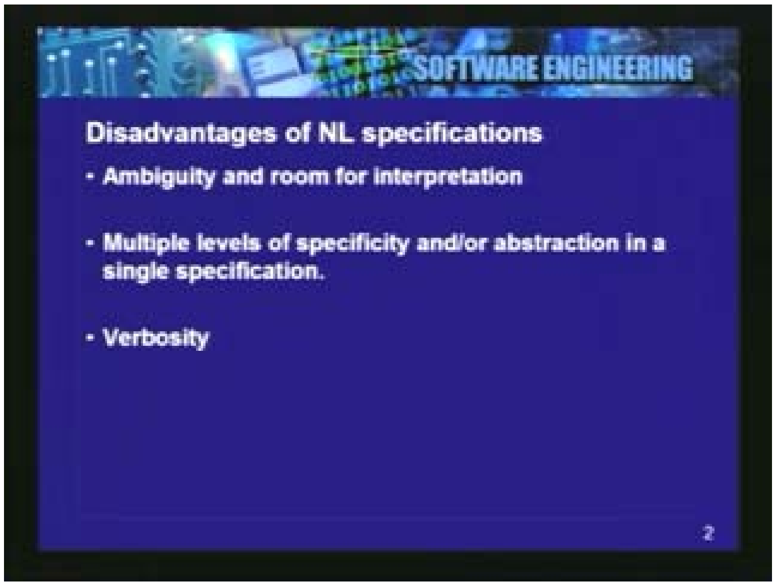
Software Engineering
Prof. Umesh Bellur
Computer Science & Engineering
Indian Institute of Technology, Bombay
Lecture - 7
Algebraic Specification Methods

Last lecture we took a look at the need for formal specification and formal methods in the software engineering processes. In this lecture we are going to go a little deeper into the two modes of formal specifications, the algebraic formal specifications as well as model based specifications. We started out with trying to specify requirements using natural language and there were certain disadvantages that came out of this which were quite significant and we summarize those disadvantages in this slide.

(Refer Slide Time: 1:24)



SOFTWARE ENGINEERING

Disadvantages of NL specifications

- Ambiguity and room for interpretation
- Multiple levels of specificity and/or abstraction in a single specification.
- Verbosity

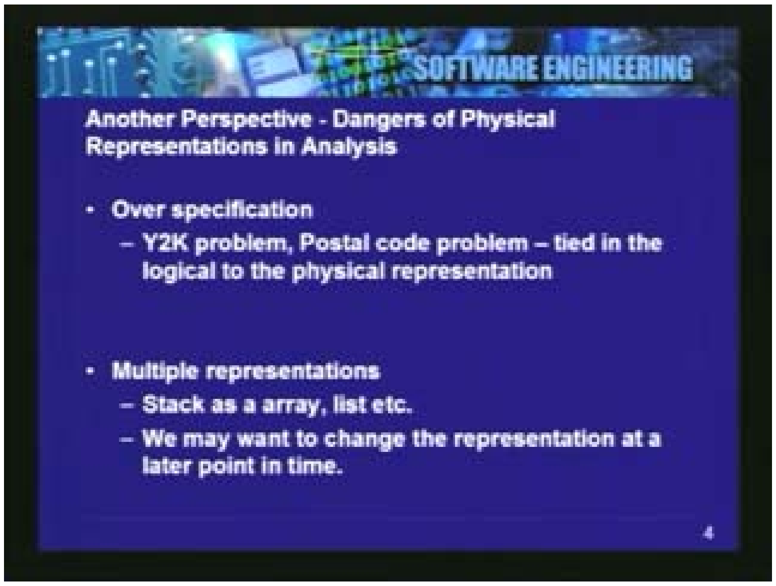
2

The first one was clearly the ambiguity. The same statement written in natural language could be interpreted in multiple ways by multiple developers or multiple analysts as a result of which, what came out of the whole process was not necessarily what was intended by client during the specification. The second thing the problem was the multiple levels of specificity within a single specification. That is there could be something at a very high level of abstraction such as the grid editor example that we talked about. At the same time there would be something with a very detailed level of detail that was not commensurate with the earlier parts of the specification. The last disadvantage clearly was one of verbosity, where using natural languages to specify what the software is meant to do was going to result in significantly larger documents than using formal methods.

We saw that there were several types of formal methods. The two main types were based on the state of the objects that were being described. We took a look at the library example the last time. These were called state based specifications. Family of languages belonging to this is Z, VDM and so on. There are also algebraic specifications that are based on operational definitions. That is what we are going to take a look at today and getting into a little bit of detail on how to specify software modules using either algebraic specifications or state based specifications. One of the things that we have to keep in mind is that specifications lead us to a higher level of abstraction than using either natural language or tying down ourselves to a particular implementation representation early in the cycle. That is the other advantage that it tends to give us and it saves us from the danger of using physical representations in analysis.

A good example of over specification is the Y2K problem. What happened was instead of representing the date as an abstract entity, what happened was there was the commitment for the date being having a year field which was specifically represented in two digits and that representation of date with two digits was used all over the program code. As a result of which when the date had to change to four digits the obvious problem followed that there were hundreds of thousands of places in any given programmer application that had to be modified. Because they all assumed the internal structure of what the date contained. This was the problem of what is called over specification. You are not confining yourself to the interfaces of the entities that you are talking about. The date in this particular case which could just have been represented as something that contained a year field and how the year field was implemented whether it was done using two digits, four digits or whatever should have been completely left out of the specification, but it wasn't. Tying the logical into the physical representation was a problem of over specification.

(Refer Slide Time: 4:54)



SOFTWARE ENGINEERING

Another Perspective - Dangers of Physical Representations in Analysis

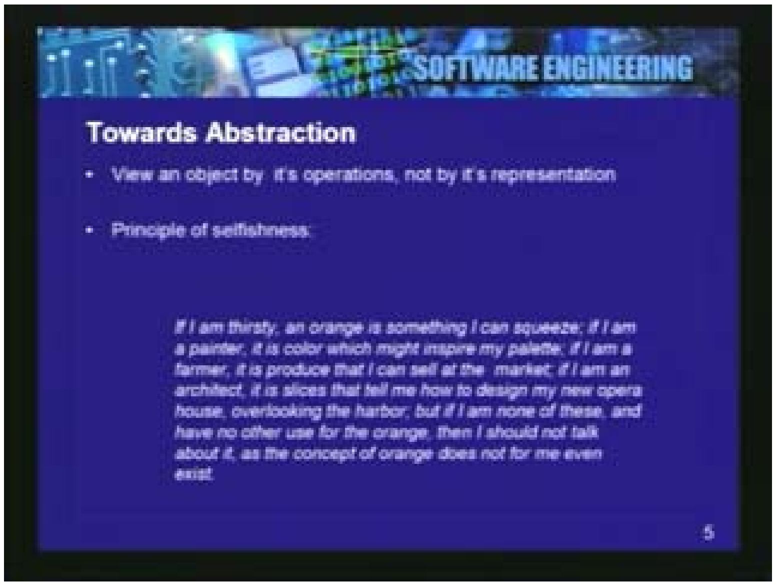
- Over specification
  - Y2K problem, Postal code problem – tied in the logical to the physical representation
- Multiple representations
  - Stack as a array, list etc.
  - We may want to change the representation at a later point in time.

4

Another problem is that there could be many different implementations that are possible. Each implementation can be tied into the logical representation at a point in time of the design cycle when it is sense to make the decision. For example, if you consider a stack data structure, which is a commonly used data structure in most applications. The stack data structure can be implemented either using ‘list’ for example or it can be implemented using an ‘array’. But the decision of what the physical representation of the stack is going to be and when this gets to be bound to the logical definition of the stack should be done as late as possible though in the software development life cycle.

Again, this is to some extent, it doesn’t belong in the specification. But if you assume the internal representation very early on the cycle it could lead to several problems. Also the notion of reuse has to be taken into account here. An abstract definition of the data structure such as a stack or a date or an insulin pump example that we saw is something that can to be reused over several different projects or even in the same projects across different modules. But if it is tied to the physical representation then that may not be suitable from one module to the next. Then the reuse is no longer going to be possible and it is going to get cut off. As a result of which we need to be moving towards a higher level of abstraction and that is what the notion of abstract specification allows us to do. They allow us to view an object and entity, a module within the specification by its operations and not by the implementation or the representation that is going to get tied to at some later point of time in the life cycle of the software. This is also well illustrated by what is called the principle of selfishness and the way it is stated here is that a fruit orange can be viewed in many different forms.

