Software Engineering Mathematics 17

Key Topics

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

17.1 Introduction

The NATO Science Committee organized two famous conferences on software engineering in the late 1960s. The first conference was held in Garmisch, Germany, in 1968, and it was followed by a second conference in Rome in 1969. The Garmisch conference was attended by over fifty 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 conference led to the birth of *software engineering* as a separate discipline, and the realization that programming is quite distinct from science and mathematics. Programmers are like engineers in the sense that they design and build products. Therefore, they need an appropriate

G. O'Regan, *Guide to Discrete Mathematics*, Texts in Computer Science,
DOI 10.1007/978-3-319-44561-8_17

software engineering education (not just on the latest technologies but on the fundamentals of engineering) in order to properly design and develop software.

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 prior to practicing 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.[1] However, best practice in modern HR places a strong emphasis on the qualification of staff.

The Standish group has conducted research since the late 1990s [1] on the extent of problems with schedule and budget overruns of IT projects. The results indicate serious problems with on-time delivery, cost overruns and quality.[2] 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 projects such as schedule overruns and software quality problems [2, 3].

Poor quality software can at best cause minor irritation to clients, and in some circumstances it may seriously disrupt the work of the client organization leading to injury or even the death of individuals (e.g. as in the case of the Therac-25[3] radiotherapy machine). The Y2K problem occurred due to poor design, as the representation of the date used two digits to record the year rather than four. Its correction required major rework, as it was necessary to examine all existing software code to determine how the date was represented, and to make appropriate

[1]Modern HR recruitment specifies the requirements for a particular role, and the interviews establish whether the candidate is suitably qualified, and has the appropriate experience for the role. Parnas is arguing against the content of courses that emphasize the latest technologies rather than the fundamentals of engineering.

[2]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.

[3]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.

corrections. Clearly, well-designed programs would have hidden the representation of the date thereby minimizing the changes required for year 2000 compliance.

Mathematics plays a key role in engineering, and it may potentially assist software engineers in delivering high-quality software products that are safe to use. Several mathematical approaches that may assist in delivering high-quality software are described in [4]. However, it is important to recognise that while the use of mathematics is suitable for some areas of software engineering (especially in the safety and security critical fields), less rigorous techniques (such as software inspections and testing) are sufficient in most other areas of software engineering.

There is a lot of industrial interest in approaches to mature software engineering practices in software organizations (e.g. the use of software process maturity models such as the CMMI). These include approaches to assess and mature the software engineering processes in software companies, and they are described in [5, 6].[4] Software process improvement focuses mainly on improving the effectiveness of the management, engineering and organization practices related to software engineering.

17.2 What Is Software Engineering?

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

Definition 17.1 (*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 the following:

1. Methodologies to determine requirements, design, develop, implement and test software to meet customers' needs.
2. The philosophy of engineering: i.e. an engineering approach to developing software is adopted. That is, products are properly designed, developed, tested, with quality and safety properly addressed.
3. Mathematics[5] may be employed to assist with the design and verification of software products. The level of mathematics to be employed will depend on the

[4]The process maturity models focus mainly on the management, engineering and organizational practices required in software engineering. The models focus on what needs to be done rather how it should be done.

[5]There is no consensus at this time as to the appropriate role of mathematics in software engineering. The use of mathematics is invaluable in the safety critical and security critical fields as it provides an extra level of confidence in the correctness of the software.

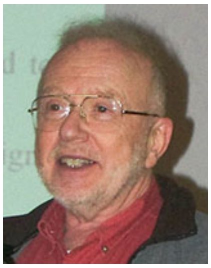
Fig. 17.1 David Parnas

safety critical nature of the product, as systematic peer reviews and testing are often sufficient.

4. Sound project 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 validating its correctness; finally, the design is implemented and testing is performed to verify the correctness of the implementation with respect to the requirements. The software requirements needs to be unambiguous, and should clearly state what is and what is not required.

Classical engineers produce the product design, and then analyse their design for correctness. They 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[6] (Fig. 17.1) 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 [7] 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 advocates a solid 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, and testing. The education should enable the software engineer to produce well-designed programs that will correctly implement the requirements.

[6]Parnas has made important contributions to software engineering including information hiding which is used in the object-oriented world.

He argues that software engineers have individual responsibilities as professional engineers.[7] 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,[8] and have a responsibility to object to decisions that violate professional standards. Professional engineers need to be honest about current capabilities, especially when asked to work on problems that have no appropriate technical solution. Another words, they should be honest and avoid 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. Professional engineers are required to follow rules of good practice, and to object when the rules are violated.[9] 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.

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, software design, coding, and testing. Most companies use a set of templates for the various phases. The waterfall model [8] and spiral model [9] are popular software development lifecycles.

