Software Development & Quality Handbook

Lamar Lacy
Lane Galvin

First Edition, 2012

ISBN 978-81-323-1447-9

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Software Development Process

A **software development process**, also known as a **software development lifecycle**, is a structure imposed on the development of a software product. Similar terms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. Some people consider a lifecycle model a more general term and a software development process a more specific term. For example, there are many specific software development processes that 'fit' the spiral lifecycle model.

Overview

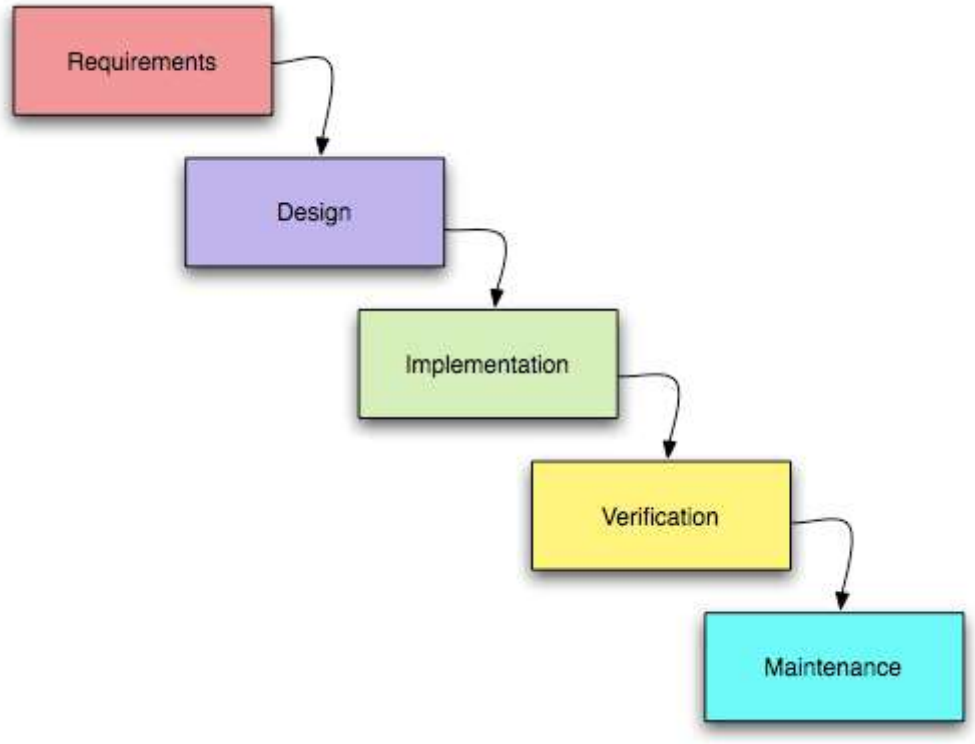
The large and growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts.

The international standard for describing the method of selecting, implementing and monitoring the life cycle for software is ISO 12207.

A decades-long goal has been to find repeatable, predictable processes that improve productivity and quality. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management techniques to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management appears to be lacking.

Organizations may create a Software Engineering Process Group (SEPG), which is the focal point for process improvement. Composed of line practitioners who have varied skills, the group is at the center of the collaborative effort of everyone in the organization who is involved with software engineering process improvement.

Software development activities



The activities of the software development process represented in the waterfall model. There are several other models to represent this process.

Planning

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Once the general requirements are gathered from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document.

Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.

Implementation, testing and documenting

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important part of the software development process. This part of the process ensures that defects are recognized as early as possible.

Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the writing of an API, be it external or internal. It is very important to document everything in the project.

Deployment and maintenance

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

Software Training and Support is important and a lot of developers fail to realize that. It would not matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality of at least one prior phase is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

Software Development Models

Several models exist to streamline the development process. Each one has its pros and cons, and it's up to the development team to adopt the most appropriate one for the project. Sometimes a combination of the models may be more suitable.

Waterfall Model

The waterfall model shows a process, where developers are to follow these phases in order:

1. Requirements specification (Requirements analysis)
2. Software Design
3. Integration

4. Testing (or Validation)
5. Deployment (or Installation)
6. Maintenance

In a strict Waterfall model, after each phase is finished, it proceeds to the next one. Reviews may occur before moving to the next phase which allows for the possibility of changes (which may involve a formal change control process). Reviews may also be employed to ensure that the phase is indeed complete; the phase completion criteria are often referred to as a "gate" that the project must pass through to move to the next phase. Waterfall discourages revisiting and revising any prior phase once it's complete. This "inflexibility" in a pure Waterfall model has been a source of criticism by supporters of other more "flexible" models.

Spiral Model

The key characteristic of a Spiral model is risk management at regular stages in the development cycle. In 1988, Barry Boehm published a formal software system development "spiral model", which combines some key aspect of the waterfall model and rapid prototyping methodologies, but provided emphasis in a key area many felt had been neglected by other methodologies: deliberate iterative risk analysis, particularly suited to large-scale complex systems.

The Spiral is visualized as a process passing through some number of iterations, with the four quadrant diagram representative of the following activities:

1. formulate plans to: identify software targets, selected to implement the program, clarify the project development restrictions;
2. Risk analysis: an analytical assessment of selected programs, to consider how to identify and eliminate risk;
3. the implementation of the project: the implementation of software development and verification;

Risk-driven spiral model, emphasizing the conditions of options and constraints in order to support software reuse, software quality can help as a special goal of integration into the product development. However, the spiral model has some restrictive conditions, as follows:

1. The spiral model emphasizes risk analysis, and thus requires customers to accept this analysis and act on it. This requires both trust in the developer as well as the willingness to spend more to fix the issues, which is the reason why this model is often used for large-scale internal software development.
2. If the implementation of risk analysis will greatly affect the profits of the project, the spiral model should not be used.
3. Software developers have to actively look for possible risks, and analyze it accurately for the spiral model to work.

The first stage is to formulate a plan to achieve the objectives with these constraints, and then strive to find and remove all potential risks through careful analysis and, if necessary, by constructing a prototype. If some risks can not be ruled out, the customer has to decide whether to terminate the project or to ignore the risks and continue anyway. Finally, the results are evaluated and the design of the next phase begins.

Iterative and Incremental development

Iterative development prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes are preferred by commercial developers because it allows a potential of reaching the design goals of a customer who does not know how to define what they want.

Agile development

Agile software development uses iterative development as a basis but advocates a lighter and more people-centric viewpoint than traditional approaches. Agile processes use feedback, rather than planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software.

There are many variations of agile processes:

- In Extreme Programming (XP), the phases are carried out in extremely small (or "continuous") steps compared to the older, "batch" processes. The (intentionally incomplete) first pass through the steps might take a day or a week, rather than the months or years of each complete step in the Waterfall model. First, one writes automated tests, to provide concrete goals for development. Next is coding (by a pair of programmers), which is complete when all the tests pass, and the programmers can't think of any more tests that are needed. Design and architecture emerge out of refactoring, and come after coding. Design is done by the same people who do the coding. (Only the last feature — merging design and code — is common to *all* the other agile processes.) The incomplete but functional system is deployed or demonstrated for (some subset of) the users (at least one of which is on the development team). At this point, the practitioners start again on writing tests for the next most important part of the system.

- Scrum

Process Improvement Models

Capability Maturity Model Integration

The Capability Maturity Model Integration (CMMI) is one of the leading models and based on best practice. Independent assessments grade organizations on how well they follow their defined processes, not on the quality of those processes or the software produced. CMMI has replaced CMM.

ISO 9000

ISO 9000 describes standards for a formally organized process to manufacture a product and the methods of managing and monitoring progress. Although the standard was originally created for the manufacturing sector, ISO 9000 standards have been applied to software development as well. Like CMMI, certification with ISO 9000 does not guarantee the quality of the end result, only that formalized business processes have been followed.

ISO 15504

ISO 15504, also known as Software Process Improvement Capability Determination (SPICE), is a "framework for the assessment of software processes". This standard is aimed at setting out a clear model for process comparison. SPICE is used much like CMMI. It models processes to manage, control, guide and monitor software development. This model is then used to measure what a development organization or project team actually does during software development. This information is analyzed to identify weaknesses and drive improvement. It also identifies strengths that can be continued or integrated into common practice for that organization or team.

Formal methods

Formal methods are mathematical approaches to solving software (and hardware) problems at the requirements, specification and design levels. Examples of formal methods include the B-Method, Petri nets, Automated theorem proving, RAISE and VDM. Various formal specification notations are available, such as the Z notation. More generally, automata theory can be used to build up and validate application behavior by designing a system of finite state machines.

Finite state machine (FSM) based methodologies allow executable software specification and by-passing of conventional coding.

Formal methods are most likely to be applied in avionics software, particularly where the software is safety critical. Software safety assurance standards, such as DO178B demand formal methods at the highest level of categorization (Level A).

Formalization of software development is creeping in, in other places, with the application of Object Constraint Language (and specializations such as Java Modeling Language) and especially with Model-driven architecture allowing execution of designs, if not specifications.

Another emerging trend in software development is to write a specification in some form of logic (usually a variation of FOL), and then to directly execute the logic as though it were a program. The OWL language, based on Description Logic, is an example. There is also work on mapping some version of English (or another natural language) automatically to and from logic, and executing the logic directly. Examples are Attempto Controlled English, and Internet Business Logic, which does not seek to control the vocabulary or syntax. A feature of systems that support bidirectional English-logic

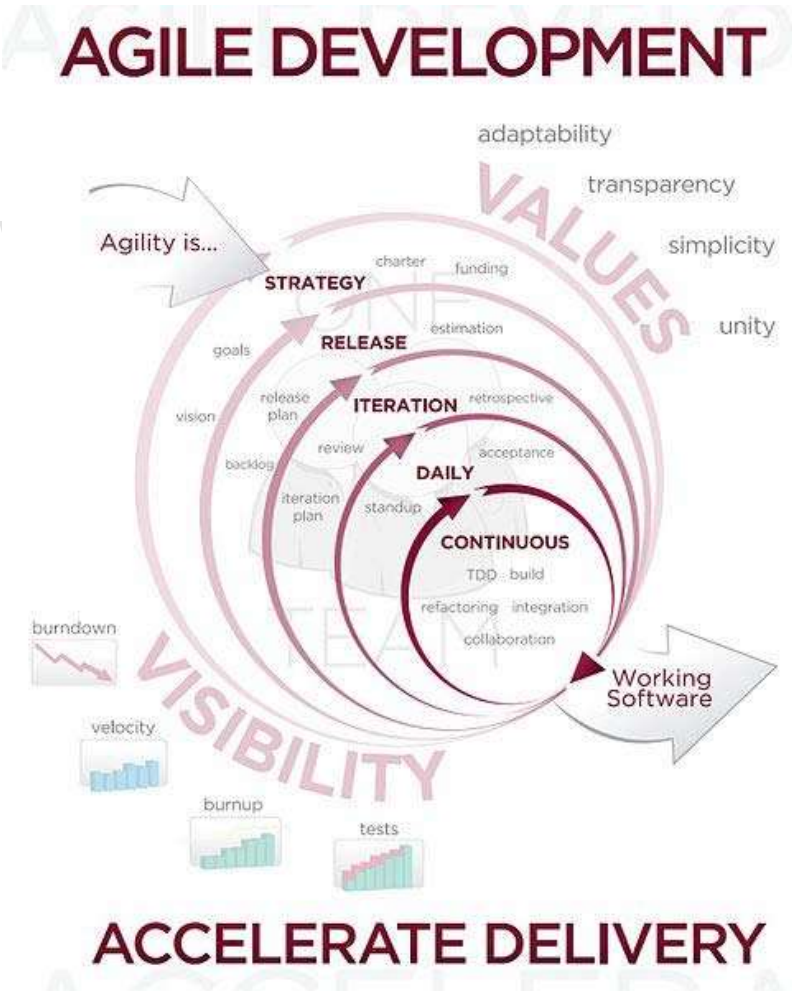
mapping and direct execution of the logic is that they can be made to explain their results, in English, at the business or scientific level.

The Government Accountability Office, in a 2003 report on one of the Federal Aviation Administration's air traffic control modernization programs, recommends following the agency's guidance for managing major acquisition systems by

- establishing, maintaining, and controlling an accurate, valid, and current performance measurement baseline, which would include negotiating all authorized, unpriced work within 3 months;
- conducting an integrated baseline review of any major contract modifications within 6 months; and
- preparing a rigorous life-cycle cost estimate, including a risk assessment, in accordance with the Acquisition System Toolset's guidance and identifying the level of uncertainty inherent in the estimate.

Chapter 2

Agile Software Development

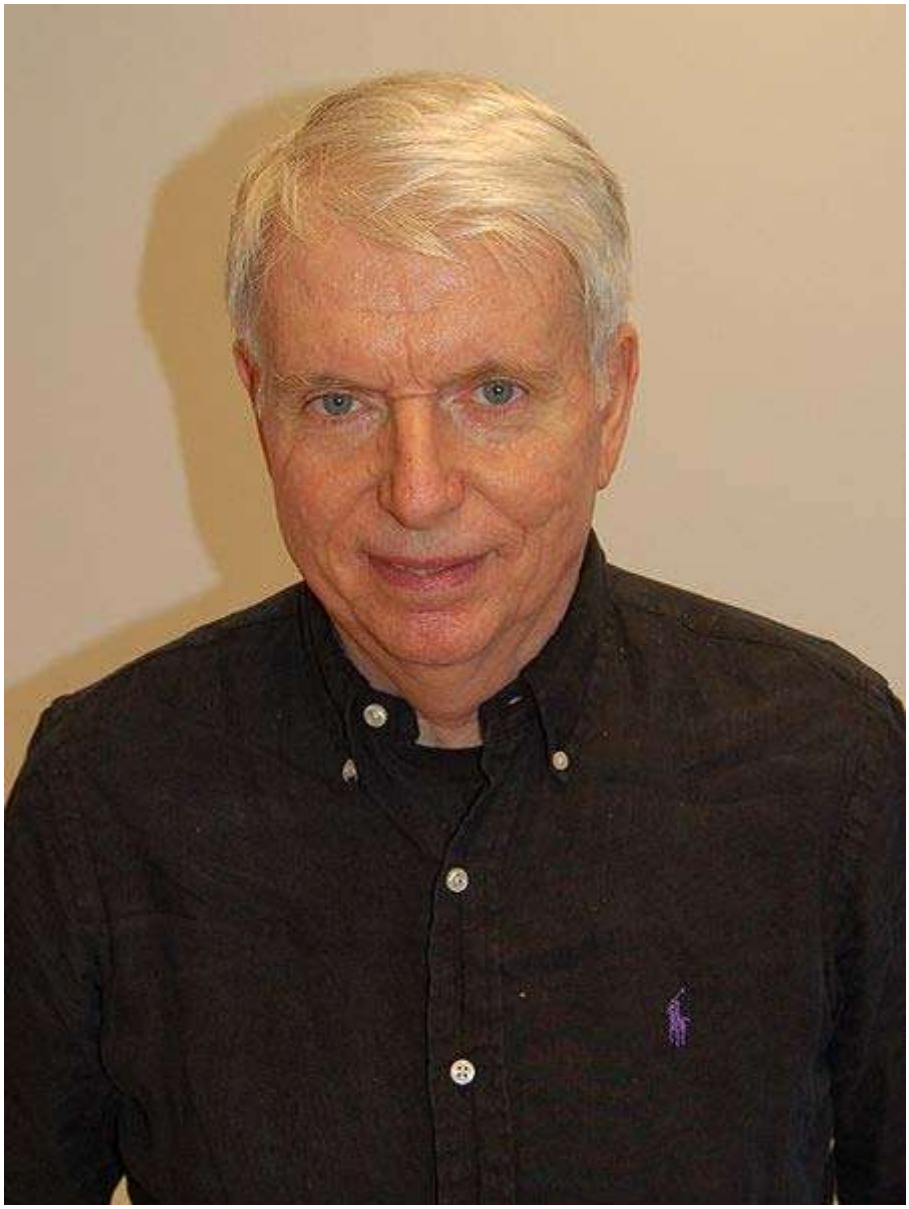


Agile software development poster

Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. The *Agile Manifesto* introduced the term in 2001.

History

Predecessors



Jeff Sutherland, one of the developers of the Scrum agile software development process

Incremental software development methods have been traced back to 1957. In 1974, a paper by E. A. Edmonds introduced an adaptive software development process.

So-called *lightweight* software development methods evolved in the mid-1990s as a reaction against *heavyweight* methods, which were characterized by their critics as a heavily regulated, regimented, micromanaged, waterfall model of development. Proponents of lightweight methods (and now *agile* methods) contend that they are a return to development practices from early in the history of software development.

Early implementations of lightweight methods include Scrum (1995), Crystal Clear, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (DSDM) (1995). These are now typically referred to as agile methodologies, after the Agile Manifesto published in 2001.

Agile Manifesto

In February 2001, 17 software developers met at the Snowbird, Utah resort, to discuss lightweight development methods. They published the *Manifesto for Agile Software Development* to define the approach now known as agile software development. Some of the manifesto's authors formed the Agile Alliance, a nonprofit organization that promotes software development according to the manifesto's principles.

Agile Manifesto reads, in its entirety, as follows:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

> **Individuals and interactions** over processes and tools
> **Working software** over comprehensive documentation
> **Customer collaboration** over contract negotiation
> **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Twelve principles underlie the Agile Manifesto, including:

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily co-operation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances

In 2005, a group headed by Alistair Cockburn and Jim Highsmith wrote an addendum of project management principles, the Declaration of Interdependence, to guide software project management according to agile development methods.

Characteristics

Pair programming, an XP development technique used by agile

There are many specific agile development methods. Most promote development, teamwork, collaboration, and process adaptability throughout the life-cycle of the project.

Agile methods break tasks into small increments with minimal planning, and do not directly involve long-term planning. Iterations are short time frames (timeboxes) that typically last from one to four weeks. Each iteration involves a team working through a full software development cycle including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders. This minimizes overall risk and allows the project to adapt to changes quickly. Stakeholders produce documentation as required. An iteration may not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal bugs) at the end of each iteration. Multiple iterations may be required to release a product or new features.

Team composition in an agile project is usually cross-functional and self-organizing without consideration for any existing corporate hierarchy or the corporate roles of team members. Team members normally take responsibility for tasks that deliver the functionality an iteration requires. They decide individually how to meet an iteration's requirements.

Agile methods emphasize face-to-face communication over written documents when the team is all in the same location. Most agile teams work in a single open office (called a bullpen), which facilitates such communication. Team size is typically small (5-9 people) to simplify team communication and team collaboration. Larger development efforts may be delivered by multiple teams working toward a common goal or on different parts of an effort. This may require a co-ordination of priorities across teams. When a team works in different locations, they maintain daily contact through videoconferencing, voice, e-mail, etc.

No matter what development disciplines are required, each agile team will contain a customer representative. This person is appointed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer mid-iteration problem-domain questions. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the return on investment (ROI) and ensuring alignment with customer needs and company goals.

Most agile implementations use a routine and formal daily face-to-face communication among team members. This specifically includes the customer representative and any interested stakeholders as observers. In a brief session, team members report to each other what they did the previous day, what they intend to do today, and what their roadblocks are. This face-to-face communication exposes problems as they arise.

Agile development emphasizes working software as the primary measure of progress. This, combined with the preference for face-to-face communication, produces less written documentation than other methods. The agile method encourages stakeholders to prioritize wants with other iteration outcomes based exclusively on business value perceived at the beginning of the iteration (also known as *value-driven*).

Specific tools and techniques such as continuous integration, automated or xUnit test, pair programming, test driven development, design patterns, domain-driven design, code refactoring and other techniques are often used to improve quality and enhance project agility.

Comparison with other methods

Agile methods are sometimes characterized as being at the opposite end of the spectrum from *plan-driven* or *disciplined* methods; agile teams may, however, employ highly disciplined formal methods. A more accurate distinction is that methods exist on a continuum from *adaptive* to *predictive*. Agile methods lie on the *adaptive* side of this continuum. Adaptive methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team will have difficulty describing exactly what will happen in the future. The further away a date is, the more vague an adaptive method will be about what will happen on that date. An adaptive team cannot report exactly what tasks are being done next week, but only which features are planned for next month. When asked about a release six months from now,

an adaptive team may only be able to report the mission statement for the release, or a statement of expected value vs. cost.

Predictive methods, in contrast, focus on planning the future in detail. A predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive teams have difficulty changing direction. The plan is typically optimized for the original destination and changing direction can require completed work to be started over. Predictive teams will often institute a change control board to ensure that only the most valuable changes are considered.

Formal methods, in contrast to adaptive and predictive methods, focus on computer science theory with a wide array of types of provers. A formal method attempts to prove the absence of errors with some level of determinism. Some formal methods are based on model checking and provide counter examples for code that cannot be proven. Generally, mathematical models map to assertions about requirements. Formal methods are dependent on a tool driven approach, and may be combined with other development approaches. Some provers do not easily scale. Like agile methods, manifestos relevant to high integrity software have been proposed in Crosstalk.

Agile methods have much in common with the *Rapid Application Development* techniques from the 1980/90s as espoused by James Martin and others.

Agile methods

Well-known agile software development methods include:

- Agile Modeling
- Agile Unified Process (AUP)
- Dynamic Systems Development Method (DSDM)
- Essential Unified Process (EssUP)
- Extreme Programming (XP)
- Feature Driven Development (FDD)
- Open Unified Process (OpenUP)
- Scrum
- Velocity tracking

Method tailoring

In the literature, different terms refer to the notion of method adaptation, including ‘method tailoring’, ‘method fragment adaptation’ and ‘situational method engineering’. Method tailoring is defined as:

A process or capability in which human agents through responsive changes in, and dynamic interplays between contexts, intentions, and method fragments determine a system development approach for a specific project situation.

Potentially, almost all agile methods are suitable for method tailoring. Even the DSDM method is being used for this purpose and has been successfully tailored in a CMM context. Situation-appropriateness can be considered as a distinguishing characteristic between agile methods and traditional software development methods, with the latter being relatively much more rigid and prescriptive. The practical implication is that agile methods allow project teams to adapt working practices according to the needs of individual projects. Practices are concrete activities and products that are part of a method framework. At a more extreme level, the philosophy behind the method, consisting of a number of principles, could be adapted (Aydin, 2004).

Extreme Programming (XP) makes the need for method adaptation explicit. One of the fundamental ideas of XP is that no one process fits every project, but rather that practices should be tailored to the needs of individual projects. Partial adoption of XP practices, as suggested by Beck, has been reported on several occasions. A tailoring practice is proposed by Mehdi Mirakhorli which provides sufficient roadmap and guideline for adapting all the practices. RDP Practice is designed for customizing XP. This practice, first proposed as a long research paper in the APSO workshop at the ICSE 2008 conference, is currently the only proposed and applicable method for customizing XP. Although it is specifically a solution for XP, this practice has the capability of extending to other methodologies. At first glance, this practice seems to be in the category of static method adaptation but experiences with RDP Practice says that it can be treated like dynamic method adaptation. The distinction between static method adaptation and dynamic method adaptation is subtle. The key assumption behind static method adaptation is that the project context is given at the start of a project and remains fixed during project execution. The result is a static definition of the project context. Given such a definition, route maps can be used in order to determine which structured method fragments should be used for that particular project, based on predefined sets of criteria. Dynamic method adaptation, in contrast, assumes that projects are situated in an emergent context. An emergent context implies that a project has to deal with emergent factors that affect relevant conditions but are not predictable. This also means that a project context is not fixed, but changing during project execution. In such a case prescriptive route maps are not appropriate. The practical implication of dynamic method adaptation is that project managers often have to modify structured fragments or even innovate new fragments, during the execution of a project (Aydin et al., 2005).

Measuring agility

While agility can be seen as a means to an end, a number of approaches have been proposed to quantify agility. *Agility Index Measurements* (AIM) score projects against a number of agility factors to achieve a total. The similarly named *Agility Measurement Index*, scores developments against five dimensions of a software project (duration, risk, novelty, effort, and interaction). Other techniques are based on measurable goals. Another study using fuzzy mathematics has suggested that project velocity can be used as a metric of agility. There are agile self assessments to determine whether a team is using agile practices (Nokia test, Karlskrona test, 42 points test).