(Refer Slide Time: 6:51)


SOFTWARE ENGINEERING

Towards Abstraction

- View an object by it's operations, not by it's representation
- Principle of selfishness:

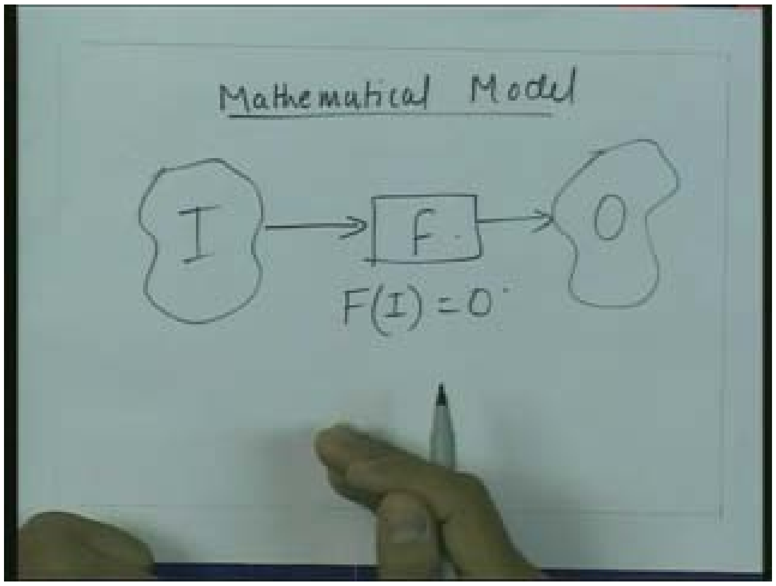
If I am thirsty, an orange is something I can squeeze; if I am a painter, it is color which might inspire my palette; if I am a farmer, it is produce that I can sell at the market; if I am an architect, it is slices that tell me how to design my new opera house, overlooking the harbor; but if I am none of these, and have no other use for the orange, then I should not talk about it, as the concept of orange does not for me even exist.

5

It can be viewed as a color orange, it can be viewed as a fruit that is going to give some juice to a person who is thirsty, if you are a painter it is viewed by the color, if you are farmer it is something that you can sell in the market and make a profit out of and so on.

Depending on the perspective, then a particular logical representation of this can be laid out. And you only want to stick to that particular perspective and therefore you are moving towards a higher level of abstraction as you go down this path and that is what we are interested in doing. Before we get into the details of what algebraic specifications consists of, it is useful to take a look at what exactly the mathematical model of software represent. The mathematical model of software essentially involves translating an input space which is represented here by I which is given to a function and this is the F that is doing the translation.

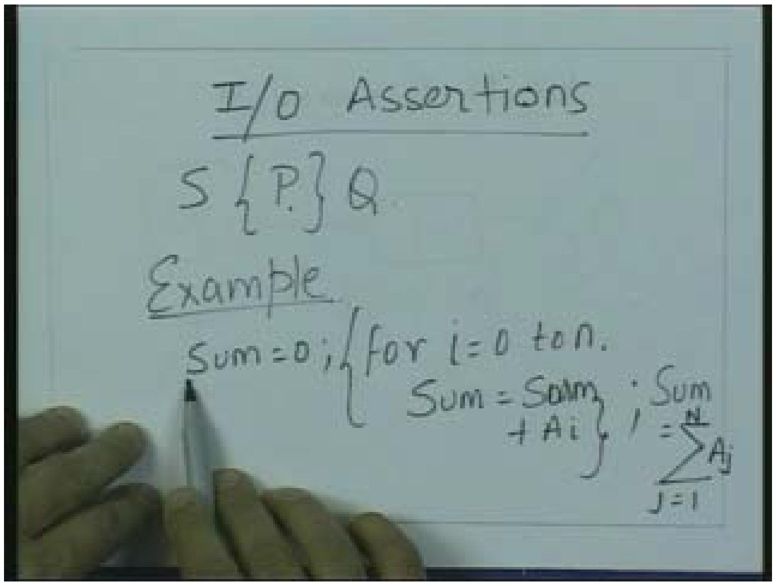
(Refer Slide Time: 8:17)



This function is translating the input space to some kind of output space O. So a program is in sense, a mathematical object and we can write F (I) is equal to O and the language that is being used here is nothing but a mathematical notation and therefore we can prove certain properties about it. For example does the software do what they supposed to do, the second is: is it doing something that it is not supposed to do. Both of these things are something that has to be proved about the software and the mathematical model will help us create this proof. With that we can take a look at different types of mathematical models. One of them is algebraic specification that we are going to be looking at shortly. But before that we can look at something much simpler which is called IO assertions it is a part of algebraic specifications as well but IO assertions are much simpler. IO assertion simply take the form “S {P} Q” and what this is trying to say is if S holds before the program P is executed, then Q is going to hold after the program P is executed.

An example of IO assertion which is a way of formally modeling software can be that, sum is equal to 0 that is the pre conditional assertion. And the function that is going to take place at the software program that is going to run says for i is equal to 0 to n sum is equal to sum + Ai. This is the definition of the program that is going to run and after the program is executed a certain condition is going to hold true and that condition in this case is going to be sum is equal to $\sum_{J=1}^{N} Aj$. Essentially it is software specification which has a precondition that is the variable whose value is null or equal to zero.

(Refer Slide Time: 10:31)



I/O Assertions

$S \{P\} Q$

Example

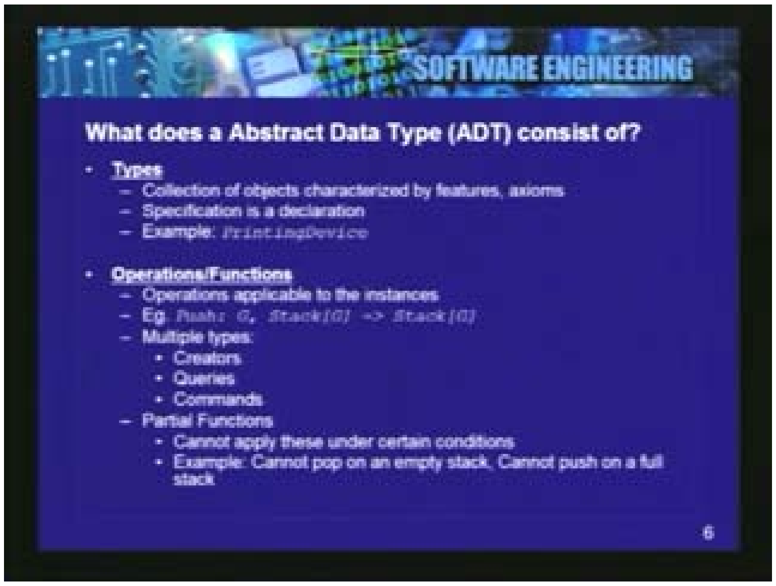
Sum = 0 ; { for i = 0 to n. Sum = Sum + Ai } ; Sum = $\sum_{J=1}^{N} Aj$

There is a program that is going to execute and a guarantee that is given by the specification S P Q is that, after this particular program executes which is you are running a loop over a series of numbers that are being added to the sum. Then the sum is going to contain the sum total of all the numbers within that particular array. That is the kind of assertion that you are making. It is called IO assertion. It is also another form of software specification that is used fairly common. Next we will start taking look at what abstract data types are made of and what exactly are abstract data types, how do they fit into algebraic specifications.

Remember we said that algebraic specifications focus on the operational interfaces or the notations that are used to lay out the different operations that it can be supported by a data type or it can be supported by an entity. For example a print server entity that can support a bunch of operations and the interface specification of the print server would go something like it would accept a particular printer to manage, then it would allow you to queue documents to the printer, it would allow you to dequeue documents from the print server queue, it will allow you to monitor the status of the queue at any given point in time and so on.

An algebraic specification essentially allows this specification to be decomposed into a set of different objects and every object is represented by an abstract data type. So there can be, for example a stack object, it can be a list object, it can be a date object, it can be UNIX directory object, and it can be a print server object and so on. Every object is represented by what is called an ADT or an abstract data type and will see what are the different elements of the ADT are. The first element of the ADT as is shown on the slide is the collection of objects that are characterized by features and axioms of the type of the object.

(Refer Slide Time: 12:50)



SOFTWARE ENGINEERING

What does a Abstract Data Type (ADT) consist of?

