Chapter 9
Overview of Software Engineering

Key Topics

Standish Chaos Report
Software Lifecycles
Waterfall Model
Spiral Model
Rational Unified Process
Agile Development
Software Inspections
Software Testing
Project Management
CMMI

9.1 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' or lack of method was applied to projects that were running late, and it involved adding many inexperienced programmers to the project, with the expectation that this would allow 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 solve problems, and they inevitably required significant time from the other project team members resulting in further delays.

This led to 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 philosophy of software development back in the 1950/60s was characterised by:

[1]The 'Mongolian Hordes' management myth is the belief that adding more programmers to a software project that is running late will allow catch-up. In fact, as Brooks says adding people to a late software project actually makes it later.

© Springer International Publishing AG, part of Springer Nature 2018

G. O'Regan, *World of Computing*,
https://doi.org/10.1007/978-3-319-75844-2_9

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. It, therefore, made sense to code as quickly as possible, and to then identify the defects that were present in order 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 organised two famous conferences to discuss critical issues in software development (Buxton 1975). The first conference was held at Garmisch, Germany, in 1968, and it was followed by a second conference in Rome in 1969. Over 50 people from 11 countries attended the Garmisch conference, including Edsger Dijkstra, who did important theoretical work on formal specification and verification. The NATO 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. There were problems with budget and schedule overruns, as well as problems with the quality and reliability of the delivered software.

The conference led to the birth of *software engineering* as a discipline, and the realisation that programming is quite distinct from science and mathematics. Programmers are like engineers in that they build software products, and so they need an 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 essential engineering foundations for 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. These engineering skills include a solid foundation on design and on the mathematics required for building safe software products. Mathematics plays a key role in classical engineering, and it has a role to play in some situations (especially in the safety-critical and security-critical fields) in developing high-quality software products. Several mathematical approaches that may assist software engineers are described in (O'Regan 2017b).

There are parallels between the software crisis of the late 1960s, and the crisis with bridge construction in the nineteenth century. Several bridges collapsed or were delivered late or overbudget in the nineteenth century, because people involved in their design and construction did not have the required engineering knowledge. This led to bridges that were poorly designed and constructed, leading to their collapse and loss of life, as well as endangering the lives of the public.

This led to legislation requiring engineers to be licensed by the Professional Engineering Association prior to practicing as engineers. This organisation 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

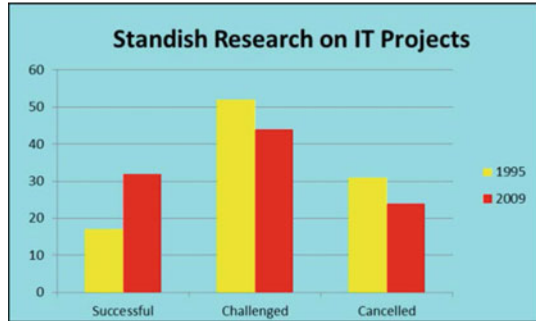

Fig. 9.1 Standish report—results of 1995 and 2009 survey

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 designed and built, and are safe for the public to use.

The Standish group has conducted research (Fig. 9.1) on the extent of problems with IT projects since the mid-1990s. These studies were conducted in the United States, but there is no reason to believe that European or Asian companies perform any better. The results indicate serious problems with on time delivery of projects, and projects being cancelled prior to completion.[2] However, the comparison between 1995 and 2009 suggests that there have been some 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 problems associated with software development such as schedule or budget overruns (Brooks 1975; Brooks 1986). Poor software quality can lead to defects in the software that may adversely impact the customer, or even cause loss of life. It is, therefore, essential that software development organisations place sufficient emphasis on quality throughout the software development process.

The Y2K problem was caused by a two-digit representation of dates, and it required major rework to enable legacy software to function for the new millennium. Clearly, well-designed programs would have hidden the representation of the date, which would have led to 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 impressive.[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 organisations, and various approaches to assess and mature

[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.

[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).

software companies are described in (O'Regan 2010, 2014).[4] These models focus on improving the effectiveness of the management, engineering and organisation practices related to software engineering, and in introducing best practice in software engineering into the organisation. The disciplined use of the mature software processes by the software engineers plays a key role in the consistent delivery of high-quality software.

9.2 What Is Software Engineering?

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

> Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software, and the study of such approaches.

Software engineering includes:

1. Methodologies to 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 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.

