Fundamentals of Software Engineering

2

Key Topics

Standish report
Software lifecycles
Waterfall model
Spiral model
Rational unified process
Agile development
Software inspections
Software testing
Project management

2.1 Introduction

The approach to software development in the 1950s and 1960s has been described as the "*Mongolian Hordes Approach*" by Brooks (1975).[1] The "method" was applied to projects that were running late, and it involved adding a large number of programmers to the project, with the expectation that this would enable the project schedule to be recovered. However, this approach was deeply flawed as it led to inexperienced programmers with inadequate knowledge of the project attempting to

[1]Brooks was the project manager for the IBM System 360 project. The "Mongolian Hordes" management myth is the belief that adding more programmers to a software project that is running late will allow catchup. Brooks confirmed that adding people to a late software project actually makes it later.

G. O'Regan, *Concise Guide to Software Testing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-28494-7_2

solve problems, and they inevitably required significant time from the other project team members.

This resulted in the project being delivered even later, as well as subsequent problems with quality (i.e. the approach of throwing people at a problem does not work). The approach to software development back in the 1950/60s was characterized by the philosophy:

The completed code will always be full of defects.
The coding should be finished quickly to correct these defects.
Design as you code approach.

This philosophy accepted defeat in software development, and suggested that irrespective of a solid engineering approach, that the completed software would always contain lots of defects, and that it therefore made sense to code as quickly as possible, and to then identify the defects so as to correct them as quickly as possible.

In the late 1960s, it was clear that the existing approaches to software development were deeply flawed, and that there was an urgent need for change. The NATO Science Committee organized two famous conferences to discuss critical issues in software development (Naur and Randell 1975). The first conference was held at Garmisch, Germany, in 1968, and it was followed by the second conference in Rome in 1969. Over fifty people from eleven countries attended the Garmisch conference, including Peter Naur who produced a report on the conference (Naur and Randell 1969). The NATO conferences highlighted problems that existed in the software sector in the late 1960s, and the term "*software crisis*" was coined to refer to these. There were problems with budget and schedule overruns, as well as the quality and reliability of the delivered software.

The conference led to the birth of *software engineering* as a discipline in its own right, and the realization that programming is quite distinct from science to mathematics. Programmers are like engineers in that they build software products, and they therefore need education in traditional engineering as well as on the latest technologies. The education of a classical engineer includes product design and mathematics. However, often computer science education places an emphasis on the latest technologies, rather than on the important engineering foundations of designing and building high-quality products that are safe for the public to use.

Programmers, therefore, need to learn the key engineering skills to enable them to build products that are safe for the public to use. This includes a solid foundation on design and on the mathematics required for building safe software products. Mathematics plays a key role in classical engineering, and in some situations it may also assist software engineers in the delivery of high-quality software products. Several mathematical approaches to assist software engineers are described in O'Regan (2006, 2017b).

There are parallels between the software crisis in the late 1960s and serious problems with bridge construction in the nineteenth century. Several bridges collapsed, or were delivered late or overbudget, due to the fact that people involved in

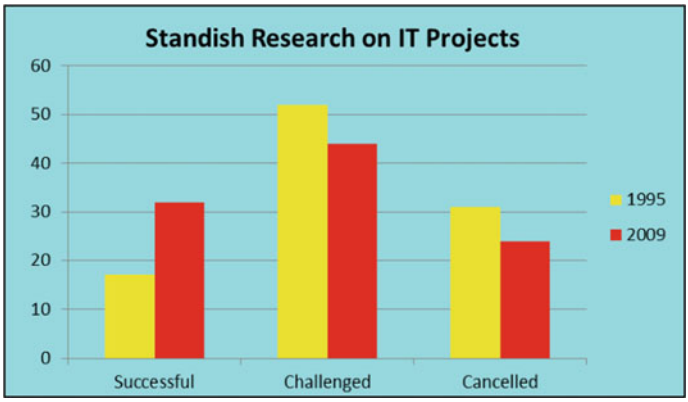

Fig. 2.1 Standish report—results of 1995 and 2009 Survey

their design and construction did not have the required engineering knowledge. This led to poorly designed and constructed bridges that later collapsed leading to loss of life, as well as endangering the lives of the public.

This led to legislation requiring engineers to be licenced by the Professional Engineering Association prior to practicing as engineers. This organization specified 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. This helps to ensure that only personnel competent to design and build products actually do so. Engineers have a professional responsibility to ensure that the products are properly built and are safe for the public to use.

The Standish group has conducted research (Fig. 2.1) on the extent of problems with IT projects since the mid-1990s. These studies were conducted in the USA, but there is no reason to believe that European or Asian companies perform any better. The results indicate serious problems with on-time delivery of projects and projects being cancelled prior to completion.[2] However, the comparison between 1995 and 2009 suggests that there have been improvements with a greater percentage of projects being delivered successfully and a reduction in the percentage of projects being cancelled.

Fred Brooks argues that software is inherently complex, and that there is no *silver bullet* that will resolve all of the problems associated with software development such as schedule or budget overruns (Brooks 1975, 1986). Poor software quality may adversely impact the customer and even cause loss of life. It is, therefore, essential that software development organizations place sufficient emphasis on quality throughout the software development lifecycle.

The Y2K problem was caused by a two-digit representation of dates, and it required major rework of legacy software for the new millennium. Clearly, well-designed programs would have hidden the representation of the date, which

[2]These are IT projects covering diverse sectors including banking, telecommunications, etc., rather than pure software companies. Software companies following maturity frameworks such as the CMMI generally achieve more consistent results.

would have required minimal changes for year 2000 compliance. Instead, companies spent vast sums of money to rectify the problem.