- Types
  - Collection of objects characterized by features, axioms
  - Specification is a declaration
  - Example: `PrintingDevice`
- Operations/Functions
  - Operations applicable to the instances
  - Eg. `Push: G, Stack[G] -> Stack[G]`
  - Multiple types:
    - Creators
    - Queries
    - Commands
  - Partial Functions
    - Cannot apply these under certain conditions
    - Example: Cannot pop on an empty stack, Cannot push on a full stack

6

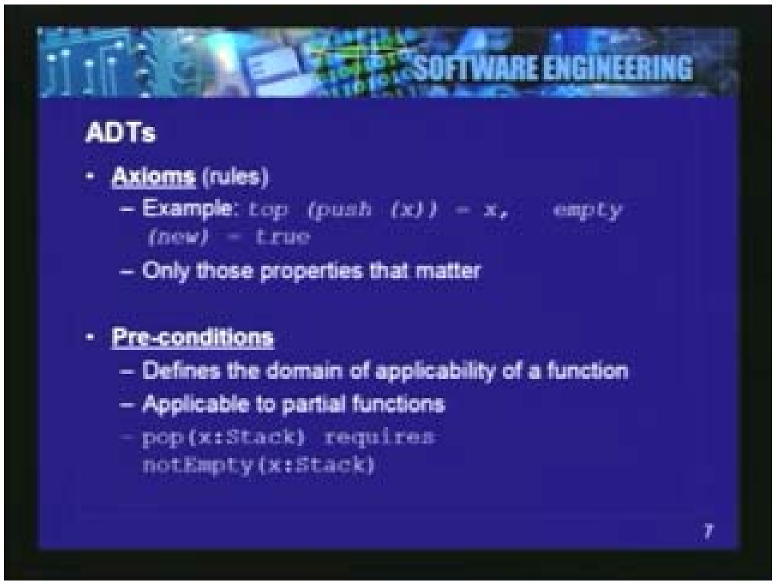
For example, if it is a print server then you will say that the type here is a print server, if it was a stack that you were trying to describe then you would say that the type was stack, if it was a list then the type will be a list. The first thing that the abstract data type consists of is just the definition of the type that you are trying to lay out and this can be a parameterized type or it can be a raw type just by itself. A raw type is something like a date for example, it is not a parameterized by anything and it is just a date. However a list could be a generic list. It could be a list of some specific type of objects only, say a list of integers or a list of rational numbers. In that case it would be parameterized by a list of some type ‘t’ and that type ‘t’ would be filled in at a later point in time.

The second thing that is important about abstract data type is the set of operations of the functions that the data type is going to support. Remember the important thing about algebraic specifications is that it is not based on the state. So you don’t expose the state of the object or the state of the type in the case of algebraic specifications. It is purely based on the interfaces or signatures of the functions themselves. Together with the constraints on what these functions are allowed to do on objects of the type.

The operations of the functions are essentially the operations that are applicable to every single instance of the type and one of the examples here can be that if you take the stack abstract data type you can push them on to the stack. If you took a look at the date data type, you can create a new instance of the date you can read the current instance of the date, change the month, you can change the year and so on. Each one of these essentially represents an operation and the operation is characterized by a certain signature which says; what is the name of the operation? What are the different input parameters that this operation expects or takes? And what are the types of those input parameters? And then what is the output that is returned by the operation?

There are different types or functions that can exist within the description of an ADT. The creators are those which create new instances of the ADT. So a new stack for example, will return a new instance of the stack and new date to return new instances of the date and so on. Then there can be what is called commands. Commands are those operations that change the state of the type, change the state of the instance of the type.

(Refer Slide Time: 16:23)


SOFTWARE ENGINEERING

ADTs

- Axioms (rules)
  - Example: top (push (x)) = x, empty (new) = true
  - Only those properties that matter
- Pre-conditions
  - Defines the domain of applicability of a function
  - Applicable to partial functions
  - pop(x:Stack) requires notEmpty(x:Stack)

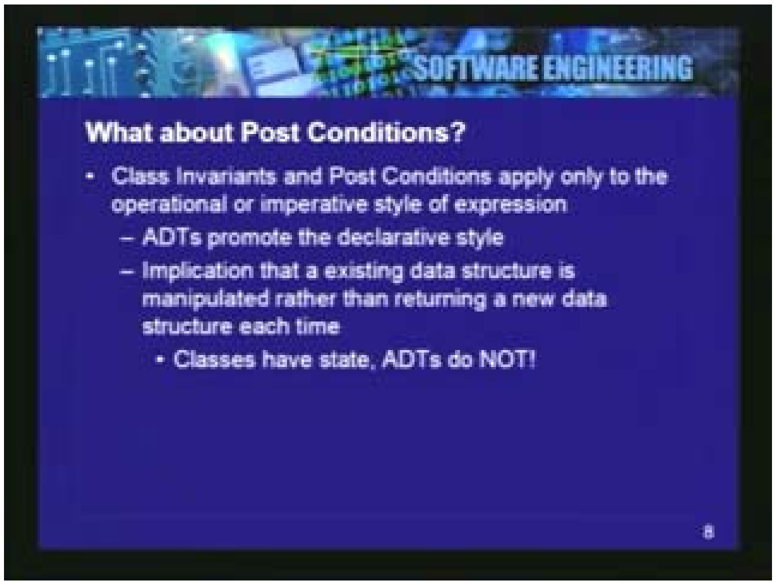
7

For example, setting the date or setting the month in a Date ADT would be a command function or command operation. Lastly we have query operations that are used to extract the state of whatever the data type already contains at given point in time. Even though the state itself is not explicitly being described here, the operations that are possible to manipulate the state in any which way, either change the state or read what the state is, are given out. And the types of the input parameters and the types of the output parameters also described, as part of the operational signature. There can also be partial functions and these are typically described using constraints of preconditions that will take a look later.

The third important things that an ADT contains are a set of rules. The instances of the ADT have to behave in accordance with these set of rules. A good example for the set of rules would be that if you took an empty stack or a new stack that is just been constructed and we pushed an element X on to the stack and if you pop the stack you should get the element X. That is a simple rule that describes about the properties of pushing an element. Only those properties that really matter need to be described. It is easy to get carried away when you are describing axioms here. There are certain laws that govern how many axioms are just right when you are describing an abstract data type representation. Lastly, we talked about partial functions; functions can be partial in nature.

Partial functions means, the function cannot be executed under all conditions. The result of the execution of the function under certain conditions is undefined, that you will not get any result out of it. Preconditions are the ways that help us define functions which functions are partial functions. Popping an empty stack say for example makes no sense at all. Pop is an example of partial functions which basically will work only under those conditions that the stack is not empty. Pop of a stack would require, ‘NOT empty of the stack’ to be TRUE, would be a precondition in this particular case. What we are trying to do here is, create intuitive definitions of the stack operations as we go along or the operations of an ADT as we go along.

(Refer Slide Time: 18:20)



SOFTWARE ENGINEERING

What about Post Conditions?

- Class Invariants and Post Conditions apply only to the operational or imperative style of expression
  - ADTs promote the declarative style
  - Implication that a existing data structure is manipulated rather than returning a new data structure each time
    - Classes have state, ADTs do NOT!

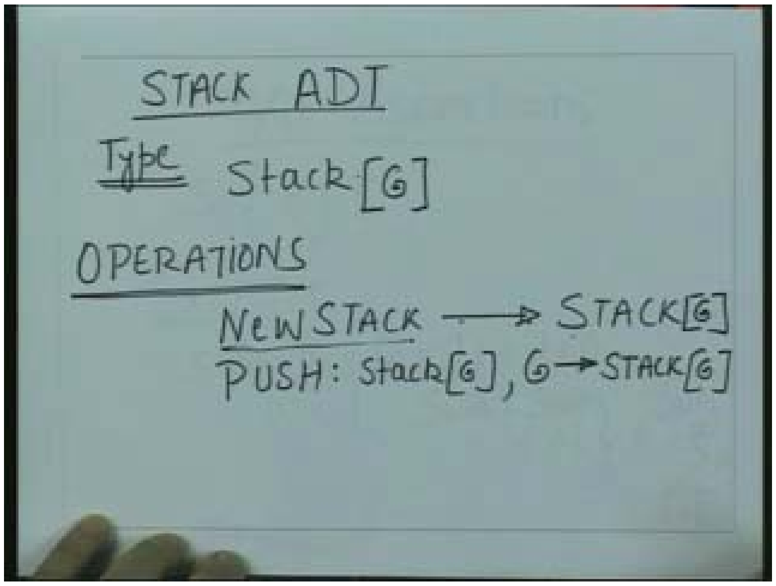
8

What would probably be helpful would be actually to take a look at the entire stack and see how this thing turns out. One of the things that we may ask ourselves when we are trying to do this is, that we saw that in IO assertion type of specification, we had something like S P followed by Q and there the notation was that you had S, you run the program P and Q was really the resulting state of the word. So the question we may ask is, how about applicability of a post condition to the abstract data type as well.