While such approaches have been proposed to measure agility, the practical application of such metrics has yet to be seen.

Experience and reception

One of the early studies reporting gains in quality, productivity, and business satisfaction by using Agile methods was a survey conducted by Shine Technologies from November 2002 to January 2003. A similar survey conducted in 2006 by Scott Ambler, the Practice Leader for Agile Development with IBM Rational's Methods Group reported similar benefits. In a survey conducted by VersionOne in 2008, 55% of respondents answered that Agile methods had been successful in 90-100% of cases. Others claim that agile development methods are still too young to require extensive academic proof of their success.

Suitability

Large-scale agile software development remains an active research area.

Agile development has been widely documented as working well for small (<10 developers) co-located teams.

Some things that may negatively impact the success of an agile project are:

- Large-scale development efforts (>20 developers), through scaling strategies and evidence of some large projects have been described.
- Distributed development efforts (non-colocated teams). Strategies have been described in *Bridging the Distance* and *Using an Agile Software Process with Offshore Development*
- Forcing an agile process on a development team
- Mission-critical systems where failure is not an option at any cost (e.g. software for surgical procedures).

Several successful large-scale agile projects have been documented. BT has had several hundred developers situated in the UK, Ireland and India working collaboratively on projects and using Agile methods.

In terms of outsourcing agile development, Michael Hackett, Sr. Vice President of LogiGear Corporation has stated that "the offshore team ... should have expertise, experience, good communication skills, inter-cultural understanding, trust and understanding between members and groups and with each other."

Risk analysis can also be used to choose between adaptive (*agile* or *value-driven*) and predictive (*plan-driven*) methods.. Barry Boehm and Richard Turner suggest that each side of the continuum has its own *home ground*, as follows:

Agile home ground:

- Low criticality
- Senior developers
- Requirements change often
- Small number of developers
- Culture that thrives on chaos

Plan-driven home ground:

- High criticality
- Junior developers
- Requirements do not change often
- Large number of developers
- Culture that demands order

Formal methods:

- Extreme criticality
- Senior developers
- Limited requirements, limited features
- Requirements that can be modeled
- Extreme quality

Experience reports

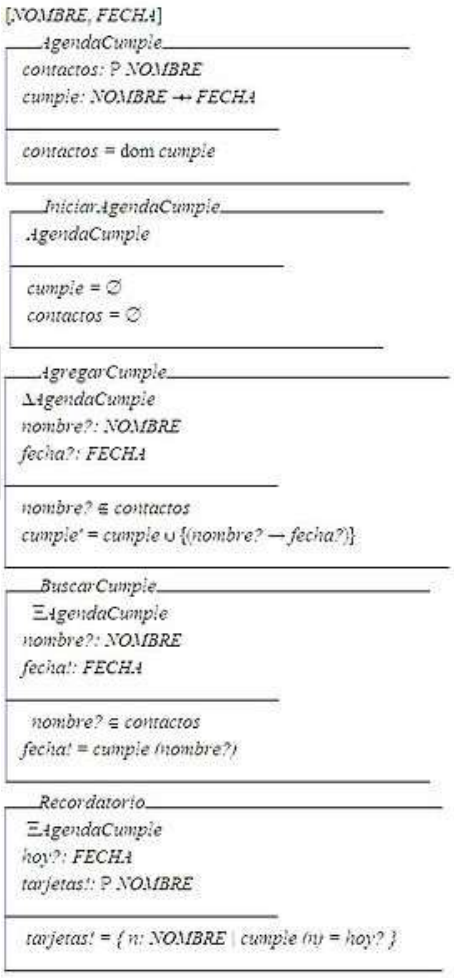
Agile development has been the subject of several conferences. Some of these conferences have had academic backing and included peer-reviewed papers, including a peer-reviewed experience report track. The experience reports share industry experiences with agile software development.

As of 2006, experience reports have been or will be presented at the following conferences:

- XP (2000, 2001, 2002, 2003, 2004, 2005, 2006, 2010 (proceedings published by IEEE))
- XP Universe (2001)
- XP/Agile Universe (2002, 2003, 2004)
- Agile Development Conference (2003,2004,2007,2008) (peer-reviewed; proceedings published by IEEE)

Chapter 3

Formal Methods



An example formal specification using the Z notation

In computer science and software engineering, **formal methods** are a particular kind of mathematically-based techniques for the specification, development and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and

robustness of a design. However, the high cost of using formal methods means that they are usually only used in the development of high-integrity systems, where safety or security is of utmost importance.

Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification.

Taxonomy

Formal methods can be used at a number of levels:

Level 0: Formal specification may be undertaken and then a program developed from this informally. This has been dubbed *formal methods lite*. This may be the most cost-effective option in many cases.

Level 1: Formal development and formal verification may be used to produce a program in a more formal manner. For example, proofs of properties or refinement from the specification to a program may be undertaken. This may be most appropriate in high-integrity systems involving safety or security.

Level 2: Theorem provers may be used to undertake fully formal machine-checked proofs. This can be very expensive and is only practically worthwhile if the cost of mistakes is extremely high (e.g., in critical parts of microprocessor design).

As with programming language semantics, styles of formal methods may be roughly classified as follows:

- Denotational semantics, in which the meaning of a system is expressed in the mathematical theory of domains. Proponents of such methods rely on the well-understood nature of domains to give meaning to the system; critics point out that not every system may be intuitively or naturally viewed as a function.
- Operational semantics, in which the meaning of a system is expressed as a sequence of actions of a (presumably) simpler computational model. Proponents of such methods point to the simplicity of their models as a means to expressive clarity; critics counter that the problem of semantics has just been delayed (who defines the semantics of the simpler model?).
- Axiomatic semantics, in which the meaning of the system is expressed in terms of preconditions and postconditions which are true before and after the system performs a task, respectively. Proponents note the connection to classical logic; critics note that such semantics never really describe what a system *does* (merely what is true before and afterwards).

Lightweight formal methods

Some practitioners believe that the formal methods community has overemphasized full formalization of a specification or design. They contend that the expressiveness of the languages involved, as well as the complexity of the systems being modelled, make full formalization a difficult and expensive task. As an alternative, various *lightweight* formal methods, which emphasize partial specification and focused application, have been proposed. Examples of this lightweight approach to formal methods include the Alloy object modelling notation, Denney's synthesis of some aspects of the Z notation with use case driven development, and the CSK VDM Tools.

Uses

Formal methods can be applied at various points through the development process.

Specification

Formal methods may be used to give a description of the system to be developed, at whatever level(s) of detail desired. This formal description can be used to guide further development activities, additionally, it can be used to verify that the requirements for the system being developed have been completely and accurately specified.

The need for formal specification systems has been noted for years. In the ALGOL 58 report, John Backus presented a formal notation for describing programming language syntax (later named Backus Normal Form then renamed Backus-Naur Form (BNF)). Backus also wrote that a formal description of the meaning of syntactically-valid ALGOL programs wasn't completed in time for inclusion in the report. "Therefor the formal treatment of the semantics of legal programs will be included in a subsequent paper." It never appeared.

Development

Once a formal specification has been produced, the specification may be used as a guide while the concrete system is developed during the design process (i.e., realized typically in software, but also potentially in hardware). For example:

- If the formal specification is in an operational semantics, the observed behavior of the concrete system can be compared with the behavior of the specification (which itself should be executable or simulateable). Additionally, the operational commands of the specification may be amenable to direct translation into executable code.
- If the formal specification is in an axiomatic semantics, the preconditions and postconditions of the specification may become assertions in the executable code.

Verification

Once a formal specification has been developed, the specification may be used as the basis for proving properties of the specification (and hopefully by inference the developed system).

Human-directed proof

Sometimes, the motivation for proving the correctness of a system is not the obvious need for re-assurance of the correctness of the system, but a desire to understand the system better. Consequently, some proofs of correctness are produced in the style of mathematical proof: handwritten (or typeset) using natural language, using a level of informality common to such proofs. A "good" proof is one which is readable and understandable by other human readers.

Critics of such approaches point out that the ambiguity inherent in natural language allows errors to be undetected in such proofs; often, subtle errors can be present in the low-level details typically overlooked by such proofs. Additionally, the work involved in producing such a good proof requires a high level of mathematical sophistication and expertise.

Automated proof

In contrast, there is increasing interest in producing proofs of correctness of such systems by automated means. Automated techniques fall into two general categories:

- Automated theorem proving, in which a system attempts to produce a formal proof from scratch, given a description of the system, a set of logical axioms, and a set of inference rules.
- Model checking, in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution.

Neither of these techniques work without human assistance. Automated theorem provers usually require guidance as to which properties are "interesting" enough to pursue; model checkers can quickly get bogged down in checking millions of uninteresting states if not given a sufficiently abstract model.

Proponents of such systems argue that the results have greater mathematical certainty than human-produced proofs, since all the tedious details have been algorithmically verified. The training required to use such systems is also less than that required to produce good mathematical proofs by hand, making the techniques accessible to a wider variety of practitioners.

Critics note that some of those systems are like oracles: they make a pronouncement of truth, yet give no explanation of that truth. There is also the problem of "verifying the

verifier"; if the program which aids in the verification is itself unproven, there may be reason to doubt the soundness of the produced results. Some modern model checking tools produce a "proof log" detailing each step in their proof, making it possible to perform, given suitable tools, independent verification.

Criticisms

The field of formal methods has its critics. Handwritten proofs of correctness need significant time (and thus money) to produce, with limited utility other than assuring correctness. This makes formal methods more likely to be used in fields where it is possible to perform automated proofs using software, or in cases where the cost of a fault is high. Example: in railway engineering and aerospace engineering, undetected errors may cause death, so formal methods are more popular in this field than in other application areas.

Formal methods and notations

There are a variety of formal methods and notations available.

Specification languages

- Abstract State Machines (ASMs)
- ANSI/ISO C Specification Language (ACSL)
- Alloy
- B-Method
- CADP
- Common Algebraic Specification Language (CASL)
- Process calculi
    - CSP
    - LOTOS
    - π-calculus
- Actor model
- Esterel
- Lustre
- mCRL2
- Petri nets
- RAISE
- Specification and Description Language
- Temporal logic of actions (TLA)
- USL
- VDM
    - VDM-SL
    - VDM++
- Z notation
- Rebeca Modeling Language

Model checkers

- SPIN
- PAT is a powerful free model checker, simulator and refinement checker for concurrent systems and CSP extensions (e.g. shared variables, arrays, fairness).

Chapter 4

Best Coding Practices and Big Design Up Front

Best Coding Practices

Best coding practices for software development can be broken into many levels based on the coding language, the platform, the target environment and so forth. Using best practices for a given situation greatly reduces the probability of introducing errors into your applications, regardless of which software development model is being used to create that application.

There are standards that originated from the intensive study of industry experts who analyzed how bugs were generated when code was written and correlated these bugs to specific coding practices. They took these correlations between bugs and coding practices and came up with a set of rules that when used prevented coding errors from occurring. These standard practices offer incredible value to software development organizations because they are pre-packaged automated error prevention practices; they close the feedback loop between a bug and what must be done to prevent that bug from recurring.

In a team environment or group collaboration, best coding practices ensure the use of standards and uniform coding, reducing oversight errors and the time spent in code review. When work is outsourced to a third-party contractor, having a set of these best practices in place gives you the knowledge that the code produced by the contractor meets all the guidelines mandated by the client company.

It should be understood that these practices are not just a way to enforce naming conventions in your code.

Best coding practices gives you a way to analyze your source code so that certain rules and patterns can be detected automatically and that the knowledge obtained through previous years of experience by industry experts is implemented in an appropriate way.

With the foregoing in mind, here is a some base step of what is needed for a project that successfully utilizes 'Best Coding Practices':

Lifecycle

It is important to choose the appropriate development lifecycle for a given project because all other activities are derived from this process. A couple examples of this are the Rational Unified Process (RUP) and the eXtreme Programming (XP) methods. Having a well defined process is usually better than having none at all, and in many cases it is less important what process is used than how well it is executed. The methodologies above are very common and a quick Web search will turn up all kinds of information regarding how to implement them.

Requirements

Everyone needs to be on the same page before jumping into programming. This is a fundamental truth to almost any endeavor and even more so when accomplishing a group driven programming task. If you are programming alone, you may find yourself adding or tweaking your application. When you are looking at an enterprise piece of software, everyone involved in the project needs to understand clearly the requirements and goals of the project before moving forward. This is often referred to as Functional and Detailed Specifications.

Architecture

Choosing the appropriate architecture for your application is key. You have to know what you are building on before you can start a project. Check the architecture of the target. Read as much as you can about the ins and outs of the platform and note any pitfalls before you start your code. It will go a long way to heading off any bugs that might be 'show stoppers' later on.

Design

Even if you feel great about knowing the architecture of your target platform without a good design you are going to be sunk. Try not to fall into the trap though of over-designing the application. The two basic principles are to "Keep it Simple" and to utilize information hiding (Don't show the user more than they need to see). Often this is where object-oriented analysis and UML come in. Do an internet search on UML and you'll find dozens of articles on using it.

Code Building

Building the code is really just a small part of the total project effort even though it's what most people equate with the whole process since it's the most visible. Other pieces equally or even more important include what we have already gone over above namely requirements, architecture, analysis, design, and testing. A best practice for building code involves daily builds and testing.

Best coding evolves from following proper coding standards and guidelines. Appropriate Comments for each and every line of code makes code maintainability much easier. A best code should have reusable components.

Testing

Testing is an integral part of software development that needs to be planned. It is also important that testing is done proactively; meaning that test cases are planned before coding starts, and test cases are developed while the application is being designed and coded.

Deployment

Deployment is the final stage of releasing an application for users.

Big Design Up Front

Big Design Up Front (BDUF) is a term for any software development approach in which the program's design is to be completed and perfected before that program's implementation is started. It is often associated with the waterfall model of software development.

Arguments for Big Design Up Front

Proponents of BDUF argue that time spent in designing is a worthwhile investment, and reference numerous studies which have concluded that less time and effort is spent fixing a bug in the early stages of a software products lifecycle than when that same bug is found and must be fixed later. That is, it is much easier to fix a requirements bug in the requirements phase than to fix that same bug in the implementation phase, as to fix a requirements bug in the implementation phase requires scrapping at least some implementation and design work which has already been completed.

Joel Spolsky, a popular online commentator on software development, has argued strongly in favor of Big Design Up Front:

"Many times, thinking things out in advance saved us serious development headaches later on. ... [on making a particular specification change] ... Making this change in the spec took an hour or two. If we had made this change in code, it would have added weeks to the schedule. I can't tell you how strongly I believe in Big Design Up Front, which the proponents of Extreme Programming consider anathema. I have consistently saved time and made better products by using BDUF and I'm proud to use it, no matter what the XP fanatics claim. They're just wrong on this point and I can't be any clearer than that."

However, some argue that what Joel has called Big Design Up Front doesn't resemble the BDUF criticized by advocates of XP and other agile software development methodologies.

Arguments against Big Design Up Front

Critics (notably those who practice agile software development) argue that BDUF is poorly adaptable to changing requirements and that BDUF assumes that designers are able to foresee problem areas without extensive prototyping and at least some investment into implementation.

They also assert that there is an overhead to be balanced between the time spent planning and the time that fixing a defect would actually cost. This is sometimes termed analysis paralysis.

If the *cost of planning* is greater than the *cost of fixing* then time spent planning is wasted.

Continuous Deployment, Automatic Updates, Fault Tolerance, Lisp's Read-eval-print loop and related ideas seek to substantially reduce the cost of defects in production so that they become cheaper to fix at run-time than to plan out at the beginning.

Also, in most projects there is a significant lack of comprehensive written (or even well known) requirements. So in BDUF a lot of assumptions are made that later prove to be false but are designed and possibly already coded.

Chapter 5

CCU Delivery

Customer Configuration Updating (CCU) is a software development method for structuring the process of providing customers with new versions of products and updates production. This method is developed by researchers of the Utrecht University.

Introduction to the delivery process

As described in the general entry of CCU, the delivery phase is the second phase of the CCU method. In figure one the CCU method is depicted. The phases of CCU that are not covered here are concealed by a transparent grey rectangle.

As can be seen in figure one, the delivery phase is in between the release phase and the deployment phase. A software vendor develops and releases a software product and afterwards it has to be transported to the customer. This phase is the delivery process. This process is highly complex because the vendor often has to deal with a product which has multiple versions, variable features, dependency on external products, and different kinds of distribution options. The CCU method helps the software vendor in structuring this process.

In figure 2, the process-data diagram of the delivery phase within CCU is depicted. This way of modeling was invented by Saeki (2003). On the left side you can see the meta-process model and on the right side the meta-data model. The two models are linked to each other by the relationships visualized as doted lines. The meta-data model (right side) shows the concepts involved in the process and how the concepts are related to each other. For instance it is visible that a package consists of multiple parts, being the: software package, system description, manual, and license and management information. The numbers between the relations indicate in what quantity the concepts are related. For example the “1..1” between package and software package means that a package has to contain at least 1 software package and at the most 1 software package. So in this case a package just has to contain 1 software package. On the left side of the picture the process-data model is depicted. This consists of all the activities within the delivery process. This is based on this process-data model. The meta-process model (left side of the process-data diagram) is divided into several parts which are presented along with the corresponding paragraphs throughout the article to make it easier to understand.

The tables that describe the concepts of the meta-data model and the activities of the process-data model are presented beneath figure 2.

Table of concepts

The table of concepts contains all concepts used in the meta-data model with their explanations along with the source from which the explanations are derived.

Table 1: Table of concepts

Concept	Definition (Source)
REPOSITORY	Also called a vault. A repository contains only one complete version of a configuration item (CI). Differences between versions are usually stored using a delta algorithm.
package	Collection of records describing resources A collection of different related items combined for transferring purposes to the customer.
SOFTWARE PACKAGE	A collection of different related software components combined for transferring purposes to the customer.
SOFTWARE COMPONENTS	The different components of which software consists related through dependencies.
VERSION	A version is a state of an object or concept that varies from its previous state or condition.
SYSTEM DESCRIPTION	Description of the system including its requirements and the dependencies on other external components.
manual	A technical communication document intended to give assistance to people using a particular system.
LICENSE	Type of proprietary or gratuitous license as well as a memorandum of contract between a producer and a user of computer software that specifies the perimeters of the permission granted by the owner to the user.
MANAGEMENT INFORMATION	All the information that is relevant for the management of the system at the customer site.
CUSTOMER RELATIONSHIP MANAGEMENT SYSTEM	A system which maintains all information about the customers.
CUSTOMER	A company or person that bought some product or made use of one of your companies services before.

LICENSE TYPE	In this case it can either be a long term license, an expired license or a temporary license.
CUSTOMER DATA	All known information about the customers in the customer relationship management system.
CONFIGURATION MANAGEMENT SYSTEM	A system maintaining information about the software configurations at the customer sites.
PRODUCT	An element of software or a document placed under version control.
UPDATES	An update also called a patch is a small piece of software designed to update or fix problems with a computer program
CONFIGURATION	A configuration is an arrangement of functional units according to their nature, number, and chief characteristics.
MODIFICATION	Modification is the act of applying change to an original.
FEEDBACK	Feedback allows a vendor to gather large amounts of data about its customers and its product as it acts in the field
BUG REPORT	A report of the problems users encountered using the product. This can mean a problem with a certain function or dead links within the system. This information is collected manually.
PRODUCT USAGE DATA	This data contains information about the actual product usage. This reflects on options that are most used within the program.
ERROR REPORT	When the software product gets an error it will automatically send an error report to the vendor.
USAGE QUESTIONS	Questions users have about handling the product etc.

Activity table

The activity table contains the explanations of the activities along with the source from which the explanations are derived. Because the method is quite innovative a lot of the activity's are designed especially for this model and therefore the explanations do not have a source.

Table 2: Activity Table

Activity	Sub-Activity	Description (Source)
package		Packaging the system so that it can be transferred to the customers' site.
	package software	Combining different software components into one package that can be delivered to the customer.
	package system description	Add a system description to the package.
	package manual	Add a manual to the package.
	package license	Add a license to the package.
	package management information	Add a management information document to the package.
	Check package	Make sure that the package is complete and ready for deployment at the customer site.
Advertise update		When a vendor wishes to provide updates to its customers, the customers first need to be informed through the available communication channels.
Prepare distribution		Prepare measures be able to get the software to the customer.
	Set package in repository	A finished software component will be made available in some release repository.
	Create transfer channels	The vendor needs to create channels through which the software can be transferred to the customer.
Distribute		Getting the software to the different customers.
	customer request	A customer makes the vendor aware of his interest for a certain product or update.
	Determine configuration needs	It is determined which software components are needed for a successful configuration update.
	Determine configuration constraints	It is determined to which constraints the infrastructure of the customer has to suffice in order to run the new product or update.
	Check customers license	It is checked if the customer is in possession of the right license for the new configuration update.
	Deliver update	Getting the software components at the customer site.
Inform customer		Providing the customer with information about in this case the status of its request.
Update CRM		Add information to the CRM system so that it contains the most current information available.
Receive delivery and		Getting a report (automatically or manually) about

deployment report	the success of the delivery and deployment from the customer.
Update license type	Add information about the license obtained by the customer so that the system contains the most current information available.
Update configuration management	Add recent information in the configuration management system, so that the customer most recent configuration is stored.
Update product properties	Renew the information about the products in use by the customer so that the system contains the most current information available.

Package software

In order to deliver the developed product to the customer, the vendor needs to package the different components of its product into a package. By doing this, the customer will receive all the information and software components at once fulfilling al its needs. After combining all elements into one package the software vendor will carefully have to check if the package is complete. The package will have to provide the customer with all the tools and information to use the product. When this is not the case the software vendor will get a lot of questions from its customers which will consume a lot of time. It is therefore very important that the package is checked carefully before it is shipped. The package can be a physical combination of different elements packed into for example a box, but it can also be a digital combination of files which contain all the elements. Within the CCU process it is stated that a package will consist of five elements, being: software package, system description, manual, and license and management information. In the following paragraphs is explained how these elements fit into the CCU delivery phase.

Software package

One of the elements of the package will be the software package. The software package is a package in itself, because it consists of the different software components that together form the product. In contrast with the overall package, the software package is always a technical package in which all the files needed are combined in order to run the software product. Another concept of the software package is the version. This keeps track of the modifications made to the software product. By relating it to the software package the vendor and the customer are able to keep track of the functionality and properties of the product the customer is using.

System description

It is a general description of what the product and its functionalities. In addition it will also describe of what components, the product consists and how these are related to other product software already in place. In case of a software update it will for example

describe how the previous version of the software is modified by this product. Besides this, it will also describe the requirements needed to run the software product properly. For example what other products and configurations need to be in place in order to let this product run properly.

Manual

The manual is the document that will provide the customer with guidance in deploying and using the product.

License

The license is in this case a Software license agreement in which is stated how the customer is permitted to use the product. For example it can state how many users are permitted to use the software product. In this situation the license agreement is a contract or a certificate which is the customers prove of its using permits. The software vendor has its own part of the agreement which in most cases is stored in a system. An elaboration of this part can be found at the receive feedback section. The license agreement shipped to the customer can be a digital document as well as a physical document.

Management information

This piece of information should contain the information that is relevant for managing the system at the customer site. In many cases this information is already part of the manual. However in particular situations this information is meant only for the management of the system and not for the users of the system and is therefore supplied as a separate document.

Distribution

After the package is assembled it needs to be distributed to the customers. This section within the delivery process is about the actual delivery of the package to the customers.