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 clear and unambiguous, and reflect the customer's needs. The next step is then to create

[4]Approaches such as the CMM or SPICE (ISO 15504) focus mainly on the management and organisational practices required in software engineering. The emphasis is on defining software processes that are fit for the 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 organisation the freedom to choose the appropriate implementation to meet its needs. The models provide useful information on practices to consider in the implementation.

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, where it may be appropriate to employ mathematical techniques. However, it will usually be sufficient to employ peer reviews or software inspections for verification and validation, as these methodologies provide a high degree of rigour. This may include approaches 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 classical 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 (such as 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 with modelling and understanding the behaviour or properties of the proposed software system.

Software engineers need education[6] on specification, design, 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 engineers.[7] They are responsible for designing and implementing high-quality and

---

[5]Parnas has made important contributions to computer science. He 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 consider the education and qualification of staff prior to assigning staff to performing specific roles. The appropriate qualifications and experience for the specific role are considered prior to appointing a person to carry out the role. Many companies are committed to the education and continuous development of their staff.

[7]The ancient Babylonians used the concept of accountability in the 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.

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 licensed, and therefore Parnas argues that a similar licensing approach be adopted for professional software engineers[10] to provide confidence that they are competent for the assignment. Professional software engineers are required to follow best practice in software engineering and the defined software processes.[11] Chapter 16 discusses ethics and professional responsibility.

Many software companies invest heavily in training, as the education and knowledge of its staff are essential to delivering high-quality products and services. Further, as the computer sector is rapidly changing, employees need to regularly reskill during their careers. Employees receive professional training related to the roles that they are performing, and the fact that the employees are professionally qualified increases 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 obtain poor results, and suffer a loss of reputation and market share.

[8]However, it is unlikely that an individual programmer would be subject to litigation where a program failure causes 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 failure was due to the professional negligence of an individual software engineer, as there are many others involved in the process such as reviewers, testers and the entire project team that developed the software. 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 licensing 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]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 and to be respected worldwide.

[10]The British Computer Society (BCS) has introduced a qualification system for computer science professionals that it 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.

9.3 Challenges in Software Engineering

The challenge in software engineering is to deliver high-quality software on time and on budget to customers. The research done by the Standish Group was discussed earlier in this chapter, and their 1998 research (Fig. 9.2) on project cost overruns in the US indicated that 33% of projects are between 21 and 50% overestimate, 18% are between 51 and 100% overestimate and 11% of projects are between 101 and 200% overestimate.

The accurate estimation of project cost, effort and schedule is a challenge in software engineering. Therefore, project managers need to determine how good their estimation process is and to make appropriate improvements. The use of software metrics is an objective way to do this, and improvements in estimation will be evident from a reduced variance between the estimated and actual effort. The project manager will determine and 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 damage the customer relationship leading to a loss of market share.

There is a strong economic case for 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 are acted upon appropriately. This helps to promote a culture of continuous improvement.

There have been several high-profile software failures (O'Regan 2014) such as the millennium bug (Y2K) problem; the floating-point bug in the Intel

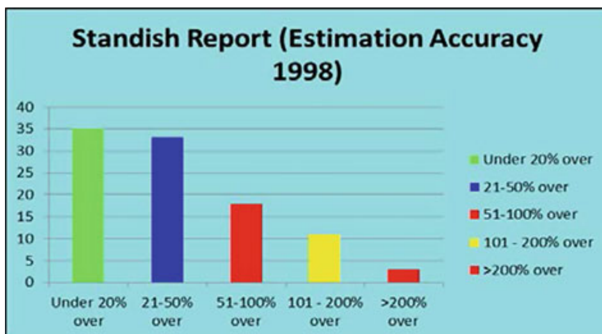

Fig. 9.2 Standish 1998 report—estimation accuracy

microprocessor in the mid-1990s; and the European Space Agency Ariane-5 disaster in 1996. 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 the necessary changes; and verifying the correctness of the changes. 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 on its customers. This led to a large financial cost in replacing microprocessors for its customers. 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 4 June 1996, after a flight time of just 40 s.

These failures indicate that 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.

9.4 Software Processes and Life Cycles

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

It is important that the processes employed are fit for purpose, and a key premise in the software quality field is that the quality of the resulting software is 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 their use is institutionalised in the organisation. That is, all employees need to follow the processes consistently. This requires that people are trained on the new processes and that process discipline is instilled by an appropriate audit strategy.

