Chapter 10
History of Software Engineering

Key Topics

Birth of Software Engineering
Floyd
Hoare
Formal Methods
Cleanroom and Software Reliability
Software Inspections and Testing
Project Management
Software Process Maturity Models

10.1 Introduction

The NATO Science Committee organised two famous conferences on software engineering in the late 1960s. The first conference was held in Garmisch, Germany, in 1968, and this was followed by a second conference in Rome in 1969. The Garmisch conference was attended by over 50 people from 11 countries.

The conferences highlighted the problems that existed in the software sector in the late 1960s, and the term '*software crisis*' was coined to refer to these problems. These included budget and schedule overruns of projects and problems with the quality and reliability of the delivered software. This led to the birth of *software engineering* as a separate discipline, and the realisation that programming is quite distinct from science and mathematics. Programmers are like engineers in the sense

G. O'Regan, *A Brief History of Computing*, DOI 10.1007/978-1-4471-2359-0_10,

that they design and build products; however, they need an appropriate education to design and develop software.[1]

The construction of bridges was problematic in the nineteenth century, and many people who presented themselves as qualified to design and construct bridges did not have the required knowledge and expertise. Consequently, many bridges collapsed, endangering the lives of the public. This led to legislation requiring an engineer to be licensed by the Professional Engineering Association prior to practising as an engineer. These engineering associations identify a core body of knowledge that the engineer is required to possess, and the licensing body verifies that the engineer has the required qualifications and experience. The licensing of engineers by most branches of engineering ensures that only personnel competent to design and build products actually do so. This in turn leads to products that the public can safely use. In other words, the engineer has a responsibility to ensure that the products are properly built and are safe for the public to use.

Parnas argues that traditional engineering be contrasted with the software engineering discipline where there is no licensing mechanism and where individuals with no qualifications can participate in the design and building of software products.[2] However, the fact that the maturity frameworks such as the CMMI or ISO 9000 place a strong emphasis on qualifications and training may help to deal with this.

The Standish Group conducted research in the late 1990s [Std:99] on the extent of current problems with schedule and budget overruns of IT projects. This study was conducted in the United States, but there is no reason to believe that European or Asian companies perform any better. The results indicate serious problems with on-time delivery.[3] Fred Brooks has argued that software is inherently complex, and that there is no silver bullet that will resolve all of the problems associated with software such as schedule overruns and software quality problems [Brk:75, Brk:86].

Poor-quality software can cause minor irritation or it may seriously disrupt the work of an organisation leading. It has in a very small number of cases led to the

[1] Software companies that are following approaches such as the CMM or ISO 9000:2000 consider the qualification of staff before assigning staff to performing specific tasks. The qualifications and experience required for the role are considered prior to appointing a person to carry out a particular role. Mature companies place significant emphasis on the education and continuous development of their staff and in introducing best practice in software engineering into their organisation. There is a growing trend among companies to mature their software processes to enable them to deliver superior results. One of the purposes that the original CMM served was to enable the US Department of Defense (DOD) to have a mechanism to assess the capability and maturity of software subcontractors.

[2] Modern HR recruitment specifies the requirements for a particular role, and interviews with candidates aim to establish that the candidate has the right education and experience for the role.

[3] It should be noted that these are IT projects covering diverse sectors including banking, telecommunications, etc., rather than pure software companies. Mature software companies using the CMM tend to be more consistent in project delivery with high quality.

deaths of individuals, for example, in the case of the Therac-25.[4] The Y2K problem, where dates were represented in a 2-digit format, required major rework for year 2000 compliance. Clearly, well-designed programs would have hidden the representation of the date, thereby minimising the changes required for year 2000 compliance. The quality of software produced by mature software companies committed to continuous improvement tends to be superior.

Mathematics plays a key role in engineering and may potentially assist software engineers deliver high-quality software products that are safe to use. Several mathematical approaches that can assist in delivering high-quality software are described in [ORg:06]. There is a lot of industrial interest in software process maturity for software organisations, and approaches to assess and mature software companies are described in [ORg:02, ORg:10].[5] These focus mainly on improving the effectiveness of the management, engineering and organisation practices related to software engineering.

10.2 What Is Software Engineering?

Software engineering involves multi-person construction of multi-version programs. The IEEE 610.12 definition of software engineering is:

Definition 10.1 (Software Engineering). *Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software, and the study of such approaches.*

Software engineering includes:

1. Methodologies to determine requirements, design, develop, implement and test software to meet customers' needs.
2. The philosophy of engineering, that is, an engineering approach to developing software is adopted. That is, products are properly designed, developed and tested, with quality and safety properly addressed.

[4] Therac-25 was a radiotherapy machine produced by the Atomic Energy of Canada Limited (AECL). It was involved in at least six accidents between 1985 and 1987 in which patients were given massive overdoses of radiation. The dose given was over 100 times the intended dose, and three of the patients died from radiation poisoning. These accidents highlighted the dangers of software control of safety-critical systems. The investigation subsequently highlighted the poor software design of the system and the poor software development practices employed.

[5] Approaches such as the CMM or SPICE (ISO 15504) focus mainly on the management and organisational practices required in software engineering. The emphasis is on defining and following the software process. In practice, there is often insufficient technical detail on requirements, design, coding and testing in the models, as the models focus on what needs to be done rather how it should be done.

3. Mathematics[6] may be employed to assist with the design and verification of software products. The level of mathematics to be employed will depend on the safety-critical nature of the product, as systematic peer reviews and testing are often sufficient in building quality into the software product.
4. Sound project management and quality management practices are employed.

Software engineering requires the engineer to state precisely the requirements that the software product is to satisfy and then to produce designs that will meet these requirements. Engineers provide a precise description of the problem to be solved; they then proceed to producing a design and validate the correctness of the design; finally, the design is implemented, and testing is performed to verify its correctness with respect to the requirements. The software requirements need to be unambiguous and should clearly state what is required, and it should also be evident what is not required.

Classical engineers produce the product design, and then analyse their design for correctness. Classical engineers will always use mathematics in their analysis as this is the basis of confirming that the specifications are met. The level of mathematics employed will depend on the particular application and calculations involved. The term '*engineer*' is generally applied only to people who have attained the necessary education and competence to be called engineers and who base their practice on mathematical and scientific principles. Often, in computer science, the term engineer is employed rather loosely to refer to anyone who builds things rather than to an individual with a core set of knowledge, experience and competence.

Parnas[7] is a strong advocate of the classical engineering approach, and he argues that computer scientists should have the right education to apply scientific and mathematical principles to their work. This includes mathematics and design, to enable them to be able to build high-quality and safe products. Baber has argued [Bab:11] that mathematics is the language of engineering. He argues that students should be shown how to turn a specification into a program using mathematics.

