Course: Software Engineering – Design I

What is a specification

Description of a (computer) system, which:

- is *precise*;
- defines the *behaviour* of the system;
  (*what*, not *how*)
- requires an *understanding* of the problem;
  (*why*)
- has *formal semantics* and *reasoning laws*.
  (*quality* & *correctness*)

Introduction

Slide Number 1

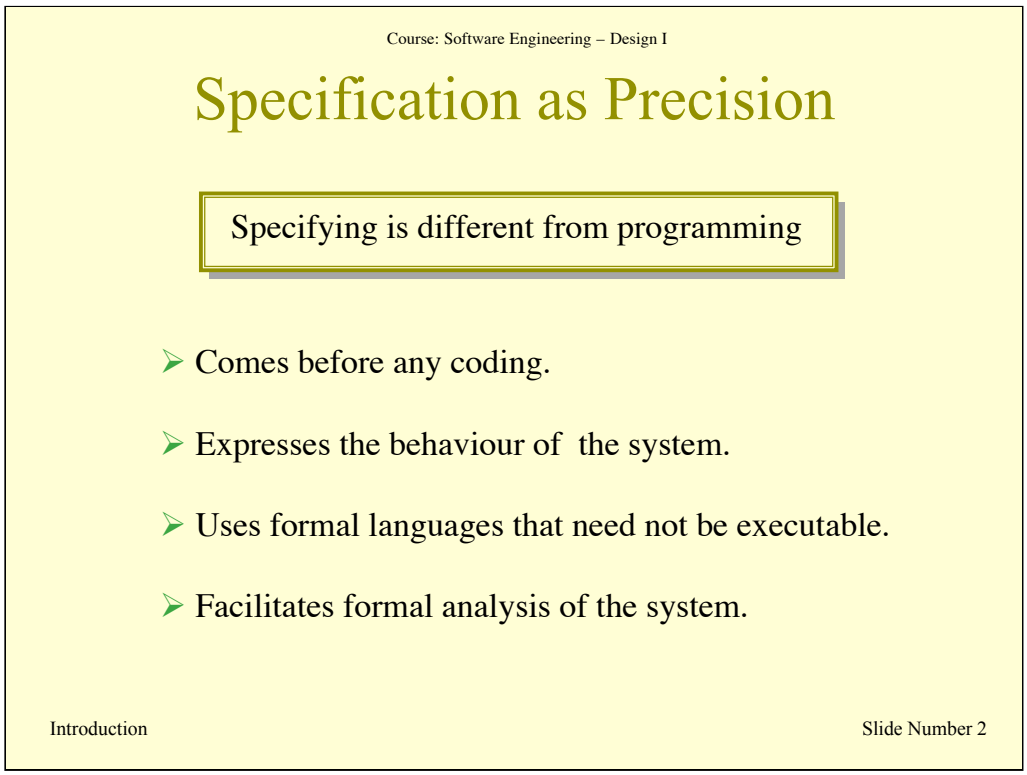
A dictionary definition might say that specification is the act of specifying, making something specific, i.e. *precise*. We could then define a specification of a computer system as "a precise description of the system". But this definition does not capture all that we really mean by a specification. Specifications are *descriptions* of a (computer) system that satisfy the following characteristics.

They are *precise*. Formal specifications are descriptions written using formulae. Formulae are indeed precise, unambiguous sequences of predefined symbols, combined according to certain rules. Computer codes are examples of precise descriptions written using formulae, expressed in certain programming languages. But they are not formal specifications.

Formal specifications use formulae to define, in a precise manner, the behavior of the system. Writing a specification means looking at *what* the system is supposed to achieve when used, rather then *how* the problem is solved. In this sense, formal specifications can be seen as mathematical models that represent the intended behavior of the system.

The process of defining such a mathematical model (or formal specification) , that we call *specification process*, requires capturing requirements. This means coming to *understand* what the customer wants or will find good. Therefore *specifying* (a computer system) is a task that requires *understanding* the problem clearly enough to be able to document it as a formal specification.

Formal notation has a precise formal semantics and reasoning laws, which facilitate precise reasoning about the specifications. This helps develop software packages that do something of value for the user and that do conform with the specification (*quality* & *correctness*).