The waterfall model (Fig. 17.2) starts with requirements, followed by specification, design, implementation, and testing. It is typically used for projects where the requirements can be identified and it is often called the 'V' life cycle model. The left-hand side of the 'V' involves requirements, specification, design, and coding and the right-hand side is concerned with unit tests, integration tests, system tests and acceptance testing. Each phase has entry and exit criteria that must be satisfied before the next phase commences. There are several variations of the waterfall model.

The spiral model (Fig. 17.3) 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

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

[8]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. A comprehensive disclaimer of responsibility for problems rather than a guarantee of quality accompany most software products. 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.

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

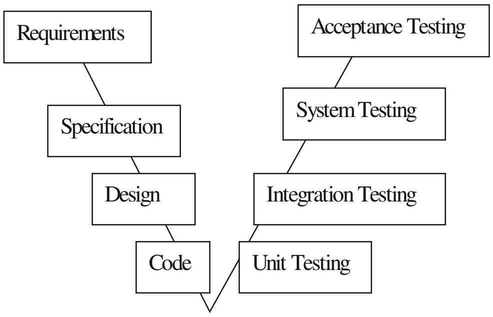

Fig. 17.2 Waterfall lifecycle model (V-model)

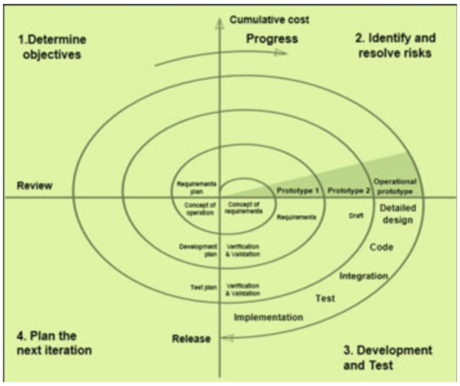

Fig. 17.3 Spiral lifecycle model

involves updates to the requirements, design, code, testing and a user review of the particular iteration or spiral.

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 has been a growth of popularity among software developers in lightweight methodologies such as *Agile*. This is a software development methodology that claims to be more responsive to customer needs than traditional methods such as the waterfall model. *The waterfall development model is similar to a wide and slow moving value stream*, and halfway through the project 100 % if the requirements are typically 50 % done. *However, for agile development 50 % of requirements are typically 100 % done halfway through the project.*

Ongoing changes to requirements are considered normal in the Agile world, and it is believed to be more realistic to change requirements regularly throughout the project rather than attempting to define all of the requirements at the start of the project. The methodology includes controls to manage changes to the requirements,

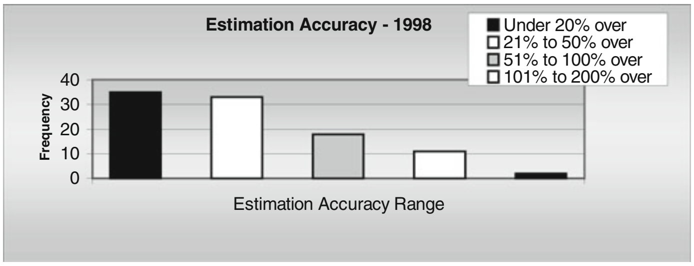

Fig. 17.4 Standish group report estimation accuracy

and good communication and early regular feedback is an essential part of the process.

A story may be a new feature or a modification to an existing feature. It is reduced to the minimum scope that can deliver business value, and a feature may give rise to several stories. Stories often build upon other stories and the entire software development lifecycle is employed for the implementation of each story. *Stories are either done or not done*, i.e. *there is such thing as a story being 80 % done*. The story is complete only when it passes its acceptance tests. For more details on Agile see [6, 10].

The challenge in software engineering is to deliver high-quality software on time to customers. The Standish Group research (Fig. 17.4) on project cost overruns in the US during 1998 showed that 33 % of projects are between 21 and 50 % over estimate, 18 % are between 51 and 100 % over estimate, and 11 % of projects are between 101 and 200 % overestimate.

The accurate estimation of project cost and effort are key challenges, and project managers need to determine how good their current estimation process actually is and to make improvements. Many companies today employ formal project management methodologies such as Prince 2 or Project Management Professional (PMP). These methodologies allow projects to be rigorously managed and include processes for initiating a project, planning a project, executing a project, monitoring and controlling a project and closing a project.

The Capability Maturity Model developed by the Software Engineering Institute (SEI) has become useful in software engineering. 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. The CMMI enables the organization to improve processes as follows:

- Developing and managing requirements
- Design activities
- Configuration Management
- Selection and Management of Suppliers
- Planning and Managing projects
- Building quality into the product with peer reviews

- Performing rigorous testing
- Performing independent audits

The rest of this chapter is focused on mathematical techniques to support software engineering to improve software quality, and the chapter concludes with a short discussion on software inspections and testing and process maturity models. For a more detailed account of software engineering see [6].