The quality of software produced by some companies is quite good.[3] These companies employ mature software processes and are committed to continuous improvement. There is a lot of industrial interest in software process maturity models for software organizations, and various approaches to assess and mature software companies are described in O'Regan (2010, 2014).[4] These models focus on improving the effectiveness of the management, engineering, and organization processes related to software engineering and in introducing best practice in software engineering. It is a key tenet of the software quality movement that the disciplined use of mature software processes by the software engineers enables high-quality software to be consistently produced.

2.2 What Is Software Engineering?

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

> 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 as follows:

1. Methodologies to design, develop, and test software to meet customers' needs.
2. Software is engineered. That is, the software products are properly designed, developed, and tested in accordance with sound engineering principles.
3. Quality and safety are properly addressed.
4. Mathematics may be employed to assist with the design and verification of software products. The level of mathematics employed will depend on the *safety-critical* nature of the product. Systematic peer reviews and rigorous testing will often be sufficient to build quality into the software, with heavy *mathematical techniques reserved for safety and security critical software*.
5. Sound project management and quality management practices are employed.
6. Support and maintenance of the software are properly addressed.

[3] I recall projects at Motorola that regularly achieved 5.6σ-quality in a L4 CMM environment (i.e. approx. 20 defects per million lines of code. This represents very high quality).

[4] Approaches such as the CMM or SPICE (ISO 15504) focus mainly on maturing management and organizational practices used in software development. The emphasis is on defining software processes that are fit for purpose and consistently following them. The process maturity models focus on what needs to be done rather how it should be done. This gives the organization the freedom to choose the appropriate implementation to meet its needs. The models provide useful information on practices to consider in the implementation.

Software engineering is not just programming. It 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. The project needs to be planned and delivered on time and budget. The requirements must provide a precise description of the problem to be solved, i.e. *it should be evident from the requirements what is and what is not required*.

The requirements need to be rigorously reviewed to ensure that they are stated clearly and unambiguously and reflect the customer's needs. The next step is then to create the design that will solve the problem, and it is essential to validate the correctness of the design. Next, the software code to implement the design is written, and peer reviews and software testing are employed to verify and validate the correctness of the software.

The verification and validation of the design are rigorously performed for safety-critical systems, and it is sometimes appropriate to employ mathematical techniques for these systems. However, it will often be sufficient to employ peer reviews (software inspections) and testing as these methodologies provide a high degree of rigour. These may include methodologies such as Fagan inspections (Fagan 1976), Gilb inspections (Gilb and Graham 1994), or Prince 2's approach to quality reviews (Office of Government Commerce 2004).

The term "*engineer*" is a title that is awarded on merit in classical engineering. It is generally applied only to people who have attained the necessary education and competence to be called engineers and who base their practice on sound engineering principles. The title places responsibilities on its holder to behave professionally and ethically. Often in computer science, the term "*software engineer*" is employed loosely to refer to anyone who builds things, rather than to an individual with a core set of knowledge, experience, and competence.

Several computer scientists (e.g. Parnas[5]) have argued that computer scientists should be educated as engineers to enable them to apply appropriate scientific principles to their work. They argue that computer scientists should receive a solid foundation in mathematics and design, to enable them to have the professional competence to perform as engineers in building high-quality products that are safe for the public to use. The use of mathematics is an integral part of the engineer's work in other engineering disciplines, and so the *software engineer* should be able to use mathematics to assist in the modelling or understanding of the behaviour or properties of the proposed software system.

Software engineers need education[6] on specification, design, and turning designs into programs, software inspections, and testing. The education should enable the software engineer to produce well-structured programs that are fit for purpose. Parnas has argued that software engineers have responsibilities as professional

[5]Parnas advocates a solid engineering approach with the extensive use of classical mathematical techniques in software development. He also introduced information hiding in the 1970s, which is now a part of object-oriented design.

[6]Software companies that are following approaches such as the CMM or ISO 9001 considering the education and qualification of staff prior to assigning staff to performing specific tasks.

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 decisions and actions[8] and have a responsibility to object to decisions that violate professional standards.

Engineers are required to behave professionally and ethically with their clients. The membership of the professional engineering body requires the member to adhere to the code of ethics[9] of the profession. Engineers in other professions are licenced, and therefore Parnas argues that a similar licencing approach be adopted for professional software engineers[10] to provide confidence that they are competent for the particular assignment. Professional software engineers are required to follow best practice in software engineering and the defined software processes.[11]

Many companies invest heavily in training, as educated and knowledgeable employees are essential in the delivery of high-quality products and services. Employees need to receive appropriate training related to the roles that they perform, such as project management, software design and development, software testing, and service management. The fact that the employees are professionally qualified increases the confidence in the ability of the company to deliver high-quality products and services. A company that pays little attention to the competence and continuous development of its staff will achieve poor results and suffer a loss of reputation and market share.

[7]The ancient Babylonians were familiar with the concept of accountability, and they employed a code of laws (known as the Hammurabi Code) c. 1750 B.C. It included a law that stated that if a house collapsed and killed the owner, then the builder of the house would be executed.

[8]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 accompanies most software products. Software engineering is a team-based activity involving many engineers in various parts of the project, and it would 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 a company is a wealthier entity than one of its employees. The legal aspects of licencing software may protect software companies from litigation. However, greater legal protection for the customer can be built into the contract between the supplier and the customer for bespoke software development.

[9]Today, many software companies have a defined code of ethics that employees are expected to adhere. Larger companies will wish to project a good corporate image to be respected worldwide.

[10]The British Computer Society (BCS) has introduced a qualification system for computer science professionals, which is used to show that professionals are properly qualified. The most important of these is the BCS Information Systems Examination Board (ISEB) which allows IT professionals to be qualified in service management, project management, software testing, and so on.