Employees need to be trained on the processes, and audits are conducted to ensure process compliance. Data will be collected to improve the process. The software process assets in an organisation generally consist of:

- A software development policy for the organisation.
- 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 employed to develop high-quality software generally include:

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

The software development process has an associated life cycle that consists of various phases. There are several well-known life cycles employed such as the waterfall model, the spiral model (Boehm 1988), the Rational Unified Process (Jacobson et al. 1999) and the Agile methodology which has become popular in recent years. The choice of the software development life cycle is determined from the needs of the specific project, and various life cycles are described in more detail in the following sections.

9.4.1 Waterfall Life Cycle

The origins of the waterfall model[12] (Fig. 9.3) are in the manufacturing and construction industry, and Winston Royce defined it formally for software development in 1970 (Royce 1970). It starts with requirements gathering and definition. It is followed by the functional specification, the design and implementation of the software and comprehensive testing. The testing generally includes unit, system and user acceptance testing.

It is employed for projects where the requirements can be identified early in the project life cycle or are known in advance. It is also called the 'V' life cycle model, with the left-hand side of the 'V' detailing requirements, specification, design and coding and the right-hand side detailing unit tests, integration tests, system tests and acceptance testing. Each phase has entry and exit criteria that must be satisfied before the next phase commences.

Many companies employ a set of templates to enable the activities in the 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 on industrial best practice.

[12] We treat the waterfall model as identical to the V model in this text.

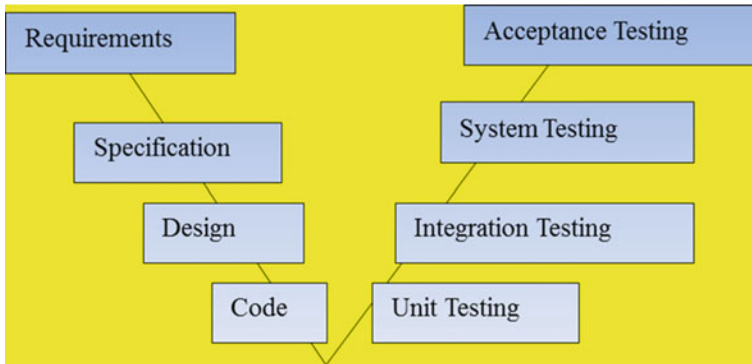

Fig. 9.3 Waterfall versus life cycle model

9.4.2 Spiral Life Cycles

The spiral model (Fig. 9.4) was developed by Barry Boehm in the mid-1980s, and it is useful when the requirements are not fully known at project initiation, or where the requirements evolve as a part of the development life cycle. The software development involves several spirals, where each spiral typically involves objectives and an analysis of the risks, updates to the requirements, design, code, testing and a user review of the iteration or spiral. The early spirals are concerned with prototyping, and the later spirals are concerned with the full implementation of the system.

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 is 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. Agile methods have become popular in recent years and these generally employ sprints (or iterations) of 2 weeks duration to implement several user stories.

There are other life cycle models, for example the iterative development process that combines the waterfall and spiral life cycle model. The cleanroom approach developed by Harlan Mills at IBM includes a phase for formal specification, and its approach to software testing is based on the predicted usage of the software product. The Rational Unified Process is discussed in the next section.

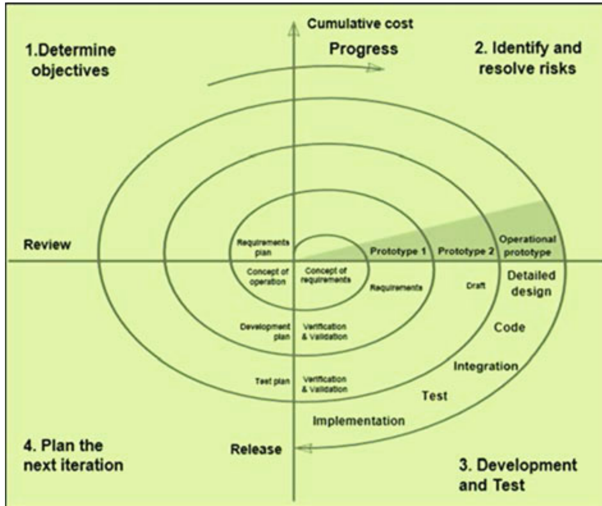

Fig. 9.4 SPIRAL life cycle model. Public domain

9.4.3 Rational Unified Process

The *rational unified process* (Jacobson et al. 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, where UML is a visual modelling language for software systems that provide a means of specifying, constructing and documenting the object-oriented system. UML was developed by James Rumbaugh, Grady Booch and Ivar Jacobson, and it facilitates understanding of the architecture and complexity of the system.

