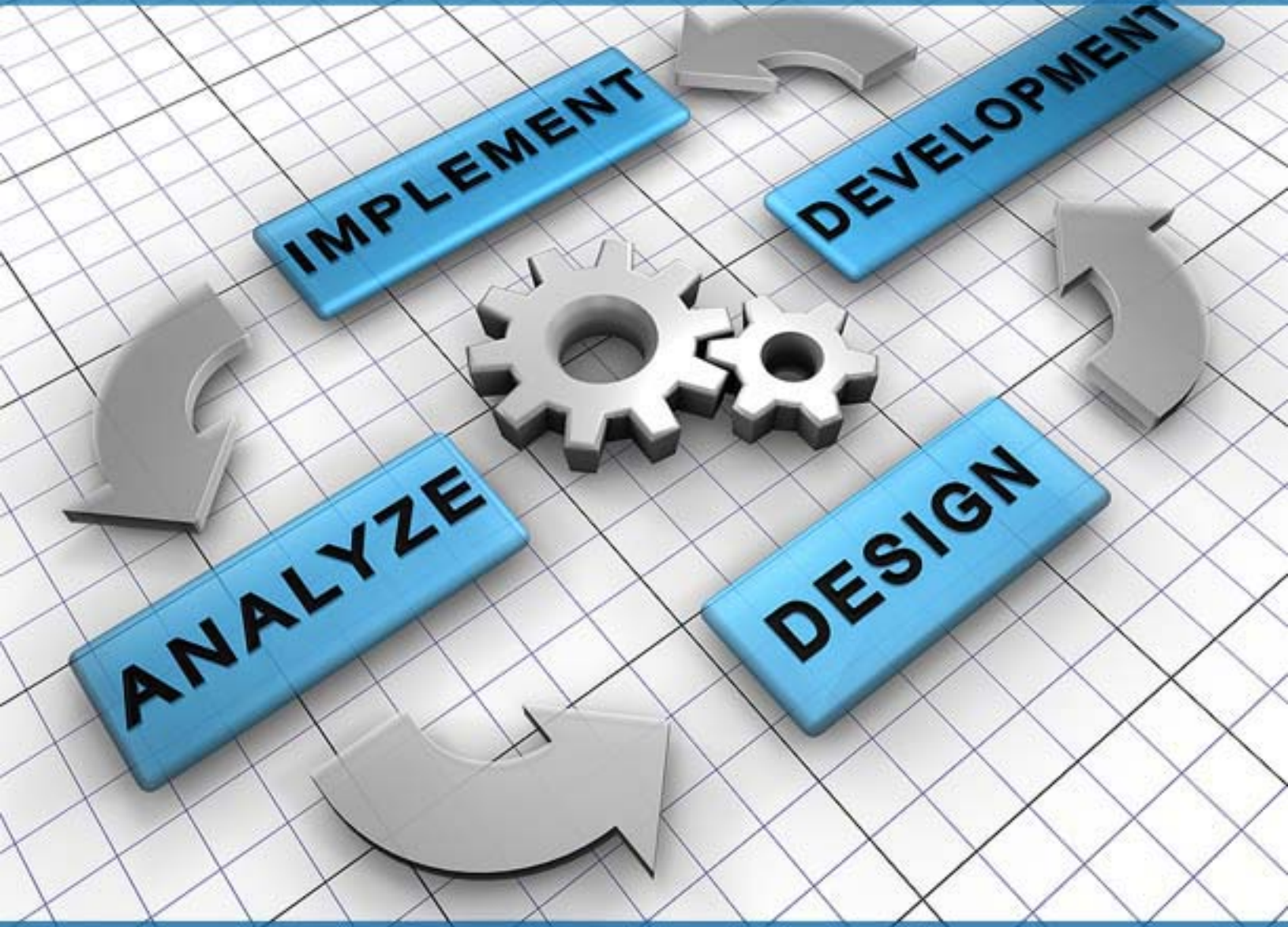
Software Development Process



Lamar Lacy

First Edition, 2012

ISBN 978-81-323-4228-1

Published by:
White Word Publications
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

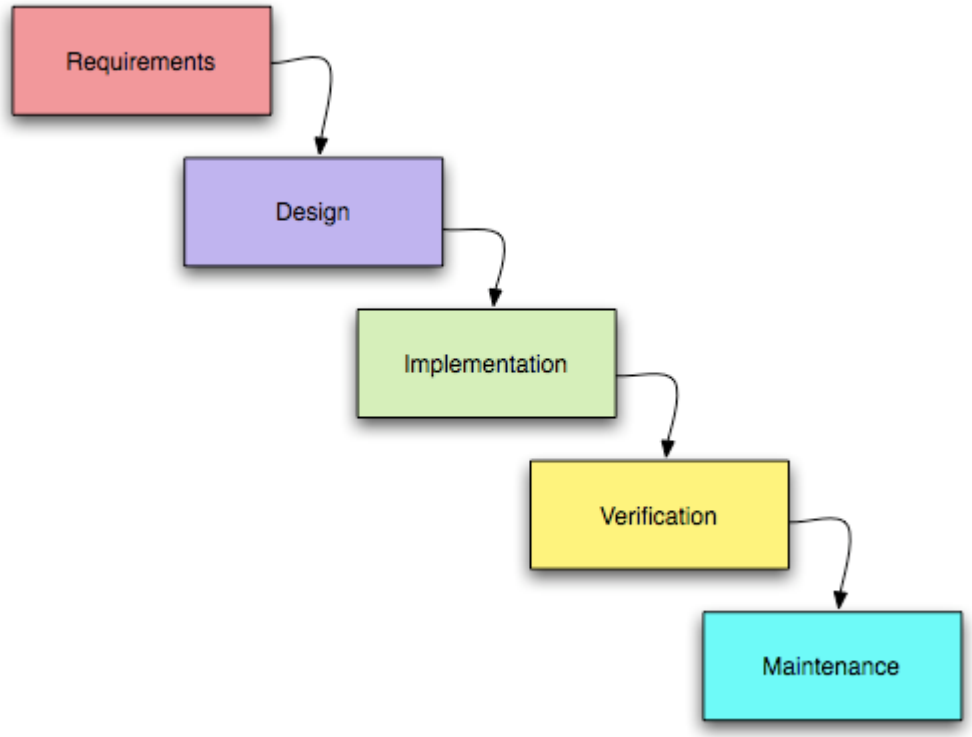
The large and growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts.

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

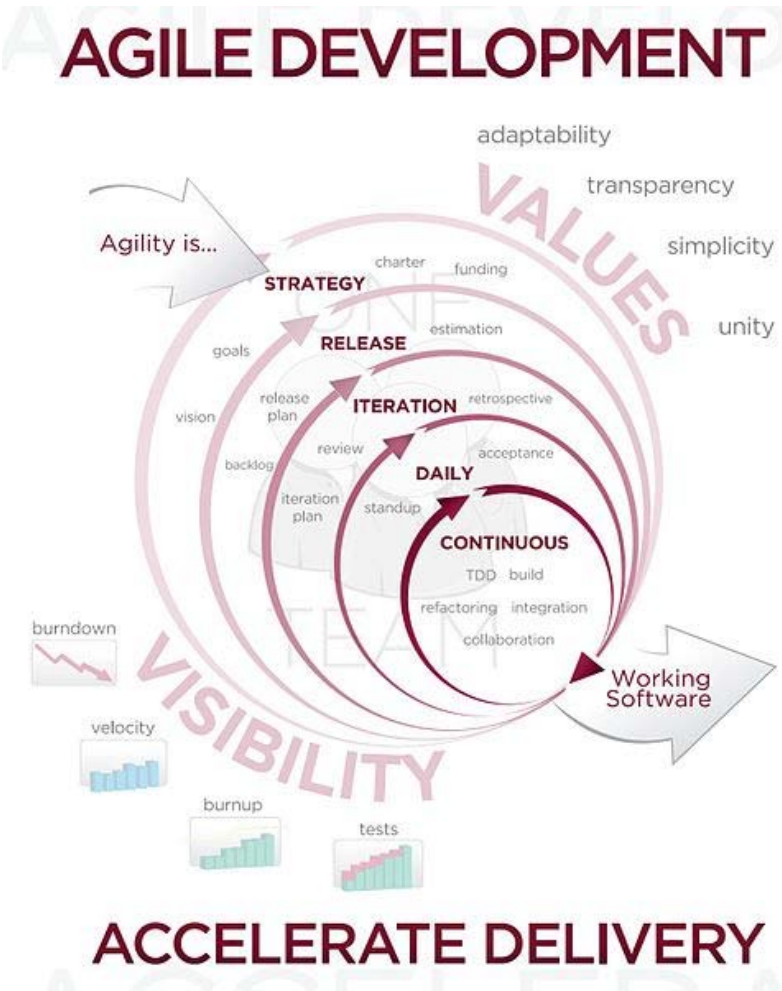
mapping and direct execution of the logic is that they can be made to explain their results, in English, at the business or scientific level.