[11]Software companies that are following the CMMI or ISO 9001 standards will employ audits to verify that the processes and procedures have been followed. Auditors report their findings to management, and the findings are addressed appropriately by the project team and affected individuals.

2.3 Challenges in Software Engineering

The challenge in software engineering is to deliver high-quality software on time and on budget to customers. We discussed the research done by the Standish Group earlier in this chapter, and their 1998 research (Fig. 1.1) on project cost overruns in the USA indicated 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, effort, and schedule is a challenge in project management. Therefore, project managers need to determine how good their estimation process actually is and to make appropriate improvements. Software metrics provide an objective way to see improvements in estimation from a reduced variance between estimated and actual effort. The project manager will report the actual versus estimated effort and schedule for the project.

Risk management is an important part of project management, and the objective is to identify potential risks early and throughout the project and to manage them appropriately. The probability of each risk occurring and its impact is determined, and the risks are managed during project execution.

Software quality needs to be properly planned to enable the project to deliver a quality product. Flaws with poor quality software lead to a negative perception of the company and may potentially lead damage to the customer relationship with a subsequent loss of market share.

There is a strong economic case to building quality into the software, as less time is spent in reworking defective software. The cost of poor quality (COPQ) should be measured and targets set for its reductions. It is important that lessons are learned during the project and acted upon appropriately. This helps to promote a culture of continuous improvement.

Several high-profile software failures were discussed in Chap. 1. These include the millennium bug (Y2K) problem, the floating-point maths bug in the Intel Pentium microprocessor in the early 1990s, the European Space Agency Ariane-5 disaster, and so on. These failures led to the embarrassment for the organizations involved, as well as the associated cost of replacement and correction.

The millennium bug was due to the use of two digits to represent dates rather than four digits. The solution involved finding and analysing all code that had a Y2K impact, planning and making all necessary changes, and verifying the correctness of the changes made. The worldwide cost of correcting the millennium bug is estimated to have been in billions of dollars.

The Intel Corporation was slow to acknowledge the floating-point problem in its Pentium microprocessor and in providing adequate information on its impact to its customers. It incurred a large financial cost in replacing microprocessors for its customers as well as damage to its reputation. The Ariane 5 failure caused major embarrassment and damage to the credibility of the European Space Agency (ESA). Its maiden flight ended in failure on June 4, 1996, after a flight time of just 40 s.

These failures indicate that the quality needs to be carefully considered when designing and developing software. The effect of software failure may be large costs to correct the software, loss of credibility of the company, or even loss of life.

2.4 Software Processes and Lifecycles

Organizations vary by size and complexity, and the processes they employ will reflect the nature of their business. The development of software involves many processes such as those for defining requirements, processes for project estimation and planning, processes for design, implementation, testing, and so on.

It is important that the processes are fit for purpose, and a key premise in the software quality field is that the quality of the resulting software is closely influenced by the quality and maturity of the underlying processes and compliance to them. Therefore, it is necessary to focus on the quality of the processes as well as the quality of the resulting software.

There is, of course, little point in having high-quality processes unless the employees in the organization use them consistently. This requires that the employees are trained on the processes, and that process discipline is instilled into an appropriate audit strategy. The software process assets in an organization generally consist of as follows:

– A software development policy for the organization
– Process maps that describe the flow of activities
– Procedures and guidelines that describe the processes in more detail
– Checklists to assist with the performance of the process
– Templates for the performance of specific activities (e.g. design, testing)
– Training materials.

The processes used to develop high-quality software generally include as follows:

– Project management process
– Requirements process
– Design process
– Coding process
– Peer review process
– Testing process
– Supplier selection and management processes
– Configuration management process
– Audit process
– Measurement process
– Improvement process
– Customer support and maintenance processes

There are several well-known lifecycles employed such as the waterfall model (Royce 1970), the spiral model (Boehm 1988), the Rational Unified Process (Rumbaugh 1999), and the Agile methodology (Beck 2000). The choice of a particular software development lifecycle for is determined from the needs of the specific project. The various lifecycles are described in more detail in the following sections.

2.4.1 Waterfall Lifecycle

The waterfall model (Fig. 2.2) starts with requirements gathering and definition. It is followed by the system specification (of the functional and non-functional requirements), the design and implementation of the software, and comprehensive testing. The testing generally includes unit, system, and user acceptance testing.

The waterfall model is employed for projects where the requirements can be identified early in the project lifecycle or are known in advance. We are treating the waterfall model as the "V" lifecycle 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 that must be satisfied before the next phase commences. There are several variations to the waterfall model.

Many companies employ a set of templates to enable the activities in various phases to be consistently performed. Templates may be employed for project planning and reporting, requirements definition, design, testing, and so on. These templates may be based on the IEEE standards or industrial best practice.

2.4.2 Spiral Lifecycles

The spiral model (Fig. 2.3) was developed by Barry Boehm in the 1980s (Boehm 1988), and it is useful for projects where the requirements are not fully known at