Parnas has argued that computer science tends to include a small amount of mathematics, whereas mathematics is a significant part of an engineering course and is the language of classical engineering. He argues that students are generally taught programming and syntax, but not how to design and analyse software. He advocates an engineering approach to the teaching of mathematics with an emphasis on its application to developing and analysing product designs.

He argues that software engineers need education on engineering mathematics, specification and design, converting designs into programs, software inspections

[6] There is no consensus at this time as to the appropriate role of mathematics in software engineering. My view is that the use of mathematics should be mandatory in the safety-critical and security-critical fields as it provides an extra level of quality assurance in these critical fields.

[7] Parnas's key contribution to software engineering is information hiding which is used in the object-oriented world. He has also done work (mainly of academic interest) on mathematical approaches to software quality.

and testing. The education should enable the software engineer to produce well-designed programs that will correctly implement the requirements.

Parnas argues that software engineers have individual responsibilities as professional engineers.[8] They are responsible for designing and implementing high-quality and reliable software that is safe to use. They are also accountable for their own decisions and actions[9] and have a responsibility to object to decisions that violate professional standards. Professional engineers have a duty to their clients to ensure that they are solving the real problem of the client. Engineers need to be honest about current capabilities, and when asked to work on problems that have no appropriate technical solution, this should be stated honestly rather than accepting a contract for something that cannot be done.

The licensing of a professional engineer provides confidence that the engineer has the right education and experience to build safe and reliable products. Otherwise, the profession gets a bad name as a result of poor work carried out by unqualified people. Professional engineers are required to follow rules of good practice and to object when the rules are violated.[10] The professional engineering body is responsible for enforcing standards and certification. The term 'engineer' is a title that is awarded on merit, but it also places responsibilities on its holder.

Engineers have a professional responsibility and are required to behave ethically with their clients. The membership of the professional engineering body requires the member to adhere to the code of ethics of the profession. Most modern companies have a code of ethics that employees are required to adhere to. It details the required ethical behaviour and responsibilities.

The approach used in current software engineering is to follow a well-defined software engineering process. The process includes activities such as project management, requirements gathering, requirements specification, architecture design,

[8] The concept of accountability is not new; indeed, the ancient Babylonians employed a code of laws ca. 1750 B.C. known as the Hammurabi Code. This code included the law that if a house collapsed and killed the owner, then the builder of the house would be executed.

[9] However, it is unlikely that an individual programmer would be subject to litigation in the case of a flaw in a program causing damage or loss of life. Most software products are accompanied by a comprehensive disclaimer of responsibility for problems rather than a guarantee of quality. Software engineering is a team-based activity involving several engineers in various parts of the project, and it could be potentially difficult for an outside party to prove that the cause of a particular problem is due to the professional negligence of a particular software engineer, as there are many others involved in the process such as reviewers of documentation and code and the various test groups. Companies are more likely to be subject to litigation, as a company is legally responsible for the actions of their employees in the workplace, and the fact that a company is a financially richer entity than one of its employees. However, the legal aspects of licensing software may protect software companies from litigation including those companies that seem to place little emphasis on software quality. However, greater legal protection for the customer can be built into the contract between the supplier and the customer for bespoke software development.

[10] Software companies that are following the CMMI or ISO 9000 will employ audits to verify that the rules have been followed. Auditors report their findings to management, and the findings are addressed appropriately by the project team and affected individuals.

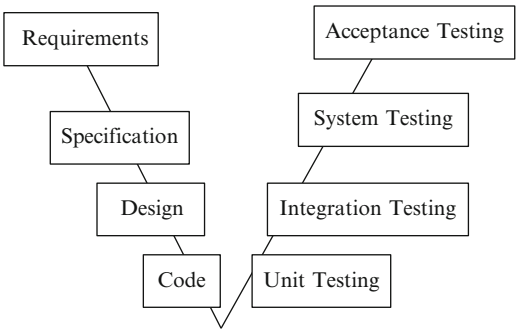

Fig. 10.1 Waterfall life cycle model (V-model)

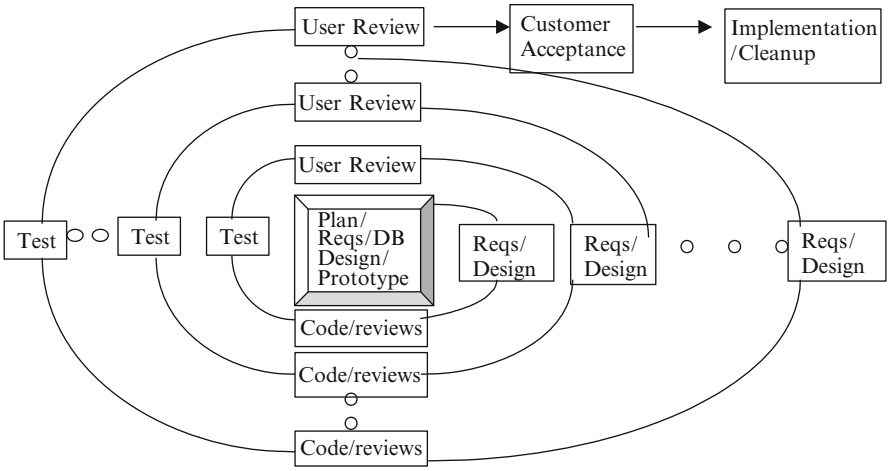

Fig. 10.2 Spiral life cycle model

software design, coding and testing. Most companies use a set of templates for the various phases. The waterfall model [Roy:70] and spiral model [Boe:88] are popular software development life cycles.

The waterfall model (Fig. 10.1) starts with requirements, followed by specification, design, implementation and testing. It is typically used for projects where the requirements can be identified early in the project life cycle or are known in advance. The waterfall model is also called the 'V' life cycle model, with the left-hand side of the 'V' detailing requirements, specification, design and coding and the right-hand side detailing unit tests, integration tests, system tests and acceptance testing. Each phase has entry and exit criteria which must be satisfied before the next phase commences. There are several variations of the waterfall model.

The spiral model (Fig. 10.2) is useful where the requirements are not fully known at project initiation. There is an evolution of the requirements during development which proceeds in a number of spirals, with each spiral typically involves updates to the requirements, design, code, testing and a user review of the particular iteration or spiral.

