Unit 1: Introduction

- **Course**: C525 *Software Engineering*
- **Lecturer**: Alessandra Russo
  - email: a.russo@imperial.ac.uk
  - office hours: available in my office (room 560) between 1:00-2:00pm on Tuesday.
- **Duration**: 12 lectures and 6 tutorials

Aims and Objectives

This part of the course is about formal specifications and their role in (formal) development of software systems.

It aims to illustrate:

- Basic approaches to formal specification.
  e.g.: "state-based" and "class-based" approaches
- Derivations of formal specifications from informal requirements.
- Refinement of formal specifications into program structures and specification of programs.
- Basic techniques for reasoning about programs and their correctness with respect to the specifications.
  i.e. product meets its requirements

This course focuses on formal development of software systems. By formal development we mean development processes that make use of formal techniques for defining system requirements, specifying programs and reason about the correctness of programs with respect to given formal requirements. We will therefore look at different types of formal specifications, chosen according to the development stage under consideration and/or the type of development approach adopted (i.e. "state based" or "object" based approaches). We will also see how some of these different types of formal specifications can be related with each other.

The **aims** are:

❖ Illustrate the role of formal specifications within the development process of computer systems, in terms of helping understand **what** the system is supposed to achieve and **why**.

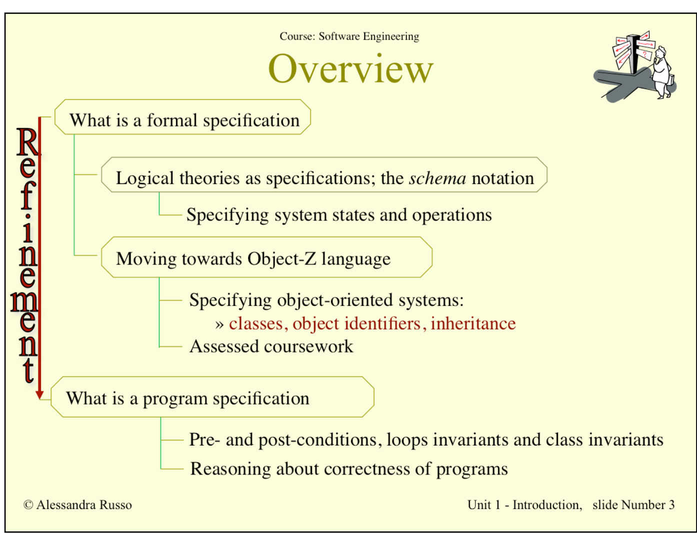
❖ Show the basic principles and logical ideas underlying **state-based formal specifications**. Conventional logic will be used, adopting some of the most common features of Z specification language.

❖ Extend the principles of state-based specifications to **class-based formal specifications**. Show how some standard object-oriented features (e.g., inheritance and polymorphism) can be captured in class-based formal specifications. A specific language, called Object-Z, will be used, which builds upon notations introduced in the state-based specifications.

❖ Show how class-based formal specifications can be mapped into specifications of object-oriented programs and illustrate basic techniques for using program specifications to reason about the correctness of object-oriented programs.

Objectives: At the end of this part of the course, you should be able to:

- ✓ Understand basic ideas underlying formal specifications,
- ✓ Write state-based and class-based specifications from informal requirements,
- ✓ Generate Java program specifications from class-based formal specifications
- ✓ Reason about the correctness of simple Java programs given their specification.



This part of the course will begin with a general introduction of what formal specifications are, why and when they are useful for the development of (large) computer systems.

We will then concentrate on the use of **logical theories** as formal specifications, building on basic logic concepts, that, in the case of MSc conversion students, have been covered in the first term. Much of the notation is borrowed from the Z specification language, for instance the *schema* notation. It is not the purpose of this course to teach you how to use Z, but rather to learn some of the basic notations common in most formal specifications languages, and see how these notations can be used to write state-based specifications using classical logic. In particular we will define the concepts of state schema and operation schemas.

We will then extend these notations in order to define **class-based formal specifications**, which are often used to formalise object-oriented systems. We will in this case borrow notations of class schemas and related features from the Object-Z specification language.