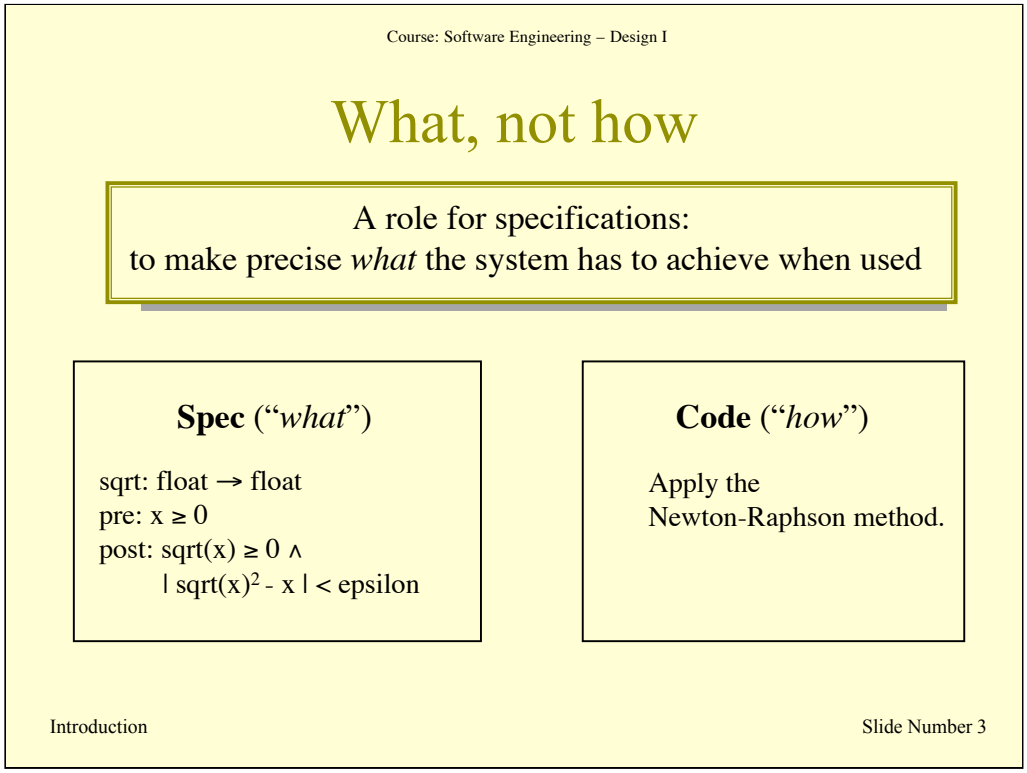


In the previous slide, we stated that specifications are precise descriptions of a computer system, which should not be confused with implementations or programming codes. The question is, then, "*what distinguishes specifying from programming*".

Programming and specifications both use formal notations. An implementation is in fact an extremely precise description, just about as precise as you'll ever get. Specifications are *less* precise than implementations, as they leave unspecified a lot of implementation details. However, formal specifications are used before coding. It is not necessary to execute a formal specification to know its meaning. One of the advantages of formal specifications is therefore to be able to understand what a program will do without running any code – in fact, without writing any. This means that we can discover many errors without having to run any test. This ability to model behaviour also distinguishes formal specification from diagrammatic methods (.e.g data-flow diagrams), which can only model program structures.

Specification languages are usually different from programming languages. They need not be executable, and resemble traditional symbolic logic more than any programming language. They can be more expressive, more concise and easy to understand than a programming language and they usually have less syntactic clutter.