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



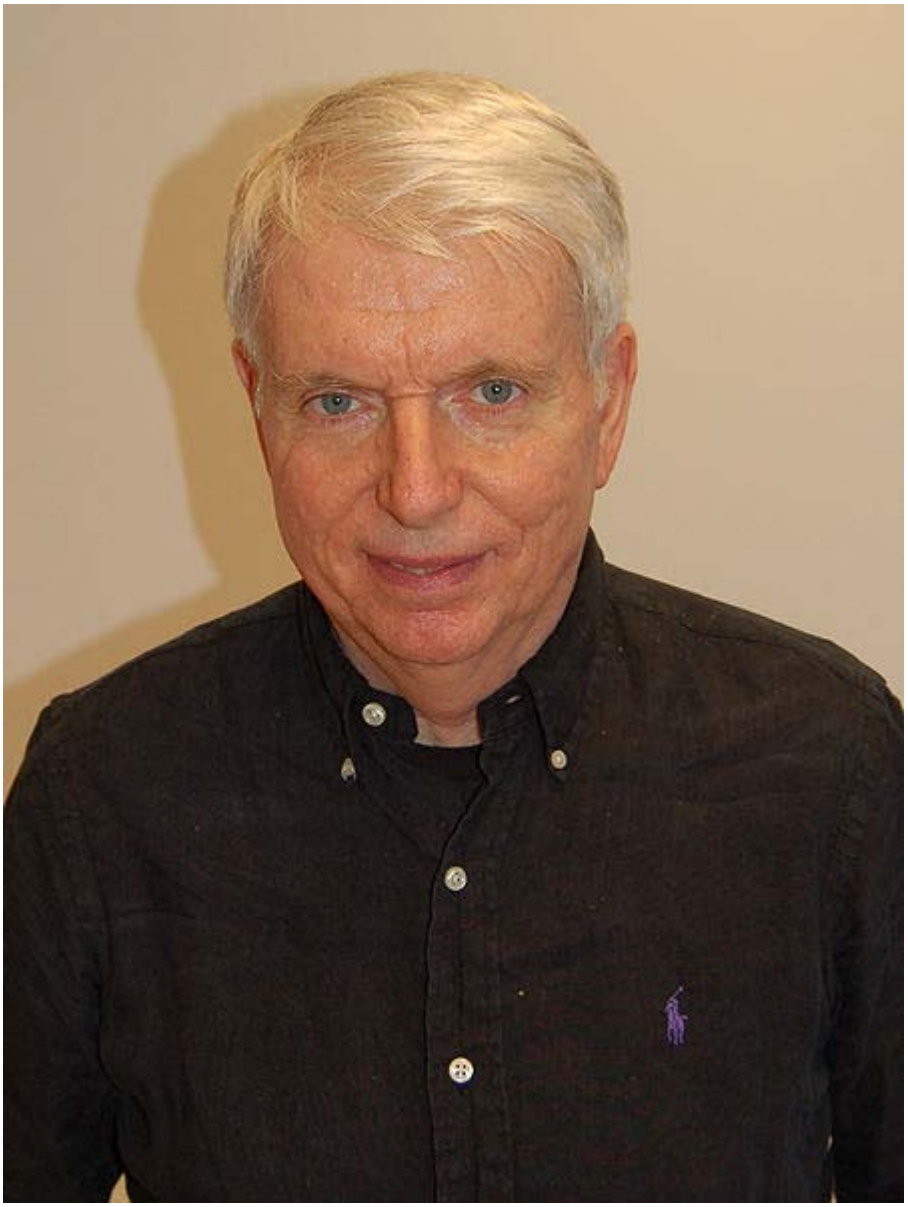
Agile software development poster

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.

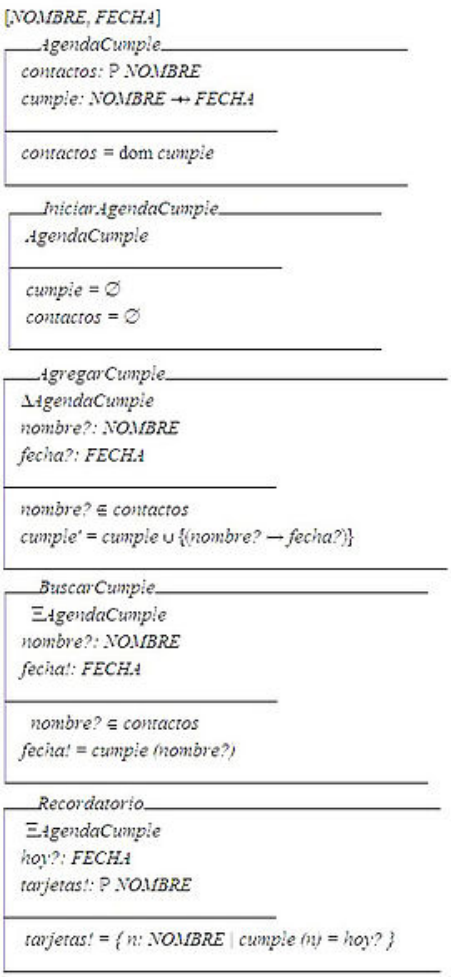
As of 2006, experience reports have been or will be presented at the following conferences:

- 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 deployment report	Getting a report (automatically or manually) about 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

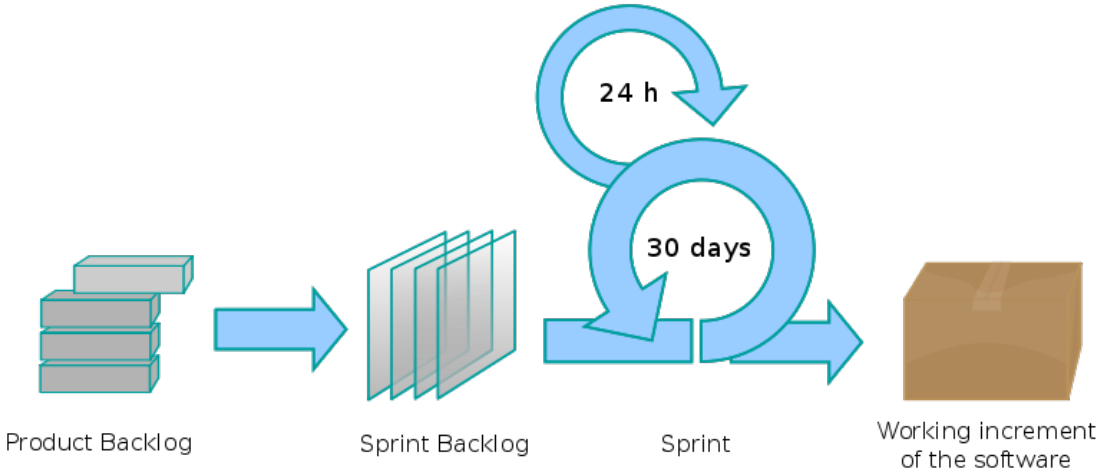
Example

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

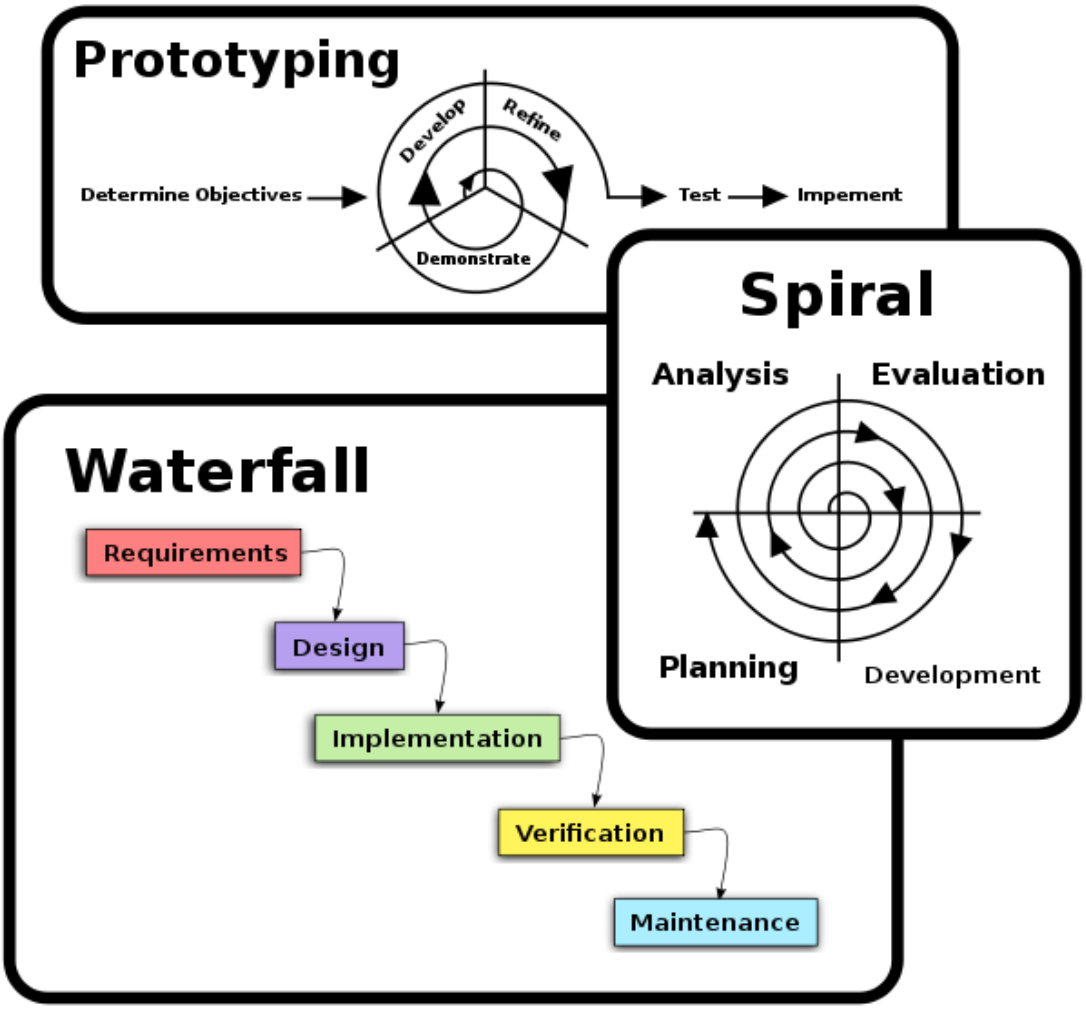
A **software development methodology** or **system development methodology** in software engineering is a framework that is used to structure, plan, and control the process of developing an information system.

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

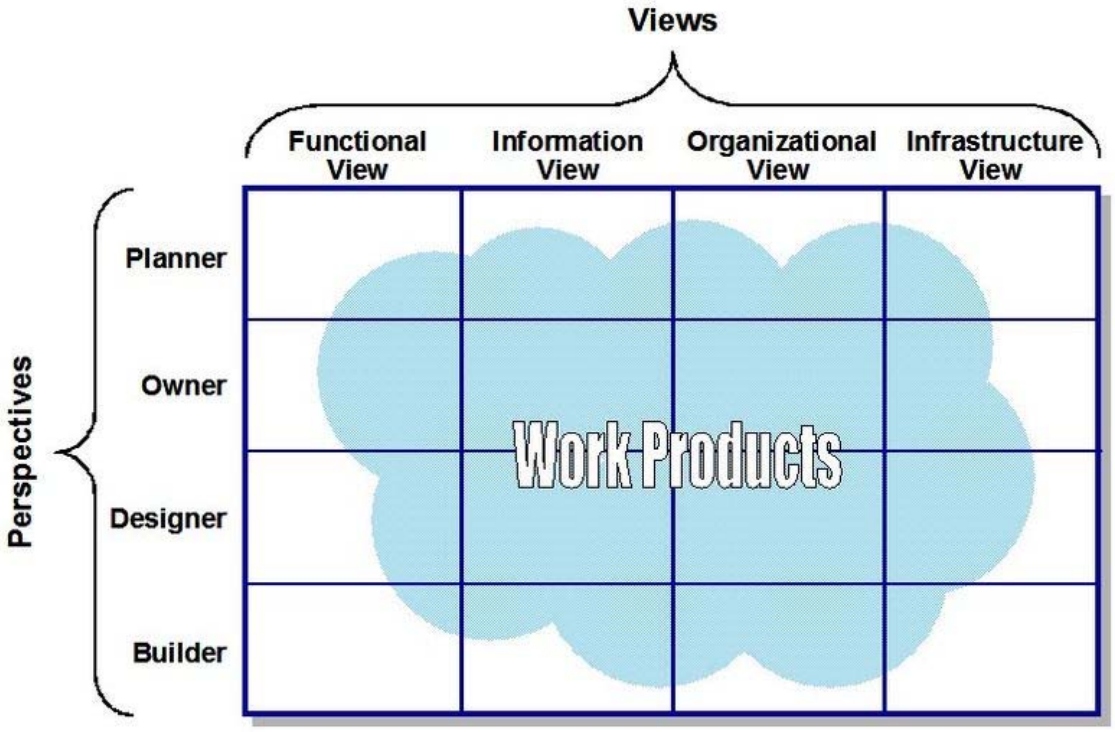
Other methodology practices include:

- 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.