Class invariants and Post conditions apply mainly to the operational or the imperative style of the programming or style of expression. Those things are not generally expressed in algebraic specifications. Because implication here is that, instead of manipulating an existing structure or an existing instance of the type, they are always handing back a new instance.

So the state of the existing type never, ever gets changed and we will take a look at what that mean shortly in the case of stack and how we write out the specifications of the stack. In fact may be it is a good time to go into that right now. What we are going to do here is, to write down the complete ADT for the stack and see what the different parts of the ADT look like when it is a stack. The first part is the denotation of the type. The type that we are trying to write down here is that of the stack. It can be parameterized by the type G in this case. What we are trying to say is that, stack only takes elements contains elements of the type G. So that is the only type information that would be needed in this particular case. The next things that we have to move on to are the different operations that are going to be possible on the stack.

(Refer Slide Time: 22:17)



STACK ADT

Type Stack[G]

OPERATIONS

NEWSTACK → STACK[G]

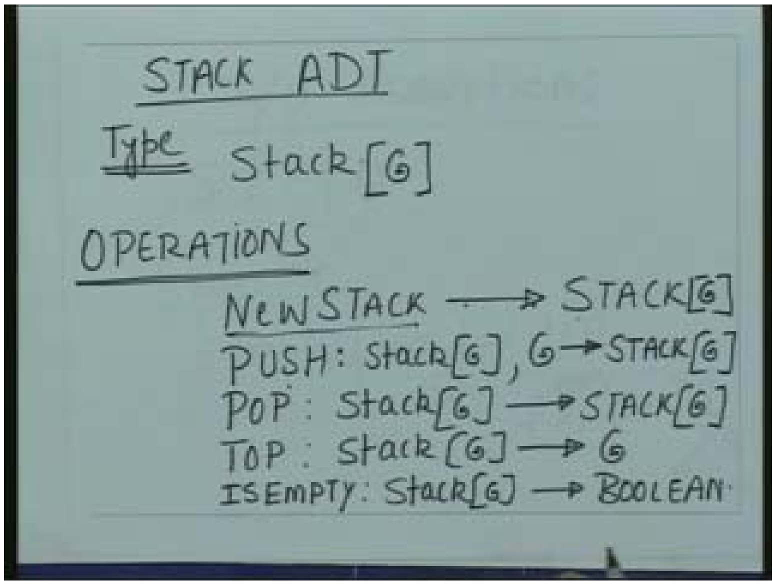
PUSH: Stack[G], G → STACK[G]

The first operation would be that of being able to create a stack. Let us call it as 'NEW STACK' and a 'NEW STACK' is an operation that takes no input parameters and in fact returns a stack. This notation essentially means the first part is the name of the operation followed by a set of input parameters may or may not take, in this example no input parameters and that returns, what is the type of the return value that is expected. In this case it returns a stack of type G.NEW STACK → STACK [G] similarly we can write out a bunch of different operations. Intuitively the next one that comes to mind is PUSH and in the case of PUSH it takes input parameters as two different arguments. One is an existing stack and one is an element that gets to be pushed on to the stack.

Taking these two essentially returns another stack G.PUSH: STACK [G], G → STACK [G] this is what we meant by its non imperative or the declarative style of programming in ADT's for this kind of algebraic specifications. What this essentially means that, we are not manipulating the same stack, but instead we are returning a new stack which has the element pushed on to it. That is the implication in the signature of the PUSH operation that we have just written out.

Similarly we can write out a POP operation for the stack. Which basically takes a stack as an input argument and what is the pop operation expected to return? Typically it returns back the stack but with the top element of it removed. So it does not really return the top element of the stack, we have to be careful about. This is why we end up writing query operation. Remember the types of operations that can exist. One of them is a creator operation, which is a new stack operation. The PUSH operation and the POP operation are essentially command operations. And the TOP is an example of a query operation and TOP will take a stack and it is expected to return the top most element of the stack which is of type G.TOP: STACK [G] → G The last operation that we can think of that would be useful in the case of a stack is 'IS EMPTY' operation and this is a query against a stack which will simply return a Boolean that says whether the stack is empty or not. These five operations are intuitively those that can be written out for a stack data type.

(Refer Slide Time: 23:49)



STACK ADT

Type Stack[G]

OPERATIONS

NEWSTACK → STACK[G]
PUSH: Stack[G], G → STACK[G]
POP: Stack[G] → STACK[G]
TOP: Stack[G] → G
ISEMPTY: Stack[G] → BOOLEAN

Remember that the ADT is not yet complete and we need to still write out for the stack ADT. We need to write out two more things those are preconditions and axioms. Let us start by writing out the preconditions in the case of a stack. You cannot pop an empty stack is one of the preconditions. So the pop operation has a precondition which is 'IS EMPTY' returns false. Similarly if we had the size element that was associated with the stack, then we can also say that we cannot push on to a stack that is already full.

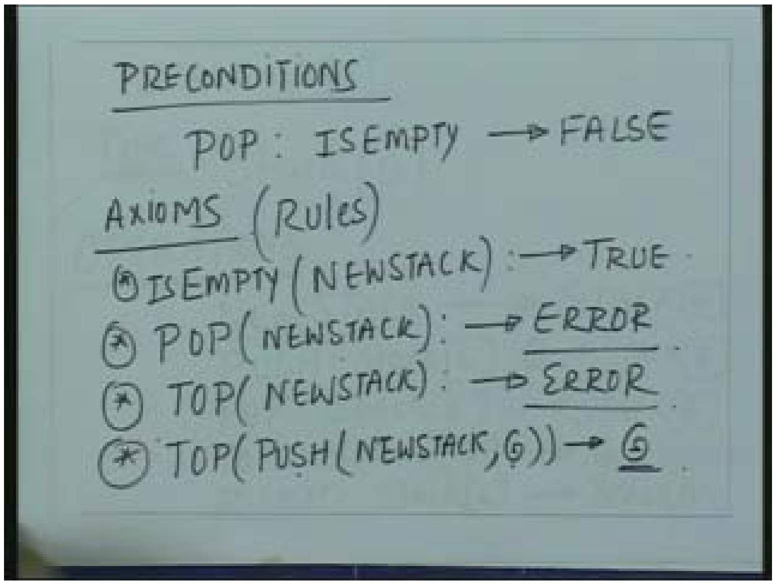
But in this particular case we have not really taken account of the fact that the stack actually has some kind of a pre build size associated with it. So we are assuming infinitely large stack as a result of which the only precondition for this stack is that 'IS EMPTY' returns false.

The more interesting aspect of an ADT is really the rules that we talked about or the axioms. These are rules that write out constraints on the behavior of the stack. Even though we are not actually tying it down to a particular representation, we are still trying to put down certain rules that say that, no matter what representation we end up using. Suppose the stack implemented is 'list', the stack implemented is an 'array' or what ever, you still got to make sure that these rules are valid and that is what is important about it.

Some of the rules that we can start writing down immediately are, IS EMPTY (NEW STACK)→ TRUE 'IS EMPTY' of a 'NEW STACK' always have to return TRUE. Because as soon as the new stack is constructed and nothing has been pushed on to it, 'IS EMPTY' has always got to be true. That is one of the axioms. And given the precondition, if we apply the pop operation to a new stack, remember we said that the precondition 'IS EMPTY' is false.

So in the new stack it is always going to be true. As a result of which the POP operation has to always result in an error in this case. POP (NEW STACK) → ERROR These are two axioms. The third axiom is that the same thing can apply to top as well. The top of a new stack is going to also result in an error. TOP(NEW STACK) → ERROR Finally we can we can write out one rule that basically says that when you create a new stack, and then you push an element G onto the new stack, and you take that top of the stack, it always necessarily has to return G. TOP(PUSH(NEW STACK,G)) → G What this rule is trying to say is that, when you push an element on to a stack that is empty, there is going to be only one element on the stack and nothing else can exist.