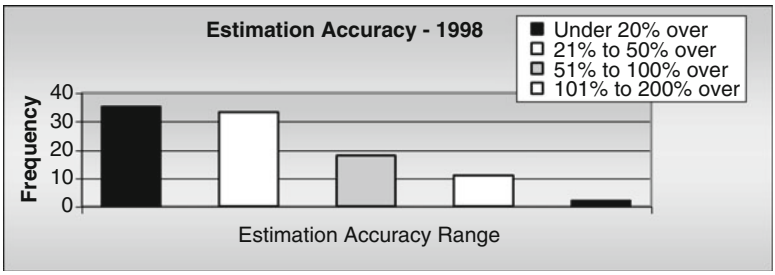

Fig. 10.3 Standish Group report: estimation accuracy

The spiral is, in effect, a reusable prototype, and the customer examines the current iteration and provides feedback to the development team to be included in the next spiral. The approach is to partially implement the system. This leads to a better understanding of the requirements of the system, and it then feeds into the next cycle in the spiral. The process repeats until the requirements and product are fully complete.

There are other life cycle models; for example, the Cleanroom approach to software development includes a phase for formal specification, and its approach to testing is quite distinct from other models as it is based on the predicted usage of the software product. Finally, the Rational Unified Process (RUP) has become popular in recent years.

The challenge in software engineering is to deliver high-quality software on time to customers. The Standish Group research (Fig. 10.3) on project cost overruns in the USA during 1998 indicates that 33% of projects are between 21% and 50% overestimate, 18% are between 51% and 100% overestimate and 11% of projects are between 101% and 200% overestimate.

Accurate project estimation of cost and effort are key challenges, and organisations need to determine how good their current estimation process actually is and to make improvements as appropriate. The use of software metrics allows effort estimation accuracy to be determined by computing the variance between actual project effort and the estimated project estimate.

Risk management is a key part of project management, and its objectives are to identify potential risks to the project, determine the probability of the risks occurring, assess the impact of each risk if it materialises, identify actions to eliminate the risk or to reduce its probability of occurrence, design contingency plans in place to address the risk if it materialises and finally, track and manage the risks throughout the project.

The concept of process maturity has become popular with the Capability Maturity Model, and the SEI has collected empirical data to suggest that there is a close relationship between software process maturity and the quality and the reliability of the delivered software. However, the main focus of the CMMI is management and organisation practices rather than on the technical engineering practices.

The implementation of the CMMI helps to provide a good engineering approach, as it places strict requirements on the processes and their characteristics that a company needs to have in place to provide a good engineering solution. The processes required include:

- Developing and managing requirements
- Doing effective design
- Planning and tracking projects
- Building quality into the product with peer reviews
- Performing rigorous testing
- Performing independent audits

There has been a growth of popularity among software developers in lightweight methodologies such as XP [Bec:00]. These methodologies view documentation with distaste, and often, software development commences prior to the full specification of the requirements.

10.3 Early Software Engineering

Robert Floyd was born in New York in 1936 and attended the University of Chicago. He became a computer operator in the early 1960s, an associate professor at Carnegie Mellow University in 1963 and a full professor of computer science at Sanford University in 1969. He did pioneering work on software engineering from the 1960s and made valuable contributions to the theory of parsing, the semantics of programming languages, program verification and methodologies for the creation of efficient and reliable software.

Mathematics and computer science were regarded as two completely separate disciplines in the 1960s, and software development was based on the assumption that the completed code would always contain defects. It was therefore better and more productive to write the code as quickly as possible and to then perform debugging to find the defects. Programmers then corrected the defects, made patches and retested and found more defects. This continued until they could no longer find defects. Of course, there was always the danger that defects remained in the code that could give rise to software failures.

Floyd believed that there was a way to construct a rigorous proof of the correctness of the programs using mathematics. He showed that mathematics could be used for program verification, and he introduced the concept of assertions that provided a way to verify the correctness of programs.

Flowcharts were employed in the 1960s to explain the sequence of basic steps for computer programs. Floyd's insight was to build upon flowcharts and to apply an invariant assertion to each branch in the flowchart. These assertions state the essential relations that exist between the variables at that point in the flowchart. An example relation is '$R = Z > 0, X = 1, Y = 0$'. He devised a general flowchart

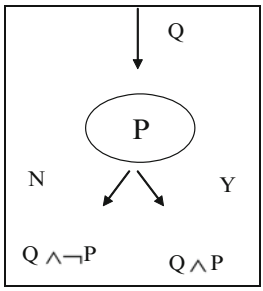

Fig. 10.4 Branch assertions in flowcharts

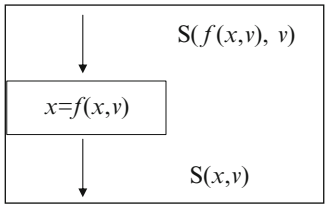

Fig. 10.5 Assignment assertions in flowcharts

language to apply his method to programming languages. The language essentially contains boxes linked by flow of control arrows [Flo:67].

Consider the assertion Q that is true on entry to a branch where the condition at the branch is P. Then, the assertion on exit from the branch is Q ∧ ¬P if P is false and Q ∧ P otherwise (Fig. 10.4).

The use of assertions may be employed in an assignment statement. Suppose x represents a variable and v represents a vector consisting of all the variables in the program. Suppose $f(x,v)$ represents a function or expression of x and the other program variables represented by the vector v. Suppose the assertion $S(f(x,v), v)$ is true before the assignment $x = f(x,v)$. Then the assertion $S(x,v)$is true after the assignment. This is given by (Fig. 10.5):

Floyd used flowchart symbols to represent entry and exit to the flowchart. This included entry and exit assertions to describe the program's entry and exit conditions.

Floyd's technique showed how a computer program is a sequence of logical assertions. Each assertion is true whenever control passes to it, and statements appear between the assertions. The initial assertion states the conditions that must be true for execution of the program to take place, and the exit assertion essentially describes what must be true when the program terminates.

His key insight was the recognition that if it can be shown that the assertion immediately following each step is a consequence of the assertion immediately preceding it, then the assertion at the end of the program will be true, provided the appropriate assertion was true at the beginning of the program.

He published an influential paper, 'Assigning Meanings to Programs', in 1967 [Flo:67], and this paper influenced Hoare's work on preconditions and post-conditions, leading to Hoare logic. This is a formal system of logic used for

programming language semantics and for program verification and was originally published in Hoare's 1969 paper 'An axiomatic basis for computer programming' [Hor:69].

Hoare recognised that Floyd's approach provided an effective method for proving the correctness of programs. He built upon Floyd's work to include all of the familiar constructs of high-level programming languages. This led to the axiomatic approach to defining the semantics of every statement in a programming language with axioms and proof rules. He introduced what has become known as the Hoare triple, and this describes how the execution of a fragment of code changes the state. A Hoare triple is of the form:

$$P\{Q\}R$$

where P and R are assertions and Q is a program or command. The predicate P is called the *precondition*, and the predicate R is called the *post-condition*.