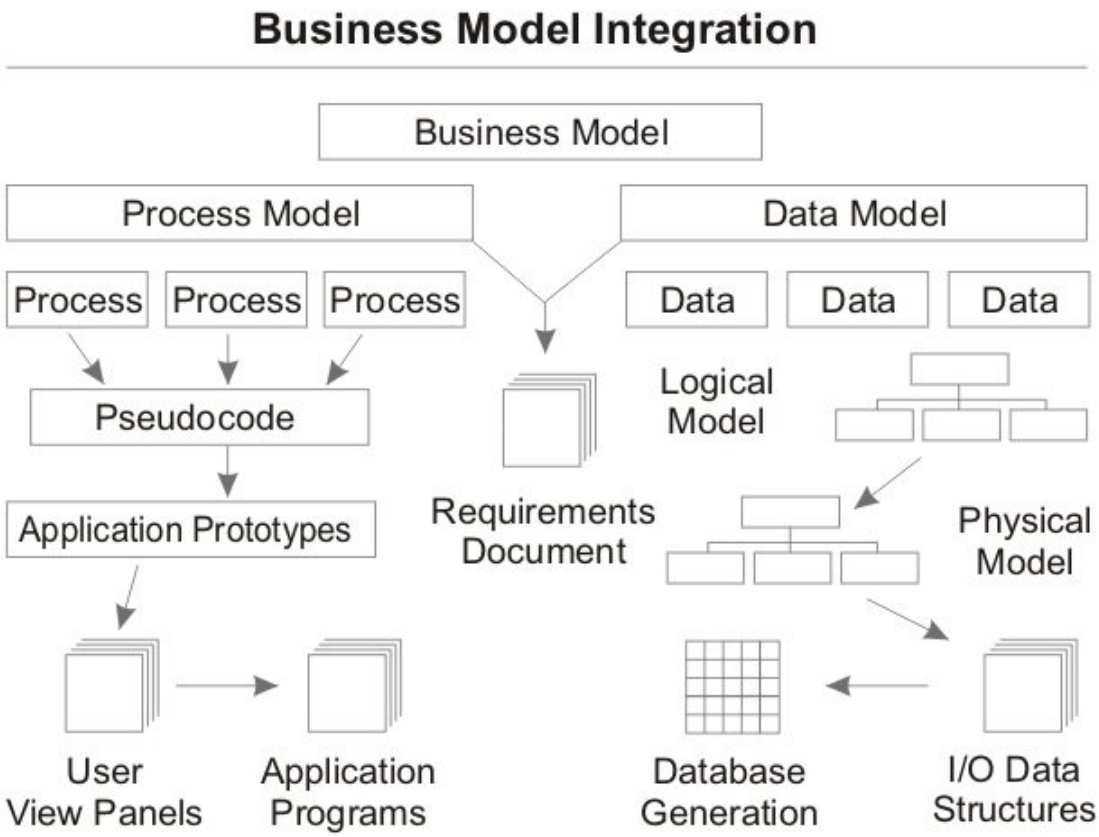
The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems, and to organize the elements of the problem and the solution around

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

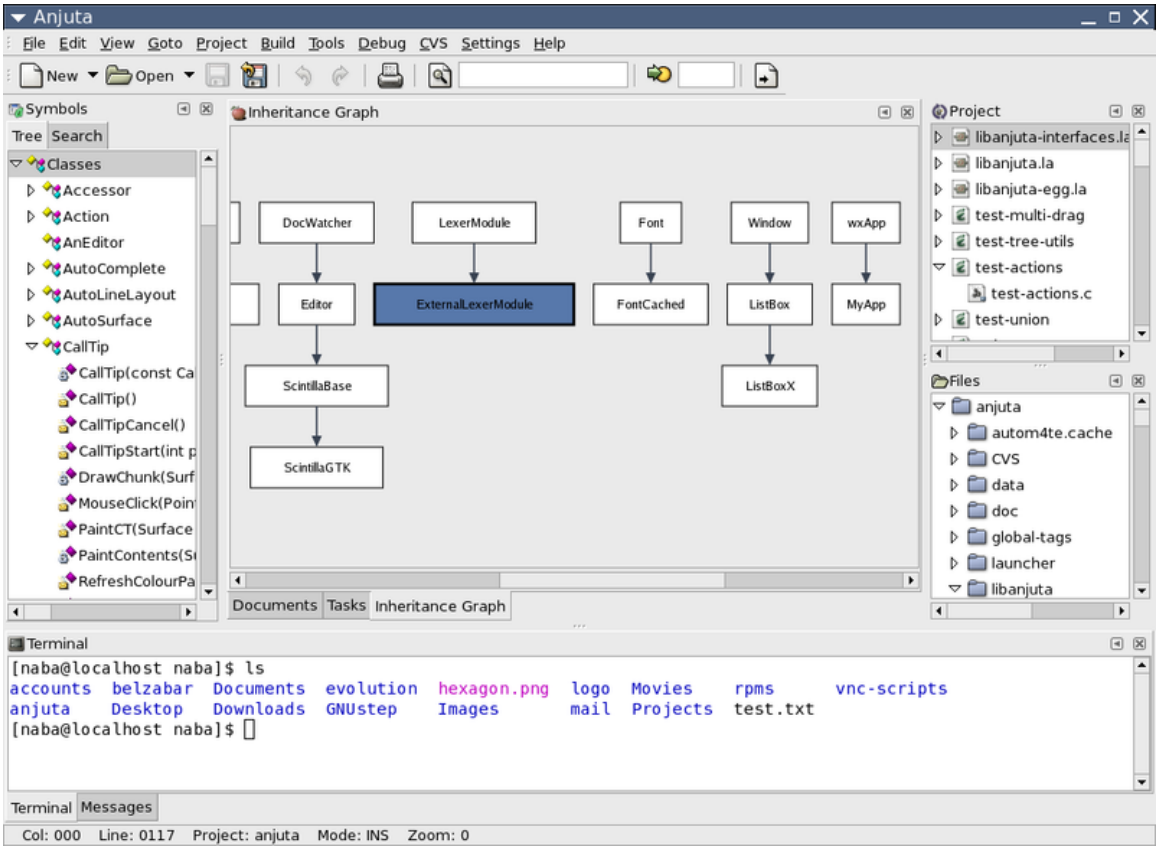
Computer-aided software engineering (CASE), in the field software engineering is the scientific application of a set of tools and methods to a software which results in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems 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?

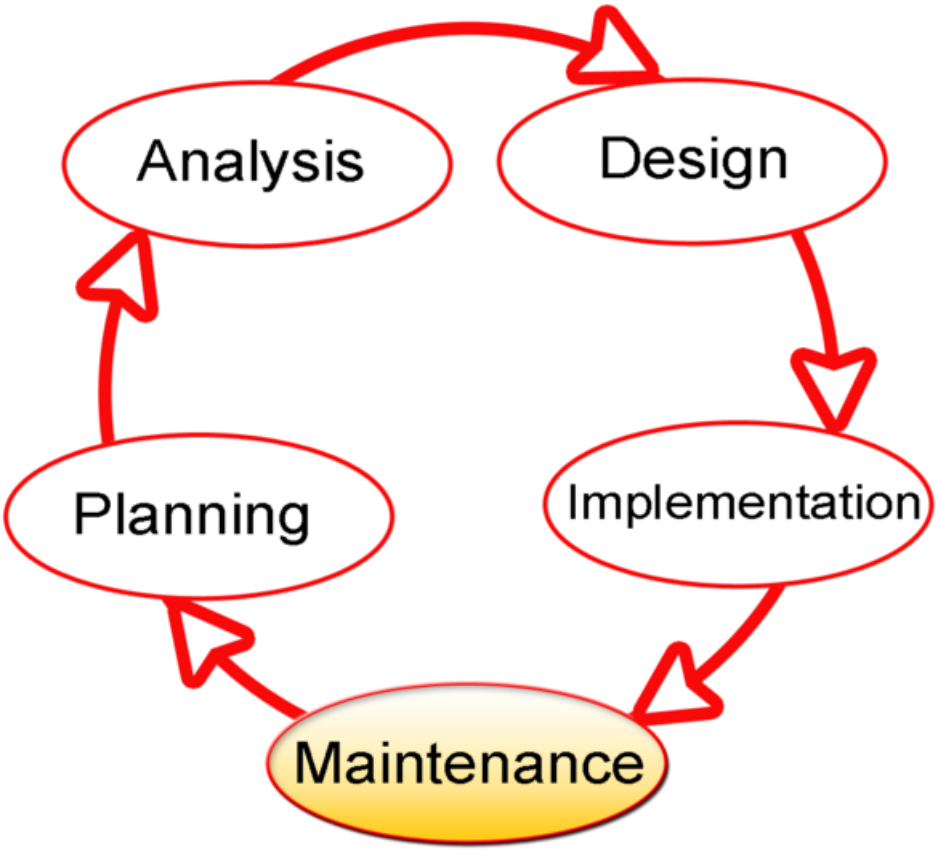
Manual testing vs. automated
: Some writers believe that test automation is so expensive relative to its value that it should be used sparingly. More in particular, test-driven development states that developers should write unit-tests of the XUnit type before coding the functionality. The tests then 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

Systems Development Life Cycle



Model of the Systems Development Life Cycle with the Maintenance bubble highlighted.