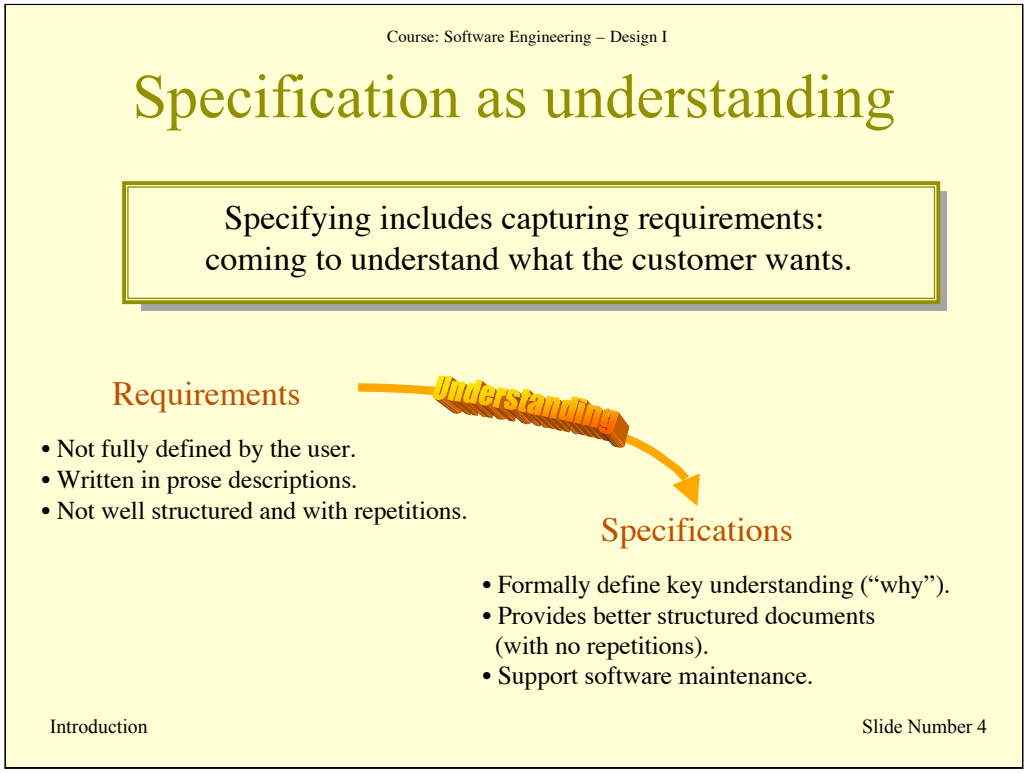
But, what most distinguishes programming from formal specifications is that the latter come with a formal semantics and reasoning laws. These laws facilitate an analysis of the system, in the sense of understanding the system without actually running programs. They enable us to simplify formulae, to derive new ones and to determine whether one formula is a consequence of others. This is important because, as we will see later in the course, it makes it possible to formally check whether a system specification meets its requirements. In this way, you will not need to rely on error to validate programs.



An implementation is there to tell the computer what to do, in a program of execution steps – i.e *how* the problem is solved. But it does not usually explain *what* is achieved by those execution steps. The example above is a program describing *how* to calculate a square root. This program does not have any formal reference to *what* is calculated – an output whose square root is the input. On the other hand, the formal specification on the left end side of the slide, defines this output together with its properties (e.g. post-condition).

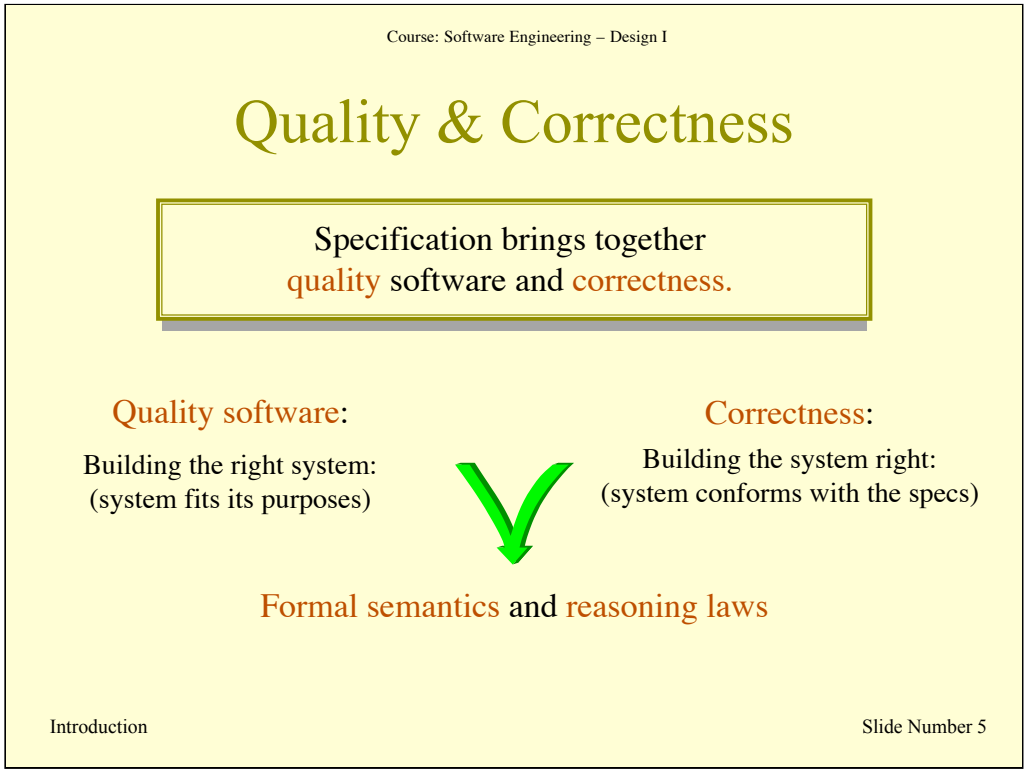
The example on the left hand side of the slide is about programming "in the small", i.e. the specification is of programs that are self-contained problems containing number, and well-defined structures such as lists of arrays. In this case it is reasonably easy to define specifications in the form of pre- and post-condition: it requires looking at and defining the properties of the underlying mathematical problem.