We will then move towards the implementation stage and introduce concepts of program specifications. We will in particular see how to specify Java classes using class invariants, pre- and post-conditions of methods, and loop invariant. We will see some reasoning techniques for checking the correctness of Java programs with respect to their specifications.

The course will then conclude with linking together these two parts (formal and program specifications), showing how formal specifications can be refined and transformed into program specifications.

Reading Material

- Books recommended are:
  - "**The Way of Z**", J. Jacky, Cambridge University Press, 1997.
  - "**Formal object-oriented specification using Object-Z**", Roger Duke and Gordon Rose, MacMillam Press Limited, 2000
  - "**Reasoned Programming**", Broda, Eisenbach, Khoshnevisan, and Vickers, Prentice Hall 1994.
  - "**Safeware: System Safety and Computers**", N. Leveson, Addison-Wesley, 1995.
  - "**Software Engineering**", Ian Sommerville, Addison-Wesley, 2008.
  - **Formal Methods** in general http://formalmethods.wikia.com/wiki/Formal_methods

- Slides and notes, complemented with information given during lectures and tutorials.

The first few chapters in the first book are basic introductions to formal specification (or more generally to formal methods). Remember that we are not adhering to the strict Z notation, but you will find in this book notations such as state schema and operation schemas, which we will also use in our formal specification examples.

The object-oriented approach to formal specifications will instead reflect pretty much the content of the second text book given here. This book presents main constructs of the Object-Z language via examples and small case studies. We will look in some detail at some of these case studies during our lectures.

The third text book covers instead the third part of our course, namely the reasoning about programs. The main reasoning techniques illustrated in this book, i.e. reasoning about correctness of individual methods and method calls within one class, are applicable to imperative type of programs as well as to object-oriented programs. The book gives examples based on Turing type of programs, whereas we will see how these reasoning techniques can be used to verify the correctness of Java programs.

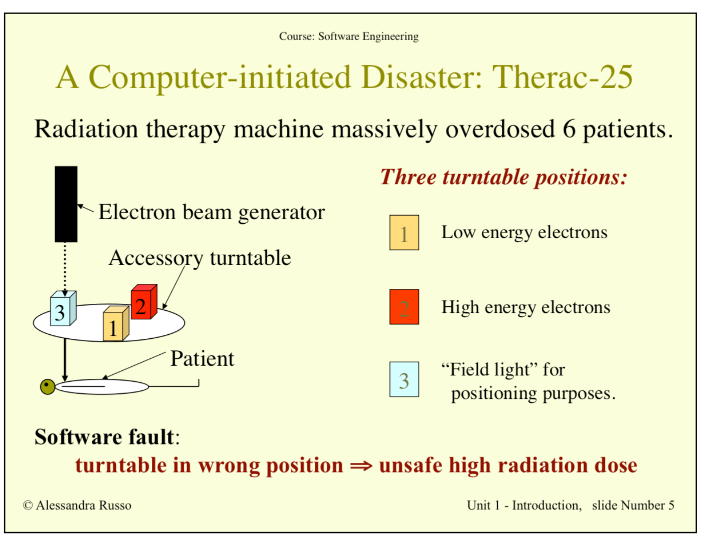
The forth text book is on formal specifications in a more general sense. It provides a collection of interesting real examples of computer-based system failures that could have been avoided if formal specifications had been used during the development cycle. It is listed here for your own general interest.

Finally, the last text book is one of the main standard reference text books for software engineering in large and it includes an introduction to formal specifications as well as an introduction of other aspects and issues related to the software development process in general.

For those who are adamant to learn more about formal specifications and formal methods in general a good source of information, with links to research papers is the Formal Methods Website listed in the slide.

The slides are available on the Web at https://wiki.imperial.ac.uk/display/~ar3/Software+Engineering,

or from my Web page at http://www-dse.doc.ic.ac.uk/cgi-bin/moin.cgi/ar3 following the Teaching link.



This case study is about a computer-controlled radiation therapy machine, called Therac-25, which has massively overdosed 6 people. This type of machine consists of both hardware and software components able to accelerate electrons to create high-energy beams that can destroy tumors with a minor impact on the surrounding healthy tissues. In this type of treatment, shallow tissues are treated with low energy accelerated electrons, whereas deeper tissues are treated with electron beams converted into X-ray photons. The special feature of this machine was to be able to deliver photons as well as electrons at various energy levels.