The **Systems Development Life Cycle (SDLC)**, or *Software Development Life Cycle* in systems engineering, information systems and software engineering, is the process of creating or altering systems, and the models and methodologies that people use to develop these systems. The concept generally refers to computer or information systems.

In software engineering the SDLC concept underpins many kinds of software development methodologies. These methodologies form the framework for planning and controlling the creation of an information system: the software development process.

Overview

History

The **Systems Life Cycle (SLC)** is a type of methodology used to describe the process for building information systems, intended to develop information systems in a very deliberate, structured and methodical way, reiterating each stage of the life cycle. The systems development life cycle, according to Elliott & Strachan & Radford (2004), "originated in the 1960s,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".

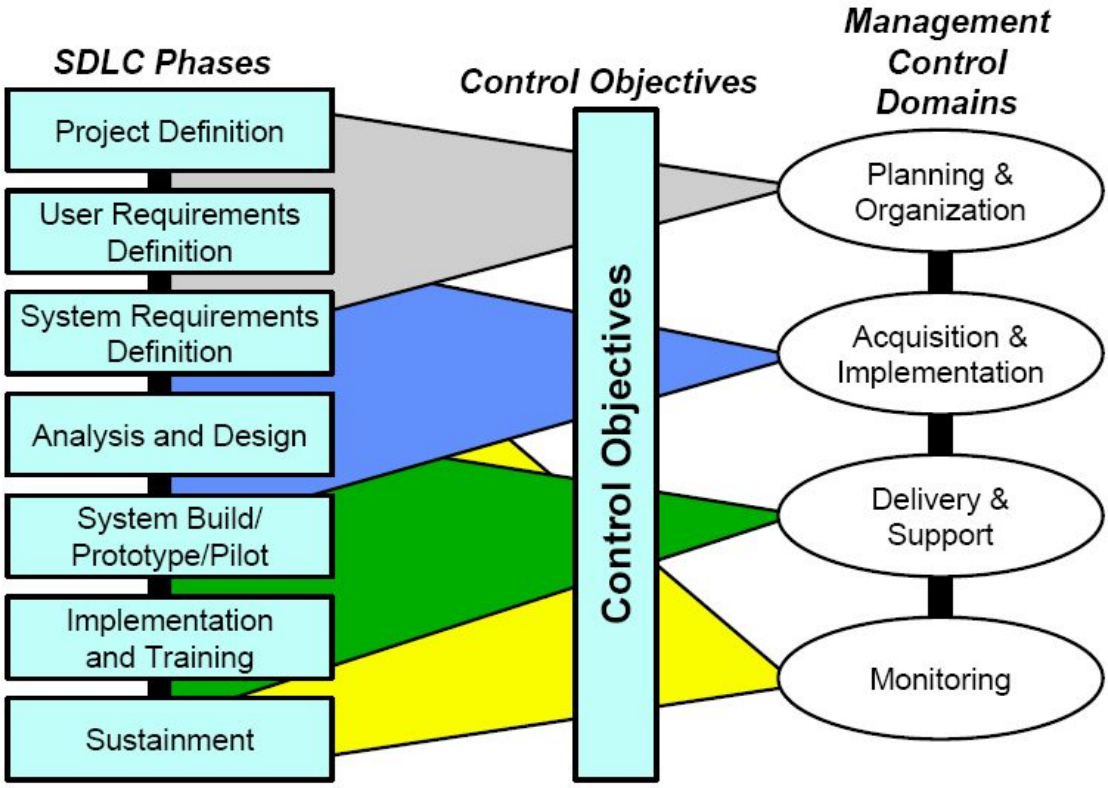
Several systems development frameworks have been partly based on SDLC, such as the Structured Systems Analysis and Design Method (SSADM) produced for the UK government Office of Government Commerce in the 1980s. Ever since, according to Elliott (2004), "the traditional life cycle approaches to systems development have been increasingly replaced with alternative approaches and frameworks, which attempted to overcome some of the inherent deficiencies of the traditional SDLC".

Systems Analysis and Design

The **Systems Analysis and Design (SAD)** is the process of developing Information Systems (IS) that effectively use of hardware, software, data, process, and people to support the company’s business objectives.

Systems development life cycle topics

Management and control



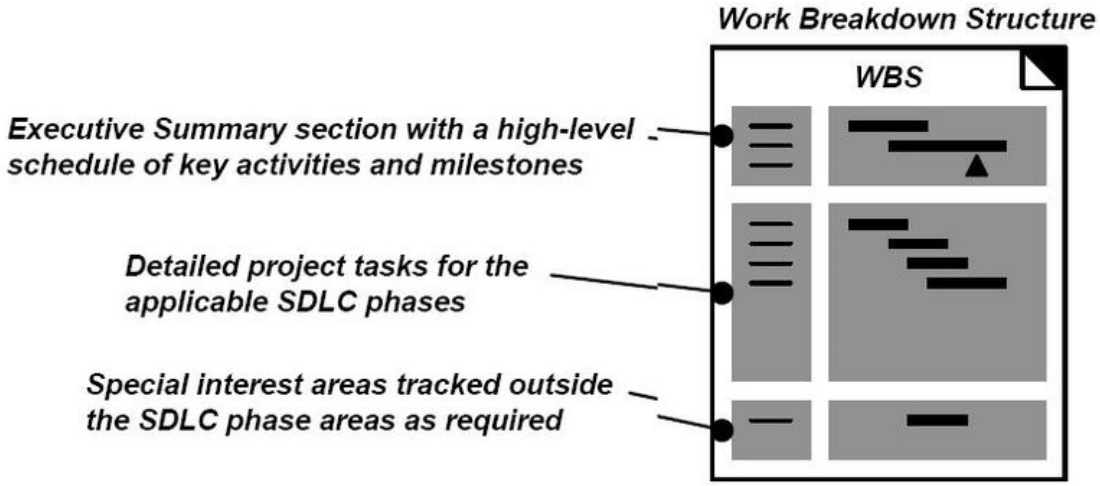
SDLC Phases Related to Management Controls.

The Systems Development Life Cycle (SDLC) phases serve as a programmatic guide to project activity and provide a flexible but consistent way to conduct projects to a depth matching the scope of the project. Each of the SDLC phase objectives are described here with key deliverables, a description of recommended tasks, and a summary of related control objectives for effective management. It is critical for the project manager to establish and monitor control objectives during each SDLC phase while executing projects. Control objectives help to provide a clear statement of the desired result or purpose and should be used throughout the entire SDLC process. Control objectives can be grouped into major categories (Domains), and relate to the SDLC phases as shown in the figure.

To manage and control any SDLC initiative, each project will be required to establish some degree of a Work Breakdown Structure (WBS) to capture and schedule the work necessary to complete the project. The WBS and all programmatic material should be kept in the "Project Description" section of the project notebook. The WBS format is mostly left to the project manager to establish in a way that best describes the project work. There are some key areas that must be defined in the WBS as part of the SDLC

policy. The following diagram describes three key areas that will be addressed in the WBS in a manner established by the project manager.

Work breakdown structured organization



Work Breakdown Structure.

The upper section of the Work Breakdown Structure (WBS) should identify the major phases and milestones of the project in a summary fashion. In addition, the upper section should provide an overview of the full scope and timeline of the project and will be part of the initial project description effort leading to project approval. The middle section of the WBS is based on the seven Systems Development Life Cycle (SDLC) phases as a guide for WBS task development. The WBS elements should consist of milestones and "tasks" as opposed to "activities" and have a definitive period (usually two weeks or more). Each task must have a measurable output (e.x. document, decision, or analysis). A WBS task may rely on one or more activities (e.g. software engineering, systems engineering) and may require close coordination with other tasks, either internal or external to the project. Any part of the project needing support from contractors should have a Statement of work (SOW) written to include the appropriate tasks from the SDLC phases. The development of a SOW does not occur during a specific phase of SDLC but is developed to include the work from the SDLC process that may be conducted by external resources such as contractors and struct.

Baselines in the SDLC

Baselines are an important part of the Systems Development Life Cycle (SDLC). These baselines are established after four of the five phases of the SDLC and are critical to the iterative nature of the model. Each baseline is considered as a milestone in the SDLC.

- Functional Baseline: established after the conceptual design phase.
- Allocated Baseline: established after the preliminary design phase.
- Product Baseline: established after the detail design and development phase.

- Updated Product Baseline: established after the production construction phase.

Complementary to SDLC

Complementary Software development methods to Systems Development Life Cycle (SDLC) are:

- Software Prototyping
- Joint Applications Design (JAD)
- Rapid Application Development (RAD)
- Extreme Programming (XP); extension of earlier work in Prototyping and RAD.
- Open Source Development
- End-user development
- Object Oriented Programming

Comparison of Methodology Approaches (Post, & Anderson 2006)							
	SDLC	**RAD**	**Open Source**	**Objects**	**JAD**	**Prototyping**	**End User**
Control	Formal	MIS	Weak	Standards	Joint	User	User
Time Frame	Long	Short	Medium	Any	Medium	Short	Short
Users	Many	Few	Few	Varies	Few	One or Two	One
MIS staff	Many	Few	Hundreds	Split	Few	One or Two	None
Transaction/DSS	Transaction	Both	Both	Both	DSS	DSS	DSS
Interface	Minimal	Minimal	Weak	Windows	Crucial	Crucial	Crucial
Documentation and training	Vital	Limited	Internal	In Objects	Limited	Weak	None
Integrity and security	Vital	Vital	Unknown	In Objects	Limited	Weak	Weak
Reusability	Limited	Some	Maybe	Vital	Limited	Weak	None

Strengths and weaknesses

Few people in the modern computing world would use a strict waterfall model for their Systems Development Life Cycle (SDLC) as many modern methodologies have superseded this thinking. Some will argue that the SDLC no longer applies to models like Agile computing, but it is still a term widely in use in Technology circles. The SDLC practice has advantages in traditional models of software development, that lends itself more to a structured environment. The disadvantages to using the SDLC methodology is when there is need for iterative development or (i.e. web development or e-commerce) where stakeholders need to review on a regular basis the software being designed. Instead of viewing SDLC from a strength or weakness perspective, it is far more important to take the best practices from the SDLC model and apply it to whatever may be most appropriate for the software being designed.

A comparison of the strengths and weaknesses of SDLC:

Strength and Weaknesses of SDLC	
Strengths	**Weaknesses**
Control.	Increased development time.
Monitor Large projects.	Increased development cost.
Detailed steps.	Systems must be defined up front.
Evaluate costs and completion targets.	Rigidity.
Documentation.	Hard to estimate costs, project overruns.
Well defined user input.	User input is sometimes limited.
Ease of maintenance.	
Development and design standards.	
Tolerates changes in MIS staffing.	