Offline vs Online

The software distribution of a product can be done offline as well as online. In an offline situation the package is a physical package which contains all the elements. The software is stored on a data carrier such as a CD or a DVD, and the documents might also be stored in a digital form on this data carrier, or they might be in physical form such as a booklet. The package as a whole is a physical product. In an online situation the entire package needs to be in a digital form. The consequences on the distribution process are described in the following paragraphs. CCU is designed to fit both situations but as bandwidth is growing it is making more sense to distribute especially updates and new versions to existing customers online. Here both ways are discussed. In the process-data model it is assumed that the software vendor conducts both distribution channels. As a practical example: HISComp, a provider of medical information systems distributes its

software straightforward via CDs. However they use their website to distribute patches for the software products.

Preparation of distribution

After a new package is assembled, the customer needs to be made aware of the new release. In the process-data model this is being depicted as a loop which states advertising the update until the customers are being properly informed. Besides this, the package ready for delivery, needs to be stored in a repository for the online distribution. In addition the vendor needs to create transfer channels. For the online distribution this means that the vendor needs to create online channels to its repository. In most cases this means that a link to the product on the website of the vendor is created. In case of updates it is largely applicable that the current version of the software product at the customer site automatically checks the repository for new updates of the product. In case of offline distribution, the vendor needs to create physical transfer channels. This can be shops or just a contract with a courier company.

The actual distribution

The distribution begins with the request for a product by the customer. This can be done automatically when the current product of the customer searches for an update at the online repository. The customer can also manually do a request for a product via the website of the vendor. A third option is that the customer does the request via telephone or e-mail.

When the vendor is aware of the customer request it will determine the customer needs. By checking what the customer current configuration is and what the customer desires. This process can also take place automatically by checking the customer configuration in the configuration management system. More information on this system is provided in the next chapter. When it is clear what product the customer needs and the possible modifications to this product it is necessary to determine if the customer current configuration suits the new product. The current configuration is compared to the constraints of the new product. This can also be done automatically by the configuration management system. When the configuration of the customer appears to be insufficient the customer is informed about this. For example the vendor can make clear to the customer that it will need an external product for this new product to run properly. Besides this the Customer Relationship Management (CRM) system of the vendor is updated. There is more information about this in the chapter about CRM.

When the customer configuration is sufficient the vendor will check the current license of the customer. If the customer does not have a proper license for the requested product the license needs to be obtained. The customer will be informed about this and the CRM system will be updated again. If the customer has the proper license or wants to buy the proper license along with the product, the product is delivered to the customer.

Software configuration management

The Software Configuration Management system, is a system at the vendor's site which keeps track of the configurations at the customer site. By storing this in a system the vendor will be able to give the customer particular service when it needs a new product. In the software configuration management system information about the products used by the customer, the version of these products, as well as which updates are already being done, is stored. In some cases it is possible that the vendor did some modifications to the product particularly for this customer. This will also have to be stored in the system. Also there needs to be configuration data, some generic information about the configuration the customer is using. For example what operating platform the customer uses for its software. What also should be stored in this system is information about the feedback that the vendor gets from the customer. This includes bug reports, product usage data, error reports and usage questions. More information about this feedback can be found in the CCU phase activation and usage.

By storing all this information the vendor can determine the customer needs very precisely whenever a customer requests a product or an update. As already stated the vendor can also easily inform the customer about some adaptations the customer needs to make to its configuration in order to let the product function properly. Another advantage of storing this information in a system is that it will ease the process of online delivery. The checking of the configuration needs and constraints can all be done automatically when a customer does a request.

CRM system

The customer relationship management system contains all kinds of data about the customers of a company. Here we will discuss the function of this customer data in the CCU delivery process. Information about the license agreement between the customer and the software vendor is stored in the CRM system. In the meta-data model this repository and online distribution is linked to the CRM system this can again be done automatically. The system will check if the license of a customer is sufficient to obtain a certain product or update.

Receiving feedback and updating the systems

In order to keep all the described systems up-to-date at the vendor site it is important that the vendor receives a lot of

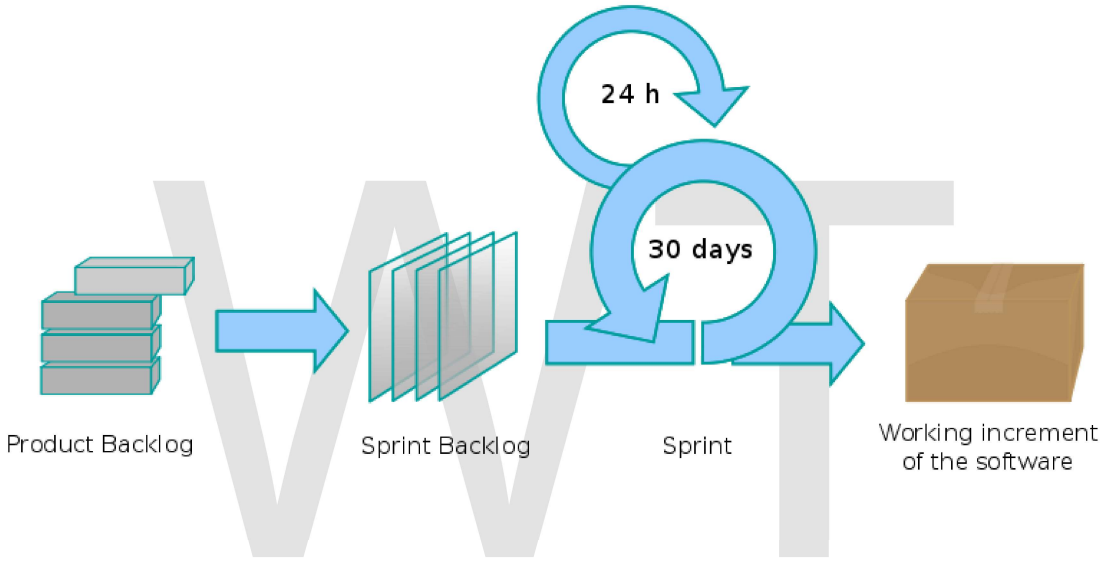
Example

An example of a successful application of the CCU method can be found at Exact Software (ES). ES is a manufacturer of accounting and enterprise resource planning software based in the Netherlands. ES has combined Product Data Management (PDM), Customer Relationship Management (CRM) and Software Configuration Management (SCM) in order to maintain the configuration at the customer site in a better and less

complex way. ES has a module in its CRM software that contains all contracts of each customer. This is linked to their PDM system. Every contract corresponds to files that can be downloaded for a new version or update of a previous version. In the delivery phase this means that the customers are able to obtain all the products through an online connection. So ES sells contracts (licenses) and stores them into their CRM system, the delivery of the actual products can be done by the customers themselves completely automated requiring little effort. The PDM system is on its turn linked to the SCM system which keeps track of the configurations the customers are using. In the delivery phase this means that ES is able to automatically determine the customer needs whenever a customer does a request.

Chapter 6

Scrum (Development)



The Scrum process

Scrum is an iterative, incremental methodology for project management often seen in agile software development, a type of software engineering.

Although the Scrum approach was originally suggested for managing product development projects, its use has focused on the management of software development projects, and it can be used to run software maintenance teams or as a general project/program management approach.

History

In 1986, Hirotaka Takeuchi and Ikujiro Nonaka described a new approach to commercial product development that would increase speed and flexibility, based on case studies from manufacturing firms in the automotive, computer, photocopier, and printer industries. They called this the *holistic or rugby approach*, as the whole process is performed by one cross-functional team across multiple overlapping phases, where the *scrum* (or whole team) "tries to go the distance as a unit, passing the ball back and forth".

In 1991, DeGrace and Stahl first referred to this as the *scrum approach*. In the early 1990s, Ken Schwaber used such an approach at his company, Advanced Development Methods, and Jeff Sutherland, with John Scumniotales and Jeff McKenna, developed a similar approach at Easel Corporation, and were the first to refer to it using the single word *Scrum*.

In 1995, Sutherland and Schwaber jointly presented a paper describing the *Scrum methodology* at the Business Object Design and Implementation Workshop held as part of OOPSLA '95 in Austin, Texas, its first public presentation. Schwaber and Sutherland collaborated during the following years to merge the above writings, their experiences, and industry best practices into what is now known as Scrum.

In 2001, Schwaber teamed up with Mike Beedle to describe the method in the book "Agile Software Development with Scrum".

Although the word is not an acronym, some companies implementing the process have been known to spell it with capital letters as SCRUM. This may be due to one of Ken Schwaber's early papers, which capitalized SCRUM in the title.

Characteristics

Scrum is a process skeleton that contains sets of practices and predefined roles. The main roles in Scrum are:

1. the "**ScrumMaster**", who maintains the processes (typically in lieu of a project manager)
2. the "**Product Owner**", who represents the stakeholders and the business
3. the "**Team**", a cross-functional group of about 7 people who do the actual analysis, design, implementation, testing, etc.

During each "sprint", typically a two to four week period (with the length being decided by the team), the team creates a potentially shippable product increment (for example, working and tested software). The set of features that go into a sprint come from the product "backlog", which is a prioritized set of high level requirements of work to be done. Which backlog items go into the sprint is determined during the sprint planning meeting. During this meeting, the Product Owner informs the team of the items in the product backlog that he or she wants completed. The team then determines how much of this they can commit to complete during the next sprint, and records this in the sprint backlog. During a sprint, no one is allowed to change the sprint backlog, which means that the requirements are frozen for that sprint. Development is timeboxed such that the sprint must end on time; if requirements are not completed for any reason they are left out and returned to the product backlog. After a sprint is completed, the team demonstrates how to use the software.

Scrum enables the creation of self-organizing teams by encouraging co-location of all team members, and verbal communication between all team members and disciplines in the project.

A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need (often called requirements churn), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements.

Like other agile development methodologies, Scrum can be implemented through a wide range of tools. Many companies use universal software tools, such as spreadsheets to build and maintain artifacts such as the sprint backlog. There are also open-source and proprietary software packages dedicated to management of products under the Scrum process. Other organizations implement Scrum without the use of any software tools, and maintain their artifacts in hard-copy forms such as paper, whiteboards, and sticky notes.

Roles

Scrum teams consist of three core roles and a range of ancillary roles—core roles are often referred to as *pigs* and ancillary roles as *chickens* after the story The Chicken and the Pig.

Core Scrum roles

The core roles in Scrum teams are those committed to the project in the Scrum process—they are the ones producing the product (objective of the project).

Product Owner
: The Product Owner represents the voice of the customer and is accountable for ensuring that the Team delivers value to the business. The Product Owner writes customer-centric items (typically user stories), prioritizes them, and adds them to the product backlog. Scrum teams should have one Product Owner, and while they may also be a member of the Development Team, it is recommended that this role not be combined with that of ScrumMaster.

Team
: The Team is responsible for delivering the product. A Team is typically made up of 5–9 people with cross-functional skills who do the actual work (analyse, design, develop, test, technical communication, document, etc.). It is recommended that the Team be self-organizing and self-led, but often work with some form of project or team management.

ScrumMaster
: Scrum is facilitated by a ScrumMaster, also written as *Scrum Master*, who is accountable for removing impediments to the ability of the team to deliver the sprint goal/deliverables. The ScrumMaster is not the team leader but acts as a

buffer between the team and any distracting influences. The ScrumMaster ensures that the Scrum process is used as intended. The ScrumMaster is the enforcer of rules. A key part of the ScrumMaster's role is to protect the team and keep them focused on the tasks at hand. The role has also been referred to as *servant-leader* to reinforce these dual perspectives.

Ancillary Scrum roles

The ancillary roles in Scrum teams are those with no formal role and infrequent involvement in the Scrum process—and must nonetheless be taken into account.

Stakeholders (customers, vendors)
: These are the people who enable the project and for whom the project will produce the agreed-upon benefit[s], which justify its production. They are only directly involved in the process during the sprint reviews.

Managers (including Project Managers)
: People who will set up the environment for product development.

Agile Project Management with Scrum

Scrum has not only reinforced the interest in software project management, but also challenged the conventional ideas about such management. Scrum focuses on project management institutions where it is difficult to plan ahead with mechanisms for *empirical process control*, such as where feedback loops constitute the core element of product development compared to traditional command-and-control-oriented management. It represents a radically new approach for planning and managing software projects, bringing decision-making authority to the level of operation properties and certainties. Scrum reduces defects and makes the development process more efficient, as well as reducing long-term maintenance costs.

Meetings

Daily Scrum
: Each day during the sprint, a project status meeting occurs. This is called a *daily scrum*, or *the daily standup*. This meeting has specific guidelines:

- The meeting starts precisely on time.
- All are welcome, but normally only the core roles speak
- The meeting is timeboxed to 15 minutes
- The meeting should happen at the same location and same time every day

During the meeting, each team member answers three questions:

- What have you done since yesterday?
- What are you planning to do today?
- Do you have any problems that would prevent you from accomplishing your goal? (It is the role of the ScrumMaster to facilitate resolution of

these impediments, although the resolution should occur outside the Daily Scrum itself to keep it under 15 minutes.)

Sprint Planning Meeting

At the beginning of the sprint cycle (every 7–30 days), a "Sprint Planning Meeting" is held.

- Select what work is to be done
- Prepare the Sprint Backlog that details the time it will take to do that work, with the entire team
- Identify and communicate how much of the work is likely to be done during the current sprint
- Eight hour time limit
  - (1st four hours) Product Owner + Team: dialog for prioritizing the Product Backlog
  - (2nd four hours) Team only: hashing out a plan for the Sprint, resulting in the Sprint Backlog

At the end of a sprint cycle, two meetings are held: the "Sprint Review Meeting" and the "Sprint Retrospective"

Sprint Review Meeting

- Review the work that was completed and not completed
- Present the completed work to the stakeholders (a.k.a. "the demo")
- Incomplete work cannot be demonstrated
- Four hour time limit

Sprint Retrospective

- All team members reflect on the past sprint
- Make continuous process improvements
- Two main questions are asked in the sprint retrospective: What went well during the sprint? What could be improved in the next sprint?
- Three hour time limit

Artifacts

Product backlog

The **product backlog** is a high-level list that is maintained throughout the entire project. It aggregates backlog items: broad descriptions of all potential features, prioritized as an absolute ordering by business value. It is therefore the "What" that will be built, sorted by importance. It is open and editable by anyone and contains rough estimates of both business value and development effort. Those estimates help the Product Owner to gauge the timeline and, to a limited extent prioritize. For example, if the "add spellcheck" and

“add table support” features have the same business value, the one with the smallest development effort will probably have higher priority, because the ROI (Return on Investment) is higher.

The Product Backlog, and business value of each listed item is the property of the product owner. The associated development effort is however set by the Team.

Sprint backlog

The **sprint backlog** is the list of work the team must address during the next sprint. Features are broken down into tasks, which, as a best practice, should normally be between four and sixteen hours of work. With this level of detail the whole team understands exactly what to do, and potentially, anyone can pick a task from the list. Tasks on the sprint backlog are never assigned; rather, tasks are signed up for by the team members as needed, according to the set priority and the team member skills. This promotes self-organization of the team, and developer buy-in.

The sprint backlog is the property of the team, and all included estimates are provided by the Team. Often an accompanying **task board** is used to see and change the state of the tasks of the current sprint, like “to do”, “in progress” and “done”.

Burn down

The sprint burn down chart is a publicly displayed chart showing remaining work in the sprint backlog. Updated every day, it gives a simple view of the sprint progress. It also provides quick visualizations for reference. There are also other types of burndown, for example the **release burndown chart** that shows the amount of work left to complete the target commitment for a Product Release (normally spanning through multiple iterations) and the **alternative release burndown chart**, which basically does the same, but clearly shows scope changes to Release Content, by resetting the baseline.

It should not be confused with an earned value chart.

Adaptive project management

The following are some general practices of Scrum:

- "Working more hours" does not necessarily mean "producing more output"
- "A happy team makes a tough task look simple"

Terminology

The following terminology is used in Scrum:

Roles

Scrum Team
: Product Owner, ScrumMaster and Team

Product Owner
: The person responsible for maintaining the Product Backlog by representing the interests of the stakeholders.

ScrumMaster
: The person responsible for the Scrum process, making sure it is used correctly and maximizing its benefits.

Team
: A cross-functional group of people responsible for managing itself to develop the product.

Artifacts

Sprint burn down chart
: Daily progress for a Sprint over the sprint's length.

Product backlog
: A prioritized list of high level requirements.

Sprint backlog
: A prioritized list of tasks to be completed during the sprint.

Others

Impediment
: Anything that prevents a team member from performing work as efficiently as possible.

Sprint
: A time period (typically 2–4 weeks) in which development occurs on a set of backlog items that the Team has committed to. Also commonly referred to as a Time-box.

Sashimi
: A report that something is "done". The definition of "done" may vary from one Scrum Team to another, but must be consistent within one team.

Abnormal Termination
: The Product Owner can cancel a Sprint if necessary. The Product Owner may do so with input from the team, scrum master or management. For instance, management may wish to cancel a sprint if external circumstances negate the value of the sprint goal. If a sprint is abnormally terminated, the next step is to conduct a new Sprint planning meeting, where the reason for the termination is reviewed.

Planning Poker
: In the Sprint Planning Meeting, the team sits down to estimate its effort for the stories in the backlog. The Product Owner needs these estimates, so that he or she is empowered to effectively prioritize items in the backlog and, as a result, forecast releases based on the team's velocity.

Point Scale
: Relates to an abstract point system, used to discuss the difficulty of the task, without assigning actual hours. Common systems of scale are linear (1,2,3,4...), Fibonacci (1,2,3,5,8...), Powers-of-2 (1,2,4,8...), and Clothes size (XS, S, M, L, XL).

Definition of Done (DoD)
: The exit-criteria to determine whether a product backlog item is complete. In many cases the DoD requires that all regression tests should be successful.

Scrum modifications

Scrum-ban

Scrum-ban is a software production model based on Scrum and Kanban. Scrum-ban is especially suited for maintenance projects or (system) projects with frequent and unexpected user stories or programming errors. In such cases the time-limited sprints of the Scrum model are of no appreciable use, but Scrum's daily meetings and other practices can be applied, depending on the team and the situation at hand. Visualization of the work stages and limitations for simultaneous unfinished user stories and defects are familiar from the Kanban model. Using these methods, the team's workflow is directed in a way that allows for minimum completion time for each user story or programming error, and on the other hand ensures each team member is constantly employed.

To illustrate each stage of work, teams working in the same space often use post-it notes or a large whiteboard. In the case of decentralized teams, stage-illustration software, such as Assembla, ScrumWorks, or JIRA in combination with GreenHopper can be used to visualize each team's user stories, defects and tasks divided into separate phases.

In their simplest, the tasks or usage stories are categorized into the work stages

- Unstarted
- Ongoing
- Completed

If desired, though, the teams can add more stages of work (such as "defined", "designed", "tested" or "delivered"). These additional phases can be of assistance if a certain part of the work becomes a bottleneck and the limiting values of the unfinished work cannot be raised. A more specific task division also makes it possible for employees to specialize in a certain phase of work.

There are no set limiting values for unfinished work. Instead, each team has to define them individually by trial and error; a value too small results in workers standing idle for lack of work, whereas values too high tend to accumulate large amounts of unfinished work, which in turn hinders completion times. A rule of thumb worth bearing in mind is that no team member should have more than two simultaneous selected tasks, and that on the other hand not all team members should have two tasks simultaneously.

The major differences between Scrum and Kanban are derived from the fact that, in Scrum work is divided into sprints that last a certain amount of time, whereas in Kanban the workflow is continuous. This is visible in work stage tables, which in Scrum are emptied after each sprint. In Kanban all tasks are marked on the same table. Scrum focuses on teams with multifaceted know-how, whereas Kanban makes specialized, functional teams possible.

Since Scrum-ban is such a new development model, there is not much reference material. Kanban, on the other hand, has been applied in software development at least by Microsoft and Corbis.

Product development

Scrum as applied to product development was first referred to in "New New Product Development Game" (Harvard Business Review 86116:137–146, 1986) and later elaborated in "The Knowledge Creating Company" both by Ikujiro Nonaka and Hirotaka Takeuchi (Oxford University Press, 1995). Today there are records of Scrum used to produce financial products, Internet products, and medical products by ADM.

Chapter 7

Software Architecture

The **software architecture** of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. The term also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects.

Overview

The field of computer science has come across problems associated with complexity since its formation. Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns. Although the term “software architecture” is relatively new to the industry, the fundamental principles of the field have been applied sporadically by software engineering pioneers since the mid 1980s. Early attempts to capture and explain software architecture of a system were imprecise and disorganized, often characterized by a set of box-and-line diagrams. During the 1990s there was a concentrated effort to define and codify fundamental aspects of the discipline. Initial sets of design patterns, styles, best practices, description languages, and formal logic were developed during that time.

The software architecture discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. To date there is still no agreement on the precise definition of the term “software architecture”.

As a maturing discipline with no clear rules on the right way to build a system, designing software architecture is still a mix of art and science. The “art” aspect of software architecture is because a commercial software system supports some aspect of a business or a mission. How a system supports key business drivers is described via scenarios as non-functional requirements of a system, also known as quality attributes, determine how a system will behave. Every system is unique due to the nature of the business drivers it supports, as such the degree of quality attributes exhibited by a system such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and such other –ilities will vary with each implementation. To bring a software architecture user's perspective into the software architecture, it can be said that software architecture gives the direction to take steps and do the tasks involved in each

such user's speciality area and interest e.g. the stakeholders of software systems, the software developer, the software system operational support group, the software maintenance specialists, the deployer, the tester and also the business end user. In this sense software architecture is really the amalgamation of the multiple perspectives a system always embodies. The fact that those several different perspectives can be put together into a software architecture stands as the vindication of the need and justification of creation of software architecture before the software development in a project attains maturity.

History

The origin of software architecture as a concept was first identified in the research work of Edsger Dijkstra in 1968 and David Parnas in the early 1970s. These scientists emphasized that the structure of a software system matters and getting the structure right is critical. The study of the field increased in popularity since the early 1990s with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods.

Research institutions have played a prominent role in furthering software architecture as a discipline. Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled *Software Architecture: Perspectives on an Emerging Discipline* in 1996, which brought forward the concepts in Software Architecture, such as components, connectors, styles and so on. The University of California, Irvine's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

The IEEE 1471: ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems is the first formal standard in the area of software architecture, and was adopted in 2007 by ISO as *ISO/IEC 42010:2007*.

Software architecture topics

Architecture description languages

Architecture description languages (**ADLs**) are used to describe a Software Architecture. Several different ADLs have been developed by different organizations, including AADL (SAE standard), Wright (developed by Carnegie Mellon), Acme (developed by Carnegie Mellon), xADL (developed by UCI), Darwin (developed by Imperial College London), DAOP-ADL (developed by University of Málaga), and ByADL (University of L'Aquila, Italy). Common elements of an ADL are component, connector and configuration.

Views

Software architecture is commonly organized in views, which are analogous to the different types of blueprints made in building architecture. A view is a representation of a set of system components and relationships among them. Within the ontology established

by ANSI/IEEE 1471-2000, *views* are responses to *viewpoints*, where a viewpoint is a specification that describes the architecture in question from the perspective of a given set of stakeholders and their concerns. The viewpoint specifies not only the concerns addressed but the presentation, model kinds used, conventions used and any consistency (correspondence) rules to keep a view consistent with other views.

Some possible views (actually, *viewpoints* in the 1471 ontology) are:

- Functional/logic view
- Code/module view
- Development/structural view
- Concurrency/process/runtime/thread view
- Physical/deployment/install view
- User action/feedback view
- Data view/data model

Several languages for describing software architectures ('architecture description language' in ISO/IEC 42010 / IEEE-1471 terminology) have been devised, but no consensus exists on which symbol-set or language should be used to describe each architecture view. The UML is a standard that can be used *"for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes."* Thus, the UML is a visual language that can be used to create software architecture views.

Architecture frameworks

Frameworks related to the domain of software architecture are:

- 4+1
- RM-ODP (Reference Model of Open Distributed Processing)
- Service-Oriented Modeling Framework (SOMF)

Other architectures such as the Zachman Framework, DODAF, and TOGAF relate to the field of Enterprise architecture.

The distinction from functional design

The IEEE Std 610.12-1990 Standard Glossary of Software Engineering Terminology defines the following distinctions:

- Architectural Design: the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.
- Detailed Design: the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to begin implementation.