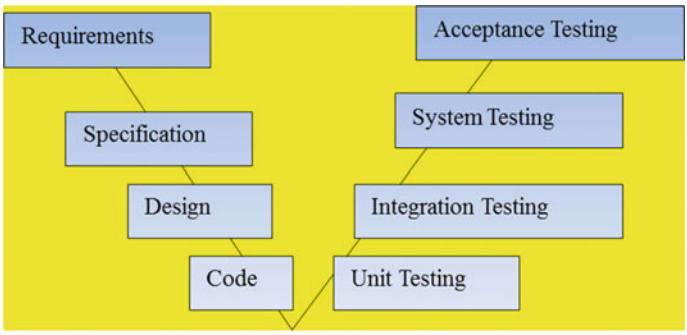

Fig. 2.2 Waterfall V lifecycle model

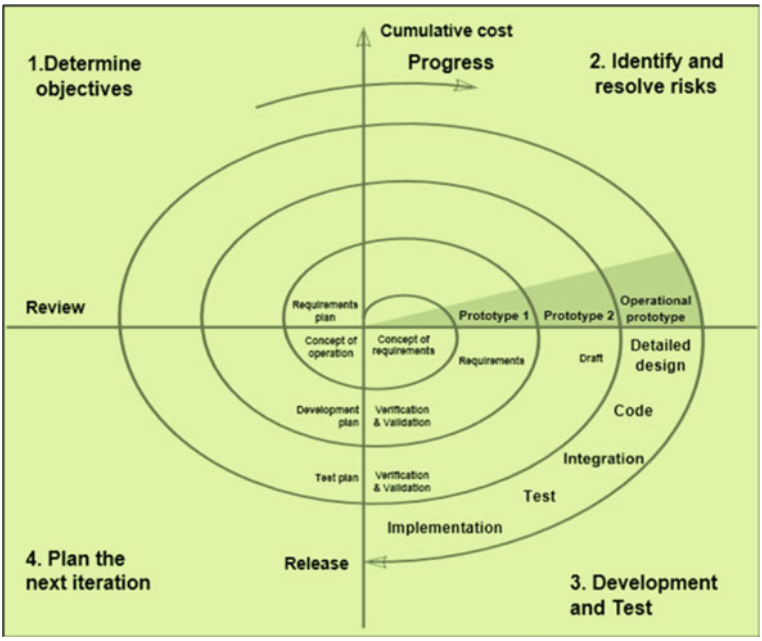

Fig. 2.3 SPIRAL lifecycle model. Public domain

project initiation, or where the requirements evolve as a part of the development lifecycle. The development proceeds in a number of spirals, where each spiral implements new functionality and typically involves determining objectives and analysing the risks, 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 with the business analysts and the customer reviewing the current iteration, and providing feedback to the development team. The feedback is analysed and used to plan the next iteration. This approach is often used in joint application development, where the usability and look and feel of the application are a key concern. This is important in web-based development and in the development of a graphical user interface (GUI). The implementation of part of the system helps in gaining a better understanding of the requirements of the system, and this feeds into subsequent development cycles. The process repeats until the requirements and the software product are fully complete.

There are several variations of the spiral model including rapid application development (RAD), joint application development (JAD) models, and the dynamic systems development method (DSDM) model. The Agile methodology is popular, and it employs sprints (or iterations) of 2–4 weeks duration to implement a number of user stories.

There are other lifecycle models such as the iterative development process that combines the waterfall and spiral lifecycle model. The Cleanroom methodology

was developed by Harlan Mills at IBM and includes a phase for formal specification. Its approach to software testing is based on the predicted usage of the software product, which allows a software reliability measure to be calculated. The Rational Unified Process (RUP) was developed by IBM Rational.

2.4.3 Rational Unified Process

The *Rational Unified Process* (Rumbaugh 1999) was developed at the Rational Corporation (now part of IBM) in the late 1990s. It uses the Unified Modelling Language (UML) as a tool for specification and design. UML is a visual modelling language for software systems that provides a means of specifying, constructing, and documenting the object-oriented system. James Rumbaugh, Grady Booch, and Ivar Jacobson developed it to facilitate the understanding of the architecture and complexity of a system.

RUP is *use case driven*, *architecture centric*, *iterative* and *incremental*, and it includes cycles, phases, workflows, risk mitigation, quality control, project management, and configuration control (Fig. 2.4). Software projects may be very complex, and there are risks that requirements may be incomplete, or that the interpretation of a requirement may differ between the customer and the project team. RUP is a way to reduce risk in software engineering.

Requirements are gathered as use cases, where the *use cases describe the functional requirements from the point of view of the user of the system*. They describe what the system will do at a high level and ensure that there is an appropriate focus on the user when defining the scope of the project. *Use cases also drive the development process*, as the developers create a series of design and implementation models that realize the use cases. The developers review each successive model for conformance to the use-case model, and the test team verifies that the implementation correctly and implements the use cases.

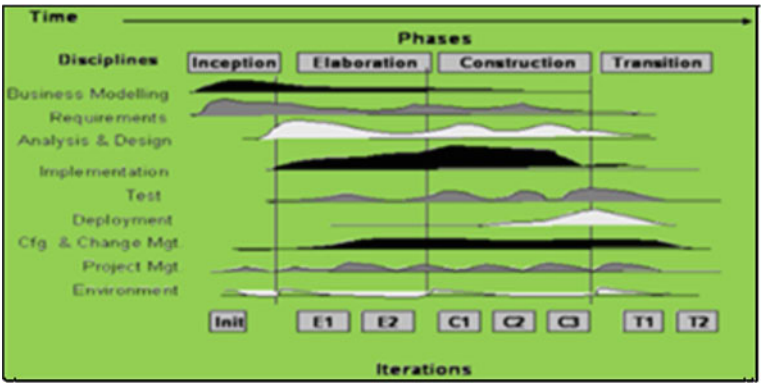

Fig. 2.4 Rational unified process

The software architecture concept embodies the most significant static and dynamic aspects of the system. The architecture grows out of the use cases and factors such as the platform that the software is to run on, deployment considerations, legacy systems, and the non-functional requirements.

RUP decomposes the work of a large project into smaller slices or mini-projects, and *each mini-project is an iteration that results in an increment to the product*. The iteration consists of one or more steps in the workflow and generally leads to the growth of the product. If there is a need to repeat an iteration, then all that is lost is the misdirected effort of one iteration, rather that the entire product. Another words—RUP is a way to mitigate risk in software engineering.