An alternative to the SDLC is Rapid Application Development, which combines prototyping, Joint Application Development and implementation of CASE tools. The advantages of RAD are speed, reduced development cost, and active user involvement in the development process.

Chapter 13

Goal-Driven Software Development Process

Goal-Driven Software Development Process (GDP) is an iterative and incremental software development technique. Although similar to other modern process models, GDP is primarily focusing on identifying goals *before* setting the requirements and explicitly utilizing the bottom-up design approach.

The following sections are based on the paper *Goal-Driven Software Development* where the GDP concept was introduced.

Justification

The first argument to embrace the GDP principles is the aspect of requirements. When developing a software, the strong concentration on requirements (e.g. typical for the waterfall model) causes excessive costs and reduced quality of the outcome, mainly due to the following reasons:

- Requirements are usually not identical with business objectives because of the author's limited knowledge about technical possibilities and their costs – such requirements tend to include unnecessary expensive wishes while excluding technically simple features that would provide substantial benefit.
- Formalization of the supported business process during development usually reveals inconsistencies and gaps within that process which need to be compensated with changes to the process itself or to the role of the software system.

The result of these two effects is usually a large number of change requests during and after development (entailing time and cost overruns), therefore the user involvement is considered to be a critical project success factor.

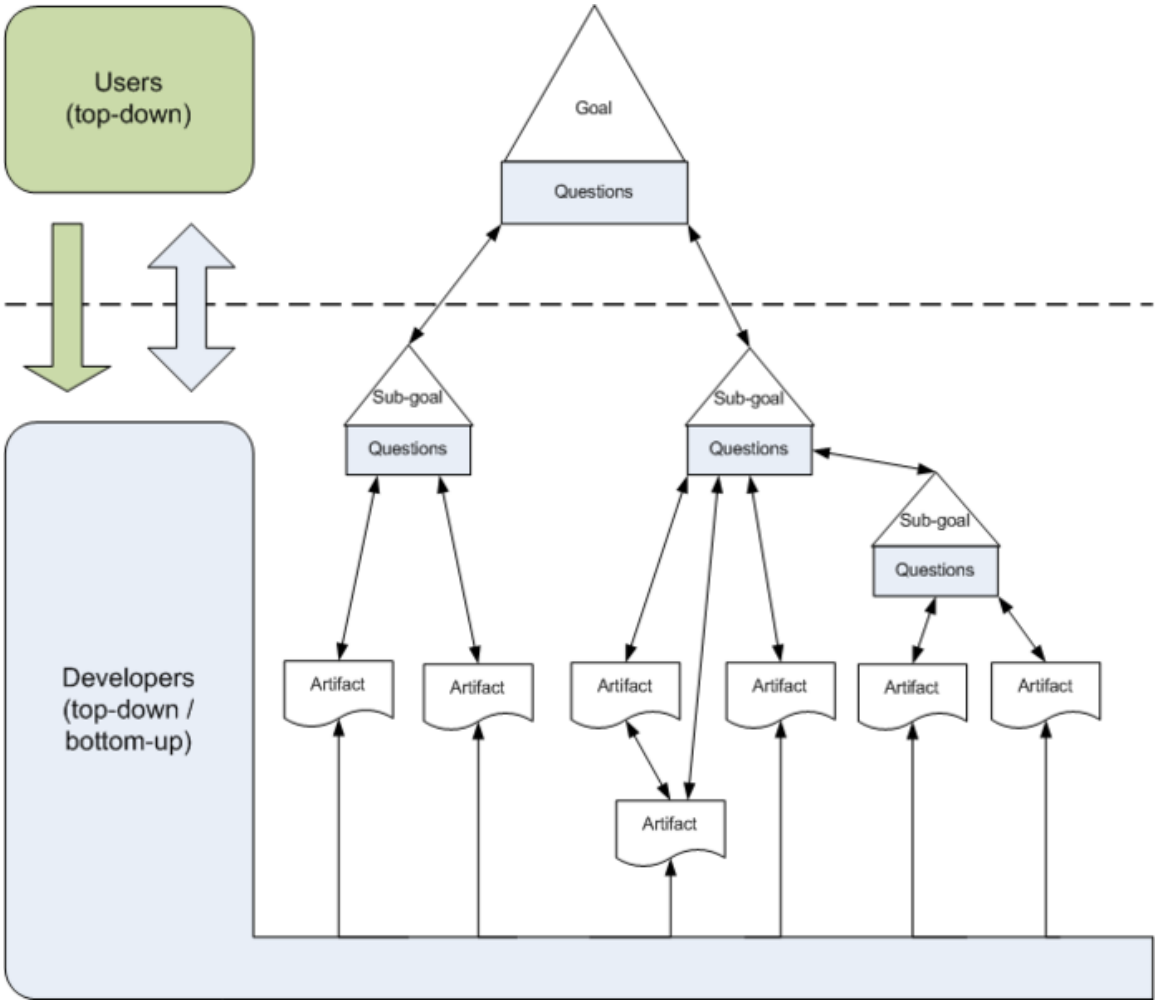
Secondly, while established software processes refine requirements down to an implementation, the Goal-driven Development Process recommends trying to find an optimal mapping between business objectives and capabilities of the technical platform in

an *iterative* process, equally considering and adjusting business goals and technical aspects to come to an optimal, *convergent* solution.

Goal-driven development process allows stakeholders to :

- Discover use cases that are tailored to the requirements according to business goals
- Establish a bridge between goals and IT architecture

Key principles



Goal-driven software development process

Collaborative goal identification

As closely related to the Goal-Question-Metric paradigm, a **top-level goal** is defined as an informal description of what a stakeholder wants to change or improve in his business environment, decomposing itself to more specific **sub-goals**. Moreover, a set of questions

is linked to every goal, which characterizes the way how software will be tested against defined goals after each iteration.

Being this the key GDP principle, the collaborative identification of goals brings knowledge of users and software developers together. While goal definition is top-down driven, deciding, if a goal is feasible is bottom-up oriented.

Top-down and bottom-up convergence

While the top-down orientation supports a horizontal team organization, bottom-up approaches try to provide generalized components or services, leading to a better user satisfaction. The collaborative identification of goals introduced by GDP allows combining top-down with bottom-up aspects ("*top-down thinking and bottom-up acting*") to support artifacts consistency and allowing vertical team organization.

Vertical team organization

In contrast to horizontally organized project teams where programmers implement the solution specified by the modeling team, the vertical organization implied by the GDP requires skilled and qualified generalists. As stated by IBM Rational Unified Process, individual developers can and *should* take multiple roles on a project to avoid unnecessary communication overhead and conflicts.

Roles and people

Because of its vertical organization the GDP requires skilled generalists with the ability to fulfill many roles of the process:

- **Programmers** (responsible for top-down and bottom-up convergence)
- **Business analysts** (collaborate with the programmers during goal identification and later-on during testing)
- **Software architects** (keep an eye on the whole project)
- **Project manager** (assigns resources, keeps track of time and effort, creates a productive environment)
- **Requirement engineer**

Minimizing project size

According to GDP, another key to success in large project is to minimize project size in all aspects, i.e. limit the number of goals and software artifacts like documents, requirement specifications, models, etc. but also to limit the number of staff, to avoid mutual waiting and the size of the code.

Minimizing size leads to an increased maintainability and changeability of the system to business processes as they are the most likely factor to change in the future.

Activities

Every iteration starts with the identification of business goals and their priorities and ends with a running version of the software system corresponding to the selected goals.

While incremental development of the software system is also done in other software processes, the scope of GDP iteration is extended to include a discussion of business objectives after *each* iteration as is believed the business objectives themselves mature with the availability of usable implementation.

The core activities are:

1. **Identification and prioritization of goals** (small groups of at most 5 people consisting of stakeholders and/or business analysts, and programmers)
2. **Vertical distribution of tasks** (selected goals are assigned to groups of at most 4 programmers)
3. **Implementation and testing** (implementation-driven tests during implementation, goal-driven tests at the end of each iteration)

These activities can be also divided into six main steps :

1. **Group business requirements by goals**
2. **Formalize goal-driven system behaviors inside processes**
3. **Monitor advancement in the realization of the goals (optional)**
4. **Assign responsibilities to participants of the processes**
5. **Plug behaviors in the goal-driven architectural backbone and play**
6. **Integrate application constraints of the actors**

Chapter 14

Lean Software Development

Lean software development is a translation of Lean manufacturing and Lean IT principles and practices to the software development domain. Adapted from the Toyota Production System, a pro-lean subculture is emerging from within the Agile community.

Origin

The term **Lean Software Development** originated in a book by the same name, written by Mary Poppendieck and Tom Poppendieck. The book presents the traditional Lean principles in a modified form, as well as a set of 22 *tools* and compares the tools to agile practices. Mary and Tom's involvement in the Agile software development community, including talks at several Agile conferences has resulted in such concepts being more widely accepted within the Agile community.

Lean principles

Lean development could be summarized by seven principles, very close in concept to lean manufacturing principles.

Eliminate waste

Everything not adding value to the customer is considered to be waste (*muda*). This includes:

- unnecessary code and functionality
- delay in the software development process
- unclear requirements
- bureaucracy
- slow internal communication

In order to be able to eliminate waste, one should be able to recognize and see it. If some activity could be bypassed or the result could be achieved without it, it is waste. Partially done coding eventually abandoned during the development process is waste. Extra processes and features not often used by customers are waste. Waiting for other activities,

teams, processes is waste. Defects and lower quality are waste. Managerial overhead not producing real value is waste. A value stream mapping technique is used to distinguish and recognize waste. The second step is to point out sources of waste and eliminate them. The same should be done iteratively until even essential-seeming processes and procedures are liquidated.

Amplify learning

Software development is a continuous learning process with the additional challenge of development teams and end product sizes. The best approach for improving a software development environment is to amplify learning. The accumulation of defects should be prevented by running tests as soon as the code is written. Instead of adding more documentation or detailed planning, different ideas could be tried by writing code and building. The process of user requirements gathering could be simplified by presenting screens to the end-users and getting their input.

The learning process is sped up by usage of short iteration cycles – each one coupled with refactoring and integration testing. Increasing feedback via short feedback sessions with customers helps when determining the current phase of development and adjusting efforts for future improvements. During those short sessions both customer representatives and the development team learn more about the domain problem and figure out possible solutions for further development. Thus the customers better understand their needs, based on the existing result of development efforts, and the developers learn how to better satisfy those needs. Another idea in the communication and learning process with a customer is set-based development – this concentrates on communicating the constraints of the future solution and not the possible solutions, thus promoting the birth of the solution via dialog with the customer.