- Functional Design: the process of defining the working relationships among the components of a system.
- Preliminary Design: the process of analyzing design alternatives and defining the architecture, components, interfaces, and timing/sizing estimates for a system or components.

Software architecture, also described as strategic design, is an activity concerned with global requirements governing *how* a solution is implemented such as programming paradigms, architectural styles, component-based software engineering standards, architectural patterns, security, scale, integration, and law-governed regularities. Functional design, also described as tactical design, is an activity concerned with local requirements governing *what* a solution does such as algorithms, design patterns, programming idioms, refactorings, and low-level implemenation.

According to the Intension/Locality Hypothesis, the distinction between architectural and detailed design is defined by the Locality Criterion, according to which a statement about software design is non-local (architectural) if and only if a program that satisfies it can be expanded into a program which does not. For example, the client–server style is architectural (strategic) because a program that is built on this principle can be expanded into a program which is not client–server; for example, by adding peer-to-peer nodes.

Architecture is design but not all design is architectural. In practice, the architect is the one who draws the line between software architecture (architectural design) and detailed design (non-architectural design). There aren't rules or guidelines that fit all cases. Examples of rules or heuristics that architects (or organizations) can establish when they want to distinguish between architecture and detailed design include:

- Architecture is driven by non-functional requirements, while functional design is driven by functional requirements.
- Pseudo-code belongs in the detailed design document.
- UML component, deployment, and package diagrams generally appear in software architecture documents; UML class, object, and behavior diagrams appear in detailed functional design documents.

Examples of architectural styles and patterns

There are many common ways of designing computer software modules and their communications, among them:

- Blackboard
- Client–server model (2-tier, n-tier, peer-to-peer, cloud computing all use this model)
- Database-centric architecture (broad division can be made for programs which have database at its center and applications which don't have to rely on databases, E.g. desktop application programs, utility programs etc.)
- Distributed computing

- Event-driven architecture
- Front end and back end
- Implicit invocation
- Monolithic application
- Peer-to-peer
- Pipes and filters
- Plug-in (computing)
- Representational State Transfer
- Rule evaluation
- Search-oriented architecture (A pure SOA implements a service for every data access point.)
- Service-oriented architecture
- Shared nothing architecture
- Software componentry
- Space based architecture
- Structured (module-based but usually monolithic within modules)
- Three-tier model (An architecture with Presentation, Business Logic and Database tiers)

Chapter 8

Software Design

Software design is a process of problem-solving and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view.

Overview

The software requirements analysis (SRA) step of a software development process yields specifications that are used in software engineering. If the software is "semiautomated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case some documentation of the plan is usually the product of the design.

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design.

Software Design Topics

Design concepts

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

1. Abstraction - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.
2. Refinement - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.

3. Modularity - Software architecture is divided into components called modules.
4. Software Architecture - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. A good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule and cost.
5. Control Hierarchy - A program structure that represent the organization of a program components and implies a hierarchy of control.
6. Structural Partitioning - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.
7. Data Structure - It is a representation of the logical relationship among individual elements of data.
8. Software Procedure - It focuses on the processing of each modules individually
9. Information Hiding - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Design considerations

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

- **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
- **Fault-tolerance** - The software is resistant to and able to recover from component failure.
- **Maintainability** - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.
- **Modularity** - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
- **Packaging** - Printed material such as the box and manuals should match the style designated for the target market and should enhance usability. All compatibility information should be visible on the outside of the package. All components required for use should be included in the package or specified as a requirement on the outside of the package.

- **Reliability -** The software is able to perform a required function under stated conditions for a specified period of time.
- **Reusability -** the software is able to add further features and modification with slight or no modification.
- **Robustness -** The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
- **Security -** The software is able to withstand hostile acts and influences.
- **Usability -** The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.

Modeling language

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual. Examples of graphical modelling languages for software design are:

- Business Process Modeling Notation (BPMN) is an example of a Process Modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across a number of layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.
- Jackson Structured Programming (JSP) is a method for structured programming based on correspondences between data stream structure and program structure
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Unified Modeling Language (UML) is a general modeling language to describe software both structurally and behaviorally. It has a graphical notation and allows for extension with a Profile (UML).
- Alloy (specification language) is a general purpose specification language for expressing complex structural constraints and behavior in a software system. It provides a concise language based on first-order relational logic.
- Systems Modeling Language (SysML) is a new general-purpose modeling language for systems engineering.

flexibility

Design patterns

A software designer or architect may identify a design problem which has been solved by others before. A template or pattern describing a solution to a common problem is known as a design pattern. The reuse of such patterns can speed up the software development process, having been tested and proved in the past.

Usage

Software design documentation may be reviewed or presented to allow constraints, specifications and even requirements to be adjusted prior to programming. Redesign may occur after review of a programmed simulation or prototype. It is possible to design software in the process of programming, without a plan or requirement analysis, but for more complex projects this would not be considered a professional approach. A separate design prior to programming allows for multidisciplinary designers and Subject Matter Experts (SMEs) to collaborate with highly-skilled programmers for software that is both useful and technically sound.

Chapter 9

Software Development Methodology

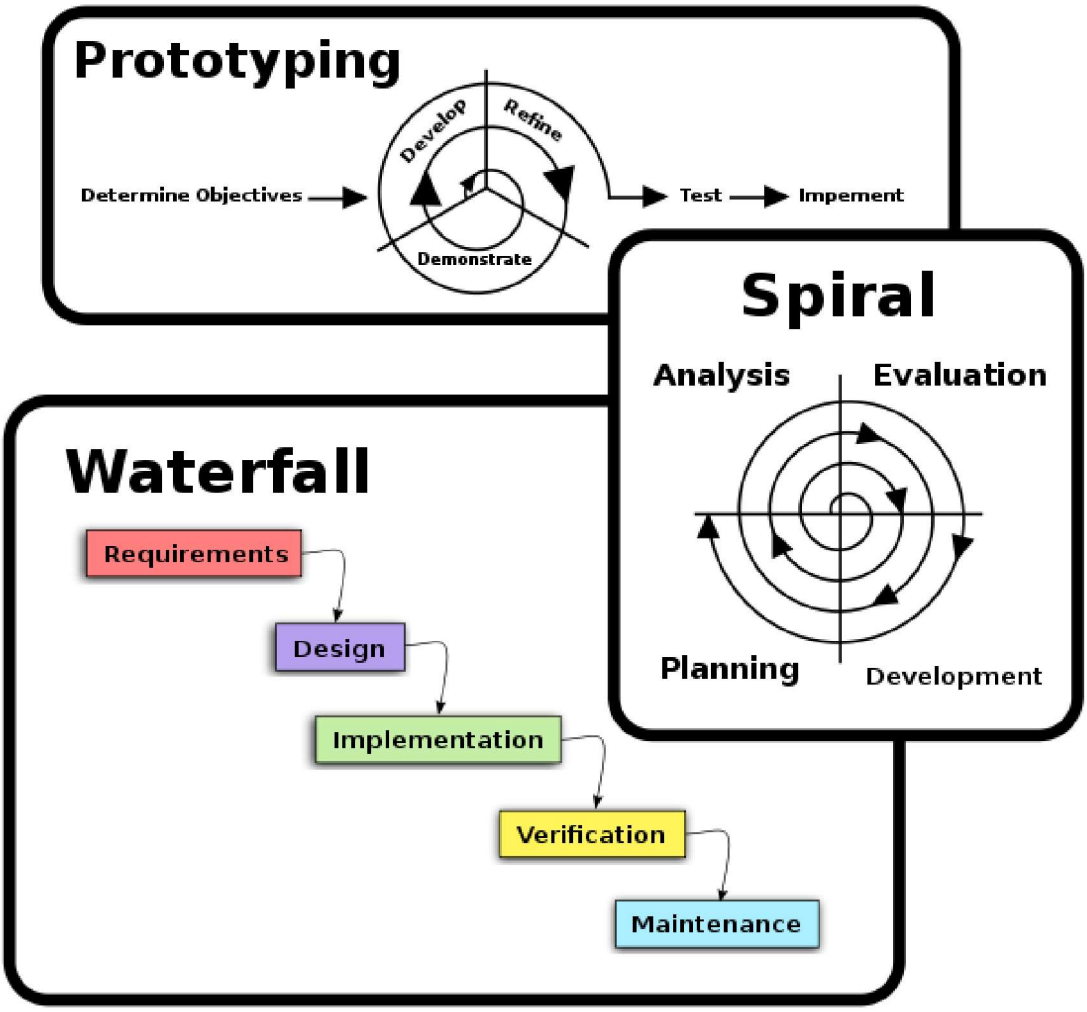
A **software development methodology** or **system development methodology** in software engineering is a framework that is used to structure, plan, and control the process of developing an information system.

History

The software development methodology framework didn't emerge until the 1960s. According to Elliott (2004) the systems development life cycle (SDLC) can be considered to be the oldest formalized methodology framework for building information systems. The main idea of the SDLC has been "to pursue the development of information systems in a very deliberate, structured and methodical way, requiring each stage of the life cycle from inception of the idea to delivery of the final system, to be carried out in rigidly and sequentially". within the context of the framework being applied. The main target of this methodology framework in the 1960s was "to develop large scale functional business systems in an age of large scale business conglomerates. Information systems activities revolved around heavy data processing and number crunching routines".

As a noun

As a noun, a software development methodology is a framework that is used to structure, plan, and control the process of developing an information system - this includes the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application.



The three basic approaches applied to software development methodology frameworks

A wide variety of such frameworks have evolved over the years, each with its own recognized strengths and weaknesses. One software development methodology framework is not necessarily suitable for use by all projects. Each of the available methodology frameworks are best suited to specific kinds of projects, based on various technical, organizational, project and team considerations.

These software development frameworks are often bound to some kind of organization, which further develops, supports the use, and promotes the methodology framework. The methodology framework is often defined in some kind of formal documentation. Specific software development methodology frameworks (noun) include

- Rational Unified Process (RUP, IBM) since 1998.
- Agile Unified Process (AUP) since 2005 by Scott Ambler

As a verb

As a verb, the software development methodology is an approach used by organizations and project teams to apply the software development methodology framework (noun). Specific software development methodologies (verb) include:

1970s

- Structured programming since 1969
- Cap Gemini SDM, originally from PANDATA, the first English translation was published in 1974. SDM stands for System Development Methodology

1980s

- Structured Systems Analysis and Design Methodology (SSADM) from 1980 onwards
- Information Requirement Analysis/Soft systems methodology

1990s

- Object-oriented programming (OOP) has been developed since the early 1960s, and developed as a dominant programming approach during the mid-1990s
- Rapid application development (RAD) since 1991
- Scrum, since the late 1990s
- Team software process developed by Watts Humphrey at the SEI
- Extreme Programming since 1999

Verb approaches

Every software development methodology framework acts as a basis for applying specific approaches to develop and maintain software. Several software development approaches have been used since the origin of information technology. These are:

- Waterfall: a linear framework
- Prototyping: an iterative framework
- Incremental: a combined linear-iterative framework
- Spiral: a combined linear-iterative framework
- Rapid application development (RAD): an iterative framework
- Extreme Programming

Waterfall development

The Waterfall model is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The

first formal description of the method is often cited as an article published by Winston W. Royce in 1970 although Royce did not use the term "waterfall" in this article.

The basic principles are:

- Project is divided into sequential phases, with some overlap and splashback acceptable between phases.
- Emphasis is on planning, time schedules, target dates, budgets and implementation of an entire system at one time.
- Tight control is maintained over the life of the project via extensive written documentation, formal reviews, and approval/signoff by the user and information technology management occurring at the end of most phases before beginning the next phase.

Prototyping

Software prototyping, is the development approach of activities during software development, the creation of prototypes, i.e., incomplete versions of the software program being developed.

The basic principles are:

- Not a standalone, complete development methodology, but rather an approach to handling selected parts of a larger, more traditional development methodology (i.e. incremental, spiral, or rapid application development (RAD)).
- Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
- User is involved throughout the development process, which increases the likelihood of user acceptance of the final implementation.
- Small-scale mock-ups of the system are developed following an iterative modification process until the prototype evolves to meet the users' requirements.
- While most prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.
- A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem.

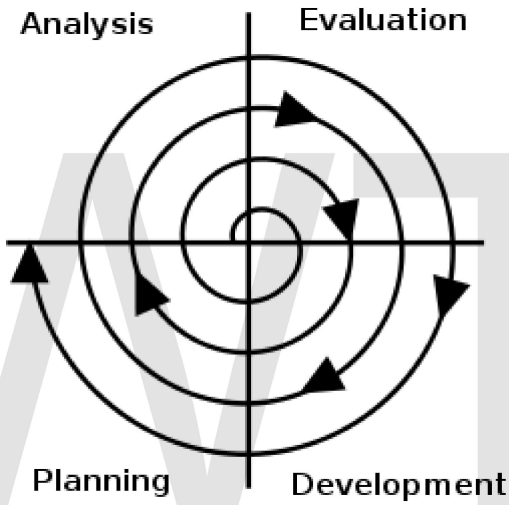
Incremental development

Various methods are acceptable for combining linear and iterative systems development methodologies, with the primary objective of each being to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

The basic principles are:

- A series of mini-Waterfalls are performed, where all phases of the Waterfall are completed for a small part of a system, before proceeding to the next increment, or
- Overall requirements are defined before proceeding to evolutionary, mini-Waterfall development of individual increments of a system, or
- The initial software concept, requirements analysis, and design of architecture and system core are defined via Waterfall, followed by iterative Prototyping, which culminates in installing the final prototype, a working system.

Spiral development



The spiral model

The spiral model is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. It is a meta-model, a model that can be used by other models.

The basic principles are:

- Focus is on risk assessment and on minimizing project risk by breaking a project into smaller segments and providing more ease-of-change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle.
- "Each cycle involves a progression through the same sequence of steps, for each part of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program."
- Each trip around the spiral traverses four basic quadrants: (1) determine objectives, alternatives, and constraints of the iteration; (2) evaluate alternatives;

Identify and resolve risks; (3) develop and verify deliverables from the iteration; and (4) plan the next iteration.
- Begin each cycle with an identification of stakeholders and their win conditions, and end each cycle with review and commitment.

Rapid application development

Rapid application development (RAD) is a software development methodology, which involves iterative development and the construction of prototypes. Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991.

The basic principles are:

- Key objective is for fast development and delivery of a high quality system at a relatively low investment cost.
- Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
- Aims to produce high quality systems quickly, primarily via iterative Prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include Graphical User Interface (GUI) builders, Computer Aided Software Engineering (CASE) tools, Database Management Systems (DBMS), fourth-generation programming languages, code generators, and object-oriented techniques.
- Key emphasis is on fulfilling the business need, while technological or engineering excellence is of lesser importance.
- Project control involves prioritizing development and defining delivery deadlines or “timeboxes”. If the project starts to slip, emphasis is on reducing requirements to fit the timebox, not in increasing the deadline.
- Generally includes joint application design (JAD), where users are intensely involved in system design, via consensus building in either structured workshops, or electronically facilitated interaction.
- Active user involvement is imperative.
- Iteratively produces production software, as opposed to a throwaway prototype.
- Produces documentation necessary to facilitate future development and maintenance.
- Standard systems analysis and design methods can be fitted into this framework.

Other practices

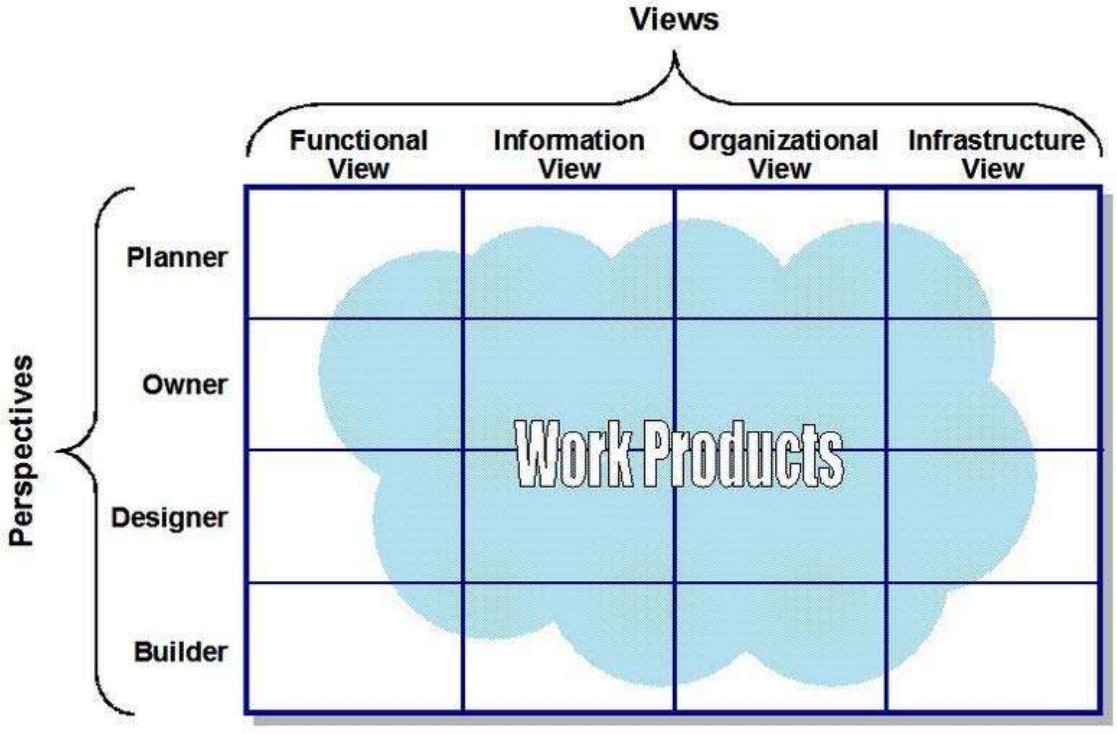
Other methodology practices include:

- Object-oriented development methodologies, such as Grady Booch's object-oriented design (OOD), also known as object-oriented analysis and design (OOAD). The Booch model includes six diagrams: class, object, state transition, interaction, module, and process.

- Top-down programming: evolved in the 1970s by IBM researcher Harlan Mills (and Niklaus Wirth) in developed structured programming.
- Unified Process (UP) is an iterative software development methodology framework, based on Unified Modeling Language (UML). UP organizes the development of software into four phases, each consisting of one or more executable iterations of the software at that stage of development: inception, elaboration, construction, and guidelines. Many tools and products exist to facilitate UP implementation. One of the more popular versions of UP is the Rational Unified Process (RUP).
- Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve via collaboration between self-organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto was formulated.

Subtopics

View model



The TEAF Matrix of Views and Perspectives

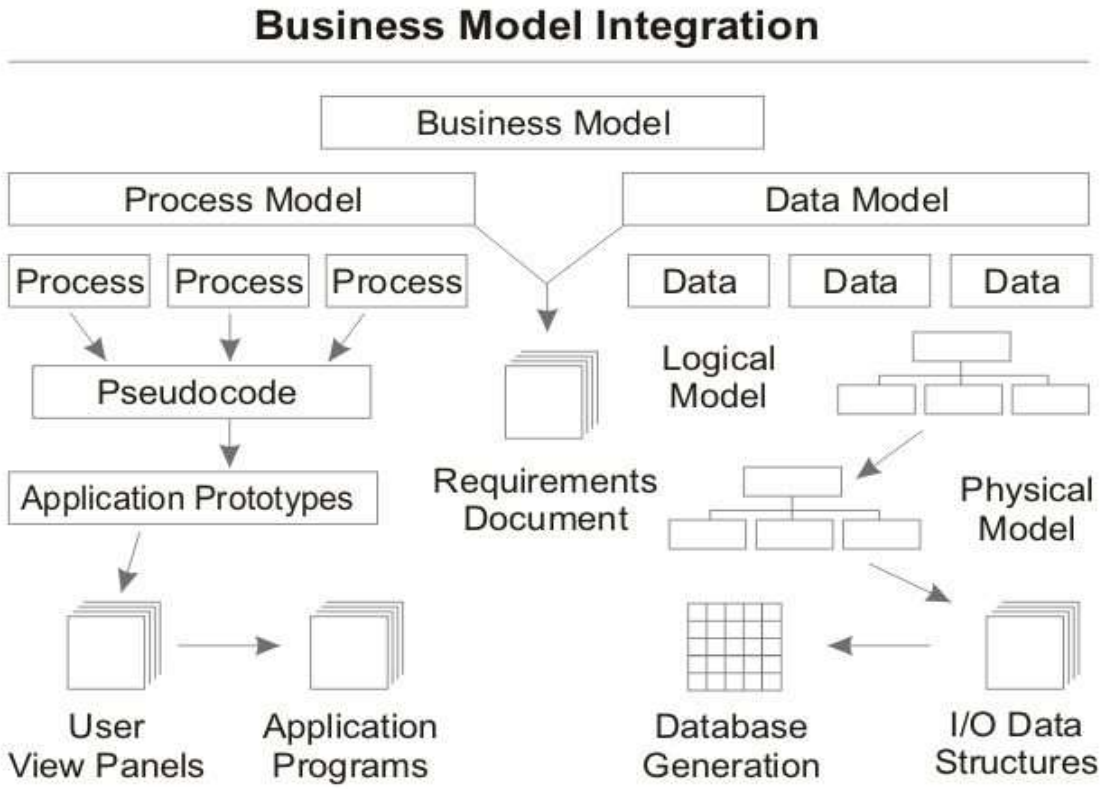
A view model is framework which provides the viewpoints on the system and its environment, to be used in the software development process. It is a graphical representation of the underlying semantics of a view.

The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems, and to organize the elements of the problem and the solution around domains of expertise. In the engineering of physically intensive systems, viewpoints often correspond to capabilities and responsibilities within the engineering organization.

Most complex system specifications are so extensive that no one individual can fully comprehend all aspects of the specifications. Furthermore, we all have different interests in a given system and different reasons for examining the system's specifications. A business executive will ask different questions of a system make-up than would a system implementer. The concept of viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system. These viewpoints each satisfy an audience with interest in some set of aspects of the system. Associated with each viewpoint is a viewpoint language that optimizes the vocabulary and presentation for the audience of that viewpoint.

Business process and data modelling

Graphical representation of the current state of information provides a very effective means for presenting information to both users and system developers.



Example of the interaction between business process and data models

- A business model illustrates the functions associated with the business process being modeled and the organizations that perform these functions. By depicting activities and information flows, a foundation is created to visualize, define, understand, and validate the nature of a process.
- A data model provides the details of information to be stored, and is of primary use when the final product is the generation of computer software code for an application or the preparation of a functional specification to aid a computer software make-or-buy decision.

Usually, a model is created after conducting an interview, referred to as business analysis. The interview consists of a facilitator asking a series of questions designed to extract required information that describes a process. The interviewer is called a facilitator to emphasize that it is the participants who provide the information. The facilitator should have some knowledge of the process of interest, but this is not as important as having a structured methodology by which the questions are asked of the process expert. The methodology is important because usually a team of facilitators is collecting information across the facility and the results of the information from all the interviewers must fit together once completed.

The models are developed as defining either the current state of the process, in which case the final product is called the "as-is" snapshot model, or a collection of ideas of what the process should contain, resulting in a "what-can-be" model. Generation of process and data models can be used to determine if the existing processes and information systems are sound and only need minor modifications or enhancements, or if re-engineering is required as a corrective action. The creation of business models is more than a way to view or automate your information process. Analysis can be used to fundamentally reshape the way your business or organization conducts its operations.