Definition 10.2 (Partial Correctness). *The meaning of the Hoare triple above is that whenever the predicate P holds of the state before the execution of the command or program Q, then the predicate R will hold after the execution of Q. The brackets indicate partial correctness as if Q does not terminate then R can be any predicate.*

Total correctness requires Q to terminate, and at termination, R is true. Termination needs to be proved separately. Hoare logic includes axioms and rules of inference rules for the constructs of imperative programming language. Hoare and Dijkstra were of the view that the starting point of a project should always be the specification, and that the proof of the correctness of the program should be developed hand in hand along with the program itself. That is, one starts off with a mathematical specification of what a program is supposed to do, and mathematical transformations are applied to the specification until it is turned into a program that can be executed. The resulting program is then known to be correct by construction.

10.4 Software Engineering Mathematics

Mathematics plays a key role in the classical engineer's work. For example, bridge designers will develop a mathematical model of a bridge prior to its construction. The model is a simplification of the reality, and an exploration of the model enables a deeper understanding of the proposed bridge to be gained. Engineers will model the various stresses on the bridge to ensure that the bridge design can deal with the projected traffic flow. The engineer applies mathematics and models to the design of the product, and the analysis of the design is a mathematical activity.

Mathematics allows a rigorous analysis to take place and avoids an over-reliance on intuition. The emphasis is on applied mathematics to solve practical problems

and to develop products that are fit for use. The objective is therefore to teach students how to use and apply mathematics to program well and to solve practical problems. There is a rich body of classical mathematics available that may be applied to software engineering. This includes:

- Set theory
- Relations
- Functions
- Logic
- Calculus
- Functions
- Probability theory
- Graph theory
- Matrix theory

Mathematical approaches to software engineering are described in [ORg:06]. Next, we consider various formal methods that may be employed to assist in the development of high-quality software.

10.5 Formal Methods

The term 'formal methods' refers to various mathematical techniques used in the software field for the specification and formal development of software. Formal methods consist of formal specification languages or notations and employ a collection of tools to support the syntax checking of the specification, as well as the proof of properties of the specification. Abstraction is employed, and this allows questions to be asked about what the system does to be answered independently of the implementation. Furthermore, the unambiguous nature of mathematical notation avoids the problem of speculation about the meaning of phrases in an imprecisely worded natural language description of a system. Natural language is inherently ambiguous, whereas mathematics employs a precise notation with sound rules of inference. Spivey [Spi:92] defines formal specification as:

Definition 10.3 (Formal Specification). *Formal specification is the use of mathematical notation to describe in a precise way the properties which an information system must have, without unduly constraining the way in which these properties are achieved.*

The formal specification thus becomes the key reference point for the different parties involved in the construction of the system. It may be used as the reference point in the requirements, program implementation, testing and program documentation. The formal specification is a valuable means of promoting a common understanding for all those concerned with the system. The term '*formal methods*' is used to describe a formal specification language and a method for the design and implementation of computer systems.

The specification is written in a mathematical language, and the implementation is derived from the specification via stepwise refinement.[11] The refinement step makes the specification more concrete and closer to the actual implementation. There is an associated proof obligation that the refinement is valid, and that the concrete state preserves the properties of the more abstract state. Thus, assuming that the original specification is correct and the proofs of correctness of each refinement step are valid, then there is a very high degree of confidence in the correctness of the implemented software. Stepwise refinement is illustrated as follows: the initial specification S is the initial model M_0; it is then refined into the more concrete model M_1, and M_1 is then refined into M_2, and so on until the eventual implementation $M_{\mathrm{n}} = E$ is produced:

$$S = M_0 \subseteq M_1 \subseteq M_2 \subseteq M_3 \subseteq \ldots \subseteq M_n = \mathrm{E}$$

Requirements are the foundation from which the system is built, and irrespective of the best design and development practices, the product will be incorrect if the requirements are incorrect. The objective of requirements validation is to ensure that the requirements are correct and reflect what is actually required by the customer (in order to build the right system). Formal methods may be employed to model the requirements, and the model exploration yields further desirable or undesirable properties. The ability to prove that certain properties are true of the specification is very valuable, especially in safety-critical and security-critical applications. These properties are logical consequences of the definition of the requirements, and, if appropriate, the requirements may need to be amended appropriately. Thus, formal methods may be employed for requirements validation and in a sense to debug the requirements.

The use of formal methods generally leads to more robust software and to increased confidence in its correctness. The challenges involved in the deployment of formal methods in an organisation include the education of staff in formal specification, as formal specification and the use of mathematical techniques may be a culture shock to many staff.

Formal methods have been applied to a diverse range of applications, including the security-critical field, the safety-critical field, the railway sector, microprocessor verification, the specification of standards and the specification and verification of programs.

Formal methods have been criticised by Parnas and others on the grounds as enumerated in Table 10.1.

[11] It is questionable whether stepwise refinement is cost effective in mainstream software engineering, as it involves rewriting a specification ad nauseam. It is time-consuming to proceed in refinement steps with significant time also required to prove that the refinement step is valid. It is more relevant to the safety-critical field. Others in the formal methods field may disagree with this position.

Table 10.1 Criticisms of formal methods

No.	Criticism
1.	Often the formal specification is as difficult to read as the program[a]
2.	Many formal specifications are wrong[b]
3.	Formal methods are strong on syntax but provide little assistance in deciding on what technical information should be recorded using the syntax[c]
4.	Formal specifications provide a model of the proposed system. However, a precise unambiguous mathematical statement of the requirements is what is needed[d]
5.	Stepwise refinement is unrealistic. It is like, e.g. deriving a bridge from the description of a river and the expected traffic on the bridge. There is always a need for a creative step in design[e]
6.	Much unnecessary mathematical formalisms have been developed rather than using the available classical mathematics[f]

[a]Of course, others might reply by saying that some of Parnas's tables are not exactly intuitive, and that the notation he employs in some of his tables is quite unfriendly. The usability of all of the mathematical approaches needs to be enhanced if they are to be taken seriously by industrialists

[b]Obviously, the formal specification must be analysed using mathematical reasoning and tools to provide confidence in its correctness. The validation of a formal specification can be carried out using mathematical proof of key properties of the specification, software inspections or specification animation

[c]Approaches such as VDM include a method for software development as well as the specification language

[d]Models are extremely valuable as they allow simplification of the reality. A mathematical study of the model demonstrates whether it is a suitable representation of the system. Models allow properties of the proposed requirements to be studied prior to implementation

[e]Stepwise refinement involves rewriting a specification with each refinement step producing a more concrete specification (that includes code and formal specification) until eventually the detailed code is produced. However, tool support may make refinement easier

[f]Approaches such as VDM or Z are useful in that they add greater rigour to the software development process. They are reasonably easy to learn, and there have been some good results obtained by their use. Classical mathematics is familiar to students, and, therefore, it is desirable that new formalisms are introduced only where absolutely necessary

However, formal methods are potentially quite useful and reasonably easy to use. The use of a formal method such as Z or VDM forces the software engineer to be precise and helps to avoid ambiguities present in natural language. Clearly, a formal specification should be subject to peer review to provide confidence in its correctness. New formalisms need to be intuitive to be usable by practitioners. The advantage of classical mathematics is that it is familiar to students.