2.4.4 Agile Development

There has been a massive growth of popularity among software developers in lightweight methodologies such as *Agile*. This is a software development methodology that is 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% of the requirements are typically 50% done. *However, for Agile development, 50% of requirements are typically 100% done halfway through the project*. Agile has a strong collaborative style of working including:

– Aims to achieve a narrow fast flowing value stream
– Feedback and adaptation employed in decision making
– User stories and sprints are employed
– Stories are either done or are not done (no such thing as 50% done)
– Iterative and incremental development is employed
– A project is divided into iterations
– An iteration has a fixed length (i.e. Time boxing is employed)
– Entire software development lifecycle is employed for the implementation of each story
– Change is accepted as a normal part of life in the Agile world
– Delivery is made as early as possible.
– Maintenance is seen as part of the development process
– Refactoring and evolutionary design employed
– Continuous integration is employed
– Short cycle times
– Emphasis on quality
– Stand-up meetings
– Plan regularly
– Direct interaction preferred over documentation
– Rapid conversion of requirements into working functionality
– Demonstrate value early
– Early decision making.

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, and good communication and early regular feedback are 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. Stories are prioritized based on a number of factors including:

– Business value of story
– Mitigation of risk
– Dependencies on other stories.

The Scrum approach is an Agile method for managing iterative development, and it consists of an outline planning phase for the project followed by a set of sprint cycles (where each cycle develops an increment). *Sprint planning* is performed before the start of the iteration, and stories are assigned to the iteration to fill the available time. Each scrum sprint is of a fixed length (usually 2–4 weeks), and it develops an increment of the system. The estimates for each story and their priority are determined, and the prioritized stories are assigned to the iteration. *A short morning stand-up meeting is held daily* during the iteration, and attended by the scrum master, the project manager[12] and the project team. It discusses the progress made the previous day, problem reporting and tracking, and the work planned for the day ahead. A separate meeting is held for issues that require more detailed discussion.

Once the iteration is complete, the latest product increment is demonstrated to an audience including the product owner. This is to receive feedback and to identify new requirements. The team also conducts a retrospective meeting to identify what went well and what went poorly during the iteration. This is for continuous improvement of the future iterations. Planning for the next sprint then commences. The scrum master is a facilitator who arranges the daily meetings and ensures that the scrum process is followed. The role involves removing roadblocks so that the team can achieve their goals and communicating with other stakeholders.

Agile employs pair programming and a collaborative style of working with the philosophy that two heads are better than one. This allows multiple perspectives in decision making and a broader understanding of the issues.

[12]Agile teams are self-organizing and the project manager role is generally not employed for small projects (<20 staff).

Software testing is very important, and Agile generally employs automated testing for unit, acceptance, performance, and integration testing. Tests are run frequently with the goal of catching programming errors early. They are generally run on a separate build server to ensure that all dependencies are checked. Tests are rerun before making a release. *Agile employs test-driven development with tests written before the code*. The developers write code to make a test pass with ideally developers only coding against failing tests. This approach forces the developer to write testable code.

Refactoring is employed in Agile as a design and coding practice. The objective is to change how the software is written without changing what it does. Refactoring is a tool for evolutionary design where the design is regularly evaluated, and improvements are implemented as they are identified. It helps in improving the maintainability and readability of the code and in reducing complexity. The automated test suite is essential in showing that the integrity of the software is maintained following refactoring.

Continuous integration allows the system to be built with every change. Early and regular integration allows early feedback to be provided. It also allows all of the automated tests to be run thereby identifying problems earlier. Agile is discussed in more detail in Chap. 18 of O'Regan (2017a).

2.5 Activities in Waterfall Lifecycle

The waterfall software development lifecycle consists of various activities including:

- User (business) requirements definition
- Specification of system requirements
- Design
- Implementation
- Unit testing
- System testing
- UAT testing
- Support and maintenance

These activities are discussed in the following sections, and the description is specific to the non-Agile world.

2.5.1 User Requirements Definition

The user (business) requirements specify what the customer wants and define what the software system is required to do (*as distinct from how this is to be done*). The requirements are the foundation for the system, and if they are incorrect, then the

implemented system will be incorrect. *Prototyping may be employed* to assist in the definition and validation of the requirements. The process of determining the requirements, analysing and validating them, and managing them throughout the project lifecycle is termed *requirements engineering*.

The *user requirements* are determined from discussions with the customer to determine their actual needs, and they are then refined into the *system requirements*, which state the *functional* and *non-functional* requirements of the system. The specification of the user requirements needs to be unambiguous to ensure that all parties involved in the development of the system share a common understanding of what is to be developed and tested.

Requirements gathering involves meetings with the stakeholders to gather all relevant information for the proposed product. The stakeholders are interviewed, and requirements workshops are conducted to elicit the requirements from them. An early working system (prototype) is often used to identify gaps and misunderstandings between developers and users. The prototype may serve as a basis for writing the specification.

The requirements workshops are used to discuss and prioritize the requirements, as well as identifying and resolving any conflicts between them. The collected information is consolidated into a coherent set of requirements. Changes to the requirements may occur during the project, and these need to be controlled. It is essential to understand the impacts (e.g. schedule, budget, and technical) of a proposed change to the requirements prior to its approval.

Requirements verification is concerned with ensuring that the requirements are properly implemented (i.e. building it right) in the design and implementation. *Requirements validation* is concerned with ensuring that the right requirements are defined (building the right system), and that they are precise, complete, and reflect the actual needs of the customer.

The requirements are validated by the stakeholders to ensure that they are actually those desired, and to establish their feasibility. This may involve several reviews of the requirements until all stakeholders are ready to approve the requirements document. Other validation activities include reviews of the prototype and the design, and user acceptance testing.