A major hardware component of the system was a turntable, which positioned accessory equipment under the beam path to produce two therapeutic modes: the electron mode (position 1) and the photon mode (position 2). The third position of the turntable did not involve any beam, but it was used to facilitate correct positioning of the patient under the machine. Therefore a correct operation of the machine strictly depended on the positioning of the turntable. Three micro switches monitored the position of the turntable.

In the *electron therapy*, the accelerator beam were able to generate different energy levels. In the *photon therapy*, only one (high) energy level was available, which was attenuated by a "beam flattener". If the machine had produced a photon beam, with the flattener (position 2) not in the correct position, then the patient would have received an unsafe high output dose. The turntable be in the wrong position meant the beam flattener be not in correct place.

When the turntable was in the third position, no accelerator beam should have ever been delivered.

In Therac-25, the computer was responsible for the positioning of the turntable.

Course: Software Engineering

Unsafe situations

A high energy electron beam could be used but mustn't hit patient directly.

High electron beam + turntable in position 1 or 3 = **unsafe**

- Predecessors Therac-6, Therac-20 had hardware interlocks. Therac-25 relied only on software checks.
- **Unsafe situations weren't detected and patients were killed.**
- **Main Cause**:

  Unsafe architecture & lack of specs → software bugs

© Alessandra Russo

As mentioned in the previous slide, to deliver X-ray (photon) treatments, a high energy electron beam was generated and flatted by a beam flattener attached to the turntable (i.e. position 2). One of the safety properties of the system was that such a high energy beam should have never hit the patient directly. Positions 1 or 3 were then wrong positions of the turntable, when the machine was programmed to deliver an X-ray treatment. **Extremely unsafe circumstances would therefore arise when the system was in X-ray mode, delivering a high energy electron beam and the turntable was in one of these wrong positions.**

Previous versions of Therac-25, called respectively Therac-6 and Therac-20 used only hardware interlocks to guarantee that the turntable and attached equipment were in the correct position before starting any treatment. The software component merely added convenience (e.g. interface) to the existing hardware system. In Therac-25, hardware interlocks were fully substituted by software checks. The software in Therac-25 had, therefore, more responsibility for maintaining safety than its predecessors. BUT software from the previous version systems were used in Therac-25 and some unsafe situations were not detected. As a consequence six accidents occurred of massive overdose, which resulted in the death of the patients.

The main failing factors were that Therac-25 was built by taking into account design features and modules of Therac-6 and by re-using some of the software routines developed for Therac-20. However, hidden bugs and problems of these two predecessors had been covered in these previous systems by their hardware interlock mechanisms. This wasn't the case for Therac-25.

The unsafe architecture of Therac-25 and the lack of formal specifications of the software developed for Therac-20 hadn't therefore revealed existing software bugs, which were aggravated by problems such as quality control, lack of good documentation of previous failures, etc.

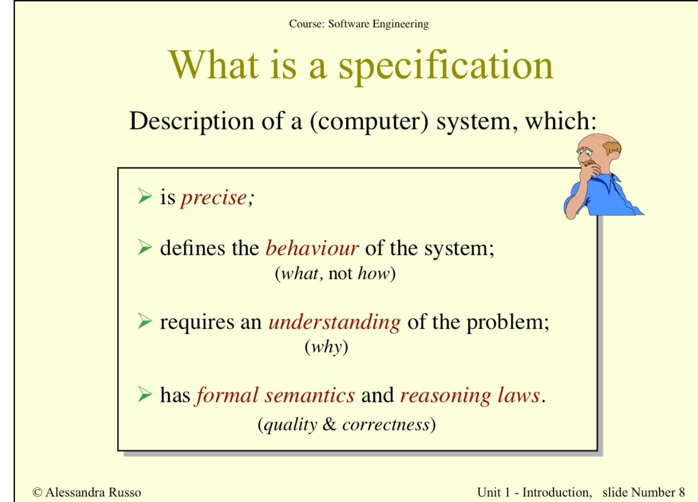
Course: Software Engineering

Lesson Learned

- Formal specifications and rigorous analysis of existing software with respect to the new system architecture would have highlighted the problems.
- A disaster could have been avoided.