10.5.1 Why Should We Use Formal Methods?

There is a very strong motivation to use best practices in software engineering in order to produce software adhering to high-quality standards. Flaws in software may at best cause minor irritations or major damage to a customer's business including loss of life. Consequently, companies need to employ best practice to develop high-quality software, and formal methods are one leading-edge technology which may

be of benefit to companies in reducing the occurrence of defects in software products. Brown [Bro:90] argues that for the safety-critical field:

Comment 10.1 (Missile Safety). *Missile systems must be presumed dangerous until shown to be safe, and that the absence of evidence for the existence of dangerous errors does not amount to evidence for the absence of danger.*

It is quite possible that a software company may be sued for software which injures a third party, and this suggests that companies will need a quality assurance program that will demonstrate that every reasonable practice was considered to prevent the occurrence of defects.

There is some evidence to suggest that the use of formal methods provides savings in the cost of the project. For example, a 9% cost saving is attributed to the use of formal methods during the CICS project; the T800 project attributes a 12-month reduction in testing time to the use of formal methods. These are discussed in more detail in chapter one of [HB:95].

10.5.2 Applications of Formal Methods

Formal methods have been employed to verify correctness in the nuclear power industry, the aerospace industry, the security technology area and the railroad domain. These sectors are subject to stringent regulatory controls to ensure safety and security. Several organisations have piloted formal methods with varying degrees of success. These include IBM, who developed VDM at its laboratory in Vienna; IBM (Hursley) piloted the *Z* formal specification language in the CICS (Customer Information Control System) project.

The mathematical techniques developed by Parnas (i.e. requirements model and tabular expressions) have been employed to specify the requirements of the A-7 aircraft as part of a research project for the US Navy.[12] Tabular expressions have also been employed for the software inspection of the automated shutdown software of the Darlington nuclear power plant in Canada.[13] These are two successful uses of mathematical techniques in software engineering.

There are examples of the use of formal methods in the railway domain, and examples dealing with the modelling and verification of a railroad gate controller and railway signalling are described in [HB:95]. Clearly, it is essential to verify safety-critical properties such as '*when the train goes through the level crossing, then the gate is closed*'.

[12] However, the resulting software was never actually deployed on the A-7 aircraft.

[13] This was an impressive use of mathematical techniques, and it has been acknowledged that formal methods must play an important role in future developments at Darlington. However, given the time and cost involved in the software inspection of the shutdown software, some managers have less enthusiasm in shifting from hardware to software controllers [Ger:94].

10.5.3 Tools for Formal Methods

One key criticism of formal methods is the lack of available or usable tools to support the software engineer in writing the formal specification or in doing the proof. Many of the early tools were criticised as not being of industrial strength. However, in recent years, more advanced tools to support the software engineer's work in formal specification and formal proof have become available, and this is likely to continue in the coming years.

The tools include syntax checkers to check that the specification is syntactically correct, specialised editors to ensure that the written specification is syntactically correct, tools to support refinement, automated code generators to generate a high-level language corresponding to the specification, theorem provers to demonstrate the presence or absence of key properties and to prove the correctness of refinement steps and to identify and resolve proof obligations and specification animation tools where the execution of the specification can be simulated.

The *B*-Toolkit from *B*-Core is an integrated set of tools that supports the *B*-Method. These include syntax and type checking, specification animation, proof obligation generator, an auto-prover, a proof assistor and code generation. Thus, in theory, a complete formal development from initial specification to final implementation may be achieved, with every proof obligation justified, leading to a provably correct program.

The IFAD Toolbox[14] is a support tool for the VDM-SL specification language, and it includes support for syntax and type checking, an interpreter and debugger to execute and debug the specification and a code generator to convert from VDM-SL to C++. It also includes support for graphical notations such as the OMT/UML design notations.

10.5.4 Model-Oriented Approach

There are two key approaches to formal methods, namely, the model-oriented approach of VDM or *Z* and the algebraic or axiomatic approach. The latter includes the process calculi such as the calculus communicating systems (CCS) or communicating sequential processes (CSP).

A model-oriented approach to specification is based on mathematical models. A mathematical model is a mathematical representation or abstraction of a physical entity or system. The representation or model aims to provide a mathematical explanation of the behaviour of the system or the physical world. A model is considered suitable if its properties closely match the properties of the system,

[14] The IFAD Toolbox has been renamed to VDM Tools as IFAD sold the VDM Tools to CSK in Japan. The tools are expected to be available worldwide and will be improved further.

and if its calculations match and simplify calculations in the real system and if predictions of future behaviour may be made. The physical world is dominated by models, for example, models of the weather system, that enable predictions of the weather to be made, and economic models that enable predictions of the future performance of the economy may be made.

It is fundamental to explore the model and to consider the behaviour of the model and the behaviour of the physical world entity. The adequacy of the model is the extent to which it explains the underlying physical behaviour and allows predictions of future behaviour to be made. This will determine its acceptability as a representation of the physical world. Models that are ineffective will be replaced with newer models which offer a better explanation of the manifested physical behaviour. There are many examples in science of the replacement of one theory by a newer one. For example, the Copernican model of the universe replaced the older Ptolemaic model, and Newtonian physics was replaced by Einstein's theories on relativity. The structure of the revolutions that take place in science is described in [Kuh:70].

The model-oriented approach to software development involves defining an abstract model of the proposed software system. The model acts as a representation of the proposed system, and the model is then explored to assess its suitability. The exploration of the model takes the form of model interrogation, that is, asking questions and determining the effectiveness of the model in answering the questions. The modelling in formal methods is typically performed via elementary discrete mathematics, including set theory, sequences, functions and relations.

The modelling approach is adopted by the Vienna Development Method (VDM) and *Z*. VDM arose from work done in the IBM laboratory in Vienna in formalising the semantics for the PL/1 compiler, and it was later applied to the specification of software systems. The *Z* specification language had its origins in work done at Oxford University in the early 1980s.