17.3 Early Software Engineering Mathematics

Robert Floyd was born in New York in 1936, and he did pioneering work on software engineering from the 1960s (Fig. 17.5). He made important 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 re-tested 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.

Fig. 17.5 Robert Floyd

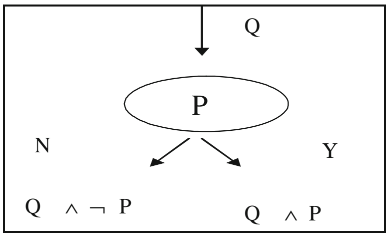

Fig. 17.6 Branch assertions in flowcharts

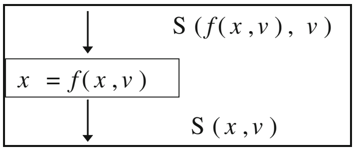

Fig. 17.7 Assignment assertions in flowcharts

An example relation is "$R = Z > 0, X = 1, Y = 0$". He devised a general flowchart language to apply his method to programming languages. The language essentially contains boxes linked by flow of control arrows [11].

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 $Q \wedge \neg P$ if P is false and $Q \wedge P$ otherwise (Fig. 17.6).

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 (Fig. 17.7). This is given by:

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.

Fig. 17.8 C.A.R Hoare

He published an influential paper, "Assigning Meanings to Programs", in 1967 [11], and this paper influenced Hoare's work on preconditions and post-conditions leading to Hoare logic [12]. Floyd's paper also presented a formal grammar for flowcharts, together with rigorous methods for verifying the effects of basic actions like assignments.

Hoare logic is a formal system of logic used for programming semantics and for program verification. It was developed by C.A.R. Hoare (Fig. 17.8), and was originally published in Hoare's 1969 paper "An axiomatic basis for computer programming" [12]. Hoare and others have subsequently refined it, and it provides a logical methodology for precise reasoning about the correctness of computer programs.

Hoare was influenced by Floyd's 1967 paper that applied assertions to flowcharts, and he recognised that this provided an effective method for proving the correctness of programs. He built upon Floyd's approach to cover 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, and the approach consists of 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 *postcondition*.

Definition 4.2 (*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. R may be chosen to be false to express that Q does not terminate.

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 program should always be the specification, and that the proof of the correctness of the program should be developed along with the program itself.

That is, the starting point is the mathematical specification of what a program is 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.

17.4 Mathematics in Software Engineering

Mathematics plays a key role in classical engineering to assist with design and verification of software products. It is therefore reasonable to apply appropriate mathematics in software engineering (especially for safety and security critical systems) to assure that the delivered systems conform to the requirements. The extent to which mathematics should be used is controversial with strong views in both camps. 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 with mathematical techniques.

Mathematics allows a rigorous analysis to take place and avoids an over-reliance on intuition. The emphasis is on applying mathematics to solve practical problems and to develop products that are fit for use. Engineers are taught how to apply mathematics in their work, and the emphasis is always on the application of mathematics to solve practical problems.

Classical mathematics may be applied to software engineering and specialized mathematical methods and notations have also been developed. The classical mathematics employed includes sets, relations, functions, logic, graph theory, automata theory, matrix theory, probability and statistics, calculus, and matrix theory. Specialized formal specification languages such as *Z* and VDM have been developed, and these allow the requirements to be formally specified in precise mathematical language.

The term 'formal method' 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 about the specification. The term 'formal method' is used to describe a formal specification language and a method for the design and implementation of computer systems.

The mathematical analysis of the formal specification allows questions to be asked about what the system does, and these questions may be answered independently of the implementation. Mathematical notation is precise, and this helps to avoid the problem of ambiguity inherent in a natural language description of a system. The formal specification may be used to promote a common understanding for all stakeholders.

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

There are various tools to support formal methods including syntax checkers; specialized editors; tools to support refinement; automated code generators; theorem provers; and specification animation tools. Formal methods need to mature further before they will be used in mainstream software engineering, and they are described in more detail in Chap. 18.

17.5 Software Inspections and Testing

Software inspections play an important role in building quality into software products. The Fagan Inspection Methodology was developed by Michael Fagan at IBM in the mid-1970s [13]. It is a seven-step process that identifies and removes defects in work products. The Fagan methodology mandates that requirement documents, design documents, source code, and test plans are all formally inspected.

There are several *roles* defined in the process including the *moderator* who chairs the inspection; the *reader* who reads or paraphrases the particular deliverable; the *author* who is the creator of the deliverable; and the *tester* who 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 several stages in the Fagan inspection process, including planning, overview, preparation, inspection, process improvement, rework, and follow-up.

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 identifying 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 and test cases prepared and executed. The results of testing are reported and any issues corrected and retested. The test cases will need to be appropriate to verify the correctness of the software. Software inspection and testing are described in more detail in [6].