© Alessandra Russo

Unit 1 - Introduction, slide Number 7



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, as we will see in the rest of this lecture, 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. They describe unambiguously the functionality 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. Reasoning about the program specification helps develop software that does something of value for the user (i.e. quality software) and that does conform with the specification (i.e. *correct software*).

Course: Software Engineering

Specification as Precision

Specifying is different from programming

- Defines abstract models capturing some view of the system.
- It can be used at different stages of the development process.
- Comes before any coding.
- Uses formal languages that need not be executable.
- Facilitates formal analysis of the system.

© Alessandra Russo

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. They are essentially **abstract models capturing some views of the system**. They can be constructed at different stages of the development process to reflect the way the various views evolve. For instance at a requirements stage they often reflect client and engineer's perception of the system and they model real-world entities. At design level, they are defined so to model software (virtual) entities, and at the implementation stage, they can be even more detailed so to capture some low-level code detail.

Formal specifications come 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.

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 programs 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 guide the implementation and to formally check whether a piece of code correctly implements a specification, i.e. meets its requirements.
In this way, you will not need to rely on error to validate programs.

Course: Software Engineering

Example

(A formal specification in BNF language)

```
<unsigned number> ::= <unsigned integer> | <unsigned real>
<unsigned integer> ::= <digit> {<digit>}
<unsigned real> ::=   <significant figure> |
                      <significant figure> <exponent part>
<significant figure> ::=  <unsigned integer> |
                          <unsigned integer>. | .<unsigned integer> |
                          <unsigned integer>.<unsigned integer> |
<exponent part>::= E <scale factor>
<scale factor> ::=  <unsigned integer> | <sign> <unsigned integer>
<sign> ::= + | -
```

Good numbers: 1E5, 0.5, 1.5E-5, 1. , .2
Bad number: E5, 1.5E5.0

© Alessandra Russo

This is an example of a formal specification written using the BNF grammar language and the BNF notation. This is just and example. Our formal specifications will not look like this!

The specification is about the definition of real numbers in Turing, with examples of good numbers and bad numbers. (Don't worry about understanding the notation!)

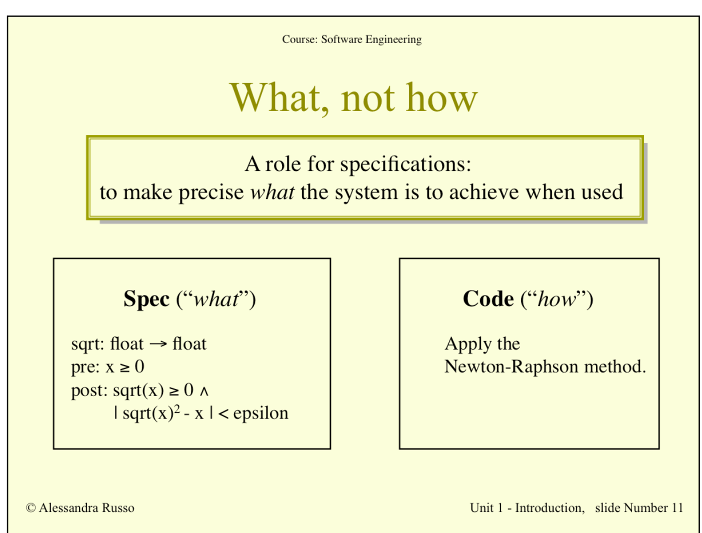
This specification expresses how numbers are syntactically constructed in Turing.

Such descriptions can be used by the programmer to *understand* the basic underlying problem of how to form numbers in Turing.

But, it also provides a formal specification for the part of a system, called the compiler, which recognizes whether numbers in a user's program are syntactically correct. This specification can be seen as a partial description of the intended behavior of a compiler, and compilers written by following this description will be correct Turing syntactic analyzers.