Decide as late as possible

As software development is always associated with some uncertainty, better results should be achieved with an options-based approach, delaying decisions as much as possible until they can be made based on facts and not on uncertain assumptions and predictions. The more complex a system is, the more capacity for change should be built into it, thus enabling the delay of important and crucial commitments. The iterative approach promotes this principle – the ability to adapt to changes and correct mistakes, which might be very costly if discovered after the release of the system. (This description - delay decisions and options - seems to suggest a strong link with the Set-Based Concurrent Engineering -SBCE- approach described in various prints related to Lean Product Development)

An agile software development approach can move the building of options earlier for customers, thus delaying certain crucial decisions until customers have realized their needs better. This also allows later adaptation to changes and the prevention of costly earlier technology-bounded decisions. This does not mean that no planning should be involved – on the contrary, planning activities should be concentrated on the different

options and adapting to the current situation, as well as clarifying confusing situations by establishing patterns for rapid action. Evaluating different options is effective as soon as it is realized that they are not free, but provide the needed flexibility for late decision making.

Deliver as fast as possible

In the era of rapid technology evolution, it is not the biggest that survives, but the fastest. The sooner the end product is delivered without considerable defect, the sooner feedback can be received, and incorporated into the next iteration. The shorter the iterations, the better the learning and communication within the team. Without speed, decisions cannot be delayed. Speed assures the fulfilling of the customer's present needs and not what they required yesterday. This gives them the opportunity to delay making up their minds about what they really require until they gain better knowledge. Customers value rapid delivery of a quality product.

The Just-in-Time production ideology could be applied to software development, recognizing its specific requirements and environment. This is achieved by presenting the needed result and letting the team organize itself and divide the tasks for accomplishing the needed result for a specific iteration. At the beginning, the customer provides the needed input. This could be simply presented in small cards or stories – the developers estimate the time needed for the implementation of each card. Thus the work organization changes into self-pulling system – each morning during a stand-up meeting, each member of the team reviews what has been done yesterday, what is to be done today and tomorrow, and prompts for any inputs needed from colleagues or the customer. This requires transparency of the process, which is also beneficial for team communication. Another key idea in Toyota's Product Development System is set-based design. If a new brake system is needed for a car, for example, three teams may design solutions to the same problem. Each team learns about the problem space and designs a potential solution. As a solution is deemed unreasonable, it is cut. At the end of a period, the surviving designs are compared and one is chosen, perhaps with some modifications based on learning from the others - a great example of deferring commitment until the last possible moment. Software decisions could also benefit from this practice to minimize the risk brought on by big up-front design.

Empower the team

There has been a traditional belief in most businesses about the decision-making in the organisation – the managers tell the workers how to do their own job. In a Work-Out technique, the roles are turned – the managers are taught how to listen to the developers, so they can explain better what actions might be taken, as well as provide suggestions for improvements. The lean approach favors the aphorism "find good people and let them do their own job," encouraging progress, catching errors, and removing impediments, but not micro-managing.

Another mistaken belief has been the consideration of people as resources. People might be resources from the point of view of a statistical data sheet, but in software development, as well as any organisational business, people do need something more than just the list of tasks and the assurance that they will not be disturbed during the completion of the tasks. People need motivation and a higher purpose to work for – purpose within the reachable reality, with the assurance that the team might choose its own commitments. The developers should be given access to customer; the team leader should provide support and help in difficult situations, as well as make sure that skepticism does not ruin the team's spirit.

Build integrity in

The customer needs to have an overall experience of the System – this is the so called perceived integrity: how it is being advertised, delivered, deployed, accessed, how intuitive its use is, price and how well it solves problems.

Conceptual integrity means that the system's separate components work well together as a whole with balance between flexibility, maintainability, efficiency, and responsiveness. This could be achieved by understanding the problem domain and solving it at the same time, not sequentially. The needed information is received in small batch pieces – not in one vast chunk with preferable face-to-face communication and not any written documentation. The information flow should be constant in both directions – from customer to developers and back, thus avoiding the large stressful amount of information after long development in isolation.

One of the healthy ways towards integral architecture is refactoring. The more features are added to the System, the more loose the starting code base for further improvements. As described above in the XP agile method refactoring is about keeping simplicity, clarity, minimum amount of features in the code. Repetitions in the code are signs for bad code designs and should be avoided. The complete and automated building process should be accompanied by a complete and automated suite of developer and customer tests, having the same versioning, synchronization and semantics as the current state of the System. At the end the integrity should be verified with thorough testing, thus ensuring the System does what the customer expects it to. Automated tests are also considered part of the production process, and therefore if they do not add value they should be considered waste. Automated testing should not be a goal, but rather a means to an end, specifically the reduction of defects.

See the whole

Software systems nowadays are not simply the sum of their parts, but also the product of their interactions. Defects in software tend to accumulate during the development process – by decomposing the big tasks into smaller tasks, and by standardizing different stages of development, the root causes of defects should be found and eliminated. The larger the system, the more organisations that are involved in its development and the more parts are developed by different teams, the greater the importance of having well defined

relationships between different vendors, in order to produce a system with smoothly interacting components. During a longer period of development, a stronger sub-contractor network is far more beneficial than short-term profit optimizing, which does not enable win-win relationships.

Lean thinking has to be understood well by all members of a project, before implementing in a concrete, real-life situation. “Think big, act small, fail fast; learn rapidly” – these slogans summarize the importance of understanding the field and the suitability of implementing lean principles along the whole software development process. Only when all of the lean principles are implemented together, combined with strong “common sense” with respect to the working environment, is there a basis for success in software development.

Lean software practices

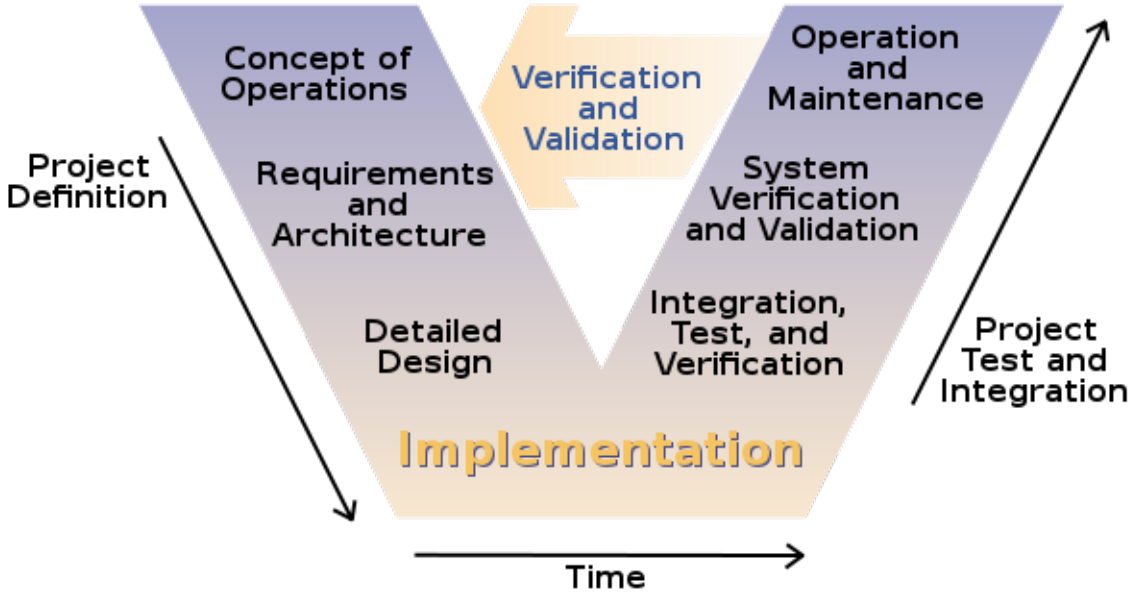
Lean software development practices, or what the Poppendiecks call "tools" are expressed slightly differently from their equivalents in agile software development, but there are parallels. Examples of such practices include:

- Seeing waste
- Value stream mapping
- Set-based development
- Pull systems
- Queuing theory
- Motivation
- Measurements

Some of the tools map quite easily to *agile* methods. Lean Workcells, for example are expressed in Agile methods as cross-functional teams.

Chapter 15

V-Model (Software Development)



The V-model of the Systems Engineering Process.

The **V-model** represents a software development process (also applicable to hardware development) which may be considered an extension of the waterfall model. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represents time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively.

Verification Phases

Requirements analysis

In the Requirements analysis phase, the requirements of the proposed system are collected by analyzing the needs of the user(s). This phase is concerned about establishing what the ideal system has to perform. However it does not determine how the software will be designed or built. Usually, the users are interviewed and a document called the user requirements document is generated.

The user requirements document will typically describe the system's functional, interface, performance, data, security, etc requirements as expected by the user. It is used by business analysts to communicate their understanding of the system to the users. The users carefully review this document as this document would serve as the guideline for the system designers in the system design phase. The user acceptance tests are designed in this phase.

There are different methods for gathering requirements of both soft and hard methodologies including; interviews, questionnaires, document analysis, observation, throw-away prototypes, use cases and status and dynamic views with users.

System Design

Systems design is the phase where system engineers analyze and understand the business of the proposed system by studying the user requirements document. They figure out possibilities and techniques by which the user requirements can be implemented. If any of the requirements are not feasible, the user is informed of the issue. A resolution is found and the user requirement document is edited accordingly.

The software specification document which serves as a blueprint for the development phase is generated. This document contains the general system organization, menu structures, data structures etc. It may also hold example business scenarios, sample windows, reports for the better understanding. Other technical documentation like entity diagrams, data dictionary will also be produced in this phase. The documents for system testing are prepared in this phase.

Architecture Design

The phase of the design of computer architecture and software architecture can also be referred to as high-level design. The baseline in selecting the architecture is that it should realize all which typically consists of the list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology details etc. The integration testing design is carried out in the particular phase.

Module Design

The module design phase can also be referred to as low-level design. The designed system is broken up into smaller units or modules and each of them is explained so that the programmer can start coding directly. The low level design document or program specifications will contain a detailed functional logic of the module, in pseudocode:

- database tables, with all elements, including their type and size
- all interface details with complete API references
- all dependency issues
- error message listings
- complete input and outputs for a module.

The unit test design is developed in this stage.

Validation Phases

Unit Testing

In computer programming, unit testing is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers. The purpose is to verify the internal logic code by testing every possible branch within the function, also known as test coverage. Static analysis tools are used to facilitate in this process, where variations of input data are passed to the function to test every possible case of execution.