In this part of the course we will look at programming "in the large". We are interested in bigger projects that have to achieve something in the real world, where objects are less well-defined. Specifying "what" the system is to do also means making reference to the real world and incorporating assumptions about it. This is one of the challenging aspects of formal specifications.



"In the large", specifying is a non-trivial task. It includes capturing requirements, coming to understand what the customer wants or will find good. For all sorts of reasons it's difficult to get a definitive spec prior to all coding. It is certainly not usually possible simply to go to the customer and say "You tell me what you want". Customers are normally uncertain about what they expect a computer system to do, and moreover their expectations will evolve as they use the system. Furthermore, user requirements often come in prose descriptions that are basically narratives, and as such they include repetitions. Writing programs directly from user requirements is a very hard work.

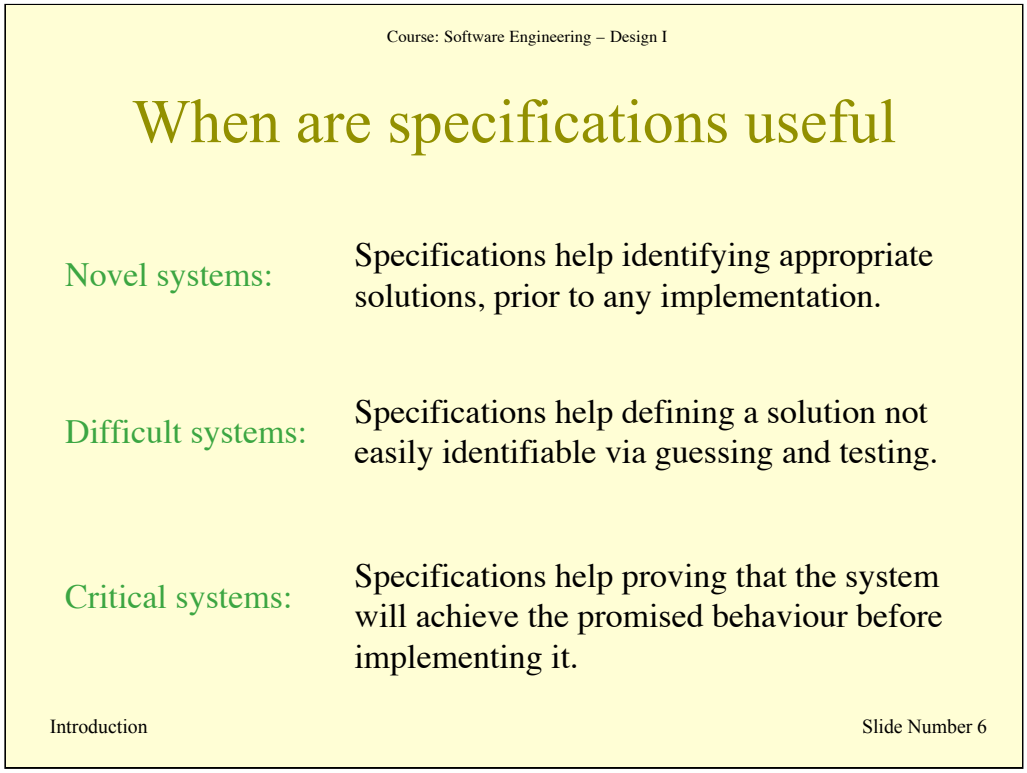
Specifying helps understanding the problem and documenting it in a clear and organised manner. True understanding is not just what, but also "*why*", e.g., why certain decisions are made. Stating why certain decisions are made in developing a system also helps backtracking and revision. A good understanding facilitates the development of a more organized description of the system behavior. For example, common features of different system operations can be identified and factored out. Remember that what the specifier goes through in coming to understand the problem is going to be repeated many times afterwards – by the implementers, by the writers of the user manuals, by the users themselves and by the software maintenance workers. To help them and to avoid misunderstandings, specifications should be make as clear as possible and include the key steps in understanding what the specifier went through.