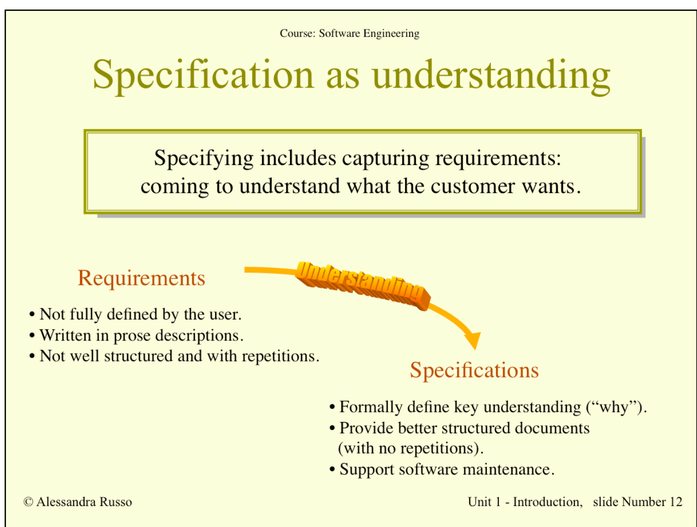
Developing syntactic analyzers without the use of a formal specification, would require, in this case, taking into account all possible numbers, and going through this (virtually) infinite collection of distinct cases to identify those that are correct and those that are syntactically illegal. Formal specifications tell us what the result of executing a program on any test case should be, without running it.

Formal specifications can help you create software so that you can understand it before you run it, and calculate its results. You shouldn't have to resort to guessing to produce programs, and neither should you have to rely on error to validate and improve them.



An implementation is there to tell the computer what to do, with 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). We will see examples of these specifications towards the end of the course.

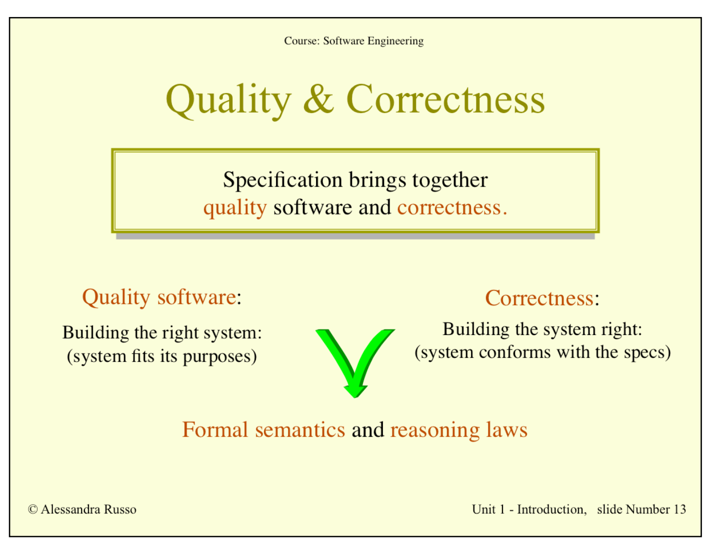
In this course we are interested in (large) systems that have to achieve something in the real world, where objects are less well-defined. Specifying "what" the system has to do also means making reference to the real world and incorporating assumptions about it. This is one of the challenging aspects 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 possible to simply 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 understand the problem and document it in a clear and organized 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 engineer goes through when specifying a system and 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 made as clear as possible and include the key steps in understanding what the engineer 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 not to loose sight of user needs. Once a formal specification of a system has been developed, system properties can be defined using the same formal language and checked whether they are met by the specification. This process, known as "**validation**", guarantees that the model of the system fits its purposes.

At the same time, it provides a means for proving that the implemented system satisfies its specified model. Refinements processes are often used to transform formal specifications into program specifications that preserve the same set of system properties and system requirements. Reasoning processes can then be used to analyze whether an implementation of the system meets its program specifications. This process is known as "**verification**" process, and guarantees that the implemented system is built correctly.

Specifications can therefore 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, and therefore to satisfy the user's requirements.

When are specifications useful

Novel systems	Specifications help identify appropriate solutions, prior to any implementation.
Difficult systems	Specifications help define solutions, not easily identifiable by guessing and testing.
Safety Critical systems	Specifications help prove that the system would achieve the promised behaviour before implementing it.