Integration Testing

In integration testing the separate modules will be tested together to expose faults in the interfaces and in the interaction between integrated components. Testing is usually black box as the code is not directly checked for errors.

System Testing

System testing will compare the system specifications against the actual system. After the integration test is completed, the next test level is the system test. System testing checks if the integrated product meets the specified requirements. Why is this still necessary after the component and integration tests? The reasons for this are as follows:

Reasons for system test

1. In the lower test levels, the testing was done against technical specifications, i.e., from the technical perspective of the software producer. The system test, though,

looks at the system from the perspective of the customer and the future user. The testers validate whether the requirements are completely and appropriately met.
    - Example: The customer (who has ordered and paid for the system) and the user (who uses the system) can be different groups of people or organizations with their own specific interests and requirements of the system.
2. Many functions and system characteristics result from the interaction of all system components, consequently, they are only visible on the level of the entire system and can only be observed and tested there.

User Acceptance Testing

Acceptance testing is the phase of testing used to determine whether a system satisfies the requirements specified in the requirements analysis phase. The acceptance test design is derived from the requirements document. The acceptance test phase is the phase used by the customer to determine whether to accept the system or not.

Acceptance testing helps

- to determine whether a system satisfies its acceptance criteria or not.
- to enable the customer to determine whether to accept the system or not.
- to test the software in the "real world" by the intended audience.

Purpose of acceptance testing:

- to verify the system or changes according to the original needs.

Procedures

1. Define the acceptance criteria:
    - Functionality requirements.
    - Performance requirements.
    - Interface quality requirements.
    - Overall software quality requirements.
2. Develop an acceptance plan:
    - Project description.
    - User responsibilities.
    - Acceptance description.
    - Execute the acceptance test plan.

Chapter 16

Test-Driven Development

Test-driven development (**TDD**) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

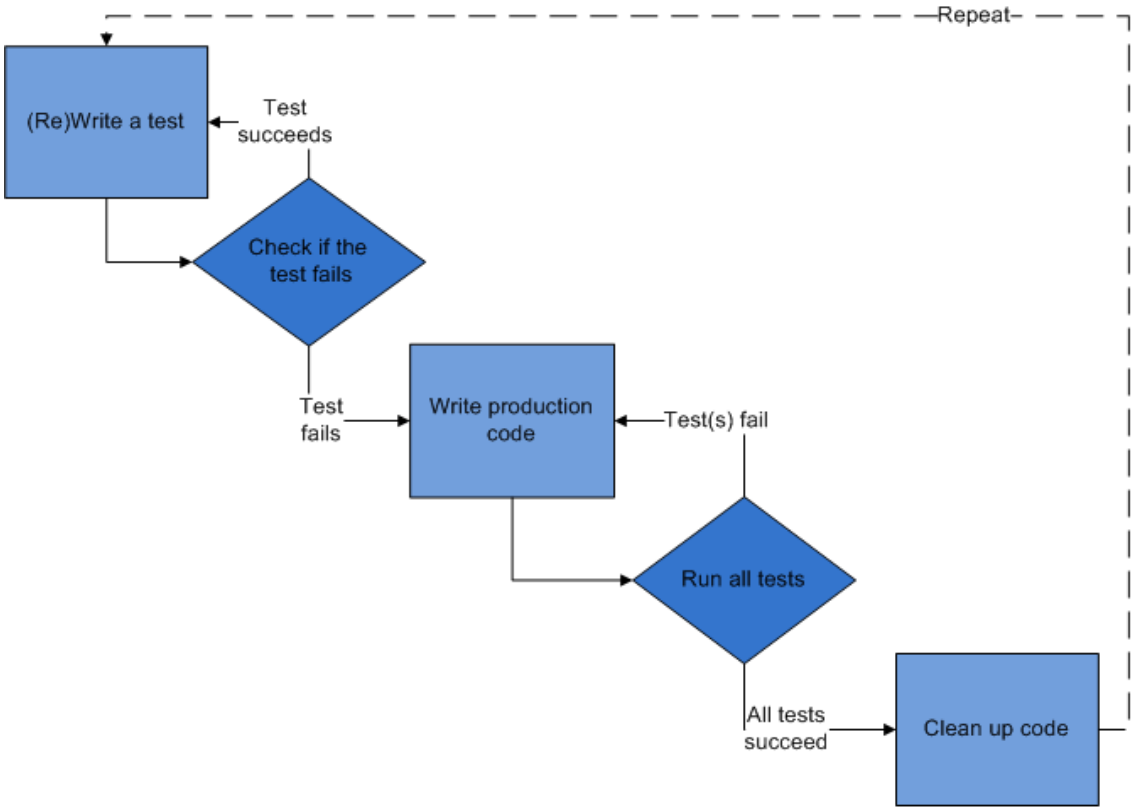
Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.

Requirements

Test-driven development requires developers to create automated unit tests that define code requirements (immediately) before writing the code itself. The tests contain assertions that are either true or false. Passing the tests confirms correct behavior as developers evolve and refactor the code. Developers often use testing frameworks, such as xUnit, to create and automatically run sets of test cases.

Test-driven development cycle



A graphical representation of the development cycle, using a basic flowchart

The following sequence is based on the book *Test-Driven Development by Example*.

Add a test

In **test-driven development**, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. (If it does not fail, then either the proposed "new" feature already exists or the test is defective.) To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories that cover the requirements and exception conditions. This could also imply a variant, or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.

Run all tests and see if the new one fails

This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. This step also tests the test itself, in the negative: it rules out the possibility that the new test will always pass, and therefore be

worthless. The new test should also fail for the expected reason. This increases confidence (although it does not entirely guarantee) that it is testing the right thing, and will pass only in intended cases.

Write some code

The next step is to write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it.

It is important that the code written is *only* designed to pass the test; no further (and therefore untested) functionality should be predicted and 'allowed for' at any stage.

Run the automated tests and see them succeed

If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

Refactor code

Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality. The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to removing any duplication between the test code and the production code — for example magic numbers or strings that were repeated in both, in order to make the test pass in step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous Integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the main program being written.

Development style

There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS) and "You ain't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can be cleaner and clearer than is often achieved by other methods. In *Test-Driven Development by Example* Kent Beck also suggests the principle "Fake it till you make it".

To achieve some advanced design concept (such as a design pattern), tests are written that will generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Write the tests first. The tests should be written before the functionality that is being tested. This has been claimed to have two benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset, rather than worrying about it later. It also ensures that tests for every feature will be written. When writing feature-first code, there is a tendency by developers and the development organisations to push the developer onto the next feature, neglecting testing entirely.

First fail the test cases. The idea is to ensure that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has been coined the "test-driven development mantra", known as red/green/refactor where red means *fail* and green is *pass*.

Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the programmer's mental model of the code, boosts confidence and increases productivity.

Advanced practices of test-driven development can lead to Acceptance Test-driven development (ATDD) where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process. This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target to satisfy, the acceptance tests, which keeps them continuously focused on what the customer really wants from that user story.

Benefits

A 2005 study found that using TDD meant writing more tests and, in turn, programmers that wrote more tests tended to be more productive. Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.

Programmers using pure TDD on new ("greenfield") projects report they only rarely feel the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality will be used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is

complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, total code implementation time is typically shorter. Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, in order for a TDD developer to add an `else` branch to an existing `if` statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they will detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Vulnerabilities

- Test-driven development is difficult to use in situations where full functional tests are required to determine success or failure. Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.
- Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.

- Unit tests created in a test-driven development environment are typically created by the developer who will also write the code that is being tested. The tests may therefore share the same blind spots with the code: If, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify these input parameters. If the developer misinterprets the requirements specification for the module being developed, both the tests and the code will be wrong.
- The high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.
- The tests themselves become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or which are themselves prone to failure, are expensive to maintain. This is especially the case with Fragile Tests. There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs it may not be detected. It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase described above.
- The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original tests become increasingly precious as time goes by. If a poor architecture, a poor design or a poor testing strategy leads to a late change that makes dozens of existing tests fail, it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

Code visibility

Test suite code clearly has to be able to access the code it is testing. On the other hand normal design criteria such as information hiding, encapsulation and the separation of concerns should not be compromised. Therefore unit test code for TDD is usually written within the same project or module as the code being tested.

In object oriented design this still does not provide access to `private` data and methods. Therefore, extra work may be necessary for unit tests. In Java and other languages, a developer can use reflection to access fields that are marked `private`. Alternatively, an inner class can be used to hold the unit tests so they will have visibility of the enclosing class's members and attributes. In the .NET Framework and some other programming languages, partial classes may be used to expose private methods and data for the tests to access.

It is important that such testing hacks do not remain in the production code. In C and other languages, compiler directives such as `#if DEBUG ... #endif` can be placed around such additional classes and indeed all other test-related code to prevent them being compiled into the released code. This then means that the released code is not exactly the same as that which is unit tested. The regular running of fewer but more comprehensive, end-to-end, integration tests on the final release build can then ensure

(among other things) that no production code exists that subtly relies on aspects of the test harness.

There is some debate among practitioners of TDD, documented in their blogs and other writings, as to whether it is wise to test private and protected methods and data anyway. Some argue that it should be sufficient to test any class through its public interface as the private members are a mere implementation detail that may change, and should be allowed to do so without breaking numbers of tests. Others say that crucial aspects of functionality may be implemented in private methods, and that developing this while testing it indirectly via the public interface only obscures the issue: unit testing is about testing the smallest unit of functionality possible.

Fakes, mocks and integration tests

Unit tests are so named because they each test *one unit* of code. A complex module may have a thousand unit tests and a simple one only ten. The tests used for TDD should never cross process boundaries in a program, let alone network connections. Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite. Introducing dependencies on external modules or data also turns *unit tests* into *integration tests*. If one module misbehaves in a chain of interrelated modules, it is not so immediately clear where to look for the cause of the failure.

When code under development relies on a database, a web service, or any other external process or service, enforcing a unit-testable separation is also an opportunity and a driving force to design more modular, more testable and more reusable code. Two steps are necessary:

1. Whenever external access is going to be needed in the final design, an interface should be defined that describes the access that will be available.
2. The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock. Fake objects need do little more than add a message such as “Person object saved” to a trace log, against which a test assertion can be run to verify correct behaviour. Mock objects differ in that they themselves contain test assertions that can make the test fail, for example, if the person's name and other data are not as expected. Fake and mock object methods that return data, ostensibly from a data store or user, can help the test process by always returning the same, realistic data that tests can rely upon. They can also be set into predefined fault modes so that error-handling routines can be developed and reliably tested. Fake services other than data stores may also be useful in TDD: Fake encryption services may not, in fact, encrypt the data passed; fake random number services may always return 1. Fake or mock implementations are examples of dependency injection.