Ultimately, what we are interested in software engineering is to have *quality* software, i.e. software that fits its purpose and that does something of value for the user, as well as *correct* software, i.e software that respects (satisfies) the specifications.

Formal specification brings these two aspects together. It helps achieve quality by facilitating an understanding of the user requirements and therefore helps to not loose sight of user needs. At the same time, it provides a means for proving that the system satisfies the users' requests in a correct way.

Specifications can be used to both **validate** and **verify** a (computer) system. It's the precise semantics and reasoning laws of a formal specification that enable these two tasks to be performed. Users' needs can be formalized and proved to be satisfied by the specifications of the system. In the same way, program code can be proved to verify the specifications. Those of you who took the course of reasoning about programs in the first year have already seen some simple examples of verification when you were proving that (little) procedures satisfy their pre- and post-conditions. The definition of specifications helps implementers to focus on small parts of the system and forget about the rest, just working blindly to what has been decided. The verification of each part of the system with respect to its specifications contribute towards the correctness of the entire system.

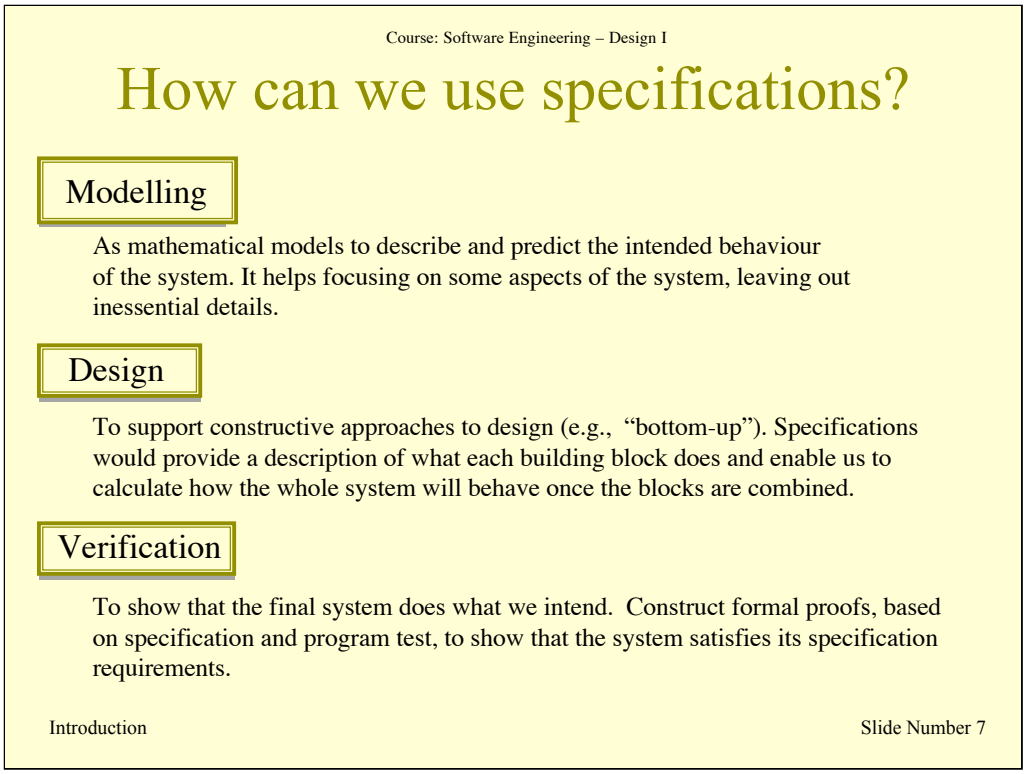


Formal specifications are particularly useful for novel systems, difficult systems and critical systems.