The requirements for a system are generally documented in a natural language such as “English”. Other notations that are employed include the visual modelling language UML (Jacobson et al. 1999) and formal specification languages such as VDM or Z for the safety-critical field.

The Agile software development methodology argues that as requirements change so quickly that a requirements document is unnecessary, such a document would be out of date as soon as it was written.

2.5.2 Specification of System Requirements

The specification of the system requirements of the product is essentially a statement of what the software development organization will provide to meet the

business (user) requirements. That is, the detailed business requirements are a statement of what the customer wants; whereas, the specification of the system requirements is a statement of what will be delivered by the software development organization.

It is essential that the system requirements are valid with respect to the user requirements, and the stakeholders review them to ensure their validity. Traceability is employed to show that the business requirements are addressed by the system requirements.

There are two categories of system requirements, namely functional and non-functional requirements. The *functional requirements* define the functionality that is required of the system, and it may include screen shots, report layouts, or desired functionality specified as use cases. The *non-functional requirements* will generally include security, reliability, availability, performance and portability requirements, as well as usability and maintainability requirements.

2.5.3 Design

The design of the system consists of engineering activities to describe the architecture or structure of the system, as well as activities to describe the algorithms and functions required to implement the system requirements. It is a creative process concerned with how the system will be implemented, and its activities include architecture design, interface design, and data structure design. There are often several possible design solutions for a particular system, and the designer will need to decide on the most appropriate approach.

The design may be specified in various ways such as graphical notations that display the relationships between the components making up the design. The notation may include flow charts or various UML diagrams such as sequence diagrams, state charts, and so on. Program description languages or pseudo code may be employed to define the algorithms and data structures that are the basis for implementation.

Function-oriented design is mainly historical, and it involves starting with a high-level view of the system and refining it into a more detailed design. The system state is centralized and shared between the functions operating on that state.

Object-oriented design (OOD) is based on the concept of *information hiding* as developed by Parnas (Parnas 1972). The system is viewed as a collection of objects rather than functions, with each object managing its own state information. The system state is decentralized, and an object is a member of a class. The definition of a class includes attributes and operations on class members, and these may be inherited from super classes. Objects communicate by exchanging messages, and object-oriented design has largely replaced function-oriented design,

It is essential to verify and validate the design with respect to the system requirements, and this may be done by traceability of the design to the system requirements and design reviews to ensure that the design is fit for purpose.

2.5.4 Implementation

This phase is concerned with implementing the design in the target language and environment (e.g. C++ or Java) and involves writing or generating the actual code. The development team divides up the work to be done, with each programmer responsible for one or more modules. The coding activities generally include code reviews or walkthroughs to ensure that quality code is produced and to verify its correctness. The code reviews will verify that the source code conforms to the coding standards and that maintainability issues are addressed. They will also verify that the code produced is a valid implementation of the software design, and that it is fit for purpose.

Software reuse provides a way to speed up the development process. Components or objects that may be reused need to be identified and handled accordingly. The implemented code may use software components that have either being developed internally or purchased off the shelf. Open-source software has become popular in recent years, and it allows software developed by others to be used (*under an open-source licence*) in the development of applications.

The benefits of software reuse include increased productivity and a faster time to market. There are inherent risks with customized-off-the shelf (COTS) software, as the supplier may decide to no longer support the software, or there is no guarantee that the software that has worked successfully in one domain will work correctly in a different domain. It is therefore important to consider the risks as well as the benefits of software reuse and open-source software.

2.5.5 Software Testing

Software testing is employed to verify that the requirements have been correctly implemented, and that the software is fit for purpose, as well as identifying defects present in the software. There are various types of testing that may be conducted including *unit testing, integration testing, system testing, performance testing, and user acceptance testing*. These are described below:

Unit Testing
Unit testing is performed by the programmer on the completed unit (or module) prior to its integration with other modules. The programmer writes these tests, and the objective is to show that the code satisfies the design. The unit test case is generally documented, and it should include the test objectives and the expected results.

Code coverage and branch coverage metrics are often generated to give an indication of how comprehensive the unit testing has been. These provide visibility into the number of lines of code executed, as well as the branches covered during unit testing. Test tools are often employed to provide code coverage and branch coverage metrics.

The developer executes the unit tests; records the results; corrects any identified defects; and retests the software. *Test-driven development* (TDD) has become popular in the Agile world, this involves writing the unit test cases (and possibly other test cases) before the code, and the code is then written to pass the defined test cases.

Integration Test

The integrated system is ready to be tested once the unit testing is complete and when all of the individual units work correctly in isolation. The development team generally performs this type of testing, and the objective is to verify that all of the modules and their interfaces work correctly together. Modules that work correctly in isolation may fail when they are integrated with other modules, and the developers will analyse the cause of failure; make appropriate corrections; and retest until the integrated system works correctly.

System Test

The purpose of system testing is to verify that the implementation is valid with respect to the system requirements. It involves the specification and execution of system test cases to verify that the system requirements have been correctly implemented. An independent test group generally conducts this type of testing, and the system test cases are traceable to the system requirements.

Any system requirements that have been incorrectly implemented will be identified, and defects are logged and reported to the developers. The developers make the appropriate corrections, the test group verifies that the new version of the software is correct, and regression testing is conducted to verify system integrity. System testing may include security testing, usability testing, and performance testing.

The preparation of the system test environment requires detailed planning, and it may involve ordering special hardware and tools. It is important that the test environment is set up early to ensure that it is ready on time for the execution of the test cases.

Performance Test

The purpose of performance testing is to ensure that the performance of the system is within the bounds specified by the non-functional requirements. It may include *load performance testing*, where the system is subjected to heavy loads over a long period of time, and *stress testing*, where the system is subjected to heavy loads during a short time interval.