RUP is *use case driven*, *architecture centric*, *iterative* and *incremental*, and includes cycles, phases, workflows, risk mitigation, quality control, project management and configuration control. 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.

Requirements are gathered as use cases, which *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 realise 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 implements the use cases.

The software architecture concept embodies the most significant static and dynamic aspects of the system. The architecture grows out of the use cases and

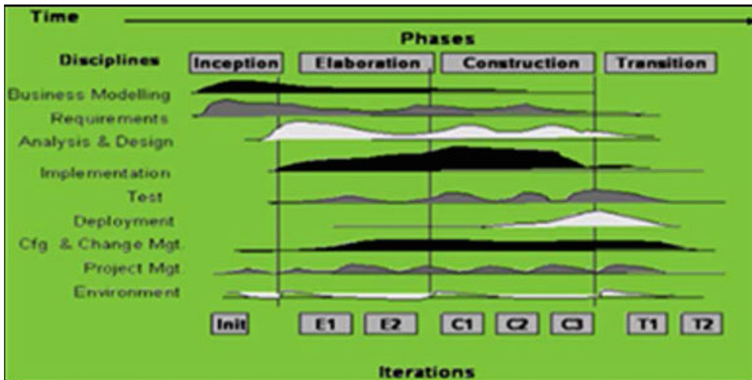

Fig. 9.5 Rational unified process

factors such as the platform that the software is to run on, deployment considerations, legacy systems and 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 than the entire product. That is, RUP is a way to mitigate risk in software engineering (Fig. 9.5).

9.4.4 Agile Development

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 like 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*.

An early version of the methodology was introduced in the late 1980s/early 1990s, and the Agile Manifesto was introduced in early 2001 (www.agilealliance.org). Agile has a strong collaborative style of working and its approach includes:

- Aim is to achieve a narrow fast flowing value stream.
- Feedback and adaptation are employed in decision-making.
- User stories and sprints are employed.
- Stories are either done or not 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 life cycle is employed in 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 considered 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 during the project rather than attempting to define all the requirements at the start of the project. The methodology includes controls to manage changes to the requirements, and good communication and 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 life cycle 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 prioritised based on several factors including:

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

Sprint planning is performed before the start of the iteration, and stories are assigned to the iteration to fill the available time. The estimates for each story and their priority are determined, and the prioritised stories are assigned to the iteration. *A short morning stand-up meeting is held daily* during the iteration, and attended by the project manager 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, and this is used for continuous improvement in future iterations.

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.

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. 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 automated tests to be run thereby identifying problems earlier.

9.5 Activities in Waterfall Life Cycle

This section describes the various activities in the waterfall software development life cycle in more detail. The activities discussed include:

- Business Requirements Definition,
- Specification of System Requirements,
- Design,
- Implementation,
- Unit Testing,
- System Testing,
- UAT Testing,
- Support and Maintenance.

9.5.1 Business Requirements Definition

The 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 specification of the 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 involve meetings with the stakeholders to gather all relevant information for the proposed product. The stakeholders are interviewed, and requirements workshops 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 prioritise the requirements, as well as identifying and resolving any conflicting requirements. The collected information is consolidated into a coherent set of requirements.

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. Changes to the requirements may occur during the project, and these need to be controlled. It is essential to understand the impacts of a change prior to its approval.

The requirements for a system are generally documented in a natural language such as 'English'. Other notations that may be employed to express the requirements include the visual modelling language UML (Jacobson et al. 2005), and formal specification languages such as VDM or Z.

9.5.2 Specification of System Requirements

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

It is essential that the system requirements are valid with respect to the business requirements, and the stakeholders review them to ensure their validity. Traceability may be employed to show how 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 screenshots, report layouts or the desired functionality specified in natural language, use cases, etc. The *non-*

functional requirements will generally include security, reliability, performance and portability requirements, as well as usability and maintainability requirements.

9.5.3 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 the proposed system, and the designer will need to decide on the most appropriate solution.

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 flowcharts, or various UML diagrams such as sequence diagrams, state charts and so on. Program description languages or pseudocode may be employed to define the algorithms and data structures that are the basis for implementation.

Functional design involves starting with a high-level view of the system and refining it into a more detailed design. The system state is centralised and shared between the functions operating on that state.

Object-oriented design is based on the concept of *information hiding* (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 decentralised 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 superclasses. Objects communicate by exchanging messages.

It is essential to verify and validate the design with respect to the system requirements, and this will be done by design reviews, and traceability of the design to the system requirements.