(Refer Slide Time: 28:12)



PRECONDITIONS

POP : ISEMPTY $\rightarrow$ FALSE

AXIOMS (Rules)

(*) ISEMPTY (NEWSTACK) : $\rightarrow$ TRUE

(*) POP(NEWSTACK) : $\rightarrow$ ERROR

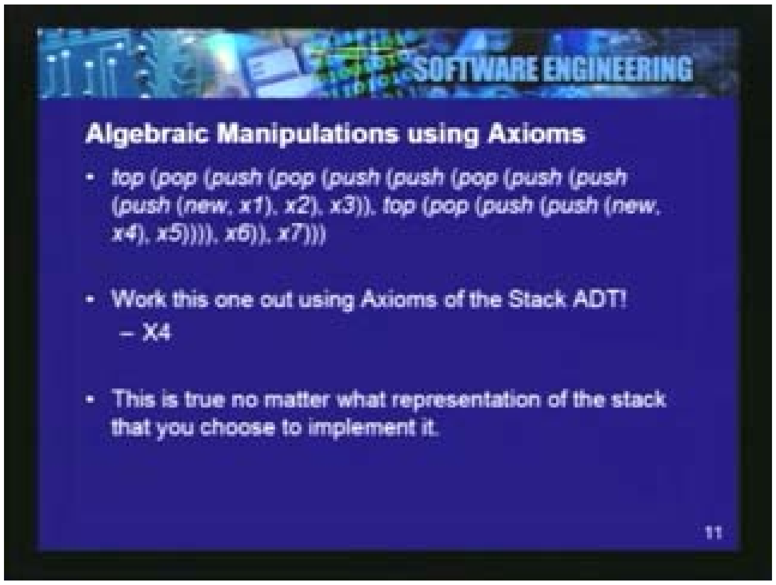
(*) TOP(NEWSTACK) : $\rightarrow$ ERROR

(*) TOP(PUSH(NEWSTACK, G)) $\rightarrow$ G

That is defining the property of the push operation per say and the way of testing the fact that push will only push the element once on to the stack and that element is always going to be the topmost element of the stack. So if you do quickly a TOP operation on the PUSH of a new stack, which is guaranteed to return single element that is pushed on to the stack. This example is trying to illustrate very simple higher level of abstraction that we are trying to create out of a stack data type. Now this can be applied to pretty much any module that you end up with specification. It could be a date specification that you are trying to write down. What you are trying to do here is put some constraints on the behavior of what this data type is going to be capable of doing when it is implemented by the designers and the developers of the system. This has to be done upfront as much as possible.

What these axioms essentially help us at the end of the day is that, we can do certain algebraic manipulations using these axioms. No matter how long an algebraic manipulation you have given, the rules will always guarantee that you end up with a valid value. That is what the rules of the axioms are meant to do. An example that is shown on this slide is a fairly long example you are taking a stack, you are doing several operations on this. You are creating a new stack, first thing is that you are pushing the value x4 on to it, then you are pushing x5 value on it, you are popping it so on. It is a huge combination of operations that are being performed on this and work this entire expression out there is nothing but an expression in the mathematical model that we have created.

(Refer Slide Time: 29:42)


SOFTWARE ENGINEERING

Algebraic Manipulations using Axioms

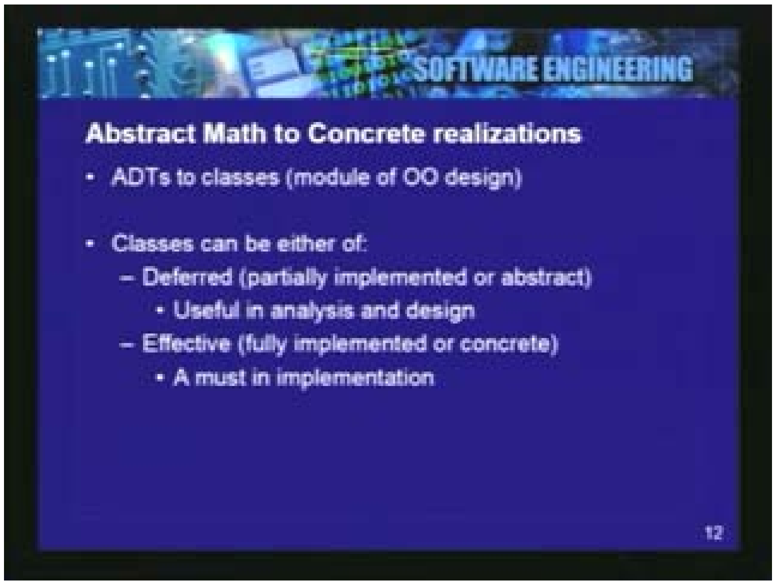
- *top (pop (push (pop (push (push (pop (push (push (push (new, x1), x2), x3)), top (pop (push (push (new, x4), x5)))), x6)), x7)))*
- Work this one out using Axioms of the Stack ADT!
  - X4
- This is true no matter what representation of the stack that you choose to implement it.

11

We can work this out just like an algebraic manipulation you can reduce certain expressions down to simpler expressions. Given the rules that we have, we can reduce this particular expression down to a single value that eventually comes out of it which happens to be x4 in this case. What is important is that this single value x4 for this particular expression is going to have to be true. No matter how the stack is going to be implemented down the line. That is what the power of the axioms gives us at the end of the day. So essentially it gives us the quality that is unchanging about stack, no matter how it is going to be implemented.

One of the things that we may ask ourselves is that what about concrete realization of this. So far we have taken look at describing very abstract representation of what a stack data type is. But how are we going to convert this into a programmatic entity that can actually be used in an application. Typically there is a very clean mapping between abstract data types and object oriented data structures. This is something that we will probably see during the later part of the course when we described object oriented design.

(Refer Slide Time: 31:29)


SOFTWARE ENGINEERING

Abstract Math to Concrete realizations

- ADTs to classes (module of OO design)
- Classes can be either of:
  - Deferred (partially implemented or abstract)
    - Useful in analysis and design
  - Effective (fully implemented or concrete)
    - A must in implementation

12

Object oriented data structures allow us to map these abstract data types very cleanly on to them and these are called classes. You must have read about this class in object oriented programming. Classes can either be deferred classes. They are deferred in the sense they are partially implemented, or there can be effective classes or fully implemented classes. Depending on how much of concreteness you want to give yourself at a particular point in time, then we can go for deferred class or an abstract class. The axioms that we saw in the stack essentially can be very simply stated in English as well. It could have been stated in a natural language but we chose to write it down in an algebraic specification.

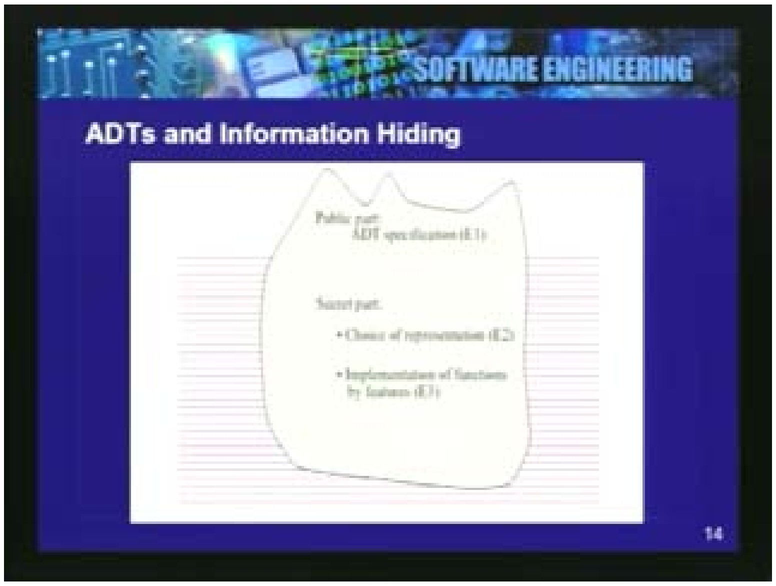
For example, one of the axioms is that the new stack is always empty. We wrote it down in a formal manner saying 'IS EMPTY (NEW STACK)' equals true. But a natural language specification that would simply be that the new stack is always empty and when you push an element on to the new stack then the top of the stack has to always contain the single element that you have pushed on to the stack. These things what you can see is the mathematical models that may have created using algebraic specifications in this case are a very powerful concise notation. Remember the characteristic of formal specifications that we talked about last time. One of them is, it has to be complete, and the second thing was it has to be concise, the third thing was it has to be correct and there were no ambiguities about it and so on. Those properties as you can see are pretty well satisfied by the algebraic specifications of the stack that we just sorted out.