Performance testing often involves the simulation of many users using the system and involves measuring the response times for various activities. Test tools are employed to simulate a large number of users and heavy loads. This type of testing is also employed to determine if the system is scalable to support future growth.

User Acceptance Test
The objective of UAT testing is to demonstrate that the product satisfies the business requirements and meets the customer expectations. Upon its successful completion, the customer will be happy to accept the product. It is usually performed under controlled conditions at the customer site, and its operation will closely resemble the real-life behaviour of the system. The customer will see the product in operation and will make an informed judgment as to whether the system is fit for purpose.

2.5.6 Support and Maintenance

This phase continues after the release of the software product to the customer. Software systems often have a long lifetime, and the software needs to be continuously enhanced over its lifetime to meet the evolving needs of the customers. This may involve regular releases of new functionality and corrections to known defects.

Any problems that the customer identifies with the software are reported as per the customer support and maintenance agreement. The support issues will require investigation, and the issue may be *a defect in the software*, *an enhancement to the software*, or *due to a misunderstanding*. The support and maintenance team will identify the causes of any identified defects and will implement an appropriate solution to resolve. Testing is conducted to verify that the solution is correct, and that the changes made have not adversely affected other parts of the system. Mature organizations will conduct post-mortems to learn lessons from the defect[13] and will take corrective action to prevent a reoccurrence.

The presence of a maintenance phase suggests an acceptance of the reality that problems with the software will be identified post-release. The goal of building a correct and reliable software product the first time is very difficult to achieve, and the customer is always likely to find some issues with the released software product. It is accepted today that the quality needs to be built into each step in the development process, with the role of software inspections and testing to identify as many defects as possible prior to release, and to minimize the risk that serious defects will be found post-release.

The effective in-phase inspections of the deliverables will influence the quality of the resulting software and lead to a corresponding reduction in the number of defects. The testing group plays a key role in verifying that the system is correct, and in providing confidence that the software is fit for purpose and ready to be released. The approach to software correctness involves testing and retesting, until

[13]This is essential for serious defects that have caused significant inconvenience to customers (e.g. a major telecom outage). The software development organization will wish to learn lessons to determine what went wrong in its processes that prevented the defect from been identified during peer reviews and testing. Actions to prevent a reoccurrence will be identified and implemented.

the testing group believes that all defects have been eliminated. Dijkstra (1972) comments on testing are well-known:

> Testing a program demonstrates that it contains errors, never that it is correct.

That is, irrespective of the amount of time spent testing, it can never be said with absolute confidence that all defects have been found in the software. Testing provides increased confidence that the program is correct, and statistical techniques may be employed to give a measure of the software reliability.

Many software companies may consider one defect per thousand lines of code (KLOC) to be reasonable quality. However, if the system contains one million lines of code, this is equivalent to a thousand post-release defects, which is unacceptable.

Some mature organizations have a quality objective of three defects per million lines of code, which was introduced by Motorola as part of its six-sigma (6σ) program. It was originally applied it to its manufacturing businesses and subsequently applied to its software organizations. The goal is to reduce variability in manufacturing processes and to ensure that the processes performed within strict process control limits.

2.6 Software Inspections

Software inspections play an important role in building quality into the software and in reducing the cost of poor quality. There are several well-known inspection methodologies such as the Fagan Methodology (Fagan 1976) and Gilb's approach (Gilb and Graham 1994), and we briefly discussed the Fagan methodology in Chap. 1.

The seven-step Fagan process identifies and removes errors in work products. It mandates that requirement documents, design documents, source code, and test plans are all formally inspected by experts independent of the author of the deliverable. 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, and *the author* is the creator of the deliverable and has a special interest in ensuring that it is correct. The *tester* role is concerned with the testing viewpoint.

The inspection meeting will consider whether the design is correct with respect to the requirements, and whether the source code is correct with respect to the design.

2.7 Software Project Management

The timely delivery of quality software requires good management and engineering processes. The project management activities include the following:

– Estimation of cost, effort, and schedule for the project
– Identifying and managing risks
– Preparing the project plan
– Preparing the initial project schedule and key milestones
– Obtaining approval for the project plan and schedule
– Staffing the project
– Monitoring progress, budget, schedule, effort, risks, issues, change requests, and quality
– Taking corrective action
– Replanning and rescheduling
– Communicating progress to affected stakeholders
– Preparing status reports and presentations.

The project plan will contain or reference several other plans such as the project quality plan; the communication plan; the configuration management plan; and the test plan.

Project estimation and scheduling are difficult as often software projects are breaking new ground, and previous estimates may not be a good basis for estimation for the current project. Often, unanticipated problems can arise for technically advanced projects, and the estimates may be optimistic. Gantt charts are employed for project scheduling, and these show the work breakdown for the project, as well as task dependencies and allocation of staff to the various tasks.

The effective management of risk during a project is essential to project success. Risks arise due to uncertainty, and the risk management cycle involves[14] risk identification; risk analysis and evaluation; identifying responses to risks; selecting and planning a response to the risk; and risk monitoring. The risks are logged, and the likelihood of each risk arising and its impact is then determined. The risk is assigned an owner and an appropriate response to the risk determined.

2.8 CMMI Maturity Model

The CMMI is a framework to assist an organization in the implementation of best practice in software and systems engineering. It is an internationally recognized model for software process improvement and assessment and is used worldwide by thousands of organizations. It provides a solid engineering approach to the development of software, and it supports the definition of high-quality processes for the various software engineering and management activities.

It was developed by the Software Engineering Institute (SEI) who adapted the process improvement principles used in the manufacturing field to the software field. They developed the original CMM in the early 1990s and its successor the

[14]These are the risk management activities in the Prince2 methodology.