9.5.4 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 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 adheres to the coding standards,

that maintainability issues are addressed, and that the code produced is a valid implementation of the software design.

Software reuse has become more important in recent times as it 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 been 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 license*) 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 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.

9.5.5 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), and prior to its integration with other modules. The programmer writes these tests, and the objective is to show that the code satisfies its design. Each unit test case is documented and it should include a test objective and the expected result.

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

The developer executes the unit tests, records the results, corrects any identified defects and retests the software. *Test-driven development* has become popular in recent years (e.g. in the Agile world), and this involves writing the unit test case before the code, and the code is written to pass the unit test cases.

Integration Test

The development team performs this type of testing on the integrated system, once all individual units work correctly in isolation. The objective is to verify that the modules and their interfaces work correctly together, and to identify and resolve

any issues. Modules that work correctly in isolation may fail when integrated with other modules.

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 tests are traceable to the system requirements.

Any system requirements that have been incorrectly implemented will be identified, and defects logged and reported to the developers. The test group will verify that the revised version of the software is correct, and regression testing is carried out to verify system integrity. System testing may include security testing, usability testing and performance testing.

The preparation of the test environment may involve ordering special hardware and tools, and it is important that the test environment is set up as early as possible to allow the timely 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 in the non-functional requirements, and to determine if the system is scalable to support future growth. It may include *load performance testing*, where the system is subjected to heavy loads over a long period, 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 measuring the response times for various activities. Test tools are employed to simulate many users and heavy loads.

User Acceptance Test

UAT testing 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 can judge if the system is fit for purpose.

The objective is to demonstrate that the product satisfies the business requirements and meets the customer expectations. Upon its successful completion, the customer should be happy to accept the product.

9.5.6 Maintenance

This phase continues after the release of the software product to the customer. Any problems that the customer notes 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. 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 organisations will conduct postmortems 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 developing 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 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 minimise the risk that that serious defects will be found post release.

The more effective the in-phase inspections of deliverables, the higher the quality of the resulting implementation, with a corresponding reduction in the number of defects detected by the test groups. 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. Testing and retesting to achieve quality, until the testing group is confident that all defects have been eliminated almost seems to be a '*brute force*' approach. Dijkstra (1972) noted that:

> 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 the program is correct, and, at best, statistical techniques may be employed to give a measure of the confidence in its correctness. That is, there is no guarantee that all defects have been found in the software.

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 organisations have a quality objective of three defects per million lines of code. This goal is known as Six Sigma (6σ), and Motorola developed it initially for its manufacturing businesses and later applied to its software organisation. The goal is to reduce variability in manufacturing processes and to ensure that the processes performed within strict process control limits. Motorola was awarded the first Malcolm Baldridge Quality Award for its Six Sigma initiative and its commitment to quality.

---

[13]This is essential for serious defects that have caused significant inconvenience to customers (e.g. a major telecom outage). The software development organisation 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.

9.6 Software Inspections

Software inspections are used to build quality into software products, and there are several well-known approaches such as the Fagan Methodology (Fagan 1976) developed by Michael Fagan of IBM. This is a seven-step process that identifies and removes errors in work products. The process mandates that requirement documents, design documents, source code and test plans are all formally inspected by experts who are independent of the author of the deliverable.

There are various *roles* defined in the process including the *moderator* who chairs the inspection. The *reader's* responsibility is to read or paraphrase the deliverable, and *the author* is the creator of the deliverable and the *tester* role is concerned with the testing viewpoint.

The inspection process will consider whether the design is correct with respect to the requirements, and whether the source code is correct with respect to the design. Software inspections play an important role in building quality into the software, and in reducing the cost of poor quality in the organisation. For more detailed information on software inspections see (O'Regan 2014).

9.7 Software Project Management

The timely delivery of quality software requires good project management and engineering processes. Software projects have a history of being delivered late or overbudget, and good project management practices include activities such as:

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

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 for software projects as often these involve new technologies and are breaking new ground. This means that they are often may be quite different from previous projects, and so historical estimates may not be a good basis for estimation. Further, unanticipated problems may arise with technically advanced projects, and so the estimates may be overly optimistic.

Gantt charts are generally employed for project scheduling, and these show the work breakdown for the project, as well as task dependencies. The Gantt chart shows the allocation of staff to the various tasks, where each task has a start date and an end date, the effort associated with it, as well as the staff involved.

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. For more detailed information on project management see (O'Regan 2017a).