10.5.5 Axiomatic Approach

The axiomatic approach focuses on the properties that the proposed system is to satisfy, and there is no intention to produce an abstract model of the system. The required properties and behaviour of the system are stated in mathematical notation. The difference between the axiomatic specification and a model-based approach is may be seen in the example of a stack.

The stack includes operators for pushing an element onto the stack and popping an element from the stack. The properties of *pop* and *push* are explicitly defined in the axiomatic approach. The model-oriented approach constructs an explicit model of the stack, and the operations are defined in terms of the effect that they have on the model. The specification of the *pop* operation on a stack is given by axiomatic properties, for example, $pop(push(s,x)) = s$.

Comment 10.2 (Axiomatic Approach). *The property-oriented approach has the advantage that the implementer is not constrained to a particular choice of implementation, and the only constraint is that the implementation must satisfy the stipulated properties.*

The emphasis is on the identification and expression of the required properties of the system, and implementation issues are avoided. The focus is on the specification of the underlying behaviour, and properties are typically stated using mathematical logic or higher-order logics. Mechanised theorem-proving techniques may be employed to prove results.

One potential problem with the axiomatic approach is that the properties specified may not be satisfiable in any implementation. Thus, whenever a 'formal axiomatic theory' is developed, a corresponding 'model' of the theory must be identified, in order to ensure that the properties may be realised in practice. That is, when proposing a system that is to satisfy some set of properties, there is a need to prove that there is at least one system that will satisfy the set of properties.

10.5.6 Proof and Formal Methods

The word *proof* has several connotations in various disciplines; for example, in a court of law, the defendant is assumed innocent until proven guilty. The proof of the guilt of the defendant may take the form of certain facts in relation to the movements of the defendant, the defendant's circumstances, the defendant's alibi, statements taken from witnesses, rebuttal arguments from the defence and certain theories produced by the prosecution or defence. Ultimately, in the case of a trial by jury, the defendant is judged guilty or not guilty depending on the extent to which the jury has been convinced by the arguments made by the prosecution and defence.

A mathematical proof typically includes natural language and mathematical symbols, and often, many of the tedious details of the proof are omitted. The strategy of proof in proving a conjecture is often *a divide and conquer* technique, that is, breaking the conjecture down into subgoals and then attempting to prove the subgoals. Most proofs in formal methods are concerned with cross-checking on the details of the specification or are concerned with checking the validity of refinement steps or proofs that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and theorem provers[15] are essential in assisting with this. Machine proof needs to be explicit, and reliance on some brilliant insight is avoided. Proofs by hand are notorious for containing errors or jumps in reasoning, while machine proofs are often extremely lengthy and unreadable.

[15] Most existing theorem provers are difficult to use and are for specialist use only. There is a need to improve the usability of theorem provers.

A mathematical proof consists of a sequence of formulae, where each element is either an axiom or derived from a previous element in the series by applying a fixed set of mechanical rules. One well-known theorem prover is the Boyer/Moore theorem prover [BoM:79]. There is an interesting case in the literature concerning the proof of correctness of the VIPER microprocessor[16] [Tie:91], and the actual machine proof consisted of several million formulae.

Theorem provers are invaluable in resolving many of the thousands of proof obligations that arise from a formal specification, and it is not feasible to apply formal methods in an industrial environment without the use of machine assisted proof. Automated theorem proving is difficult, as often mathematicians prove a theorem with an initial intuitive feeling that the theorem is true. Human intervention to provide guidance or intuition improves the effectiveness of the theorem prover.

The proof of various properties about a program increases confidence in its correctness. However, an absolute proof of correctness[17] is unlikely except for the most trivial of programs. A program may consist of legacy software which is assumed to work; it is created by compilers which are assumed to work correctly. Theorem provers are programs which are assumed to function correctly. The best that formal methods can claim is increased confidence in correctness of the software rather than an absolute proof of correctness.

10.5.7 The Future of Formal Methods

The debate concerning the level of use of mathematics in software engineering is still ongoing. Most practitioners are against the use of mathematics and avoid its use. They tend to employ methodologies such as software inspections and testing to improve confidence in the correctness of the software. They argue that in the current competitive industrial environment where time to market is a key driver, the use of such formal mathematical techniques would seriously impact the market opportunity. Industrialists often need to balance conflicting needs such as quality, cost and delivering on time. They argue that the commercial necessities require methodologies and techniques that allow them to achieve their business goals.

The other camp argues that the use of mathematics helps in the delivery of high-quality and reliable software, and that if a company does not place sufficient emphasis on quality will pay a price in terms of a poor reputation in the marketplace.

[16] This verification was controversial with RSRE and Charter overselling VIPER as a chip design that conforms to its formal specification.

[17] This position is controversial with others arguing that if correctness is defined mathematically, then the mathematical definition (i.e. formal specification) is a theorem, and the task is to prove that the program satisfies the theorem. They argue that the proofs for non-trivial programs exist, and that the reason why there are not many examples of such proofs is due to a lack of mathematical specifications.

It is generally accepted that mathematics and formal methods must play a role in the safety-critical and security-critical fields. Apart from that, the extent of the use of mathematics is a hotly disputed topic. The pace of change in the world is extraordinary, and companies face competitive forces in a global marketplace. It is unrealistic to expect companies to deploy formal methods unless they have clear evidence that it will support them in delivering commercial products to the marketplace ahead of their competition, at the right price and with the right quality. Formal method needs to prove that it can do this if it wishes to be taken seriously in mainstream software engineering. The issue of technology transfer of formal methods to industry is discussed in [ORg:06].

10.6 Propositional and Predicate Calculus

Propositional calculus associates a truth value with each proposition and is widely employed in mathematics and logic. There are a rich set of connectives employed in the calculus for truth functional operations, and these include $A \Rightarrow B, A \wedge B, A \vee B$ which denote, respectively, the conditional of A and B, the conjunction of A and B and the disjunction of A and B. A truth table may be constructed to show the results of these operations on the binary values of A and B. That is, A and B have the binary truth values of *true* and *false*, and the result of the truth functional operation is to yield a binary value. There are other logics that allow more than two truth values. These include, for example, the logic of partial functions which is a 3-valued logic. This logic allows a third truth value (the undefined truth value) for the proposition as well as the standard binary values of true and false.

Predicate calculus includes variables, and a formula in predicate calculus is built up from the basic symbols of the language. These symbols include variables; predicate symbols, including equality; function symbols, including the constants; logical symbols, for example, $\exists$, $\wedge$, $\vee$, $\neg$, etc.; and the punctuation symbols, for example, brackets and commas. The formulae of predicate calculus are built from terms, where a *term* is a key construct and is defined recursively as a variable or individual constant or as some function containing terms as arguments. A formula may be an atomic formula or built from other formulae via the logical symbols. Other logical symbols are then defined as abbreviations of the basic logical symbols.