In the case of novel systems, formal specifications facilitate an analysis of the system prior to any implementation. In particular for large systems, it's not good practice to just plunge ahead and implement the first idea that comes along. In real industrial environments, it cannot be afforded to build several versions of entire systems. Analysis of the specifications are needed instead.

Difficult systems are those systems (not necessarily large) that tackle complex problems, problems with a multitude of intricate details. Because of the complexity, guessing and testing might well not cover all possible scenarios and therefore converge to a useful solution. Formal specifications, in this case, help identify a solution and validate it.

Particularly relevant is the case of critical systems. These are systems where safety and security are essential. In this case, it is extremely valuable to know prior to any implementation what the system will do and what it will not do. Developers are required to show that the promised behavior can be achieved before implementing it or using it. Formal specifications are useful for investigating how our system will behave, even before we begin design and coding. The analytical techniques of formal specifications can be used to confirm that the system will meet its critical requirements such as safety or security.



In previous slides we have emphasized the importance of developing specification before coding. The use of specification is, however, not confined to a specific phase of the software life cycle. Formal specifications are useful throughout the entire development process.

Modeling. Starting from modeling, specifications can be used to describe and predict the behavior of the system. They can be seen as mathematical models that describe the system behavior accurately and comprehensively, and that are much shorter and clearer than the code. Developing formal specifications helps focus on some aspects of the system, by leaving out all the details that are inessential to that aspect. Complex systems might well be described by several specifications, each focusing on a different aspect.

Design. In design different approaches can be taken. But in the case of large systems, it might be useful to take a "bottom-up" approach, i.e. to identify building blocks and to assemble them together to obtain the whole system. Specifications can, in this case, be used to describe the behavior of each individual block and enable us to calculate how the entire system would work once the blocks are combined.

Verification. The use of formal specification for verification is quite renown. Verification deals with the final product of the development process and basically consists in constructing proofs, which are convincing demonstrations that the code does what its specification requires. Such proofs use only the specification and a program test. Proving the correctness of the program is better than testing the program. Testing considers only a finite number of cases, while a proof considers all (possibly infinitely many) cases.

A formal language

Z is one of the languages commonly used for writing specifications. It contains a lot of special-purpose notation.

We will

- borrow some good notational ideas from Z
- and apply to conventional logic.

Result

- To be able to write specifications that look like Z specifications, without being correct Z.

But

- Formal logical notation is *not clever per se*. It can be only useful if accompanied with a good understanding of the underlying problem.

A language commonly used for writing specifications is the Z language. This contains a lot of special-purpose notation that takes a while to learn, but it also has good notational ideas that we are going to borrow and apply to the conventional logic. At the end of the course you should be writing specifications that look somewhat like Z but without being correct Z.

This is because *our aim is to teach you the underlying logical ideas, not the Z notation* itself. But after that, you should have no trouble learning Z itself.

We are going to be using formal logical notation, and it takes some training to be able to do this. It is therefore easy to flatter yourself that merely writing something in formal logic takes cleverness and is a worthwhile thing to do. The reality is otherwise. When you don't yet understand the underlying problem, this only serves to complicate and obscure it. A good rule is that it is better to understand the important things imprecisely than to understand the unimportant things precisely!

Summary

Specification is

- understanding the problem and writing it down clearly,
- answering to “why” and “what” questions,
- reasoning about the system.

It is useful for

- modelling and validation,
- design,
- verification.

These additional notes provide you with a basic and general introduction to what formal specifications are, in terms of (1) the actual process of specifying as understanding the problem and answering “why” and “what” questions, (2) the final product as a clear, mathematical description of the behaviour of the system and (3) its use for analysing the system with respect to the user needs (i.e. validation of the spec) and with respect to the specification requirements (i.e. verification). Some of the circumstances in which formal specification is particularly useful are also mentioned here, together with a brief discussion of the role of formal specifications throughout the life cycle of a (computer) system.

In the first two lectures we are going to define a formal language for system specifications based on classical logic and some Z notational features.