Computer-aided software engineering

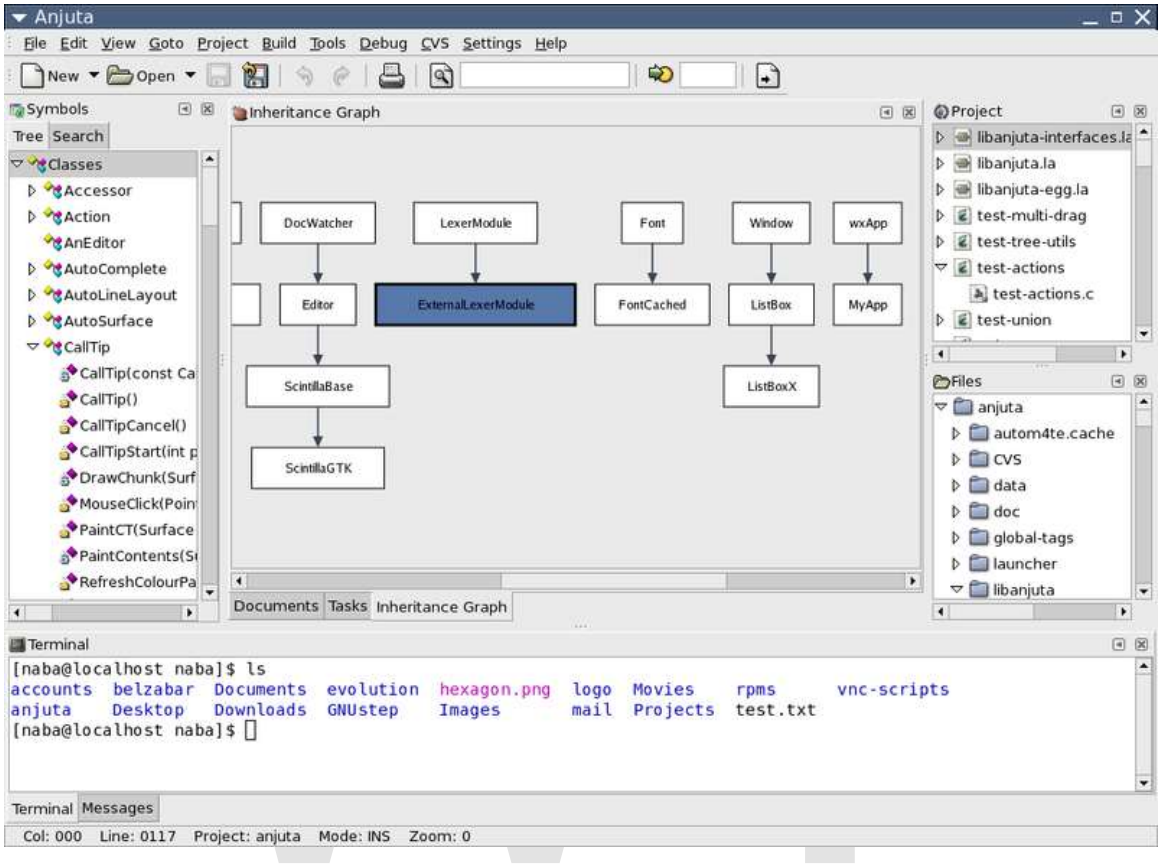
Computer-aided software engineering (CASE), in the field software engineering is the scientific application of a set of tools and methods to a software which results in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software development process. The term "computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.

Two key ideas of Computer-aided Software System Engineering (CASE) are:

- Foster computer assistance in software development and or software maintenance processes, and
- An engineering approach to software development and or maintenance.

Typical CASE tools exist for configuration management, data modeling, model transformation, refactoring, source code generation, and Unified Modeling Language.

Integrated development environment



Anjuta, a C and C++ IDE for the GNOME environment

An integrated development environment (IDE) also known as *integrated design environment* or *integrated debugging environment* is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a:

- source code editor,
- compiler and/or interpreter,
- build automation tools, and
- debugger (usually).

IDEs are designed to maximize programmer productivity by providing tight-knit components with similar user interfaces. Typically an IDE is dedicated to a specific programming language, so as to provide a feature set which most closely matches the programming paradigms of the language.

Modeling language

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual. Graphical modeling languages use a diagram techniques with named symbols that represent concepts and lines that connect the symbols and that represent relationships and various other graphical annotation to represent constraints. Textual modeling languages typically use standardised keywords accompanied by parameters to make computer-interpretable expressions.

Example of graphical modelling languages in the field of software engineering are:

- Business Process Modeling Notation (BPMN, and the XML form BPML) is an example of a process modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Specification and Description Language(SDL) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems.
- Unified Modeling Language (UML) is a general-purpose modeling language that is an industry standard for specifying software-intensive systems. UML 2.0, the current version, supports thirteen different diagram techniques, and has widespread tool support.

Not all modeling languages are executable, and for those that are, using them doesn't necessarily mean that programmers are no longer needed. On the contrary, executable modeling languages are intended to amplify the productivity of skilled programmers, so that they can address more difficult problems, such as parallel computing and distributed systems.

Programming paradigm

A programming paradigm is a fundamental style of computer programming, in contrast to a software engineering methodology, which is a style of solving specific software

engineering problems. Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints...) and the steps that compose a computation (assignation, evaluation, continuations, data flows...).

A programming language can support multiple paradigms. For example programs written in C++ or Object Pascal can be purely procedural, or purely object-oriented, or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements. In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures.

Just as different groups in software engineering advocate different *methodologies*, different programming languages advocate different *programming paradigms*. Some languages are designed to support one paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Python, Ruby, and Oz).

Many programming paradigms are as well known for what methods they *forbid* as for what they enable. For instance, pure functional programming forbids using side-effects; structured programming forbids using goto statements. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles. Avoiding certain methods can make it easier to prove theorems about a program's correctness, or simply to understand its behavior.

Software framework

A software framework is a re-usable design for a software system or subsystem. A software framework may include support programs, code libraries, a scripting language, or other software to help develop and *glue together* the different components of a software project. Various parts of the framework may be exposed via an API.

Software development process

A software development process is a framework imposed on the development of a software product. Synonyms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

A largely growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts. The international standard describing the method to select, implement and monitor the life cycle for software is ISO 12207.

A decades-long goal has been to find repeatable, predictable processes that improve productivity and quality. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management methods to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management appears to be lacking.

Chapter 10

Software Maintenance

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes.

A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997). This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

Software maintenance and evolution of systems was first addressed by Meir M. Lehman in 1969. Over a period of twenty years, his research led to the formulation of eight Laws of Evolution (Lehman 1997). Key findings of his research include that maintenance is really evolutionary developments and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Lehman demonstrated that systems continue to evolve over time. As they evolve, they grow more complex unless some action such as code refactoring is taken to reduce the complexity.

The key software maintenance issues are both managerial and technical. Key management issues are: alignment with customer priorities, staffing, which organization does maintenance, estimating costs. Key technical issues are: limited understanding, impact analysis, testing, maintainability measurement. Best and Worst Practices in Software Maintenance Because maintenance of aging legacy software is very labour intensive it is quite important to explore the best and most cost effective methods available for dealing with the millions of applications that currently exist. The sets of best and worst practices are not the same. Just as practice that has the most positive impact on maintenance productivity is the use of trained maintenance experts, while the factor that has the greatest negative impact is the presence error-prone modules in application being maintained.

The Importance of Software Maintenance

In the late 1970s, a famous and widely cited survey study by Lientz and Swanson, exposed the very high fraction of life-cycle costs that were being expended on maintenance. They categorized maintenance activities into four classes:

- Adaptive – dealing with changes and adapting in the software environment
- Perfective – accommodating with new or changed user requirements which concern functional enhancements to the software
- Corrective – dealing with errors found and fixing it
- Preventive – concerns activities aiming on increasing software maintainability and prevent problems in the future

The survey showed that around 75% of the maintenance effort was on the first two types, and error correction consumed about 21%. Many subsequent studies suggest a similar magnitude of the problem. Studies show that contribution of end user are crucial during the new requirement data gathering and analysis. And this is the main cause of any problem during software evolution and maintenance. So software maintenance is important because it consumes a large part of the overall lifecycle costs and also the inability to change software quickly and reliably means that business opportunities are lost. Impact of Key Adjustment Factors on Maintenance(Sorted in order of maximum positive impact)== Maintenance Factors Plus Range

Maintenance specialists 35%

High staff experience 34%

Table-driven variables and data 33%

Low complexity of base code 32%

Y2K and special search engines 30%

Code restructuring tools 29%

Re-engineering tools 27%

High level programming languages 25%

Reverse engineering tools 23%

Complexity analysis tools 20%

Defect tracking tools 20%

Y2K “mass update” specialists 20%

Automated change control tools 18%

Unpaid overtime 18%

Quality measurements 16%

Formal base code inspections 15%

Regression test libraries 15%

Excellent response time 12%

Annual training of > 10 days 12%

High management experience 12%

HELP desk automation 12%

No error prone modules 10%

On-line defect reporting 10%

Productivity measurements 8%

Excellent ease of use 7%

User satisfaction measurements 5%

High team morale 5%

Sum 503%

The table below summarizes the major factors that degrade software maintenance performance. Not only are error-prone modules troublesome, but many other factors can degrade performance too. For example, very complex “spaghetti code” is quite difficult to maintain safely. A very common situation which often degrades performance is lack of suitable maintenance tools, such as defect tracking software, change management software, test library software, and so forth. Below describe some of the factors and the range of Impact on maintenance of software.

Impact of Key Adjustment Factors on Maintenance(Sorted in order of maximum negative impact) Maintenance Factors

Maintenance Factors Minus Range

Error prone modules -50%

Embedded variables and data -45%

Staff inexperience -40%

High complexity of base code -30%

No Y2K of special search engines -28%

Manual change control methods -27%

Low level programming languages -25%

No defect tracking tools -24%

No Y2K “mass update” specialists -22%

Poor ease of use -18%

No quality measurements -18%

No maintenance specialists -18%

Poor response time -16% No base code inspections -15%

No regression test libraries -15%

No HELP desk automation -15%

No on-line defect reporting -12%

Management inexperience -15%

No code restructuring tools -10%

No annual training -10%

No reengineering tools -10%

No reverse engineering tools -10%

No complexity analysis tools -10%

No productivity measurements -7%

Poor team morale -6%

No user satisfaction measurements -4%

No unpaid overtime 0%

Sum -500%

Software maintenance planning

The integral part of software is the maintenance part which requires accurate maintenance plan to be prepared during software development and should specify how users will request modifications or report problems and the estimation of resources such as cost should be included in the budget and a new decision should address to develop a new system and its quality objectives .The software maintenance which can last for 5–6 years after the development calls for an effective planning which addresses the scope of software maintenance, the tailoring of the post delivery process, the designation of who will provide maintenance, an estimate of the life-cycle costs .

Software maintenance processes

Here we, describes the six software maintenance processes as:

1. The implementation processes contains software preparation and transition activities, such as the conception and creation of the maintenance plan, the preparation for handling problems identified during development, and the follow-up on product configuration management.
2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and propose a solution, document the request and the solution proposal, and, finally, obtain all the required authorizations to apply the modifications.
3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process (platform migration, for example) is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.
6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

There are a number of processes, activities and practices that are unique to maintainers, for example:

- Transition: a controlled and coordinated sequence of activities during which a system is transferred progressively from the developer to the maintainer;
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts negotiated by maintainers;
- Modification Request and Problem Report Help Desk: a problem-handling process used by maintainers to prioritize, documents and route the requests they receive;

- Modification Request acceptance/rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

Categories of maintenance in ISO/IEC 14764

E.B. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective. These have since been updated and ISO/IEC 14764 presents:

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems.
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability.
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

There is also a notion of pre-delivery/pre-release maintenance which is all the good things you do to lower the total cost of ownership of the software. Things like compliance with coding standards that includes software maintainability goals. The management of coupling and cohesion of the software. The attainment of software supportability goals (SAE JA1004, JA1005 and JA1006 for example). Note also that some academic institutions are carrying out research to quantify the cost to ongoing software maintenance due to the lack of resources such as design documents and system/software comprehension training and resources (multiply costs by approx. 1.5-2.0 where there is no design data available.).

Chapter 11

Software Testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects).

Software testing can also be stated as the process of validating and verifying that a software program/application/product:

1. meets the business and technical requirements that guided its design and development;
2. works as expected; and
3. can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

Different software development models will focus the test effort at different points in the development process. Newer development models, such as Agile, often employ test driven development and place an increased portion of the testing in the hands of the developer, before it reaches a formal team of testers. In a more traditional model, most of the test execution occurs after the requirements have been defined and the coding process has been completed.

Overview

Testing can never completely identify all the defects within software. Instead, it furnishes a *criticism* or *comparison* that compares the state and behavior of the product against oracles—principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected

purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. **Software testing** is the process of attempting to make this assessment.

A study conducted by NIST in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed.

History

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979. Although his attention was on breakage testing ("a successful test is one that finds a bug") it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:

- Until 1956 - Debugging oriented
- 1957–1978 - Demonstration oriented
- 1979–1982 - Destruction oriented
- 1983–1987 - Evaluation oriented
- 1988–2000 - Prevention oriented

Software testing topics

Scope

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

Functional vs non-functional testing

Functional testing refers to activities that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or other performance, behavior under certain constraints, or security. Non-functional requirements tend to be those that reflect the quality of the product, particularly in the context of the suitability perspective of its users.

Defects and failures

Not all software defects are caused by coding errors. One common source of expensive defects is caused by requirement gaps, e.g., unrecognized requirements, that result in errors of omission by the program designer. A common source of requirements gaps is non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new hardware platform, alterations in source data or interacting with different software. A single defect may result in a wide range of failure symptoms.

Finding faults early

It is commonly believed that the earlier a defect is found the cheaper it is to fix it. The following table shows the cost of fixing the defect depending on the stage it was found. For example, if a problem in the requirements is found only post-release, then it would cost 10–100 times more to fix than if it had already been found by the requirements review.

Cost to fix a defect		Time detected				
		Requirements	Architecture	Construction	System test	Post-release
Time introduced	Requirements	1×	3×	5–10×	10×	10–100×
	Architecture	-	1×	10×	15×	25–100×
	Construction	-	-	1×	10×	10–25×

Compatibility

A common cause of software failure (real or perceived) is a lack of compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a web application, which must render in a web browser). For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment was capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.

Input combinations and preconditions

A very fundamental problem with software testing is that testing under *all* combinations of inputs and preconditions (initial state) is not feasible, even with a simple product. This means that the number of defects in a software product can be very large and defects that occur infrequently are difficult to find in testing. More significantly, non-functional dimensions of quality (how it is supposed to *be* versus what it is supposed to *do*)—usability, scalability, performance, compatibility, reliability—can be highly subjective; something that constitutes sufficient value to one person may be intolerable to another.

Static vs. dynamic testing

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example,

spreadsheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

Software verification and validation

Software testing is used in association with verification and validation:

- Verification: Have we built the software right? (i.e., does it match the specification).
- Validation: Have we built the right software? (i.e., is this what the customer wants).

The terms verification and validation are commonly used interchangeably in the industry; it is also common to see these two terms incorrectly defined. According to the IEEE Standard Glossary of Software Engineering Terminology:

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

The software testing team

Software testing can be done by software testers. Until the 1980s the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing, different roles have been established: *manager*, *test lead*, *test designer*, *tester*, *automation developer*, and *test administrator*.

Software quality assurance (SQA)

Though controversial, software testing is a part of the software quality assurance (SQA) process. In SQA, software process specialists and auditors are concerned for the software development process rather than just the artefacts such as documentation, code and systems. They examine and change the software engineering process itself to reduce the amount of faults that end up in the delivered software: the so-called *defect rate.*

What constitutes an "acceptable defect rate" depends on the nature of the software; A flight simulator video game would have much higher defect tolerance than software for an actual airplane.

Although there are close links with SQA, testing departments often exist independently, and there may be no SQA function in some companies.

Software testing is a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs. By contrast, QA (quality assurance) is the implementation of policies and procedures intended to prevent defects from occurring in the first place.

Testing methods

The box approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White box testing

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

Types of white box testing

The following types of white box testing exist:

- API testing (application programming interface) - testing of the application using public and private APIs
- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods - improving the coverage of a test by introducing faults to test code paths
- Mutation testing methods
- Static testing - White box testing includes all static testing

Test coverage

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.

Two common forms of code coverage are:

- *Function coverage*, which reports on functions executed
- *Statement coverage*, which reports on the number of lines executed to complete the test

They both return a code coverage metric, measured as a percentage.

Black box testing

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, exploratory testing and specification-based testing.

Specification-based testing: Specification-based testing aims to test the functionality of software according to the applicable requirements. Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.
Specification-based testing is necessary, but it is insufficient to guard against certain risks.
Advantages and disadvantages: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code *must* have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. On the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion", on the one hand, and the disadvantage of "blind exploring", on the other.

Grey box testing

Grey box testing (American spelling: **gray box testing**) involves having knowledge of internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

Testing levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process as defined by the SWEBOK guide are unit-, integration-, and system testing that

are distinguished by the test target without impliying a specific process model. Other test levels are classified by the testing objective.

Test target

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

These type of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other.

Unit testing is also called *component testing*.

Integration testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be localised more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

System testing

System testing tests a completely integrated system to verify that it meets its requirements.

System integration testing

System integration testing verifies that a system is integrated to any external or third-party systems defined in the system requirements.

Objectives of testing

Regression testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, to very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk.

Acceptance testing

Acceptance testing can mean one of two things:

1. A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
2. Acceptance testing is performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Non-functional testing

Special methods exist to test non-functional aspects of software. In contrast to functional testing, which establishes the correct operation of the software (correct in that it matches the expected behavior defined in the design requirements), non-functional testing verifies that the software functions properly even when it receives invalid or unexpected inputs. Software fault injection, in the form of fuzzing, is an example of non-functional testing. Non-functional testing, especially for software, is designed to establish whether the device under test can tolerate invalid or unexpected inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. Various commercial non-functional testing tools are linked from the software fault injection page; there are also numerous open-source and free software tools available that perform non-functional testing.

Software performance testing and load testing

Performance testing is executed to determine how fast a system or sub-system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Load testing is primarily concerned with testing that can continue to operate under a specific load, whether that be large quantities of data or a large number of users. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as *endurance testing*.

Volume testing is a way to test functionality. *Stress testing* is a way to test reliability. *Load testing* is a way to test performance. There is little agreement on what the specific goals of load testing are. The terms load testing, performance testing, reliability testing, and volume testing, are often used interchangeably.

Stability testing

Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of non-functional software testing is often referred to as load (or endurance) testing.

Usability testing

Usability testing is needed to check if the user interface is easy to use and understand.It approach towards the use of the application.

Security testing

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

Internationalization and localization

The general ability of software to be internationalized and localized can be automatically tested without actual translation, by using pseudolocalization. It will verify that the application still works, even after it has been translated into a new language or adapted for a new culture (such as different currencies or time zones).

Actual translation to human languages must be tested, too. Possible localization failures include:

- Software is often localized by translating a list of strings out of context, and the translator may choose the wrong translation for an ambiguous source string.
- Technical terminology may become inconsistent if the project is translated by several people without proper coordination or if the translator is imprudent.
- Literal word-for-word translations may sound inappropriate, artificial or too technical in the target language.
- Untranslated messages in the original language may be left hard coded in the source code.
- Some messages may be created automatically in run time and the resulting string may be ungrammatical, functionally incorrect, misleading or confusing.
- Software may use a keyboard shortcut which has no function on the source language's keyboard layout, but is used for typing characters in the layout of the target language.
- Software may lack support for the character encoding of the target language.
- Fonts and font sizes which are appropriate in the source language, may be inappropriate in the target language; for example, CJK characters may become unreadable if the font is too small.
- A string in the target language may be longer than the software can handle. This may make the string partly invisible to the user or cause the software to crash or malfunction.
- Software may lack proper support for reading or writing bi-directional text.
- Software may display images with text that wasn't localized.
- Localized operating systems may have differently-named system configuration files and environment variables and different formats for date and currency.

To avoid these and other localization problems, a tester who knows the target language must run the program with all the possible use cases for translation to see if the messages are readable, translated correctly in context and don't cause failures.

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail, in order to test its robustness.

The testing process

Traditional CMMI or waterfall development model

A common practice of software testing is that testing is performed by an independent group of testers after the functionality is developed, before it is shipped to the customer. This practice often results in the testing phase being used as a project buffer to compensate for project delays, thereby compromising the time devoted to testing.

Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

Agile or Extreme development model

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process, unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process). The ultimate goal of this test process is to achieve continuous deployment where software updates can be published to the public frequently.

A sample testing cycle

Although variations exist between organizations, there is a typical cycle for testing. The sample below is common among organizations employing the Waterfall development model.

- **Requirements analysis**: Testing should begin in the requirements phase of the software development life cycle. During the design phase, testers work with developers in determining what aspects of a design are testable and with what parameters those tests work.
- **Test planning**: Test strategy, test plan, testbed creation. Since many activities will be carried out during testing, a plan is needed.
- **Test development**: Test procedures, test scenarios, test cases, test datasets, test scripts to use in testing software.
- **Test execution**: Testers execute the software based on the plans and test documents then report any errors found to the development team.
- **Test reporting**: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.

- **Test result analysis**: Or Defect Analysis, is done by the development team usually along with the client, in order to decide what defects should be treated, fixed, rejected (i.e. found software working properly) or deferred to be dealt with later.
- **Defect Retesting**: Once a defect has been dealt with by the development team, it is retested by the testing team. AKA Resolution testing.
- **Regression testing**: It is common to have a small test program built of a subset of tests, for each integration of new, modified, or fixed software, in order to ensure that the latest delivery has not ruined anything, and that the software product as a whole is still working correctly.
- **Test Closure**: Once the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

Automated testing

Many programming groups are relying more and more on automated testing, especially groups that use test-driven development. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system.

While automation cannot reproduce everything that a human can do (and all the ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

Testing tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

- Program monitors, permitting full or partial monitoring of program code including:
    - Instruction set simulator, permitting complete instruction level monitoring and trace facilities
    - Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
    - Code coverage reports
- Formatted dump or symbolic debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional GUI testing tools are used to repeat system-level tests through the GUI
- Benchmarks, allowing run-time performance comparisons to be made
- Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Some of these features may be incorporated into an Integrated Development Environment (IDE).

- A regression testing technique is to have a standard set of tests, which cover existing functionality that result in persistent tabular data, and to compare pre-change data to post-change data, where there should not be differences, using a tool like diffkit. Differences detected indicate unexpected functionality changes or "regression".

Measurement in software testing

Usually, quality is constrained to such topics as correctness, completeness, security, but can also include more technical requirements as described under the ISO standard ISO/IEC 9126, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

There are a number of frequently-used software measures, often called *metrics*, which are used to assist in determining the state of the software or the adequacy of the testing.

Testing artifacts

Software testing process can produce several artifacts.

Test plan
: A test specification is called a test plan. The developers are well aware what test plans will be executed and this information is made available to management and the developers. The idea is to make them more cautious when developing their code or making additional changes. Some companies have a higher-level document called a test strategy.

Traceability matrix
: A traceability matrix is a table that correlates requirements or design documents to test documents. It is used to change tests when the source documents are changed, or to verify that the test results are correct.

Test case
: A test case normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result. This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step, or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or

steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repository. In a database system, you may also be able to see past test results, who generated the results, and what system configuration was used to generate those results. These past results would usually be stored in a separate table.

Test script
: The test script is the combination of a test case, test procedure, and test data. Initially the term was derived from the product of work created by automated regression test tools. Today, test scripts can be manual, automated, or a combination of both.

Test suite
: The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Test data
: In most cases, multiple sets of values or data are used to test the same functionality of a particular feature. All the test values and changeable environmental components are collected in separate files and stored as test data. It is also useful to provide this data to the client and with the product or a project.

Test harness
: The software, tools, samples of data input and output, and configurations are all referred to collectively as a test harness.

Certifications

Several certification programs exist to support the professional aspirations of software testers and quality assurance specialists. No certification currently offered actually requires the applicant to demonstrate the ability to test software. No certification is based on a widely accepted body of knowledge. This has led some to declare that the testing field is not ready for certification. Certification itself cannot measure an individual's productivity, their skill, or practical knowledge, and cannot guarantee their competence, or professionalism as a tester.