An interpretation gives meaning to a formula. If the formula is a sentence (i.e. it does not contain any free variables), then the given interpretation is true or false. If a formula has free variables, then the truth or falsity of the formula depends on the values given to the free variables. A formula with free variables essentially describes a relation say, $R(x_1,x_2,\ldots, x_n)$ such that $R(x_1,x_2,\ldots, x_n)$ is true if $(x_1,x_2,\ldots, x_n)$ is in relation R. If a formula with free variables is true irrespective of the values given to the free variables, then the formula is true in the interpretation.

A valuation function is associated with the interpretation, and this gives meaning to the formulae in the language. Thus, associated with each constant c is a constant c_Σ in some universe of values Σ; with each function symbol f, we have a function

symbol f_Σ in Σ; and for each predicate symbol P, we have a relation P_Σ in Σ. The valuation function, in effect, gives a semantics to the language of the predicate calculus L. The truth of a proposition P is then defined in the natural way, in terms of the meanings of the terms, the meanings of the functions, predicate symbols and the normal meanings of the connectives.

Mendelson [Men:87] provides a rigorous though technical definition of truth in terms of satisfaction (with respect to an interpretation M). Intuitively, a formula F is *satisfiable* if it is *true* (in the intuitive sense) for some assignment of the free variables in the formula F. If a formula F is satisfied for every possible assignment to the free variables in F, then it is *true* (in the technical sense) for the interpretation M. An analogous definition is provided for *false* in the interpretation M.

A formula is *valid* if it is true in every interpretation; however, as there may be an uncountable number of interpretations, it may not be possible to check this requirement in practice. M is said to be a model for a set of formulae if and only if every formula is true in M.

There is a distinction between proof theoretic and model theoretic approaches in predicate calculus. *Proof theoretic* is essentially syntactic, and we have a list of axioms with rules of inference. In this way, the theorems of the calculus may be logically derived (written as $\vdash A$). In essence, the logical truths are a result of the syntax or form of the formulae rather than the *meaning* of the formulae. *Model theoretical*, in contrast, is essentially semantic. The truths derive essentially from the meaning of the symbols and connectives rather than the logical structure of the formulae. This is written as $\vdash_M A$.

A calculus is *sound* if all the logically valid theorems are true in the interpretation, that is, proof theoretic $\Rightarrow$ model theoretic. A calculus is complete if all the truths in an interpretation are provable in the calculus, that is, model theoretic $\Rightarrow$ proof theoretic. A calculus is *consistent* if there is no formula A such that $\vdash A$ and $\vdash \neg A$.

10.7 Unified Modelling Language

The unified modelling language (UML) is a visual modelling language for software systems, and it facilitates the understanding of the architecture of the system and in managing the complexity of large systems. It was developed by Jim Rumbaugh, Grady Booch and Ivar Jacobson [Jac:99a] as a notation for modelling object-oriented systems.

It allows the same information to be presented in several different ways, and there are UML diagrams for alternate viewpoints of the system. Use cases describe scenarios or sequences of actions for the system from the user's viewpoint. For example, typical user operations at an ATM machine include the balance inquiry operation, the withdrawal of cash and the transfer of funds from one account to another. These operations can be described with UML use case diagrams.

Class and object diagrams are a part of UML, and the concept of class and objects is taken from object-oriented design. The object diagram is related to the

class diagram in that the object is an instance of the class. There will generally be several objects associated with the class. The class diagram describes the data structure and the allowed operations on the data structure. Two key classes are customers and accounts for an ATM system, and this includes the data structure for customers and accounts and also the operations on customers and accounts. The operations include adding or removing a customer and operations to debit or credit an account. The objects of the class are the actual customers of the bank and their corresponding accounts.

Sequence diagrams show the interaction between objects/classes in the system for each use case. UML activity diagrams are similar to flowcharts. They are used to show the sequence of activities in a use case and include the specification of decision branches and parallel activities. State diagrams (or state charts) show the dynamic behaviour of a class and how different operations result in a change of state. There is an initial state and a final state, and the different operations result in different states being entered and exited.

UML offers a rich notation to model software systems and to understand the proposed system from different viewpoints. The main advantage of UML includes the fact that it is an expressive visual modelling language that allows a study of the proposed system prior to implementation. It allows the system to be visualised from different viewpoints and provides an effective mechanism to communicate the proposed behaviour of the software system.

10.8 Software Inspections and Testing

Software inspections and testing play a key role in building quality into software products and verifying that the products are of high quality. The Fagan Inspection Methodology was developed by Michael Fagan of IBM in the mid-1970s [Fag:76]. It is a seven-step process that identifies and removes errors in work products. There is a strong economic case for identifying defects as early as possible, as the cost of correction increases the later a defect is discovered in the life cycle. The methodology mandates that requirement documents, design documents, source code and test plans are all formally inspected by independent experts to ensure quality.

There are several *roles* defined in the process including the *moderator* who chairs the inspection; the *reader's* responsibility is to read or paraphrase the particular deliverable; the *author* is the creator of the deliverable and has a special interest in ensuring that it is correct; and the *tester* role is concerned with the testing viewpoint.

The inspection process will consider whether a design is correct with respect to the requirements and whether the source code is correct with respect to the design. There are seven stages in the inspection process [ORg:02]:

- Planning
- Overview

- Prepare
- Inspect
- Process improvement
- Rework
- Follow-up

The errors identified in an inspection are classified into various types, and mature organisations record the inspection data in a database for further analysis. Measurement allows the effectiveness of the organisation in identifying errors in phase and detecting defects out of phase to be determined and improved. Tom Gilb has defined an alternate inspection methodology [Glb:94].

Software testing plays a key role in verifying that a software product is of high quality and conforms to the customer's quality expectations. Testing is both a constructive activity in that it is verifying the correctness of functionality, and it is also a destructive activity in that the objective is to find as many defects as possible in the software. The testing verifies that the requirements are correctly implemented as well as identifies whether any defects are present in the software product.

There are various types of testing such as unit testing, integration testing, system testing, performance testing, usability testing, regression testing and customer acceptance testing. The testing needs to be planned to ensure that it is effective. Test cases will need to be prepared and executed, the results reported and any issues corrected and retested. The test cases will need to be appropriate to verify the correctness of the software. The quality of the testing is dependent on the maturity of the test process, and a good test process will include:

- Test planning and risk management
- Dedicated test environment and test tools
- Test case definition
- Test automation
- Formality in handover to test department
- Test execution
- Test result analysis
- Test reporting
- Measurements of test effectiveness
- Postmortem and test process improvement

Metrics are generally maintained to provide visibility into the effectiveness of the testing process. Testing is described in more detail in [ORg:02, ORg:10].