CMMI in 2001. The CMMI states *what the organization needs to do* to mature its processes rather than *how this should be done*.

The CMMI consists of five maturity levels with each maturity level consisting of several process areas. Each process area consists of a set of goals, which are implemented by practices related to that process area. Level two is focused on management practices; level three is focused on engineering and organization practices; level four is concerned with ensuring that key processes are performing within strict quantitative limits; and level five is concerned with continuous process improvement. Maturity levels may not be skipped in the staged representation of the CMMI, as each maturity level is the foundation for the next level. The CMMI and Agile are compatible, and CMMI v1.3 (released in 2010) supports Agile software development.

The CMMI allows organizations to benchmark themselves against other organizations. This is done by a formal SCAMPI appraisal conducted by an authorized lead appraiser. The results of the appraisal are generally reported back to the SEI, and there is a strict qualification process to become an *authorized lead 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. The CMMI is discussed in more detail in Chap. 16 of O'Regan (2019a).

2.9 Formal Methods

Dijkstra and Hoare have argued that the way to develop correct software is to derive the program from its specifications using mathematics, and to employ *mathematical proof* to demonstrate its correctness with respect to the specification. This offers a rigorous framework to develop programs adhering to the highest quality constraints. However, in practice, mathematical techniques have proved to be cumbersome to use, and they are used in specialized areas in industry.

The *safety-critical area* is one area in which mathematical techniques have been successfully applied. There is a need for extra rigour in this domain, and mathematical techniques can demonstrate the presence or absence of certain desirable or undesirable properties (e.g. "*when a train is in a level crossing, then the gate is closed*").

Spivey (1992) defines a "*formal specification*" as 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. It describes *what* the system must do, as distinct from *how* it is to be done. This abstraction away from implementation enables questions about what the system does to be answered, independently of the detailed code. Further, the unambiguous nature of mathematical notation avoids the problem of ambiguity in an imprecisely worded natural language description of a system.

The formal specification thus becomes the key reference point for the different parties concerned with the construction of the system and is a useful way 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 precisely in a mathematical language. The derivation of an implementation from the specification may be achieved via *step-wise refinement*. Each refinement step makes the specification more concrete and closer to the actual implementation. There is an associated *proof obligation* that the refinement be valid, and that the concrete state preserves the properties of the more abstract state. Thus, assuming 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.

Formal methods have been applied to a diverse range of applications, including circuit design, artificial intelligence, specification of standards, specification, and verification of programs, etc. They are discussed in more detail in Chap. 13.

2.10 Review Questions

1. Discuss the research results of the Standish Group the current state of IT project delivery?
2. What are the main challenges in software engineering?
3. Describe various software lifecycles such as the waterfall model and the spiral model.
4. Discuss the benefits of Agile over conventional approaches. List any risks and disadvantages?
5. Describe the purpose of the CMMI? What are the benefits?
6. Describe the main activities in software inspections.
7. Describe the main activities in software testing.
8. Describe the main activities in project management?
9. What are the advantages and disadvantages of formal methods?

2.11 Summary

The birth of software engineering at the NATO conference in 1968 in Germany highlighted the problems that existed in the software sector in the late 1960s, and the term "*software crisis*" was coined to refer to these. The conference led to the realization that programming is quite distinct from science to mathematics, and that

software engineers need to be properly trained to enable them to build high-quality products that are safe for the public.

Programmers are like engineers in the sense that they build products. Therefore, programmers need to receive an appropriate education in engineering as part of their training. The education of traditional engineers includes training on product design, and an appropriate level of mathematics.

Software engineering is a systematic approach to the development and maintenance of the software, and it requires a precise statement of the requirements of the software product, and then the design and development of a solution to meet these requirements. It includes methodologies to design, develop, implement, and test software as well as sound project management, quality management, and configuration management practices. Support and maintenance of the software need to be properly addressed.

Software process maturity models such as the CMMI have become popular in recent years. They place an emphasis on understanding and improving the software process to enable software engineers to be more effective in their work.

References

Beck K (2000) Extreme programming explained. Embrace change. Addison Wesley, Boston
Boehm B (1988) A spiral model for software development and enhancement. Computer
Brooks F (1975) The mythical man month. Addison Wesley, Boston
Brooks F (1986) No silver bullet. Essence and accidents of software engineering. Information processing. Elsevier, Amsterdam
Dijkstra EW (1972) Structured programming. Academic Press, Cambridge
Fagan M (1976) Design and code inspections to reduce errors in software development. IBM Syst J 15(3)
Gilb T, Graham D (1994) Software inspections. Addison Wesley, Boston
Jacobson I, Booch G, Rumbaugh J (1999) The unified software modelling language user guide. Addison-Wesley, Boston
Naur P, Randell B (1969) Software engineering: report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Scientific Affairs Division, NATO, Brussels
Naur P, Randell B (1975) Software engineering. Petrocelli, IN, Buxton. Report on two NATO conferences held in Garmisch, Germany (October1968) and Rome, Italy (October 1969)
Office of Government Commerce (2004) Managing successful projects with PRINCE2. Office of Government Commerce, UK
O'Regan G (2006) Mathematical approaches to software quality. Springer, London
O'Regan G (2010) Introduction to software process improvement. Springer, London
O'Regan G (2014) Introduction to software quality. Springer, London
O'Regan G (2017a) Concise guide to software engineering. Springer, Berlin
O'Regan G (2017b) Concise guide to formal methods. Springer, Berlin
Parnas D (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12)

Royce W (1970) The software lifecycle model (waterfall model). In: Proceedings of WESTCON, August, 1970
Rumbaugh J et al (1999) The unified software development process. Addison Wesley, Boston
Spivey JM (1992) The Z notation. A reference manual. Prentice Hall International Series in Computer Science