© Alessandra Russo

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 to identify a solution and to validate it.

Particularly relevant is the case of safety critical systems, such as aircraft control systems, or power station monitoring 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. It would be unprofessional to attempt the engineering of such software systems without producing a detailed and complete formal specification. Formal specifications are in this case useful for investigating how a software solution will behave, even before we begin design and coding. The formal reasoning techniques can be used to confirm that the software solution model described in the specification will meet its safety and critical requirements.

Course: Software Engineering

How can we use specifications?

Modelling

As mathematical models, to describe and predict the intended behaviour of the system. It helps focusing on some aspects of the system, leaving out inessential details.

Design

To support constructive approaches to design (e.g., "bottom-up"). Specifications would provide a description of what each building block does and enable us to calculate how the whole system will behave once the blocks are combined.

Verification

To show the final system does what we intend. Construct formal proofs, based on specification and program test, to show that the system satisfies its specification requirements.

© Alessandra Russo

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 to 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. This leads to the following use of specification as support for design.

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. This is particularly the case for class-based formal specifications, which can indeed be seen as a first definition of the architectural and design of the system in terms of classes and their inter-relationships.

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.

Course: Software Engineering

Formal languages

Z is one of the languages commonly used for writing state-based specifications. It contains a lot of special-purpose notation. Object-Z is an extension of Z, which facilitates the definition of class-based formal specifications.

We will

- Borrow some good notational ideas from Z and apply to classical logic.
- Learn in more details some main features of Object-Z.
- Use conventional logic to define program specifications

Results:

- Write state-based specifications that look like Z specifications.
- Write class-based Object-Z specifications.
- Map Object-Z specifications into (Java) program specifications.
- Use basic techniques for reasoning about correctness of Java programs.

© Alessandra Russo

A language commonly used for writing state-based 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 standard classical logic. Object-Z is instead an extension of Z that allows the definition of class-based specifications by providing appropriate constructs for specific object-oriented features, such as classes and inheritance.

In this course we will learn some of the main notations of Z and apply them to classical logic. We will then be able to write state-based specifications that look somewhat like Z but without being correct Z. We'll then look in more detail at part of the Object-Z language and object-oriented features in formal specifications, enough to enable you to write class-based formal specifications of simple real systems.

As for the specification of programs we will use mainly classical logic and we will see a (simple) approach for mapping Object-Z specifications into program specifications. Time permitting we will also see a particular formal language for specifying Java programs called JML language which is supported by an automated tool for reasoning about the correctness of Java programs

We are going to be using formal logical notations, 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 we 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

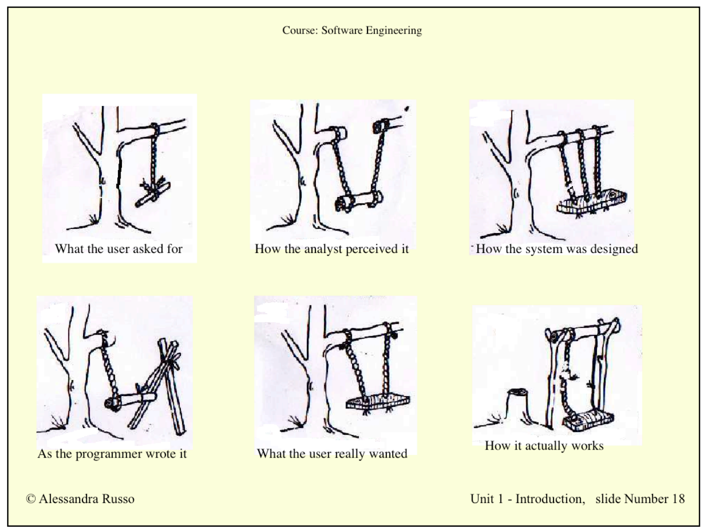
- modelling and validation,
- design,
- verification.

© Alessandra Russo

In this unit lecture, we have given 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).

We have also briefly seen some of the circumstances in which formal specification results to be very useful, and its location within the life cycle of a (computer) system.

In the next lecture we will introduce the schema notation and how classical logic can be used to write formal specifications.



© Alessandra Russo