10.9 Process Maturity Models

The Software Engineering Institute (SEI) developed the Capability Maturity Model (CMM) in the early 1990s as a framework to help software organisations to improve their software process maturity and to implement best practice in software and

Fig. 10.6 W. Edwards Deming (Courtesy of W. Edwards Deming Institute)

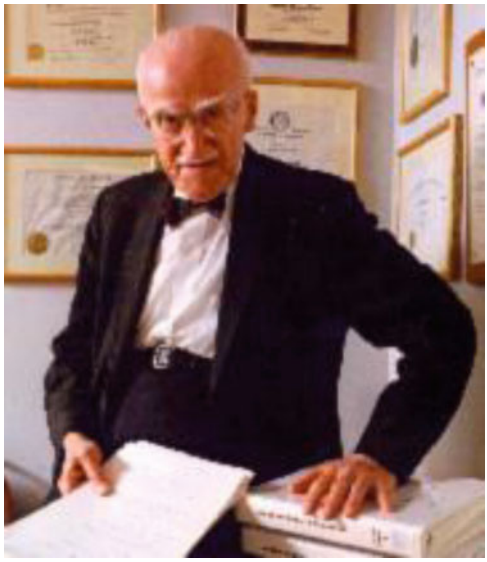

Fig. 10.7 W. Joseph Juran (Courtesy of Juran Institute)

systems engineering. The SEI believes that there is a close relationship between the maturity of software processes and the quality of the delivered software product. The CMM applied the ideas of Deming [Dem:86], Juran [Jur:00] and Crosby [Crs:79] to the software field. These quality gurus were influential in transforming manufacturing companies with quality problems to effective quality-driven organisations with a reduced cost of poor quality (Fig. 10.6).

They recognised the need to focus on the process, and software organisations need to improve their software development processes as well as the product. Watt Humphries did early work on software process improvement at IBM [Hum:89], and he moved to the SEI in the late 1980s. This led to the first version of the CMM in 1991. It is now called the Capability Maturity Model Integration (CMMI®) [CKS:11] (Fig. 10.7).

Fig. 10.8 Watts Humphrey (Courtesy of Watts Humphrey)

The CMMI consists of five maturity levels, with each maturity level (except level 1) consisting of several process areas. Each process area consists of a set of goals that must be satisfied for the process area to be satisfied. The goals for the process area are implemented by practices related to that process area, and the implementation of these practices leads to an effective process. Processes need to be defined and documented. The users of the process need to receive appropriate training to enable them to carry out the process, and processes need to be enforced by independent audits (Fig. 10.8).

The emphasis on level 2 of the CMMI is on maturing management practices such as project management, requirements management, configuration management and so on. The emphasis on level 3 of the CMMI is to mature engineering and organisation practices. This maturity level includes peer reviews and testing, requirements development, software design and implementation practices and so on. Level 4 is concerned with ensuring that key processes are performing within strict quantitative limits and adjusting processes, where necessary, to perform within these defined limits. Level 5 is concerned with continuous process improvement which is quantitatively verified.

Maturity levels may not be skipped in the staged implementation of the CMMI. There is also a continuous representation of the CMMI which allows the organisation to focus on improvements to key processes. However, in practice, it is often necessary to implement several of the level 2 process areas before serious work can be done on implementing a process at a higher maturity level. The use of metrics [Fen:95, Glb:76] becomes more important as an organisation matures, as metrics allow the performance of an organisation to be objectively judged. The higher CMMI maturity levels set quantitative levels for processes to perform within.

The CMMI allows organisations to benchmark themselves against other similar organisations. This is done by formal SEI approved SCAMPI appraisals conducted by an authorised SCAMPI lead appraiser. The results of a SCAMPI appraisal are generally reported back to the SEI, and there is a strict qualification process to become an authorised SCAMPI lead appraiser. An appraisal is useful in verifying that an organisation has improved, and it enables the organisation to prioritise improvements for the next improvement cycle.

The time required to implement the CMMI in an organisation depends on the current maturity and size of the organisation. It generally takes 1–2 years to implement maturity level 2, and a further 1–2 years to implement level 3.

10.10 Review Questions

1. Describe the crisis in software in the 1960s and the birth of software engineering.
2. Describe waterfall and spiral life cycle models including their advantages and disadvantages.
3. Discuss Floyd's contribution to software engineering and how it led to Hoare's axiomatic semantics.
4. Describe the mathematics that is potentially useful in software engineering.
5. Describe formal methods and their applications to software engineering. Explain when their use should be considered in software engineering.
6. Discuss any tools to support formal methods that you are familiar with.
7. Discuss the similarities and differences between Z and VDM.
8. Discuss the similarities and differences between the model-oriented approach and the axiomatic approach to formal methods.
9. Discuss UML and its applicability to software engineering.
10. Discuss the applicability of software inspections and testing to software engineering.
11. Discuss the Capability Maturity Model and its applicability to software engineering.

10.11 Summary

This chapter considered a short history of some important developments in software engineering from its birth at the Garmisch conference in 1968. It was recognised that there was a crisis in the software field, and there was a need for sound methodologies to design, develop and maintain software to meet customer needs.

Classical engineering has a successful track record in building high-quality products that are safe for the public to use. It is therefore natural to consider using an engineering approach to developing software, and this involves identifying the customer requirements, carrying out a rigorous design to meet the requirements, developing and coding a solution to meet the design and conducting appropriate inspections and testing to verify the correctness of the solution.

Mathematics plays a key role in engineering to assist with design and verification of software products. It is therefore reasonable to apply appropriate mathematics in software engineering (especially for safety-critical systems) to assure that the delivered systems conform to the requirements. The extent to which mathematics will need to be used is controversial with strong views on both sides. In many cases, peer reviews and testing will be sufficient to build quality into the software product. In other cases, and especially with safety and security-critical applications, it is desirable to have the extra assurance that may be provided by mathematical techniques.

Various mathematical approaches were considered including Z, VDM, propositional calculus and predicate calculus. The nature of mathematical proof and supporting theorem provers were discussed.

There is a lot more to the successful delivery of a project than just the use of mathematics or peer reviews and testing. Sound project management and quality management practices are essential, as a project that is not properly managed will suffer from schedule, budget or cost overruns as well as problems with quality.

Maturity models such as the CMMI can assist organisations in maturing key management and engineering practices that are essential for the successful delivery of high-quality software. The use of the CMMI helps companies in their goals to deliver high-quality software systems that are consistently on time and consistently meet business requirements.