Software testing certification types

- *Exam-based*: Formalized exams, which need to be passed; can also be learned by self-study [e.g., for ISTQB or QAI]
- *Education-based*: Instructor-led sessions, where each course has to be passed [e.g., International Institute for Software Testing (IIST)].

Testing certifications

- Certified Associate in Software Testing (CAST) offered by the Quality Assurance Institute (QAI)
- CATe offered by the *International Institute for Software Testing*
- Certified Manager in Software Testing (CMST) offered by the Quality Assurance Institute (QAI)
- Certified Software Tester (CSTE) offered by the Quality Assurance Institute (QAI)
- Certified Software Test Professional (CSTP) offered by the *International Institute for Software Testing*
- CSTP (TM) (Australian Version) offered by *K. J. Ross & Associates*
- ISEB offered by the Information Systems Examinations Board
- ISTQB Certified Tester, Foundation Level (CTFL) offered by the International Software Testing Qualification Board
- ISTQB Certified Tester, Advanced Level (CTAL) offered by the International Software Testing Qualification Board
- TMPF TMap Next Foundation offered by the *Examination Institute for Information Science*
- TMPA TMap Next Advanced offered by the *Examination Institute for Information Science*

Quality assurance certifications

- CMSQ offered by the *Quality Assurance Institute* (QAI).
- CSQA offered by the *Quality Assurance Institute* (QAI)
- CSQE offered by the American Society for Quality (ASQ)
- CQIA offered by the American Society for Quality (ASQ)

Controversy

Some of the major software testing controversies include:

What constitutes responsible software testing?
: Members of the "context-driven" school of testing believe that there are no "best practices" of testing, but rather that testing is a set of skills that allow the tester to select or invent testing practices to suit each unique situation.

Agile vs. traditional
: Should testers learn to work under conditions of uncertainty and constant change or should they aim at process "maturity"? The agile testing movement has received growing popularity since 2006 mainly in commercial circles, whereas government and military software providers use this methodology but also the traditional test-last models (e.g. in the Waterfall model).

Exploratory test vs. scripted
: Should tests be designed at the same time as they are executed or should they be designed beforehand?

Manual testing vs. automated

Some writers believe that test automation is so expensive relative to its value that it should be used sparingly. More in particular, test-driven development states that developers should write unit-tests of the XUnit type before coding the functionality. The tests then can be considered as a way to capture and implement the requirements.

Software design vs. software implementation

Should testing be carried out only at the end or throughout the whole process?

Who watches the watchmen?

The idea is that any form of observation is also an interaction—the act of testing can also affect that which is being tested.

Chapter 12

Software Quality

In the context of software engineering, **software quality** measures how well software is designed (*quality of design*), and how well the software conforms to that design (*quality of conformance*), although there are several different definitions, including conformance to customer expectectations. It is often described as the 'fitness for purpose' of a piece of software.

Whereas *quality of conformance* is concerned with implementation, *quality of design* measures how valid the design and requirements are in creating a worthwhile product.

Definition

One of the challenges of software quality is that "everyone feels they understand it".

In addition to more software specific definitions given below, there are several applicable definitions of quality which are used in business.Quality_(business)#Definitions

Software quality may be defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

The three key points in this definition:

1. Software requirements are the foundations from which quality is measured.

   Lack of conformance to requirement is lack of quality.

2. Specified standards define a set of development criteria that guide the management in software engineering.

   If criteria are not followed lack of quality will usually result.

3. A set of implicit requirements often goes unmentioned, for example ease of use, maintainability etc.

If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspected.

A definition in Steve McConnell's *Code Complete* divides software into two pieces: **internal** and **external quality characteristics**. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.

Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better." This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality.

Another definition, coined by Gerald Weinberg in *Quality Software Management: Systems Thinking*, is "Quality is value to some person." This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as "Who are the people we want to value our software?" and "What will be valuable to *them*?"

History

Software product quality

- Product quality
  - conformance to requirements or program specification; related to Reliability
- Scalability
- Correctness
- Completeness
- Absence of bugs
- Fault-tolerance
  - Extensibility
  - Maintainability
- Documentation

The Consortium for IT Software Quality (CISQ) was launched in 2009 to standardize the measurement of software product quality. The Consortium's goal is to bring together industry executives from Global 2000 IT organizations, system integrators, outsourcers, and package vendors to jointly address the challenge of standardizing the measurement of IT software quality and to promote a market-based ecosystem to support its deployment.

It is essential to supplement traditional testing – functional, non-functional, and run-time – with measures of application structural quality. Structural quality is the quality of the application's architecture and the degree to which its implementation accords with software engineering best practices. Industry data demonstrate that poor application structural quality results in cost and schedule overruns and creates waste in the form of

rework (up to 45% of development time in some organizations). Moreover, poor structural quality is strongly correlated with high-impact business disruptions due to corrupted data, application outages, security breaches, and performance problems. As in any other field of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs.

Source code quality

A computer has no concept of "well-written" source code. However, from a human point of view source code can be written in a way that has an effect on the effort needed to comprehend its behavior. Many source code programming style guides, which often stress readability and usually language-specific conventions are aimed at reducing the cost of source code maintenance. Some of the issues that affect code quality include:

- Readability
- Ease of maintenance, testing, debugging, fixing, modification and portability
- Low complexity
- Low resource consumption: memory, CPU
- Number of compilation or lint warnings
- Robust input validation and error handling, established by software fault injection

Methods to improve the quality:

- Refactoring
- Code Inspection or software review
- Documenting the code

Software reliability

Software **reliability** is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

History

With software embedded into many devices today, software failure has caused more than inconvenience. Software errors have even caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that lead to multiple deaths is discussed in Dr. Leveson's paper (PDF). This has resulted in requirements for development of some types software. In the

United States, both the Food and Drug Administration (FDA) and Federal Aviation Administration (FAA) have requirements for software development.

Goal of reliability

The need for a means to objectively determine software reliability comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both lay-persons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviour, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Regardless of the criticality of any single software application, it is also more and more frequently observed that software has penetrated deeply into most every aspect of modern life through the technology we use. It is only expected that this infiltration will continue, along with an accompanying dependency on the software by the systems which maintain our society. As software becomes more and more crucial to the operation of the systems on which we depend, the argument goes, it only follows that the software should offer a concomitant level of dependability. In other words, the software should behave in the way it is intended, or even better, in the way it should.

Challenge of reliability

The circular logic of the preceding sentence is not accidental—it is meant to illustrate a fundamental problem in the issue of measuring software reliability, which is the difficulty of determining, in advance, exactly how the software is intended to operate. The problem seems to stem from a common conceptual error in the consideration of software, which is that software in some sense takes on a role which would otherwise be filled by a human being. This is a problem on two levels. Firstly, most modern software performs work which a human could never perform, especially at the high level of reliability that is often expected from software in comparison to humans. Secondly, software is fundamentally incapable of most of the mental capabilities of humans which separate them from mere mechanisms: qualities such as adaptability, general-purpose knowledge, a sense of conceptual and functional context, and common sense.

Nevertheless, most software programs could safely be considered to have a particular, even singular purpose. If the possibility can be allowed that said purpose can be well or even completely defined, it should present a means for at least considering objectively whether the software is, in fact, reliable, by comparing the expected outcome to the actual outcome of running the software in a given environment, with given data. Unfortunately, it is still not known whether it is possible to exhaustively determine either the expected outcome or the actual outcome of the entire set of possible environment and input data to

a given program, without which it is probably impossible to determine the program's reliability with any certainty.

However, various attempts are in the works to attempt to rein in the vastness of the space of software's environmental and input variables, both for actual programs and theoretical descriptions of programs. Such attempts to improve software reliability can be applied at different stages of a program's development, in the case of real software. These stages principally include: requirements, design, programming, testing, and runtime evaluation. The study of theoretical software reliability is predominantly concerned with the concept of correctness, a mathematical field of computer science which is an outgrowth of language and automata theory.

Reliability in program development

Requirements

A program cannot be expected to work as desired if the developers of the program do not, in fact, know the program's desired behaviour in advance, or if they cannot at least determine its desired behaviour in parallel with development, in sufficient detail. What level of detail is considered sufficient is hotly debated. The idea of perfect detail is attractive, but may be impractical, if not actually impossible. This is because the desired behaviour tends to change as the possible range of the behaviour is determined through actual attempts, or more accurately, failed attempts, to achieve it.

Whether a program's desired behaviour can be successfully specified in advance is a moot point if the behaviour cannot be specified at all, and this is the focus of attempts to formalize the process of creating requirements for new software projects. In situ with the formalization effort is an attempt to help inform non-specialists, particularly non-programmers, who commission software projects without sufficient knowledge of what computer software is in fact capable. Communicating this knowledge is made more difficult by the fact that, as hinted above, even programmers cannot always know in advance what is actually possible for software in advance of trying.

Design

While requirements are meant to specify what a program should do, design is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program.

The purposes of high-level design are as follows. It separates what are considered to be problems of architecture, or overall program concept and structure, from problems of

actual coding, which solve problems of actual data processing. It applies additional constraints to the development process by narrowing the scope of the smaller software components, and thereby—it is hoped—removing variables which could increase the likelihood of programming errors. It provides a program template, including the specification of interfaces, which can be shared by different teams of developers working on disparate parts, such that they can know in advance how each of their contributions will interface with those of the other teams. Finally, and perhaps most controversially, it specifies the program independently of the implementation language or languages, thereby removing language-specific biases and limitations which would otherwise creep into the design, perhaps unwittingly on the part of programmer-designers.

Programming

The history of computer programming language development can often be best understood in the light of attempts to master the complexity of computer programs, which otherwise becomes more difficult to understand in proportion (perhaps exponentially) to the size of the programs. (Another way of looking at the evolution of programming languages is simply as a way of getting the computer to do more and more of the work, but this may be a different way of saying the same thing). Lack of understanding of a program's overall structure and functionality is a sure way to fail to detect errors in the program, and thus the use of better languages should, conversely, reduce the number of errors by enabling a better understanding.

Improvements in languages tend to provide incrementally what software design has attempted to do in one fell swoop: consider the software at ever greater levels of abstraction. Such inventions as statement, sub-routine, file, class, template, library, component and more have allowed the arrangement of a program's parts to be specified using abstractions such as layers, hierarchies and modules, which provide structure at different granularities, so that from any point of view the program's code can be imagined to be orderly and comprehensible.

In addition, improvements in languages have enabled more exact control over the shape and use of data elements, culminating in the abstract data type. These data types can be specified to a very fine degree, including how and when they are accessed, and even the state of the data before and after it is accessed.

Software Build and Deployment

Many programming languages such as C and Java require the program "source code" to be translated in to a form that can be executed by a computer. This translation is done by a program called a compiler. Additional operations may be involved to associate, bind, link or package files together in order to create a usable runtime configuration of the software application. The totality of the compiling and assembly process is generically called "building" the software.

The software build is critical to software quality because if any of the generated files are incorrect the software build is likely to fail. And, if the incorrect version of a program is inadvertently used, then testing can lead to false results.

Software builds are typically done in work area unrelated to the runtime area, such as the application server. For this reason, a deployment step is needed to physically transfer the software build products to the runtime area. The deployment procedure may also involve technical parameters, which, if set incorrectly, can also prevent software testing from beginning. For example, a Java application server may have options for parent-first or parent-last class loading. Using the incorrect parameter can cause the application to fail to execute on the application server.

The technical activities supporting software quality including build, deployment, change control and reporting are collectively known as Software configuration management. A number of software tools have arisen to help meet the challenges of configuration management including file control tools and build control tools.

Testing

Software testing, when done correctly, can increase overall software *quality of conformance* by testing that the product conforms to its requirements. Testing includes, but is not limited to:

1. Unit Testing
2. Functional Testing
3. Regression Testing
4. Performance Testing
5. Failover Testing
6. Usability Testing

A number of agile methodologies use testing early in the development cycle to ensure quality in their products. For example, the test-driven development practice, where tests are written before the code they will test, is used in Extreme Programming to ensure quality.

Runtime

runtime reliability determinations are similar to tests, but go beyond simple confirmation of behaviour to the evaluation of qualities such as performance and interoperability with other code or particular hardware configurations.

Software quality factors

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program. Note that none of these factors are

binary; that is, they are not "either you have it or you don't" traits. Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality. So rather than asking whether a software product "has" factor *x*, ask instead the *degree* to which it does (or does not).

Some software quality factors are listed here:

Understandability
: Clarity of purpose. This goes further than just a statement of purpose; all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account: for instance, if the software product is to be used by software engineers it is not required to be understandable to the layman.

Completeness
: Presence of all constituent parts, with each part fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must also be available.

Conciseness
: Minimization of excessive or redundant information or processing. This is important where memory capacity is limited, and it is generally considered good practice to keep lines of code to a minimum. It can be improved by replacing repeated functionality by one subroutine or function which achieves that functionality. It also applies to documents.

Portability
: Ability to be run well and easily on multiple computer configurations. Portability can mean both between different hardware—such as running on a PC as well as a smartphone—and between different operating systems—such as running on both Mac OS X and GNU/Linux.

Consistency
: Uniformity in notation, symbology, appearance, and terminology within itself.

Maintainability
: Propensity to facilitate updates to satisfy new requirements. Thus the software product that is maintainable should be well-documented, should not be complex, and should have spare capacity for memory, storage and processor utilization and other resources.

Testability
: Disposition to support acceptance criteria and evaluation of performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable; a complex design leads to poor testability.

Usability
: Convenience and practicality of use. This is affected by such things as the human-computer interface. The component of the software that has most impact on this is the user interface (UI), which for best usability is usually graphical (i.e. a GUI).

Reliability
: Ability to be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whatever conditions it finds itself (sometimes termed robustness).

Efficiency
: Fulfillment of purpose without waste of resources, such as memory, space and processor utilization, network bandwidth, time, etc.

Security
: Ability to protect data against unauthorized access and to withstand malicious or inadvertent interference with its operations. Besides the presence of appropriate security mechanisms such as authentication, access control and encryption, security also implies resilience in the face of malicious, intelligent and adaptive attackers.

Measurement of software quality factors

There are varied perspectives within the field on measurement. There are a great many measures that are valued by some professionals—or in some contexts, that are decried as harmful by others. Some believe that quantitative measures of software quality are essential. Others believe that contexts where quantitative measures are useful are quite rare, and so prefer qualitative measures. Several leaders in the field of software testing have written about the difficulty of measuring what we truly want to measure well.

One example of a popular metric is the number of faults encountered in the software. Software that contains few faults is considered by some to have higher quality than software that contains many faults. Questions that can help determine the usefulness of this metric in a particular context include:

1. What constitutes “many faults?” Does this differ depending upon the purpose of the software (e.g., blogging software vs. navigational software)? Does this take into account the size and complexity of the software?
2. Does this account for the importance of the bugs (and the importance to the stakeholders of the people those bugs bug)? Does one try to weight this metric by the severity of the fault, or the incidence of users it affects? If so, how? And if not, how does one know that 100 faults discovered is better than 1000?
3. If the count of faults being discovered is shrinking, how do I know what that means? For example, does that mean that the product is now higher quality than it was before? Or that this is a smaller/less ambitious change than before? Or that fewer tester-hours have gone into the project than before? Or that this project was tested by less skilled testers than before? Or that the team has discovered that fewer faults reported is in their interest?

This last question points to an especially difficult one to manage. All software quality metrics are in some sense measures of human behavior, since humans create software. If

a team discovers that they will benefit from a drop in the number of reported bugs, there is a strong tendency for the team to start reporting fewer defects. That may mean that email begins to circumvent the bug tracking system, or that four or five bugs get lumped into one bug report, or that testers learn not to report minor annoyances. The difficulty is measuring what we mean to measure, without creating incentives for software programmers and testers to consciously or unconsciously "game" the measurements.

Software quality factors cannot be measured because of their vague definitions. It is necessary to find measurements, or metrics, which can be used to quantify them as non-functional requirements. For example, reliability is a software quality factor, but cannot be evaluated in its own right. However, there are related attributes to reliability, which can indeed be measured. Some such attributes are mean time to failure, rate of failure occurrence, and availability of the system. Similarly, an attribute of portability is the number of target-dependent statements in a program.

A scheme that could be used for evaluating software quality factors is given below. For every characteristic, there are a set of questions which are relevant to that characteristic. Some type of scoring formula could be developed based on the answers to these questions, from which a measurement of the characteristic can be obtained.

Understandability

Are variable names descriptive of the physical or functional property represented? Do uniquely recognisable functions contain adequate comments so that their purpose is clear? Are deviations from forward logical flow adequately commented? Are all elements of an array functionally related?...

Completeness

Are all necessary components available? Does any process fail for lack of resources or programming? Are all potential pathways through the code accounted for, including proper error handling?

Conciseness

Is all code reachable? Is any code redundant? How many statements within loops could be placed outside the loop, thus reducing computation time? Are branch decisions too complex?

Portability

Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alphanumeric or special characters been avoided? How much effort would be required to transfer the program from one hardware/software system or environment to another?

Consistency

Is one variable name used to represent different logical or physical entities in the program? Does the program contain only one representation for any given physical or mathematical constant? Are functionally similar arithmetic expressions similarly constructed? Is a consistent scheme used for indentation, nomenclature, the color palette, fonts and other visual elements?

Maintainability

Has some memory capacity been reserved for future expansion? Is the design cohesive—i.e., does each module have distinct, recognizable functionality? Does the software allow for a change in data structures (object-oriented designs are more likely to allow for this)? If the code is procedure-based (rather than object-oriented), is a change likely to require restructuring the main program, or just a module?

Testability

Are complex structures employed in the code? Does the detailed design contain clear pseudo-code? Is the pseudo-code at a higher level of abstraction than the code? If tasking is used in concurrent designs, are schemes available for providing adequate test cases?

Usability

Is a GUI used? Is there adequate on-line help? Is a user manual provided? Are meaningful error messages provided?

Reliability

Are loop indexes range-tested? Is input data checked for range errors? Is divide-by-zero avoided? Is exception handling provided? It is the probability that the software performs its intended functions correctly in a specified period of time under stated operation conditions, but there could also be a problem with the requirement document...

Efficiency

Have functions been optimized for speed? Have repeatedly used blocks of code been formed into subroutines? Has the program been checked for memory leaks or overflow errors?

Security

Does the software protect itself and its data against unauthorized access and use? Does it allow its operator to enforce security policies? Are security mechanisms appropriate,

adequate and correctly implemented? Can the software withstand attacks that can be anticipated in its intended environment?

User's perspective

In addition to the technical qualities of software, the end user's experience also determines the quality of software. This aspect of software quality is called usability. It is hard to quantify the usability of a given software product. Some important questions to be asked are:

- Is the user interface intuitive (self-explanatory/self-documenting)?
- Is it easy to perform simple operations?
- Is it feasible to perform complex operations?
- Does the software give sensible error messages?
- Do widgets behave as expected?
- Is the software well documented?
- Is the user interface responsive or too slow?

Also, the availability of (free or paid) support may factor into the usability of the software.

Chapter 13

Anti-Pattern

In software engineering, an **anti-pattern** (or **antipattern**) is a pattern that may be commonly used but is ineffective and/or counterproductive in practice.

The term was coined in 1995 by Andrew Koenig, inspired by Gang of Four's book *Design Patterns*, which developed the concept of design patterns in the software field. The term was widely popularized three years later by the book *AntiPatterns*, which extended the use of the term beyond the field of software design and into general social interaction. According to the authors of the latter, there must be at least two key elements present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea:

- Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results, and
- A refactored solution exists that is clearly documented, proven in actual practice and repeatable.

Often pejoratively named with clever oxymoronic neologisms, many anti-pattern ideas amount to little more than mistakes, rants, unsolvable problems, or bad practices to be avoided if possible. Sometimes called *pitfalls* or *dark patterns*, this informal use of the term has come to refer to classes of commonly reinvented bad solutions to problems. Thus, many candidate anti-patterns under debate would not be formally considered anti-patterns.

By formally describing repeated mistakes, one can recognize the forces that lead to their repetition and learn how others have refactored themselves out of these broken patterns.

Known anti-patterns

Organizational anti-patterns

- Analysis paralysis: Devoting disproportionate effort to the analysis phase of a project
- Cash cow: A profitable legacy product that often leads to complacency about new products

- Design by committee: The result of having many contributors to a design, but no unifying vision
- Escalation of commitment: Failing to revoke a decision when it proves wrong
- Management by perkele: Authoritarian style of management with no tolerance of dissent
- Matrix Management: Unfocused organizational structure that results in divided loyalties and lack of direction
- Moral hazard: Insulating a decision-maker from the consequences of his or her decision
- Mushroom management: Keeping employees uninformed and misinformed (kept in the dark and fed manure), let to stew, and finally canned.
- Stovepipe or Silos: A structure that supports mostly up-down flow of data but inhibits cross organizational communication
- Vendor lock-in: Making a system excessively dependent on an externally supplied component

Project management anti-patterns

- Death march: Everyone knows that the project is going to be a disaster – except the CEO -- so the truth is hidden to prevent immediate cancellation of the project - (although the CEO often knows and does it anyway to maximise profit). However, the truth remains hidden and the project is artificially kept alive until the Day Zero finally comes ("Big Bang"). Alternative definition: Employees are pressured to work late nights and weekends on a project with an unreasonable deadline.
- Groupthink: During groupthink, members of the group avoid promoting viewpoints outside the comfort zone of consensus thinking
- Smoke and mirrors: Demonstrating how unimplemented functions will appear
- Software bloat: Allowing successive versions of a system to demand ever more resources
- Waterfall model: An older method of software development that inadequately deals with unanticipated change

Analysis anti-patterns

- Bystander apathy: When a requirement or design decision is wrong, but the people who notice this do nothing because it affects a larger number of people

Software design anti-patterns

- Abstraction inversion: Not exposing implemented functionality required by users, so that they re-implement it using higher level functions
- Ambiguous viewpoint: Presenting a model (usually Object-oriented analysis and design (OOAD)) without specifying its viewpoint
- Big ball of mud: A system with no recognizable structure

- Database-as-IPC: Using a database as the message queue for routine interprocess communication where a much more lightweight mechanism would be suitable
- Gold plating: Continuing to work on a task or project well past the point at which extra effort is adding value
- Inner-platform effect: A system so customizable as to become a poor replica of the software development platform
- Input kludge: Failing to specify and implement the handling of possibly invalid input
- Interface bloat: Making an interface so powerful that it is extremely difficult to implement
- Magic pushbutton: Coding implementation logic directly within interface code, without using abstraction
- Race hazard: Failing to see the consequence of different orders of events
- Stovepipe system: A barely maintainable assemblage of ill-related components

Object-oriented design anti-patterns

- Anemic Domain Model: The use of domain model without any business logic. The domain model's objects cannot guarantee their correctness at any moment, because their validation and mutation logic is placed somewhere outside (most likely in multiple places).
- BaseBean: Inheriting functionality from a utility class rather than delegating to it
- Call super: Requiring subclasses to call a superclass's overridden method
- Circle-ellipse problem: Subtyping variable-types on the basis of value-subtypes
- Circular dependency: Introducing unnecessary direct or indirect mutual dependencies between objects or software modules
- Constant interface: Using interfaces to define constants
- God object: Concentrating too many functions in a single part of the design (class)
- Object cesspool: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- Object orgy: Failing to properly encapsulate objects permitting unrestricted access to their internals
- Poltergeists: Objects whose sole purpose is to pass information to another object
- Sequential coupling: A class that requires its methods to be called in a particular order
- Yo-yo problem: A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation

Programming anti-patterns

- Accidental complexity: Introducing unnecessary complexity into a solution
- Action at a distance: Unexpected interaction between widely separated parts of a system
- Blind faith: Lack of checking of (a) the correctness of a bug fix or (b) the result of a subroutine

- Boat anchor: Retaining a part of a system that no longer has any use
- Busy spin: Consuming CPU while waiting for something to happen, usually by repeated checking instead of messaging
- Caching failure: Forgetting to reset an error flag when an error has been corrected
- Cargo cult programming: Using patterns and methods without understanding why
- Coding by exception: Adding new code to handle each special case as it is recognized
- Error hiding: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message
- Hard code: Embedding assumptions about the environment of a system in its implementation
- Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences
- Loop-switch sequence: Encoding a set of sequential steps using a switch within a loop statement
- Magic numbers: Including unexplained numbers in algorithms
- Magic strings: Including literal strings in code, for comparisons, as event types etc.
- Soft code: Storing business logic in configuration files rather than source code
- Spaghetti code: Programs whose structure is barely comprehensible, especially because of misuse of code structures

Methodological anti-patterns

- Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
- Golden hammer: Assuming that a favorite solution is universally applicable (See: Silver Bullet)
- Improbability factor: Assuming that it is improbable that a known error will occur
- Not Invented Here (NIH) syndrome: The tendency towards *reinventing the wheel* (Failing to adopt an existing, adequate solution)
- Premature optimization: Coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency
- Programming by permutation (or "programming by accident"): Trying to approach a solution by successively modifying the code to see if it works
- Reinventing the wheel: Failing to adopt an existing, adequate solution
- Reinventing the square wheel: Failing to adopt an existing solution and instead adopting a custom solution which performs much worse than the existing one
- Silver bullet: Assuming that a favorite technical solution can solve a larger process or problem
- Tester Driven Development: Software projects in which new requirements are specified in bug reports

Configuration management anti-patterns

- Dependency hell: Problems with versions of required products

- DLL hell: Inadequate management of dynamic-link libraries (DLLs), specifically on Microsoft Windows
- Extension conflict: Problems with different extensions to pre-Mac OS X versions of the Mac OS attempting to patch the same parts of the operating system
- JAR hell: Overutilization of the multiple JAR files, usually causing versioning and location problems because of misunderstanding of the Java class loading model

Chapter 14

Software Bug

A **software bug** is the common term used to describe an error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result, or causes it to behave in unintended ways. Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be *buggy*. Reports detailing bugs in a program are commonly known as bug reports, fault reports, problem reports, trouble reports, change requests, and so forth.