9.8 CMMI Maturity Model

The CMMI is a framework to assist an organisation in the implementation of best practice in software and systems engineering (Chrissis et al. 2011). It is an internationally recognised model for process improvement and assessment, and is used worldwide by thousands of organisations. It provides a framework for an organisation to introduce a solid engineering approach to the development of software, and the CMMI practices support the implementation 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. It developed the original CMM model in the early 1990s, and its successor is the CMMI. The CMMI states *what the organisation needs to do* to mature its processes rather than *how this should be done*, and so the organisation has the freedom to interpret the model meets its business needs effectively.

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 organisation practices; level four is concerned with ensuring that key processes are performing within strict quantitative limits; level five is concerned with continuous process improvement. Maturity levels may not be skipped in the staged implementation of the CMMI, as each maturity level provides the foundation for the next level.

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

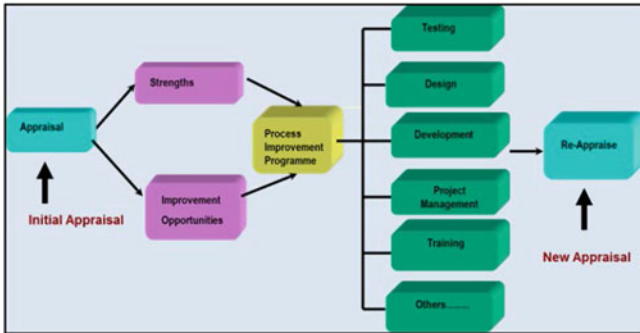

Fig. 9.6 Software process improvement

The CMMI allows organisations to benchmark themselves against other organisations. This is done by a formal appraisal conducted by an authorised lead appraiser (Standard CMMI Appraisal Method for Process Improvement 2006). The results of the appraisal are generally reported back to the SEI, and there is a strict qualification process to become an *authorised lead appraiser*.

An appraisal is useful in that it allows the organisation to determine its current software process maturity. It may be used to verify that the organisation has improved, and it enables the organisation to prioritise improvements (Fig. 9.6). The CMMI is discussed in more detail in (O'Regan 2010, 2014).

9.9 Formal Methods

Dijkstra and Hoare argued that the appropriate way to develop correct software is to derive the program from its formal mathematical specification, and to employ *mathematical proof* to demonstrate the correctness of the software with respect to the specification. This is 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 their widespread deployment in industry is unlikely at this time.

The *safety-critical area* is one domain to which mathematical techniques have been successfully used. For example, it is essential in the railway domain that a property such as '*when a train is in a level crossing, then the gate is closed*' is demonstrated to be correct, and formal methods can play a key role in the verification of safety-critical properties. There is a need for extra rigour in the software development process in the safety-critical field, and mathematical techniques can demonstrate the presence or absence of certain desirable or undesirable properties.

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 speculation about the meaning of phrases 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 in a mathematical language, and its precision helps to avoid the problem of ambiguity inherent in a natural language specification.

The derivation of the software 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 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 described in more detail in (O'Regan 2017b).

9.10 Review Questions

1. Discuss the research results of the Standish Group on the current state of IT project delivery.
2. What are the main challenges in software engineering?
3. Describe the various existing software life cycles
4. What are the advantages and disadvantages of Agile?
5. Describe the purpose of software inspections? What are the benefits?
6. Describe the main activities in software testing.
7. Describe the advantages and disadvantages of formal methods.
8. Describe the main activities in project management.
9. Explain the significance of the CMMI.

9.11 Summary

The birth of software engineering was at the NATO conference held in 1968 in Germany. This conference highlighted the problems that existed in the software sector in the late 1960s, and the term '*software crisis*' was coined to refer to these. It led to the realisation that software engineers need to be properly trained to enable them to build high-quality products that are safe to use.

The Standish Group conducts research on the extent of problems with the delivery of projects on time and budget. Their research indicates that it remains a challenge to deliver projects on time, on budget and with the right quality.

Programmers are like engineers in the sense that they build products. Therefore, programmers need to receive an appropriate education in engineering as well as on the latest technologies. Classical engineers receive training on product design, and an appropriate level of mathematics.

Software engineering involves multi-person construction of multi-version programs. It 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. The solution is verified by rigorous software testing.

Software process maturity models such as the CMMI place an emphasis on understanding and improving the software processes in an organisation. The CMMI is a framework to implement high-quality processes, and a SCAMPI appraisal allows organisations to benchmark themselves against other similar organisations.

Formal methods involve the use of mathematical techniques to provide extra confidence in the correctness of the software. They are used mainly in the safety- and security-critical field.