17.6 Process Maturity Models

The Software Engineering Institute (SEI) developed the Capability Maturity Model (CMM) in the early 1990s as a framework to help software organizations to improve their software process maturity, and to implement best practice in software and 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 [14], Juran [15] and Crosby [16] to the software field. These quality gurus were influential in transforming manufacturing companies with quality problems to effective quality driven organizations with a reduced cost of poor quality.

Watt Humphries (Fig. 17.9) did early work on software process improvement at IBM [17]. He moved to the SEI in the late 1980s and the first version of the CMM was released in 1991. It is now called the Capability Maturity Model Integration (CMMI$^{®}$) [18].

The CMMI consists of five maturity levels with each maturity level (except level one) consisting of several process areas. Each process area consists of a set of goals that are implemented by practices related to that process area leading to an effective process.

The emphasis on level two of the CMMI is on maturing management practices such as project management, requirements management, configuration management, and so on. The emphasis on level three of the CMMI is to mature engineering and organization practices. This maturity level includes peer reviews and testing, requirements development, software design and implementation practices, and so on. Level four 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 five is concerned with continuous process improvement, which is quantitatively verified.

The CMMI allows organizations to benchmark themselves against other similar organizations. This is done by appraisals conducted by an authorized SCAMPI lead appraiser. The results of an SCAMPI appraisal are generally reported back to the SEI, and there is a strict qualification process to become an authorized lead

Fig. 17.9 Watts Humphrey. Courtesy of Watts Humphrey

appraiser. An appraisal is useful in verifying that an organization has improved, and it enables the organization to prioritize improvements for the next improvement cycle.

17.7 Review Questions

1. What is software engineering? Describe the difference between classical engineers and software engineers.
2. Describe the ‘software crisis’ of the late 1960s that led to the first software engineering conference in 1968.
3. Discuss the Standish Research Report and the level of success of IT projects today. In your view is there a crisis in software engineering today? Give reasons for your answer.
4. Discuss what the role of mathematics should be in current software engineering.
5. Describe the waterfall and spiral lifecycles. What are the similarities and differences between them?
6. Discuss the contributions of Floyd and Hoare.
7. Explain the difference between partial correctness and total correctness.
8. What are formal methods?
9. Discuss the process maturity models (including the CMMI). What are their advantages and disadvantages?
10. Discuss how software inspections and testing can assist in the delivery of high-quality software.

17.8 Summary

This chapter presented a short account of some important developments in software engineering. Its birth was at the Garmisch conference in 1968, and it was recognized that there was a crisis in the software field, and 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 classical engineering to assist with the 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 should be used is controversial with strong views in both camps.

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 organizations in maturing key management and engineering practices, and may help companies in their goals to deliver high-quality software systems that are consistently delivered on time and budget.

References

1. Estimating: Art or Science. Featuring Morotz Cost Expert. Standish Group Research Note. 1999.
2. The Mythical Man Month. Fred Brooks. Addison Wesley. 1975.
3. No Silver Bullet. Essence and Accidents of Software Engineering. Fred Brooks. *Information Processing*. Elsevier. Amsterdam, 1986.
4. Mathematical Approaches to Software Quality. Gerard O' Regan. Springer. 2006.
5. Introduction to Software Process Improvement. Gerard O' Regan. Springer. 2010.
6. Introduction to Software Quality. Gerard O' Regan. Springer Verlag. 2014.
7. The Language of Mathematics. Utilizing Math in Practice. Robert L. Baber. Wiley. 2011.
8. The Software Lifecycle Model (Waterfall Model). W. Royce. In Proc. WESTCON, August, 1970.
9. A Spiral Model for software development and enhancement. Barry Boehm. *Computer*. May 1988.
10. Extreme Programming Explained. Embrace Change. Kent Beck. Addison Wesley. 2000.
11. Assigning Meanings to Programs. Robert Floyd. Proceedings of Symposia in Applied Mathematics, (19):19–32. 1967.
12. An Axiomatic Basis for Computer Programming. C.A.R. Hoare. Communications of the ACM. 12(10):576–585. 1969.
13. Design and Code Inspections to Reduce Errors in Software Development. Michael Fagan. *IBM Systems Journal* 15(3). 1976.
14. Out of Crisis. W. Edwards Deming. M.I.T. Press. 1986.
15. Juran's Quality Handbook. 5th edition. Joseph Juran. McGraw Hill. 2000.
16. Quality is Free. The Art of Making Quality Certain. Philip Crosby. McGraw Hill. 1979.
17. Managing the Software Process. Watts Humphry. Addison Wesley. 1989.
18. CMMI. Guidelines for Process Integration and Product Improvement. Third Edition. Mary Beth Chrissis, Mike Conrad and Sandy Shrum. SEI Series in Software Engineering. Addison Wesley. 2011.