Effects

Bugs trigger Type I and type II errors that can in turn have a wide variety of ripple effects, with varying levels of inconvenience to the user of the program. Some bugs have only a subtle effect on the program's functionality, and may thus lie undetected for a long time. More serious bugs may cause the program to crash or freeze leading to a denial of service. Others qualify as security bugs and might for example enable a malicious user to bypass access controls in order to obtain unauthorized privileges.

The results of bugs may be extremely serious. Bugs in the code controlling the Therac-25 radiation therapy machine were directly responsible for some patient deaths in the 1980s. In 1996, the European Space Agency's US$1 billion prototype Ariane 5 rocket was destroyed less than a minute after launch, due to a bug in the on-board guidance computer program. In June 1994, a Royal Air Force Chinook crashed into the Mull of Kintyre, killing 29. This was initially dismissed as pilot error, but an investigation by *Computer Weekly* uncovered sufficient evidence to convince a House of Lords inquiry that it may have been caused by a software bug in the aircraft's engine control computer.

In 2002, a study commissioned by the US Department of Commerce' National Institute of Standards and Technology concluded that *software bugs, or errors, are so prevalent and so detrimental that they cost the US economy an estimated $59 billion annually, or about 0.6 percent of the gross domestic product*.

Etymology

The concept that software might contain errors dates back to 1843 in Ada Byron's notes on the analytical engine in which she speaks of the difficulty of preparing program 'cards' for Charles Babbage's Analytical engine:

> “ ...an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders. ”

Use of the term "bug" to describe inexplicable defects has been a part of engineering jargon for many decades and predates computers and computer software; it may have originally been used in hardware engineering to describe mechanical malfunctions. For instance, Thomas Edison wrote the following words in a letter to an associate in 1878:

> “ It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and *[it is]* then that 'Bugs' — as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached. ”

Problems with radar electronics during World War II were referred to as *bug*s (or glitches) and there is additional evidence that the usage dates back much earlier. Baffle Ball, the first mechanical pinball game, was advertised as being "free of bugs" in 1931.

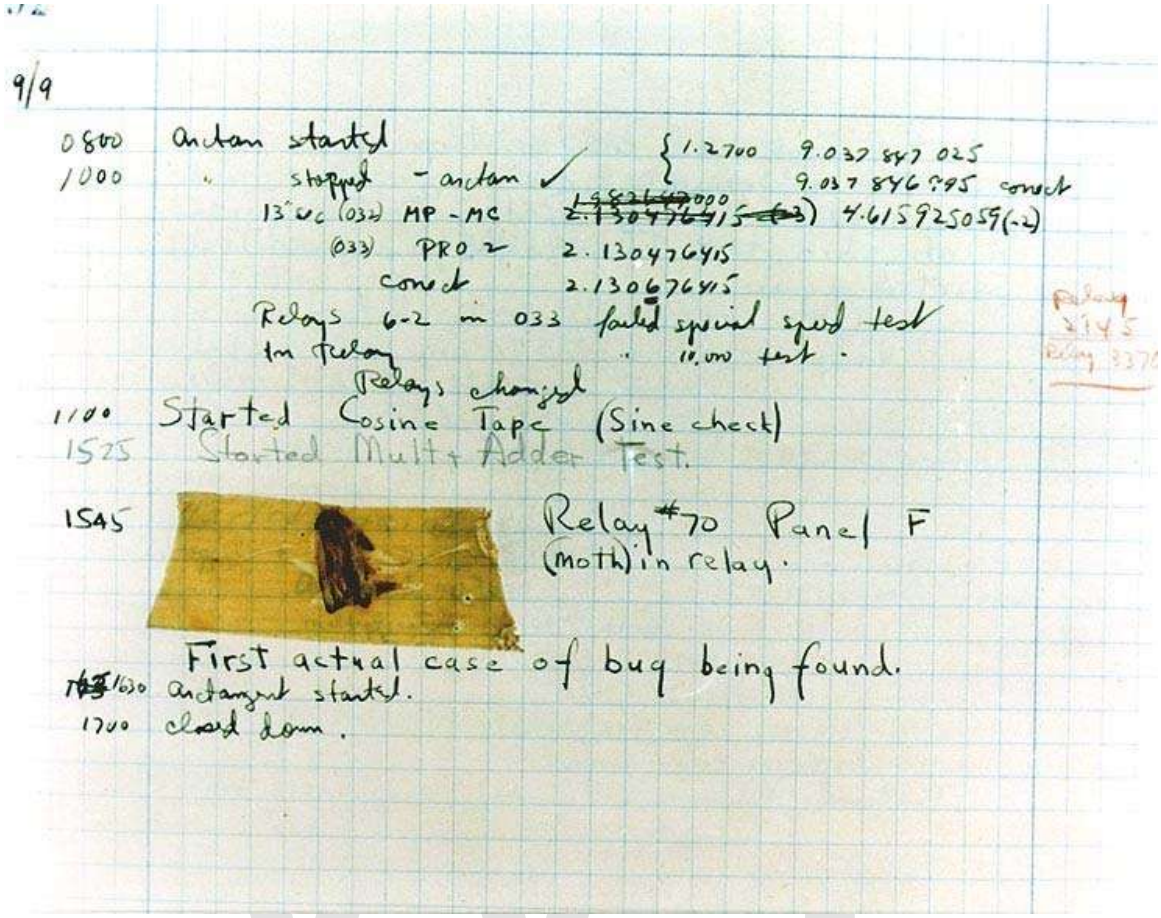


Photo of what is possibly the first real bug found in a computer

The invention of the term "bug" is often erroneously attributed to Grace Hopper, who publicized the cause of a malfunction in an early electromechanical computer. A typical version of the story is given by this quote:

> In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term *bug*. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitch's [*sic*] in a program a *bug*.

Hopper was not actually the one who found the insect, as she readily acknowledged. The date in the log book was 9 September 1947, although sometimes erroneously reported as 1945. The operators who did find it, including William "Bill" Burke, later of the Naval Weapons Laboratory, Dahlgren, Virginia, were familiar with the engineering term and, amused, kept the insect with the notation "First actual case of bug being found." Hopper loved to recount the story. This log book is on display in the Smithsonian National Museum of American History, complete with moth attached.

While it is certain that the Harvard Mark II operators did not coin the term "bug", it has been suggested that they did coin the related term, "debug". Even this is unlikely, since the Oxford English Dictionary entry for "debug" contains a use of "debugging" in the context of air-plane engines in 1945.

Prevention

Bugs are a consequence of the nature of human factors in the programming task. They arise from oversights or mutual misunderstandings made by a software team during specification, design, coding, data entry and documentation. For example: In creating a relatively simple program to sort a list of words into alphabetical order, one's design might fail to consider what should happen when a word contains a hyphen. Perhaps, when converting the abstract design into the chosen programming language, one might inadvertently create an off-by-one error and fail to sort the last word in the list. Finally, when typing the resulting program into the computer, one might accidentally type a '<' where a '>' was intended, perhaps resulting in the words being sorted into reverse alphabetical order. More complex bugs can arise from unintended interactions between different parts of a computer program. This frequently occurs because computer programs can be complex—millions of lines long in some cases—often having been programmed by many people over a great length of time, so that programmers are unable to mentally track every possible way in which parts can interact. Another category of bug called a *race condition* comes about either when a process is running in more than one thread or two or more processes run simultaneously, and the exact order of execution of the critical sequences of code have not been properly synchronized.

The software industry has put much effort into finding methods for preventing programmers from inadvertently introducing bugs while writing software. These include:

Programming style
: While typos in the program code are often caught by the compiler, a bug usually appears when the programmer makes a logic error. Various innovations in programming style and defensive programming are designed to make these bugs less likely, or easier to spot. In some programming languages, so-called typos, especially of symbols or logical/mathematical operators, actually represent logic errors, since the mistyped constructs are accepted by the compiler with a meaning other than that which the programmer intended.

Programming techniques
: Bugs often create inconsistencies in the internal data of a running program. Programs can be written to check the consistency of their own internal data while running. If an inconsistency is encountered, the program can immediately halt, so that the bug can be located and fixed. Alternatively, the program can simply inform the user, attempt to correct the inconsistency, and continue running.

Development methodologies
: There are several schemes for managing programmer activity, so that fewer bugs are produced. Many of these fall under the discipline of software engineering (which addresses software design issues as well). For example, formal program

specifications are used to state the exact behavior of programs, so that design bugs can be eliminated. Unfortunately, formal specifications are impractical or impossible for anything but the shortest programs, because of problems of combinatorial explosion and indeterminacy.

Programming language support

Programming languages often include features which help programmers prevent bugs, such as static type systems, restricted name spaces and modular programming, among others. For example, when a programmer writes (pseudocode) `LET REAL_VALUE PI = "THREE AND A BIT"`, although this may be syntactically correct, the code fails a type check. Depending on the language and implementation, this may be caught by the compiler or at runtime. In addition, many recently-invented languages have deliberately excluded features which can easily lead to bugs, at the expense of making code slower than it need be: the general principle being that, because of Moore's law, computers get faster and software engineers get slower; it is *almost always* better to write simpler, slower code than "clever", inscrutable code, especially considering that maintenance cost is considerable. For example, the Java programming language does not support pointer arithmetic; implementations of some languages such as Pascal and scripting languages often have runtime bounds checking of arrays, at least in a debugging build.

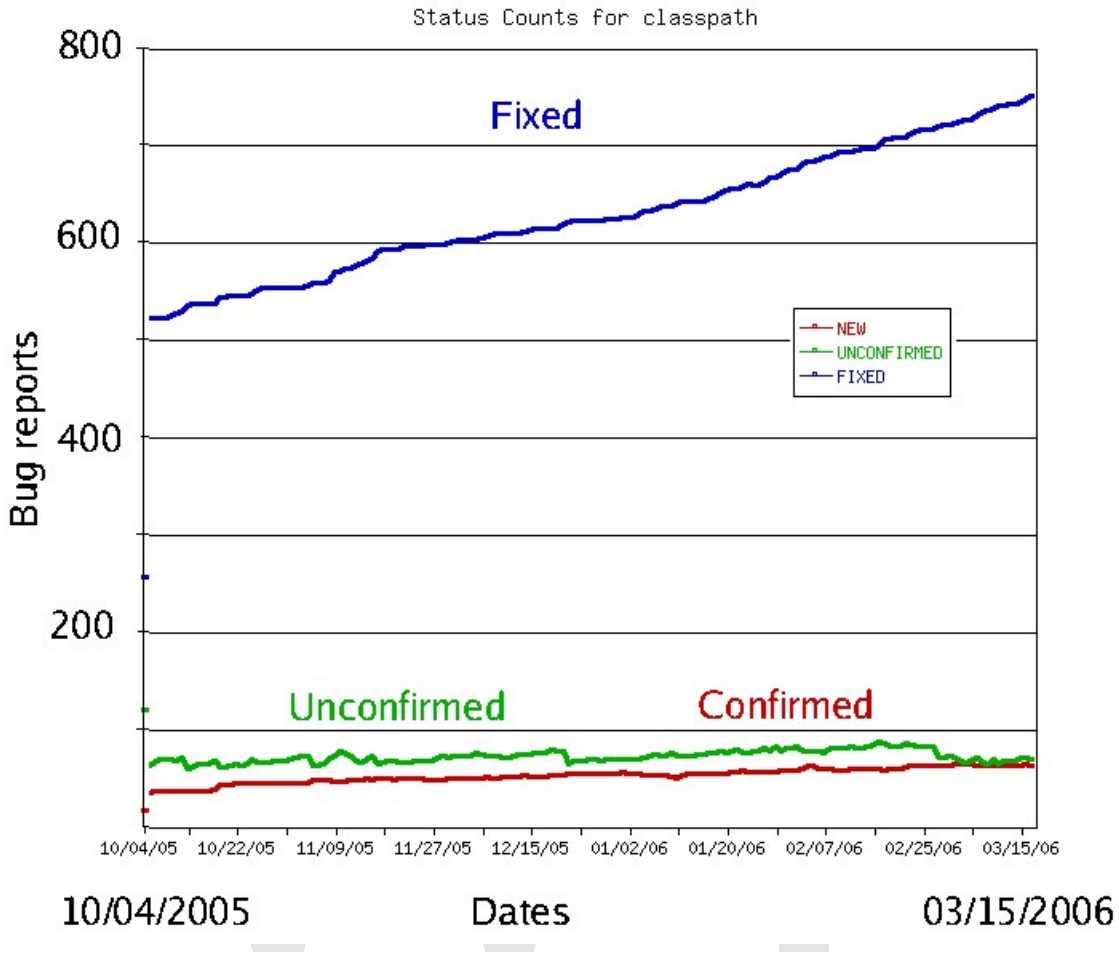
Code analysis

Tools for code analysis help developers by inspecting the program text beyond the compiler's capabilities to spot potential problems. Although in general the problem of finding all programming errors given a specification is not solvable, these tools exploit the fact that human programmers tend to make the same kinds of mistakes when writing software.

Instrumentation

Tools to monitor the performance of the software as it is running, either specifically to find problems such as bottlenecks or to give assurance as to correct working, may be embedded in the code explicitly (perhaps as simple as a statement saying `PRINT "I AM HERE"`), or provided as tools. It is often a surprise to find where most of the time is taken by a piece of code, and this removal of assumptions might cause the code to be rewritten.

Debugging



The typical bug history (GNU Classpath project data). A new bug submitted by the user is *unconfirmed.* Once it has been reproduced by a developer, it is a *confirmed* bug. The confirmed bugs are later *fixed*. Bugs belonging to other categories (unreproducible, will not be fixed, etc.) are usually in the minority

Finding and fixing bugs, or "debugging", has always been a major part of computer programming. Maurice Wilkes, an early computing pioneer, described his realization in the late 1940s that much of the rest of his life would be spent finding mistakes in his own programs. As computer programs grow more complex, bugs become more common and difficult to fix. Often programmers spend more time and effort finding and fixing bugs than writing new code. Software testers are professionals whose primary task is to find bugs, or write code to support testing. On some projects, more resources can be spent on testing than in developing the program.

Usually, the most difficult part of debugging is finding the bug in the source code. Once it is found, correcting it is usually relatively easy. Programs known as debuggers exist to help programmers locate bugs by executing code line by line, watching variable values, and other features to observe program behavior. Without a debugger, code can be added

so that messages or values can be written to a console (for example with *printf* in the C programming language) or to a window or log file to trace program execution or show values.

However, even with the aid of a debugger, locating bugs is something of an art. It is not uncommon for a bug in one section of a program to cause failures in a completely different section, thus making it especially difficult to track (for example, an error in a graphics rendering routine causing a file I/O routine to fail), in an apparently unrelated part of the system.

Sometimes, a bug is not an isolated flaw, but represents an error of thinking or planning on the part of the programmer. Such *logic errors* require a section of the program to be overhauled or rewritten. As a part of Code review, stepping through the code modelling the execution process in one's head or on paper can often find these errors without ever needing to reproduce the bug as such, if it can be shown there is some faulty logic in its implementation.

But more typically, the first step in locating a bug is to reproduce it reliably. Once the bug is reproduced, the programmer can use a debugger or some other tool to monitor the execution of the program in the faulty region, and find the point at which the program went astray.

It is not always easy to reproduce bugs. Some are triggered by inputs to the program which may be difficult for the programmer to re-create. One cause of the Therac-25 radiation machine deaths was a bug (specifically, a race condition) that occurred only when the machine operator very rapidly entered a treatment plan; it took days of practice to become able to do this, so the bug did not manifest in testing or when the manufacturer attempted to duplicate it. Other bugs may disappear when the program is run with a debugger; these are heisenbugs (humorously named after the Heisenberg uncertainty principle.)

Debugging is still a tedious task requiring considerable effort. Since the 1990s, particularly following the Ariane 5 Flight 501 disaster, there has been a renewed interest in the development of effective automated aids to debugging. For instance, methods of static code analysis by abstract interpretation have already made significant achievements, while still remaining much of a work in progress.

As with any creative act, sometimes a flash of inspiration will show a solution, but this is rare and, by definition, cannot be relied on.

There are also classes of bugs that have nothing to do with the code itself. If, for example, one relies on faulty documentation or hardware, the code may be written perfectly properly to what the documentation says, but the bug truly lies in the documentation or hardware, not the code. However, it is common to change the code instead of the other parts of the system, as the cost and time to change it is generally less. Embedded systems frequently have workarounds for hardware bugs, since to make a new version of a ROM

is much cheaper than remanufacturing the hardware, especially if they are commodity items.

Bug management

It is common practice for software to be released with known bugs that are considered non-critical, that is, that do not affect most users' main experience with the product. While software products may, by definition, contain any number of unknown bugs, measurements during testing can provide an estimate of the number of likely bugs remaining; this becomes more reliable the longer a product is tested and developed ("if we had 200 bugs last week, we should have 100 this week"). Most big software projects maintain two lists of "known bugs"— those known to the software team, and those to be told to users. This is not dissimulation, but users are not concerned with the internal workings of the product. The second list informs users about bugs that are not fixed in the current release, or not fixed at all, and a workaround may be offered.

There are various reasons for not fixing bugs:

- The developers often don't have time or it is not economical to fix all non-severe bugs.
- The bug could be fixed in a new version or patch that is not yet released.
- The changes to the code required to fix the bug could be large, expensive, or delay finishing the project.
- Even seemingly simple fixes bring the chance of introducing new unknown bugs into the system. At the end of a test/fix cycle some managers may only allow the most critical bugs to be fixed.
- Users may be relying on the undocumented, buggy behavior, especially if scripts or macros rely on a behavior; it may introduce a breaking change.
- It's "not a bug". A misunderstanding has arisen between expected and provided behavior

Given the above, it is often considered impossible to write completely bug-free software of any real complexity. So bugs are categorized by severity, and low-severity non-critical bugs are tolerated, as they do not affect the proper operation of the system for most users. NASA's SATC managed to reduce the number of errors to fewer than 0.1 per 1000 lines of code (SLOC) but this was not felt to be feasible for any real world projects.

The severity of a bug is not the same as its importance for fixing, and the two should be measured and managed separately. On a Microsoft Windows system a blue screen of death is rather severe, but if it only occurs in extreme circumstances, especially if they are well diagnosed and avoidable, it may be less important to fix than an icon not representing its function well, which though purely aesthetic may confuse thousands of users every single day. This balance, of course, depends on many factors; expert users have different expectations from novices, a niche market is different from a general consumer market, and so on. To better achieve this balance, some software developers use a formalized *bug triage* process (borrowing the medical term), in which each new

bug is assigned a priority based on its severity, frequency, risk, and other predetermined factors.

A school of thought popularized by Eric S. Raymond as Linus's Law says that popular open-source software has more chance of having few or no bugs than other software, because "given enough eyeballs, all bugs are shallow". This assertion has been disputed, however: computer security specialist Elias Levy wrote that "it is easy to hide vulnerabilities in complex, little understood and undocumented source code," because, "even if people are reviewing the code, that doesn't mean they're qualified to do so."

Like any other part of engineering management, bug management must be conducted carefully and intelligently because "what gets measured gets done" and managing purely by bug counts can have unintended consequences. If, for example, developers are rewarded by the number of bugs they fix, they will naturally fix the easiest bugs first—leaving the hardest, and probably most risky or critical, to the last possible moment ("I only have one bug on my list but it says "Make sun rise in West"). If the management ethos is to reward the number of bugs fixed, then some developers may quickly write sloppy code knowing they can fix the bugs later and be rewarded for it, whereas careful, perhaps "slower" developers do not get rewarded for the bugs that were never there.

Security vulnerabilities

Malicious software may attempt to exploit known vulnerabilities in a system — which may or may not be bugs. Viruses are not bugs in themselves — they are typically programs that are doing precisely what they were designed to do. However, viruses are occasionally referred to as such in the popular press.

Common types of computer bugs

- Conceptual error (code is syntactically correct, but the programmer or designer intended it to do something else)

Arithmetic bugs

- Division by zero
- Arithmetic overflow or underflow
- Loss of arithmetic precision due to rounding or numerically unstable algorithms

Logic bugs

- Infinite loops and infinite recursion
- Off by one error, counting one too many or too few when looping

Syntax bugs

- Use of the wrong operator, such as performing assignment instead of equality test. In simple cases often warned by the compiler; in many languages, deliberately guarded against by language syntax

Resource bugs

- Null pointer dereference
- Using an uninitialized variable
- Using an otherwise valid instruction on the wrong data type
- Access violations
- Resource leaks, where a finite system resource such as memory or file handles are exhausted by repeated allocation without release.
- Buffer overflow, in which a program tries to store data past the end of allocated storage. This may or may not lead to an access violation or storage violation. These bugs can form a security vulnerability.
- Excessive recursion which though logically valid causes stack overflow

Multi-threading programming bugs

- Deadlock
- Race condition
- Concurrency errors in critical sections, mutual exclusions and other features of concurrent processing. Time-of-check-to-time-of-use (TOCTOU) is a form of unprotected critical section.

Interfacing bugs

- Incorrect API usage
- Incorrect protocol implementation
- Incorrect hardware handling

Teamworking bugs

- Unpropagated updates; e.g. programmer changes "myAdd" but forgets to change "mySubtract", which uses the same algorithm. These errors are mitigated by the Don't Repeat Yourself philosophy.
- Comments out of date or incorrect: many programmers assume the comments accurately describe the code
- Differences between documentation and the actual product

Chapter 15

Fault-Tolerant System and Feature Interaction Problem

Fault-tolerant system

Fault-tolerance or **graceful degradation** is the property that enables a system (often computer-based) to continue operating properly in the event of the failure of (or one or more faults within) some of its components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naïvely-designed system in which even a small failure can cause total breakdown. Fault-tolerance is particularly sought-after in high-availability or life-critical systems.

Fault-tolerance is not just a property of individual machines; it may also characterise the rules by which they interact. For example, the Transmission Control Protocol (TCP) is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communications links which are imperfect or overloaded. It does this by requiring the endpoints of the communication to *expect* packet loss, duplication, reordering and corruption, so that these conditions do not damage data integrity, and only reduce throughput by a proportional amount.