We can add certain operators to this, we can add new operations to this whole thing and what we essentially end up coming up with is, a more detailed representation of one of these data structures. What it takes to get from an ADT or an abstract data type to an effective class is really the specification itself and this is the unchanging part of it. This can be handed down to a developer for example and ask him to implement it.

The developer has to make sure that when he implements, all the rules that are given in the axioms have to be true. You need some kind of a concrete representation at the end of the day and it gets turns into an executable piece of code and that needs to be a concrete representation. In the case of the stack it can be implemented using arrays, it can be implemented using lists and so on.

What you need now is also a mapping that goes from the abstract data type down to the representational capabilities that exist. For an array you can do certain set of manipulations. You have to make sure that for every operation that exists with the abstract data type, you can map it down to the set of operations that are possible on a concrete representation that you have chosen. One of the important things that you have to keep in mind is that this mapping essentially has to obey the axioms and they have to check for the preconditions that we have laid out in the notation itself. The one important thing that we will discuss pretty soon is the notion of abstract data type and information hiding. As you recall information hiding is an important characteristic of software modularization.

(Refer Slide Time: 35:07)



SOFTWARE ENGINEERING

ADTs and Information Hiding

Public part:
ADT specification (E1)

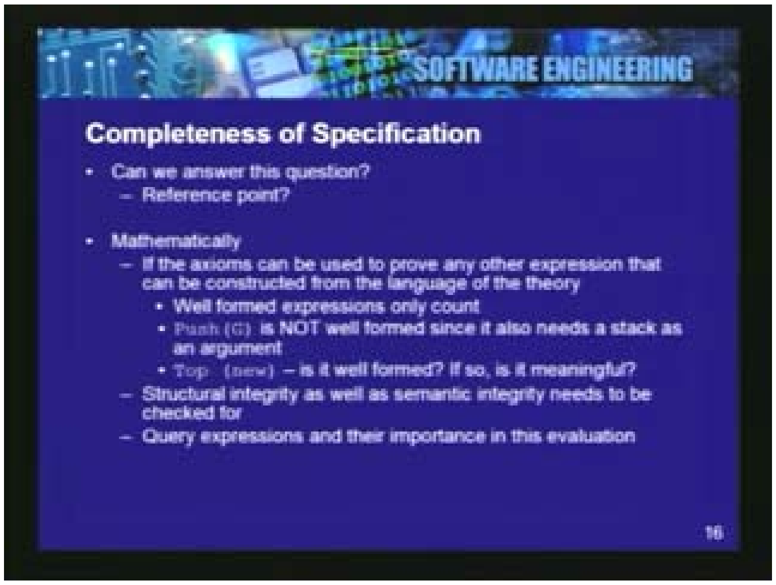
Secret part:
- Choice of representation (E2)
- Implementation of functions by features (E3)

14

What it essentially tries to say is only that part of the data type of the software module that is needed for interaction with other modules is the one that needs to be exposed and details of the internal representation must not be exposed. ADT gives a perfect way of doing that, because the ADT part of the data type is the one that will end up getting exposed. The public part of the software module that is being specified and the secret or the private part that is behind the scenes is really the implementation of it, choice of the representation that is chosen, implementations of the functions by different features is something that can be deferred to a later point of time because it’s not going to be exposed. Indeed it can even be changed at any point of time in the program as long as you stick to the specification.

Let us one of the questions that we have to ask ourselves after having just written out these specification for the task is that, for example is the specification complete?

(Refer Slide Time: 36:09)



SOFTWARE ENGINEERING

Completeness of Specification

- Can we answer this question?
  - Reference point?
- Mathematically
  - If the axioms can be used to prove any other expression that can be constructed from the language of the theory
    - Well formed expressions only count
    - `Push (G)` is NOT well formed since it also needs a stack as an argument
    - `Top (new)` – is it well formed? If so, is it meaningful?
  - Structural integrity as well as semantic integrity needs to be checked for
  - Query expressions and their importance in this evaluation

16

How can we end up answering what is the reference point for completeness? How do we know that no other operations for example are required for a stack? There are the notions of what are called atomic operations and non atomic operations. We have to consider implementation to be complete. Essentially you have described all the atomic operations. In the case of stack, we could have easily written another operation called the replace operation. The replace operation would essentially take a stack; it would take an item and replace the top item of the stack with a new item that you are giving it. So mathematically what this would look like is, it will be called replace item.

This would take two arguments, which are a given stack and some element G1.ReplaceItem (Stack [G], G1)This would end up returning a new stack, but the effect of this essentially is to say that if is empty equals false If (IS EMPTY is equal to FALSE), only if there is an item to be replaced in the stack, in that case what I am going to do is, first POP the stack that has been given to me, because I have to remove the top most element of the stack. I am going to take the result of the pop and then I am going to push the element G1 on to the resulting stack. PUSH (POP (Stack [G]), G1)

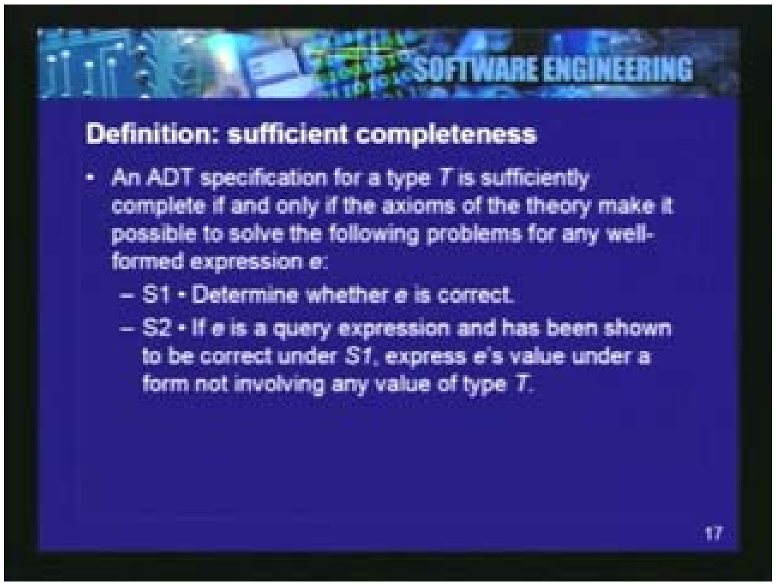
This is the definition of the operation of replace item. However what we see is that the operation replace item is the one that is entirely composed of operations that have already been defined. 'IS EMPTY' already exists within the stack definition that we gave. PUSH exists within the stack definition that we gave. POP also exists within the stack definitions we gave. Operations such as POP, PUSH and IS EMPTY can be considered to be what are called atomic operations.

In that they cannot be broken down further and they express in terms of other operations of the stack whereas operations such as ‘Replace Item’, there can be any number of these operations that we can come up with are what are called non atomic or compound operations. ‘Replace Item’ would be compound in nature whereas the other three – IS EMPTY, POP and PUSH would be atomic in nature.

Coming back to the notion of completeness of a specification, what we have to really think about is, mathematically if the axioms that have been laid out can be used to prove any expression that can be constructed out of the language of the theory then it is a complete specification. That is the definition of mathematical completeness and what this really means to say is that, given all the axioms and the operations, if i can combine these PUSHes and POPs, IS EMPTYs, NEWs and the TOPs in any particular order as long as it has to satisfy two criteria, first it syntactically it has to be a well formed operation. And secondly, semantically it has to be correct. What do we mean by that is syntactically, there are certain rules that we have laid out when we have defined the operations themselves. For example PUSH always puts two arguments which are a stack of type G and an element of type G. So a PUSH or just an element of type G is not a valid operation or is not a well formed expression in other words.

First we have to make sure that syntactic correctness exists. As long as syntactic correctness exists and it is a completely well formed expression, then you have to look at whether the expressions are meaningful. For example, whether they satisfy the preconditions, some of the operations result in error. So if there were no errors and if it was syntactically meaningful expression that was being laid out then the axioms could be use to prove that expression and that what we mean by the specification that is essentially complete. So what we are trying to look at both structural integrity of the syntactical correctness as well as semantic integrity that needs to be checked for. Query operations are obviously very important in the whole operation.