An example of graceful degradation by design in an image with transparency. The top two images are each the result of viewing the composite image in a viewer that recognises transparency. The bottom two images are the result in a viewer with no support for transparency. Because the transparency mask (centre bottom) is discarded, only the overlay (centre top) remains; the image on the left has been designed to degrade gracefully, hence is still meaningful without its transparency information.

Data formats may also be designed to degrade gracefully. HTML for example, is designed to be forward compatible, allowing new HTML entities to be ignored by Web browsers which do not understand them without causing the document to be unusable.

Recovery from errors in fault-tolerant systems can be characterised as either **roll-forward** or **roll-back**. When the system detects that it has made an error, roll-forward recovery takes the system state at that time and corrects it, to be able to move forward. Roll-back recovery reverts the system state back to some earlier, correct version, for example using checkpointing, and moves forward from there. Roll-back recovery requires that the operations between the checkpoint and the detected erroneous state can be made idempotent. Some systems make use of both roll-forward and roll-back recovery for different errors or different parts of one error.

Within the scope of an *individual* system, fault-tolerance can be achieved by anticipating exceptional conditions and building the system to cope with them, and, in general, aiming for self-stabilization so that the system converges towards an error-free state. However, if the consequences of a system failure are catastrophic, or the cost of making it sufficiently reliable is very high, a better solution may be to use some form of duplication. In any case, if the consequence of a system failure is catastrophic, the system must be able to use reversion to fall back to a safe mode. This is similar to roll-back recovery but can be a human action if humans are present in the loop.

Fault tolerance requirements

The basic characteristics of fault tolerance require:

1. No single point of failure
2. Fault isolation to the failing component
3. Fault containment to prevent propagation of the failure
4. Availability of reversion modes

In addition, fault tolerant systems are characterized in terms of both planned service outages and unplanned service outages. These are usually measured at the application level and not just at a hardware level. The figure of merit is called availability and is expressed as a percentage. For example, a five nines system would statistically provide 99.999% availability.

Fault-tolerant systems are typically based on the concept of redundancy.

Fault-tolerance by replication

Spare components addresses the first fundamental characteristic of fault-tolerance in three ways:

- Replication: Providing multiple identical instances of the same system or subsystem, directing tasks or requests to all of them in parallel, and choosing the correct result on the basis of a quorum;
- Redundancy: Providing multiple identical instances of the same system and switching to one of the remaining instances in case of a failure (failover);
- Diversity: Providing multiple *different* implementations of the same specification, and using them like replicated systems to cope with errors in a specific implementation.

All implementations of RAID, redundant array of independent disks, except RAID 0 are examples of a fault-tolerant storage device that uses data redundancy.

A lockstep fault-tolerant machine uses replicated elements operating in parallel. At any time, all the replications of each element should be in the same state. The same inputs are provided to each replication, and the same outputs are expected. The outputs of the replications are compared using a voting circuit. A machine with two replications of each element is termed Dual Modular Redundant (DMR). The voting circuit can then only detect a mismatch and recovery relies on other methods. A machine with three replications of each element is termed Triple Modular Redundancy (TMR). The voting circuit can determine which replication is in error when a two-to-one vote is observed. In this case, the voting circuit can output the correct result, and discard the erroneous version. After this, the internal state of the erroneous replication is assumed to be different from that of the other two, and the voting circuit can switch to a DMR mode. This model can be applied to any larger number of replications.

Lockstep fault tolerant machines are most easily made fully synchronous, with each gate of each replication making the same state transition on the same edge of the clock, and the clocks to the replications being exactly in phase. However, it is possible to build lockstep systems without this requirement.

Bringing the replications into synchrony requires making their internal stored states the same. They can be started from a fixed initial state, such as the reset state. Alternatively, the internal state of one replica can be copied to another replica.

One variant of DMR is **pair-and-spare**. Two replicated elements operate in lockstep as a pair, with a voting circuit that detects any mismatch between their operations and outputs a signal indicating that there is an error. Another pair operates exactly the same way. A final circuit selects the output of the pair that does not proclaim that it is in error. Pair-and-spare requires four replicas rather than the three of TMR, but has been used commercially.

No single point of repair

If a system experiences a failure, it must continue to operate without interruption during the repair process.

Fault isolation to the failing component

When a failure occurs, the system must be able to isolate the failure to the offending component. This requires the addition of dedicated failure detection mechanisms that exist only for the purpose of fault isolation.

Recovery from a fault condition requires classifying the fault or failing component. The National Institute of Standards and Technology (NIST) categorizes faults based on Locality, Cause, Duration and Effect.

Fault containment

Some failure mechanisms can cause a system to fail by propagating the failure to the rest of the system. An example of this kind of failure is the "Rogue transmitter" which can swamp legitimate communication in a system and cause overall system failure. Mechanisms that isolate a rogue transmitter or failing component to protect the system are required.

Feature interaction problem

Feature interaction is a software engineering concept. It occurs when the integration of two **features** would modify the **behavior** of one or both features.

The term *feature* is used to denote a unit of functionality of a software application. Similar to many concepts in computer science, the term can be used at different levels of abstraction. For example, the plain old telephone service (POTS) is a telephony application feature at one level, but itself is composed of originating features and terminating features. The originating features may in turn include the provide dial tone feature, digit collection feature and so on.

This definition of *feature interaction* allows one to focus on certain behavior of the interacting features such as how their response time may be changed given the integration. Many researchers in the field consider problems that arise due to change in the execution *behavior* of the interacting features. Under that context, the *behavior* of a feature is defined by its execution flow and output for a given input. In other words, the interaction changes the execution flow and output of the interacting features for a given input.

Example

In the context of telephony, a telephone line (the system) typically offers a set of features that include call forwarding and call waiting. Call waiting allows one call to be suspended while a second call is answered, while call forwarding enables a customer to specify a secondary phone number to which additional calls will be forwarded in the event that the customer is already using the phone.

To illustrate the example, we consider a telephone line provided to a customer, and we assume that both call forwarding and call waiting are enabled on the line. When a first call arrives on the line, the phone rings and is answered. Since neither feature is activated by the first call, there is no noticeable problem. When a second call arrives before the first has terminated, the telephone system has a decision to make: whether the call should be forwarded to the secondary number (call forwarding) or the person who answered the first call should be notified that another call has arrived (call waiting). Since this decision has no obvious correct answer, the optimal answer depends on the needs of the customer. This *feature interaction* is a specific example of a general and common problem that has become prevalent due to increasing system complexity.

In this situation, it is possible that the system's decision will be made in a non-deterministic fashion due to race conditions and other design factors. The consequences of feature interactions can range from minor irritations to life-threatening software failures, and therefore there is ongoing research that aims to find ways of *detecting* as well as *resolving* feature interactions.

Chapter 16

Independent Software Verification and Validation & Software Assurance

Independent software verification and validation

ISVV stands for **Independent Software Verification and Validation**. ISVV is targeted at safety-critical software systems and aims to increase the quality of software products, thereby reducing risks and costs through the operational life of the software. ISVV provides assurance that software performs to the specified level of confidence and within its designed parameters and defined requirements.

ISVV activities are performed by independent engineering teams, not involved in the software development process, to assess the processes and the resulting products. The ISVV team independency is performed at three different levels: financial, managerial and technical.

ISVV goes far beyond “traditional” verification and validation techniques, applied by development teams. While the latter aim to ensure that the software performs well against the nominal requirements, ISVV is focused on non-functional requirements such as robustness and reliability, and on conditions that can lead the software to fail. ISVV results and findings are fed back to the development teams for correction and improvement.

ISVV History

ISVV derives from the application of IV&V (Independent Verification and Validation) to the software. Early ISVV application (as known today) dates back to the early 1970s when the U.S. Army sponsored the first significant program related to IV&V for the Safeguard Anti-Ballistic Missile System.

By the end of the 1970s IV&V was rapidly becoming popular. The constant increase in complexity, size and importance of the software lead to an increasing demand on IV&V applied to software (ISVV).

Meanwhile IV&V (and ISVV for software systems) gets consolidated and is now widely used by organisations such as the DoD, FAA, NASA and ESA. IV&V is mentioned in [DO-178B], [ISO/IEC 12207] and formalised in [IEEE 1012].

Initially in 2004-2005, a European consortium led by the European Space Agency, and composed by DNV(N), Critical Software SA(P), Terma(DK) and CODA Scisys(UK) created the first version of a guide devoted to ISVV, called "ESA Guide for Independent Verification and Validation" with support from other organizations, eg SoftWcare SL (E) (), etc.

In 2008 the European Space Agency released a second version, being SoftWcare SL was the supporting editor having received inputs from many different European Space ISVV stakeholders. This guide covers the methodologies applicable to all the software engineering phases in what concerns ISVV.

ISVV Methodology

ISVV is usually composed by five principal phases, these phases can be executed sequentially or as results of a tailoring process.

ISVV Planning

- Planning of ISVV Activities
- System Criticality Analysis: Identification of Critical Components through a set of RAMS activities (Value for Money)
- Selection of the appropriate Methods and Tools

Requirements Verification

- Verification for: Completeness, Correctness, Testability

Design Verification

- Design adequacy and conformance to Software Requirements and Interfaces
- Internal and External Consistency
- Verification of Feasibility and Maintenance

Code Verification

- Verification for: Completeness, Correctness, Consistency
- Code Metrics Analysis
- Coding Standards Compliance Verification

Validation

- Identification of unstable components/functionalities
- Validation focused on Error-Handling: complementary (not concurrent!) validation regarding the one performed by the Development team (More for the Money, More for the Time)
- Compliance with Software and System Requirements
- Black box testing and White box testing techniques
- Experience based techniques

Software assurance

Software Assurance (SwA) is defined as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle, and that the software functions in the intended manner."

Alternate definitions

Department of Homeland Security (DHS)

According to the DHS, software assurance addresses:

- Trustworthiness - No exploitable vulnerabilities exist, either maliciously or unintentionally inserted;
- Predictable Execution - Justifiable confidence that software, when executed, functions as intended;
- Conformance - Planned and systematic set of multi-disciplinary activities that ensure software processes and products conform to requirements, standards/ procedures.

Contributing SwA disciplines, articulated in Bodies of Knowledge and Core Competencies: Software Engineering, Systems Engineering, Information Systems Security Engineering, Information Assurance, Test and Evaluation, Safety, Security, Project Management, and Software Acquisition.

Software Assurance is a strategic initiative of the U.S. Department of Homeland Security (DHS) to promote integrity, security, and reliability in software. The SwA Program is based upon the National Strategy to Secure Cyberspace - Action/Recommendation 2-14:

"DHS will facilitate a national public-private effort to promulgate best practices and methodologies that promote integrity, security, and reliability in software code

development, including processes and procedures that diminish the possibilities of erroneous code, malicious code, or trap doors that could be introduced during development."

United States Department of Defense (DoD)

According to the DoD, software assurance relates to "the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software."

Software Assurance Metrics and Tool Evaluation (SAMATE) project

According to the NIST SAMATE project, software assurance is "the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures to help achieve:

- Trustworthiness - No exploitable vulnerabilities exist, either of malicious or unintentional origin, and
- Predictable Execution - Justifiable confidence that software, when executed, functions as intended."

National Aeronautics and Space Administration (NASA)

According to NASA, Software Assurance is a "planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. It includes the disciplines of Quality Assurance, Quality Engineering, Verification and Validation, Nonconformance Reporting and Corrective Action, Safety Assurance, and Security Assurance and their application during a software life cycle." The NASA Software Assurance Standard also states: "The application of these disciplines during a software development life cycle is called Software Assurance."

Object Management Group (OMG)

According to the OMG, Software Assurance is "justifiable trustworthiness in meeting established business and security objectives."

OMG's SwA Special Interest Group (SIG), works with Platform and Domain Task Forces and other software industry entities and groups external to the OMG, to coordinate the establishment of a common framework for analysis and exchange of information related to software trustworthiness by facilitating the development of a specification for a Software Assurance Framework that will:

- Establish a common framework of software properties that can be used to represent any/all classes of software so software suppliers and acquirers can represent their claims and arguments(respectively), along with the corresponding evidence, employing automated tools (to address scale)

- Verify that products have sufficiently satisfied these characteristics in advance of product acquisition, so that system engineers/integrators can use these products to build (compose) larger assured systems with them
- Enable industry to improve visibility into the current status of software assurance during development of its software
- Enable industry to develop automated tools that support the common framework.

Software Assurance Forum for Excellence in Code (SAFECode)

According to SAFECode, Software Assurance is “confidence that software, hardware and services are free from intentional and unintentional vulnerabilities and that the software functions as intended.”

Webopedia

According to Webopedia, Software Quality Assurance, abbreviated as SQA, and also called "software assurance", is a level of confidence that software is free from vulnerabilities, either intentionally designed into the software or inserted at anytime during its lifecycle, and that the software functions in the intended manner."

As indicated in the Webopedia definition, the term "software assurance" has been used as a shorthand for Software Quality Assurance (SQA) when not necessarily considering security or trustworthiness. SQA is defined in the *Handbook of Software Quality Assurance* as: "the set of systematic activities providing evidence of the ability of the software process to produce a software product that is fit to use."

Chapter 17

Software Rot

Software rot, also known as **code rot** or **software erosion** or **software decay** or software entropy, is a type of bit rot. It describes the perceived slow deterioration of software over time that will eventually lead to it becoming faulty, unusable, or otherwise in need of maintenance. This is not a physical phenomenon: the software does not actually decay, but rather suffers from a lack of being updated with respect to the changing environment in which it resides.

Some software can deteriorate in performance over time as it runs and accumulates errors; this is not generally considered software rot, though it may have some of the same consequences. Usually, such a degraded state of software can be remedied by completely reinitializing its state (as by a complete reinstallation of all relevant software components, possibly including operating system software); this may also remedy some kinds of software rot, whereas other software rot is irreversible, as the operating environment of the software, or components of the software itself, have irrevocably changed.

Causes

There are several factors responsible for software rot. These include changes to the environment in which the software operates, degradation of compatibility between parts of the software itself, and the appearance of bugs in unused or rarely used code.

Environment change

When changes occur in the program's environment, particularly changes which the designer of the program did not anticipate, the software may no longer operate as originally intended. For example, many early computer game designers made assumptions about processing speed of the CPU which the games were designed for. A consequence of this was that when the CPU's clock speed was used as a timer in the game, the gameplay speed would increase with that of the CPU, making the software less usable over time.

Onceability

There are changes in the environment not related to the program's designer, but its users. A user could set up the system working once and have it working flawlessly for a time. But when the system stops working correctly, or the users want to access the configuration controls, they are unable to repeat that initial step because of the different context and the unavailable information (password lost, missing instructions, or simply a hard-to-manage user interface that was first configured by trial and error). Information Architect Jonas Söderström has named this concept *Onceability* , and defines it as "the quality in a technical system that prevents a user from restoring the system, once it has failed".

Unused code

Portions of code which are not normally executed, such as document filters or interfaces designed to be used by other programs, may contain bugs that go unnoticed. With changes in user requirements and other external factors, this code may be executed later, thereby exposing the bugs and making the software appear less functional.

Rarely updated code

Normal maintenance of software and systems may also cause software rot. In particular, when a program contains multiple parts which function at arm's length from one another, failing to consider how changes to one part affect the others may introduce bugs.

In some cases, this may take the form of libraries that the software uses being changed in a way which adversely affects the software. If the old version of a library that was previously used with the software cannot be used any longer due to conflicts with other software or security flaws that were found in the old version, there may no longer be a viable version of a needed library for the program to use.

Classification

Software rot is usually classified as being either **dormant rot** or **active rot**.

Dormant rot

Software that is not currently being used gradually becomes unusable as the remainder of the application changes. Changes in user requirements and the software environment also contribute to the deterioration.

Active rot

Software that is being continuously modified may lose its integrity over time if proper mitigating processes are not consistently applied. However, much software requires continuous changes to meet new requirements and correct bugs, and re-engineering

software each time a change is made is rarely practical. This creates what is essentially an evolution process for the program, causing it to depart from the original engineered design. As a consequence of this and a changing environment, assumptions made by the original designers may be invalidated, introducing bugs.

Developers are often encouraged to fully understand a problem before programming a solution to it, and to keep accurate and complete software documentation. In practice, however, adding new features may be prioritized over updating documentation. Without documentation, however, it is possible for specific knowledge pertaining to parts of the program to be lost. To some extent, this can be mitigated by following best current practices with regards to internal documentation and variable names.

Active software rot slows once an application is near the end of its commercial life and further development ceases. Users often learn to work around any remaining software bugs, and the behaviour of the software becomes consistent as nothing is changing.

Example

Many seminal programs from the early days of AI research have suffered from irreparable software rot. For example, the original SHRDLU program (an early natural language understanding program) cannot be run on any modern day computer or computer simulator, as it was developed during the days when LISP and PLANNER were still in development stage, and thus uses non-standard macros and software libraries which do not exist anymore.

Suppose an administrator creates a forum using phpBB or other online forum software. He then heavily modifies and "hacks" it, adding new features and options. This process requires extensive modifications to existing code and deviation from the original functionality of that software.

From here, there are several ways software rot can affect the system:

- The administrator can accidentally make changes which conflict with each other or the original software, causing the forum to behave unexpectedly or break down altogether. This leaves him in a very bad position: as he has deviated so greatly from the original code, technical support and assistance in reviving the forum will be difficult to obtain.
- A security hole may be discovered in the original forum source code, requiring a security patch. However, because the administrator has modified the code so extensively, the patch may not be directly applicable to his code, requiring the administrator to effectively rewrite the update.
- The administrator who made the modifications could vacate his position, leaving the new administrator with a convoluted and heavily-modified forum that lacks full documentation. Without fully understanding the modifications, it is difficult for the new administrator to make changes without introducing conflicts and bugs. (Furthermore, documentation of the original system might no longer be available;

it might have been abandoned, become proprietary closed software, or, with the passage of enough time, been lost.)

Refactoring

Refactoring is a means of addressing the problem of software rot. It is described as the process of rewriting existing code to improve its structure without affecting its external behaviour. This includes removing dead code and rewriting sections that have been modified extensively and no longer work efficiently. Care must be taken not to change the software's external behaviour, as this could introduce incompatibilities and thereby itself contribute to software rot.

Chapter 18

Reverse Semantic Traceability

Reverse Semantic Traceability (**RST**) is a quality control method for verification improvement that helps to insure high quality of artifacts by backward translation at each stage of the software development process.

Brief introduction

Each stage of development process can be treated as a series of "translations" from one language to another. At the very beginning a project team deals with customer's requirements and expectations expressed in natural language. These customer requirements sometimes might be incomplete, vague or even contradictory to each other. The first step is specification and formalization of customer expectations, transition ("translation") of them into a formal requirement document for the future system. Then requirements are translated into system architecture and step by step the project team generates megabytes of code written in a very formal programming language. There is always a threat of inserting mistakes, misinterpreting or losing something during the translation. Even a small defect in requirement or design specifications can cause huge amounts of defects at the late stages of the project. Sometimes such misunderstandings can lead to project failure or complete customer dissatisfaction.

The highest usage scenarios of Reverse Semantic Traceability method can be:

- Validating UML models: quality engineers restore a textual description of a domain, original and restored descriptions are compared.
- Validating model changes for a new requirement: given an original and changed versions of a model, quality engineers restore the textual description of the requirement, original and restored descriptions are compared.
- Validating a bug fix: given an original and modified Source code, quality engineers restore a textual description of the bug that was fixed, original and restored descriptions are compared.
- Integrating new software engineer into a team: a new team member gets an assignment to do Reverse Semantic Traceability for the key artifacts from the current projects.

RST roles

Main roles involved in RST session are:

- authors of project artifacts (both input and output),
- reverse engineers,
- expert group,
- project manager.

RST process

Define all project artifacts and their relationship

Reverse Semantic Traceability as a validation method can be applied to any project artifact, to any part of project artifact or even to a small piece of document or code. However, it is obvious that performing RST for all artifacts can create overhead and should be well justified (for example, for medical software where possible information loss is very critical).

It is a responsibility of company and Project manager to decide what amount of project artifacts will be “reverse engineered”. This amount depends on project specific details: trade-off matrix, project and company quality assurance policies. Also it depends on importance of particular artifact for project success and level of quality control applied to this artifact.

Amount of RST sessions for project is defined at the project planning stage.

First of all project manager should create a list of all artifacts project team will have during the project. They could be presented as a tree with dependencies and relationships. Artifacts can be present in one occurrence (like Vision document) or in several occurrences (like risks or bugs). This list can be changed later during the project but the idea behind the decisions about RST activities will be the same.

Prioritize

The second step is to analyze deliverable importance for project and level of quality control for each project artifact.

Importance of document is the degree of artifact impact to project success and quality of final product. It's measured by the scale:

- Crucial (1): the quality of deliverable is very important for overall quality of project and even for project success. Examples: Functional requirements, System architecture, critical bug fixes (show stoppers), risks with high probability and critical impact.
- High (2): the deliverable has an impact to quality of final product. Examples: Test cases, User interface requirements, major severity bug fixes, risks with high expose.
- Medium (3): the artifact has a medium or indirect impact to quality of final product. Examples: Project plan, medium severity bug fixes, risks with medium expose.
- Low (4): the artifact has insignificant impact to the final product quality. Example: employees' tasks, cosmetic bugs, risks with low probability.

Level of quality control is a measure that defines amount of verification and validation activities applied to artifact, and probability of miscommunication during artifact creation.

- Low (1): No review is supposed for the artifact, miscommunication and information loss are high probable, information channel is distributed, language barrier exists etc
- Medium (2): No review is supposed for the artifact, information channel is not distributed (e.g. creator of artifact and information provider are members of one team)
- Sufficient (3): Pair development or peer review is done, information channel is not distributed.
- Excellent (4): Pair development, peer review and/or testing are done, automation or unit testing is done, or there are some tools for artifact development and validation.

Define responsible people

Success of RST session strongly depends on correct assignment of responsible people.

Perform Reverse Semantic Traceability of artifact

Reverse Semantic Traceability starts when decision that RST should be performed is made and resources for it are available.

Project manager defines what documents will be an input for RST session. For example, it can be not only an artifact to restore but some background project information. It is recommended to give to reverse engineers number of words in original text so that they have an idea about amount of text they should get as a result: it can be one sentence or several pages of text. Though, the restored text may not contain the same number of words as original text nevertheless the values should be comparable.

After that reverse engineers take the artifact and restore the original text from it. RST itself takes about 1 hour for one page text (750 words).

Value the level of quality and make a decision

To complete RST session, restored and original texts of artifact should be compared and quality of artifact should be assessed. Decision about artifacts rework and its amount is made based on this assessment.

For assessment a group of experts is formed. Experts should be aware of project domain and be an experienced enough to assess quality level of compared artifacts. For example, business analysts will be experts for comparison of vision statement and restored vision statement from scenario.

RST assessment criteria:

1. Restored and original texts have quite big differences in meaning and crucial information loss
2. Restored and original texts have some differences in meaning, important information loss
3. Restored and original texts have some differences in meaning, some insignificant information loss
4. Restored and original texts are very close, some insignificant information loss
5. Restored and original texts are very close, none information is lost

Each of experts gives his assessment, and then the average value is calculated. Depending on this value Project Manager makes a decision should both artifacts be corrected or one of them or rework is not required.

If the average RST quality level is in range from 1 to 2 the quality of artifact is poor and it is recommended not only rework of validated artifact to eliminate defects but corrections of original artifact to clear misunderstandings. In this case one more RST session after rework of artifacts is required. For artifacts that have more than 2 but less than 3 corrections of validated artifact to fix defects and eliminate information loss is required, however review of original artifact to find out if there any vague piece of information that cause misunderstandings is recommended. No additional RST sessions is needed. If the average mark is more than 3 but less than 4 then corrections of validated artifact to remove defects and insignificant information loss is supposed. If the mark is greater than 4 it means that artifact is of good quality and no special corrections or rework is required.

Obviously the final decision about rework of artifacts is made by project manager and should be based on analysis of reasons of differences in texts.