(Refer Slide Time: 41:15)


SOFTWARE ENGINEERING

Definition: sufficient completeness

- An ADT specification for a type *T* is sufficiently complete if and only if the axioms of the theory make it possible to solve the following problems for any well-formed expression *e*:
  - S1 • Determine whether *e* is correct.
  - S2 • If *e* is a query expression and has been shown to be correct under *S1*, express *e*'s value under a form not involving any value of type *T*.

17

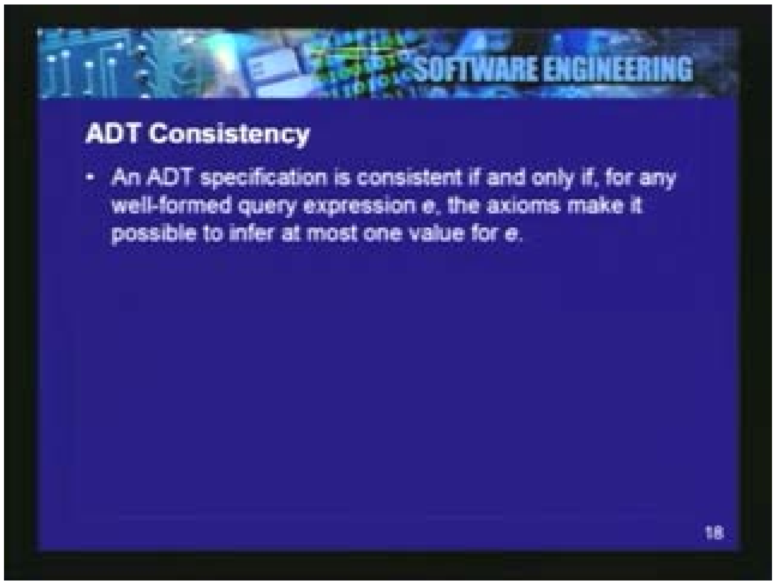
There is a definition of sufficient completeness, which is fairly formal we are looking at in the slide: The ADT specification for any given type T is sufficiently complete if and only if the axioms that have been laid out for that particular ADT make it possible to solve the following problems in any well formed expression of e: First determine whether e is correct. Given some expression e and we saw an expression that was the long sequences of pushes and the pops on an empty stack which is evaluated to this value x4.
If we can prove that indeed is the case by using reductions, having used the different axioms that are present and we can prove that value is equal to x4 then it is a valid expression and 'e' is correct. That is what we mean by prove that 'e' is correct.

The second condition that we are holding to in this case is: If e is a query expression, you remember the different types of expressions that can exist are command expression, query expressions and constructor expressions. So if e is a query expression and has shown to be correct under S1 rule that we just laid out, then we should be able to express the value of the expression in a way that does not involve the type T itself that is being tested for completeness.

Although this seems to be a little complex what this is really going to show is fact that if given any expression that can be formulated out of the language. The language that is given to us is a set of operations that are possible in the stack. 'Replace Item', for example is a new expression that we wrote out using the operations that were given to us earlier. We have to determine whether this expression is correct or wrong that is it is syntactically correct, whether there is structural integrity within this, that there are no error conditions that can exist on this expression and if it is a query expression particularly then it can be evaluated into a form that is not involved in any more operations within the definitions itself. That's the definition of completeness.

In the case there is one other property that we have to look for when we are writing out abstract data types and the other property is that of consistency. In this case, the consistency rule that can be applied is that any expression that can be formed and is evaluated, that is no two values can result from that particular expression. The axioms that have been written out for a particular abstract data type will make it possible to infer at most a single value for the expression. There is no ambiguity that can result from this. As a result of which there is a consistent set of specifications. That is the rule here.

(Refer Slide Time: 44:08)


SOFTWARE ENGINEERING

ADT Consistency

- An ADT specification is consistent if and only if, for any well-formed query expression *e*, the axioms make it possible to infer at most one value for *e*.

18

We will take a look at couple of other examples, so that the concepts become little clearer in this case. One of the first examples we look at is that of a UNIX directory. Fairly common and this can be applied to even windows directory except that we are using some syntax that is specific to UNIX directory. In the case of the type it is that of the UNIX directory and we will call the UNIX directory in a short form notation as UDIR. What are the different operations that can be performed in the case of UNIX directory?

- We should be able to go to the root of the entire file system. We will call a slash operation which will essentially give you back what the root is. by:→UDIR

- The second operation would be that you are able to form a new directory and this is commonly called mkdir in several UNIX shells. This takes two things, a U directory that is the present working directory that you are already in and the name of the directory, the cross 'X' essentially means that it is the Cartesian product, the name of the directory which is simply a string and gives you back another UNIX directory. mkdir : UDIR X NAME → UDIR

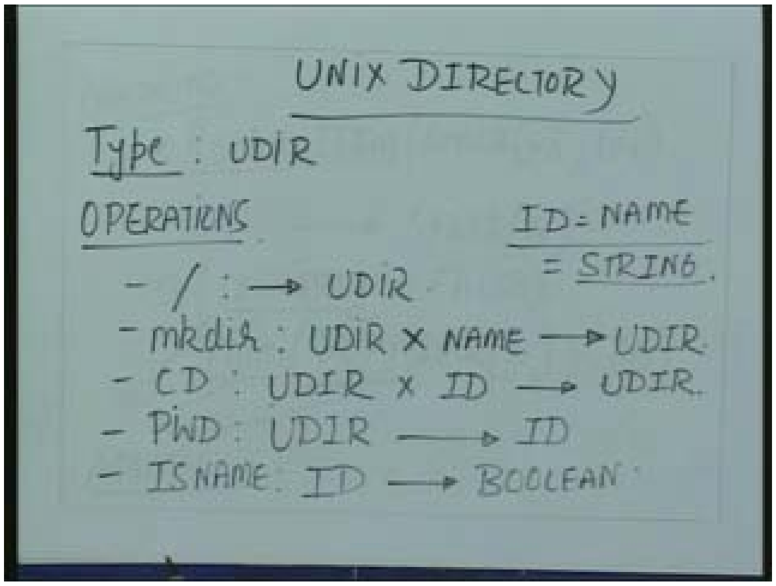
- CD which is the change directory command which is also used quite frequently takes where you are which is the UNIX directory that you are in at this point of time and it takes a ID to go to, it can be the entire path that you are giving it and then it lands you in that particular directory itself .CD: UDIR X ID → UDIR ID or a name is being used interchangeably. So, I am just going to write that down here in the top. They are both strings. The path is nothing but a string as well equals string.

  <u>ID is equal to NAME</u>
  is equal to STRING

- Another operation is PWD or present working directory which basically takes a directory as an argument and returns you back the ID or the name of the directory in the form of a string. All the way from the root, it is going to give me the path of the working directory that you are at. PWD : UDIR → ID

- One more basic operation that you can think of that might be useful is to determine whether the given directory exists on this file system. You can say that here is an ID, search the entire file system for me and return me a Boolean to say whether this is indeed a directory that exists within the file system. ISNAME: ID → BOOLEAN we have written out a few operations that we believe are atomic operations in this particular case.

(Refer Slide Time: 47:35)



UNIX DIRECTORY

Type : UDIR

OPERATIONS

ID = NAME
= STRING.

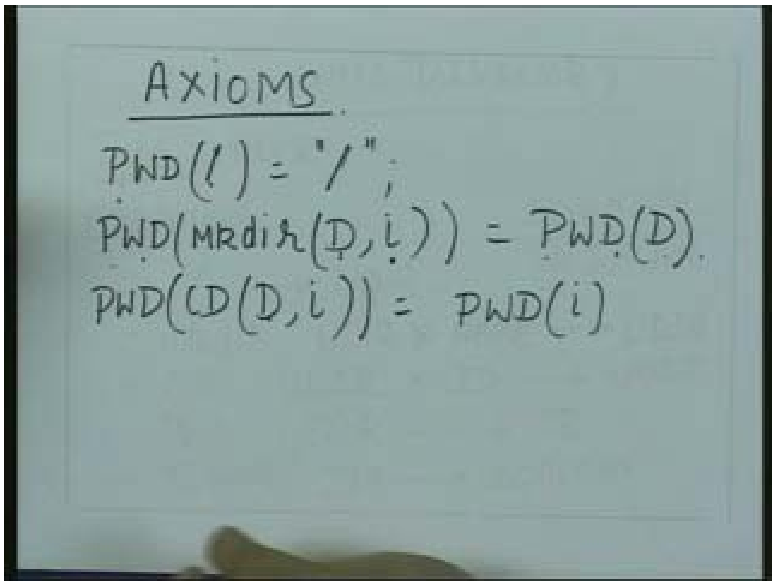
- / : → UDIR
- mkdir : UDIR × NAME → UDIR
- CD : UDIR × ID → UDIR.
- PWD : UDIR → ID
- ISNAME : ID → BOOLEAN

What we have to look at next is the notion of writing out the different axioms because that's what is important to us. What are the axioms that can be applied to different operations that we have. We had a couple of PWD operations. We can think of axioms relating to PWD.

PWD of slash will always be the string slash. For example, if this is the root directory that you are in and you ask for the present working directory in the root directory always has to give you back the string slash. PWD (/) is equal to "/"PWD of make dir (mkdir) command and 'mkdir' command takes two arguments which is some directory and some ID. This is assuming that equals PWD of D itself. PWD (mkdir(D,i)) is equal to PWD(D)This axiom is trying to point out the present working directory operation that is applied to a 'make directory' operation of D and i. That means you are in D and making a new directory whose ID is i. Then it is the same as applying the PWD operation to D, because you are not going to end up changing the directory in that particular instance.

Similarly we can apply the PWD operation to the shield directory structure as well and then change directory also takes two arguments remember it takes D and i as two arguments changing from the directory D into the directory i. Here we can say that the result of this operation PWD is nothing but the PWD of i because you are changing into the directory i, so that is the result of this particular operation. PWD (CD (D, i)) is equal to PWD (i)

(Refer Slide Time: 50:16)



AXIOMS

PWD(/) = "/";

PWD(MKdir(D, i)) = PWD(D)

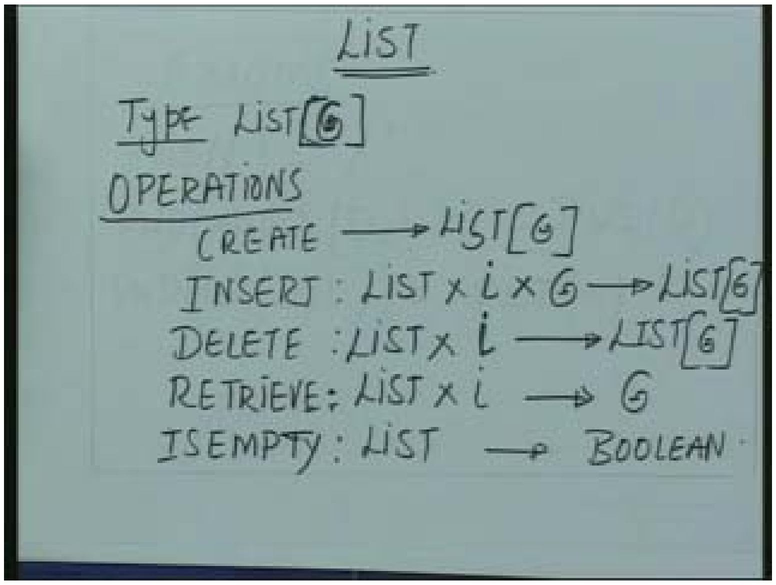
PWD(CD(D, i)) = PWD(i)

Similarly we can write out a set of axioms that will essentially give us some correctness criteria on the UNIX directory structure that we just wrote out in the last slide. One of the other operations that we can choose to define for example is the notion of 'up' operation. Instead of saying 'CD.. .' which is normally you end up trying to do in the case of UNIX directory, you can define an 'up' operation which basically takes you one level up in the directory hierarchy. And no matter how many times you go up from the root, you always have to be in root be an axiom also that you would end up considering writing down.

One more example that we would like to take a look at before we close this session on algebraic data types would be the case of the list, again a commonly used data structure. Let us first write out the operations the type is fairly obvious. The type is list of G. Type LiST [G]This is a parameterized type as you can see the UNIX directory was not a parameterized type, but in this case it is a parameterized type. The different operations or functions that can used to manipulate instances of type list of G is, you should be able to create a list or make a list new kind of operation and this takes no arguments and gives you back an item of type list of G. CREATE → LiST[G] Here we have the notion of an insert in a list. The insert takes few arguments. It takes a list. The notion of 'insert' is different from a notion of an 'append', so it takes a particular index i as to where this thing is being inserted. And it takes the item G, the item to be inserted within a list. And it returns another type list of G. INSERT: LiST X i X G → List [G] similarly we can have a delete. Delete will take a list as an argument and an index i as to which item you want to delete and this returns a new list. DELETE: LiST X i → LiST[G]

You can also have a retrieve operation and the retrieve operation taking a list and index and it returns an element of type G.RETRIEVE: LiST X i → G Similarly you can have a 'ISEMPTY' check just like that we had on the case of a stack and 'ISEMPTY' takes a list and simply returns a Boolean, to show whether this list has any elements within it or not.

(Refer Slide Time: 53:31)



LIST

Type LIST[G]

OPERATIONS

CREATE $\longrightarrow$ LIST[G]

INSERT : LIST x i x G $\longrightarrow$ LIST[G]

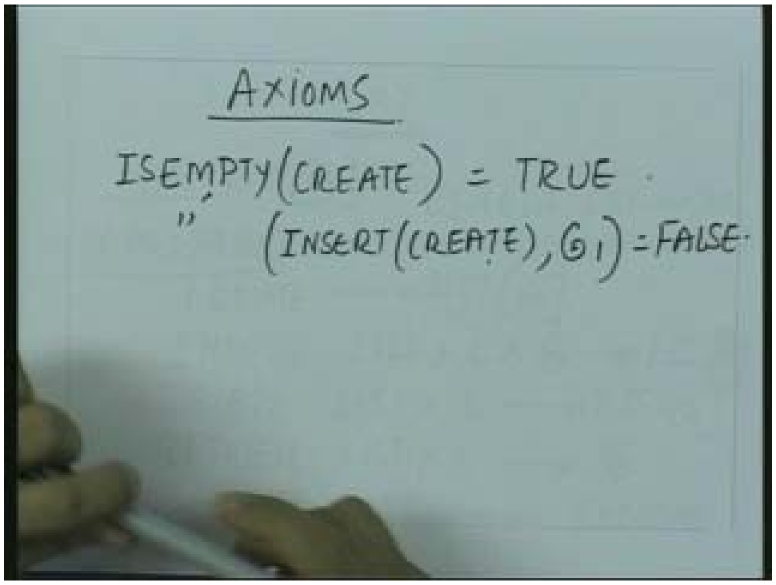
DELETE : LIST x i $\longrightarrow$ LIST[G]

RETRIEVE : LIST x i $\longrightarrow$ G

ISEMPTY : LIST $\longrightarrow$ BOOLEAN

Now one thing that is left for us is to write out the axioms for the list. The axioms for the list there are many many axioms that you can end up writing for this. One of the operations that we missed out in the case of a list is to have an operation that will return the length of the list. The length of the list will take a list as an argument and it returns an integer which denotes the length of the list. Out so I am just writing it on top of the slide in this case. LENGTH: List → INT So some of the axioms that we can write out are things like, when we create a new list 'ISEMPTY' of a 'CREATE' is indeed true. ISEMPTY (CREATE) is equal to TRUE similarly we can say 'ISEMPTY' of a 'CREATE' followed by an 'INSERT' comma G1 is false. ISEMPTY (INSERT(CREATE),G1) is equal to FALSE What this is trying to say is, if I have an empty list and I have just created it, I am checking the query is empty on the newly created list, it has to return TRUE. Whereas the 'ISEMPTY' operator on a list just created, but an element just has been inserted into it is false and so on.

(Refer Slide Time: 55:17)


Axioms

ISEMPTY(CREATE) = TRUE

" (INSERT(CREATE), G1) = FALSE

So these are the kinds of axioms that we end up writing even for the list just like we had written for stack and there are several axioms that we can to write. We have to create combinations of the operators in order to write down the axioms and there are certain rules for that as well. We would not go into those rules because of the complexity of some of the rules. But the definition for completeness and as well as consistency are two things that we have to be able to prove at the end of the day having written out an abstract data type specification. What we looked at what we have seen in this class is, some ways of actually translating the notion of writing down formal specification. What does it look like, what are the different structural elements of a formal specification and we have written that using algebraic data types essentially. We have not gone into state based models Z, VDM. But these things are something for which references will be available as part of this course and you can choose to look them up. We have seen how to write out algebraic specifications, how these can formal in the sense of creating mathematical models which we can use to prove the correctness about the language that we have just ended up creating and hopefully these methods will be useful in further specification efforts that you may be involved.