A corollary of such dependency injection is that the actual database or other external-access code is never tested by the TDD process itself. To avoid errors that may arise from this, other tests are needed that instantiate the test-driven code with the “real”

implementations of the interfaces discussed above. These tests are quite separate from the TDD unit tests, and are really integration tests. There will be fewer of them, and they need to be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, such as xUnit.

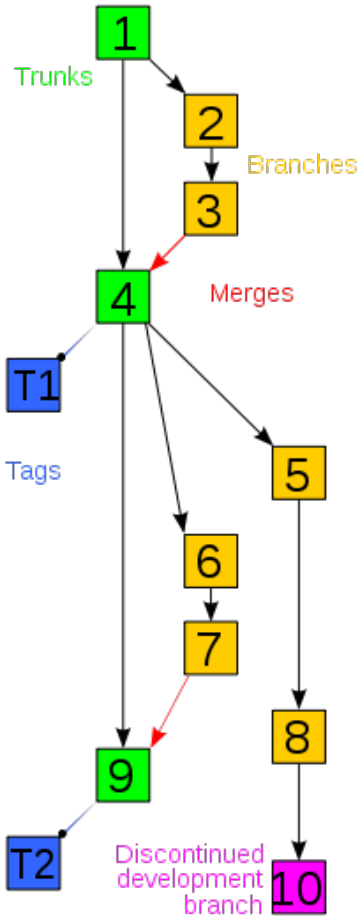
Integration tests that alter any persistent store or database should always be designed carefully with consideration of the initial and final state of the files or database, even if any test fails. This is often achieved using some combination of the following techniques:

- The `TearDown` method, which is integral to many test frameworks.
- `try...catch...finally` exception handling structures where available.
- Database transactions where a transaction atomically includes perhaps a write, a read and a matching delete operation.
- Taking a "snapshot" of the database before running any tests and rolling back to the snapshot after each test run. This may be automated using a framework such as Ant or NAnt or a continuous integration system such as CruiseControl.
- Initialising the database to a clean state *before* tests, rather than cleaning up *after* them. This may be relevant where cleaning up may make it difficult to diagnose test failures by deleting the final state of the database before detailed diagnosis can be performed.

Frameworks such as Moq, jMock, NMock, EasyMock, Typemock, jMockit, Unitils, Mockito, Mockachino, PowerMock or Rhino Mocks exist to make the process of creating and using complex mock objects easier.

Chapter 17

Revision Control



Example history tree of a revision-controlled project.

Revision control, also known as **version control** or **source control** (and an aspect of **software configuration management** or SCM), is the management of changes to documents, programs, and other information stored as computer files. It is most commonly used in software development, where a team of people may change the same files. Changes are usually identified by a number or letter code, termed the "revision

number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

Version control systems (VCSs – singular VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors (e.g., Microsoft Word, OpenOffice.org Writer, KWord, Pages, etc.), spreadsheets (e.g., Microsoft Excel, OpenOffice.org Calc, KSpread, Numbers, etc.), and in various content management systems.

Software tools for revision control are essential for the organization of multi-developer projects.

Overview

Engineering revision control developed from formalized processes based on tracking revisions of early blueprints or bluelines. This system of control implicitly allowed returning to any earlier state of the design, for cases in which an engineering dead-end was reached in the development of the design. Likewise, in computer software engineering, revision control is any practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code. Also, version control is widespread in business and law. Indeed, "contract redline" and "legal blackline" are some of the earliest forms of revision control, and are still employed with varying degrees of sophistication. An entire industry has emerged to service the document revision control needs of business and other users, and some of the revision control technology employed in these circles is subtle, powerful, and innovative. The most sophisticated techniques are beginning to be used for the electronic tracking of changes to CAD files, supplanting the "manual" electronic implementation of traditional revision control.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used on many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of

self-discipline on the part of developers, and often leads to mistakes. Consequently, systems to automate some or all of the revision control process have been developed.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even necessary in such situations.

Revision control may also track changes to configuration files, such as those typically stored in `/etc` or `/usr/local/etc` on Unix systems. This gives system administrators another way to easily track changes made and a way to roll back to earlier versions should the need arise.

Source-management models

Traditional revision control systems use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some method of managing access the developers may end up overwriting each other's work. Centralized revision control systems solve this problem in one of two different "source management models": file locking and version merging.

Atomic operations

Computer scientists speak of *atomic* operations if the system is left in a consistent state even if the operation is interrupted. The *commit* operation is usually the most critical in this sense. Commits are operations which tell the revision control system you want to make a group of changes you have been making final and available to all users. Not all revision control systems have atomic commits; notably, the widely-used CVS lacks this feature.

File locking

The simplest method of preventing "concurrent access" problems involves locking files so that only one developer at a time has write access to the central "repository" copies of those files. Once one developer "checks out" a file, others can read that file, but no one else may change that file until that developer "checks in" the updated version (or cancels the checkout).

File locking has both merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

Version merging

Most version control systems allow multiple developers to edit the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system may provide facilities to merge further changes into the central repository, and preserve the changes from the first developer when other developers check in.

Merging two files can be a very delicate operation, and usually possible only if the data structure is simple, as in text files. The result of a merge of two image files might not result in an image file at all. The second developer checking in code will need to take care with the merge, to make sure that the changes are compatible and that the merge operation does not introduce its own logic errors within the files. These problems limit the availability of automatic or semi-automatic merge operations mainly to simple text based documents, unless a specific merge plugin is available for the file types.

The concept of a *reserved edit* can provide an optional means to explicitly lock a file for exclusive write access, even when a merging capability exists.

Baselines, labels and tags

Most revision control tools will use only one of these similar terms (baseline, label, tag) to refer to the action of identifying a snapshot ("label the project") or the record of the snapshot ("try it with baseline *X*"). Typically only one of the terms *baseline*, *label*, or *tag* is used in documentation or discussion; they can be considered synonyms.

In most projects some snapshots are more significant than others, such as those used to indicate published releases, branches, or milestones.

When both the term *baseline* and either of *label* or *tag* are used together in the same context, *label* and *tag* usually refer to the mechanism within the tool of identifying or making the record of the snapshot, and *baseline* indicates the increased significance of any given label or tag.

Most formal discussion of configuration management uses the term *baseline*.

Distributed revision control

Distributed revision control (DRCS) takes a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository. Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer. This results in some important differences from a centralized system:

- No canonical, reference copy of the codebase exists by default; only working copies.

- Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server.

Rather, communication is only necessary when pushing or pulling changes to or from other peers.

- Each working copy effectively functions as a remote backup of the codebase and of its change-history, providing natural protection against data loss.

Integration

Some of the more advanced revision-control tools offer many other facilities, allowing deeper integration with other tools and software-engineering processes. Plugins are often available for IDEs such as Oracle JDeveloper, IntelliJ IDEA, Eclipse and Visual Studio. NetBeans IDE and Xcode come with integrated version control support.

Common vocabulary

Terminology can vary from system to system, but some terms in common usage include:

Baseline
: An approved revision of a document or source file from which subsequent changes can be made.

Branch
: A set of files under version control may be **branched** or **forked** at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.

Change
: A **change** (or **diff**, or **delta**) represents a specific modification to a document under version control. The granularity of the modification considered a change varies between version control systems.

Change list
: On many version control systems with atomic multi-change commits, a **changelist**, **change set**, or **patch** identifies the set of **changes** made in a single commit. This can also represent a sequential view of the source code, allowing the examination of source "as of" any particular changelist ID.

Checkout
: A **check-out** (or **co**) is the act of creating a local working copy from the repository. A user may specify a specific revision or obtain the latest. The term 'checkout' can also be used as a noun to describe the working copy.

Commit
: A **commit** (**checkin**, **ci** or, more rarely, **install**, **submit** or **record**) is the action of writing or merging the changes made in the working copy back to the repository. The terms 'commit' and 'checkin' can also used in noun form to describe the new revision that is created as a result of committing.

Conflict

A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must **resolve** the conflict by combining the changes, or by selecting one change in favour of the other.

Delta compression
: Most revision control software uses delta compression, which retains only the differences between successive versions of files. This allows for more efficient storage of many different versions of files.

Dynamic stream
: A stream in which some or all file versions are mirrors of the parent stream's versions.

Export
: **exporting** is the act of obtaining the files from the repository. It is similar to **checking-out** except that it creates a clean directory tree without the version-control metadata used in a working copy. This is often used prior to publishing the contents, for example.

Head
: Also sometime called **tip**, this refers to the most recent commit.

Import
: **importing** is the act of copying a local directory tree (that is not currently a working copy) into the repository for the first time.

Mainline
: Similar to *trunk*, but there can be a mainline for each branch.

Merge
: A **merge** or **integration** is an operation in which two sets of changes are applied to a file or set of files. Some sample scenarios are as follows:

  - A user, working on a set of files, **updates** or **syncs** their working copy with changes made, and checked into the repository, by other users.

  - A user tries to **check-in** files that have been updated by others since the files were **checked out**, and the **revision control software** automatically merges the files (typically, after prompting the user if it should proceed with the automatic merge, and in some cases only doing so if the merge can be clearly and reasonably resolved).
  - A set of files is **branched**, a problem that existed before the branching is fixed in one branch, and the fix is then merged into the other branch.
  - A **branch** is created, the code in the files is independently edited, and the updated branch is later incorporated into a single, unified **trunk**.

Promote
: The act of copying file content from a less controlled location into a more controlled location. For example, from a user's workspace into a repository, or from a stream to its parent.

Repository

The **repository** is where files' current and historical data are stored, often on a server. Sometimes also called a **depot** (for example, by SVK, AccuRev and Perforce).

Resolve
: The act of user intervention to address a conflict between different changes to the same document.

Reverse integration
: The process of merging different team branches into the main trunk of the versioning system.

Revision
: Also **version**: A version is any change in form. In SVK, a Revision is the state at a point in time of the entire tree in the repository.

Ring
: See **tag**.

Share
: The act of making one file or folder available in multiple branches at the same time. When a shared file is changed in one branch, it is changed in other branches.

Stream
: A container for branched files that has a known relationship to other such containers. Streams form a hierarchy; each stream can inherit various properties (like versions, namespace, workflow rules, subscribers, etc.) from its parent stream.

Tag
: A **tag** or **label** refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number.

Trunk
: The unique line of development that is not a branch (sometimes also called Baseline or Mainline)

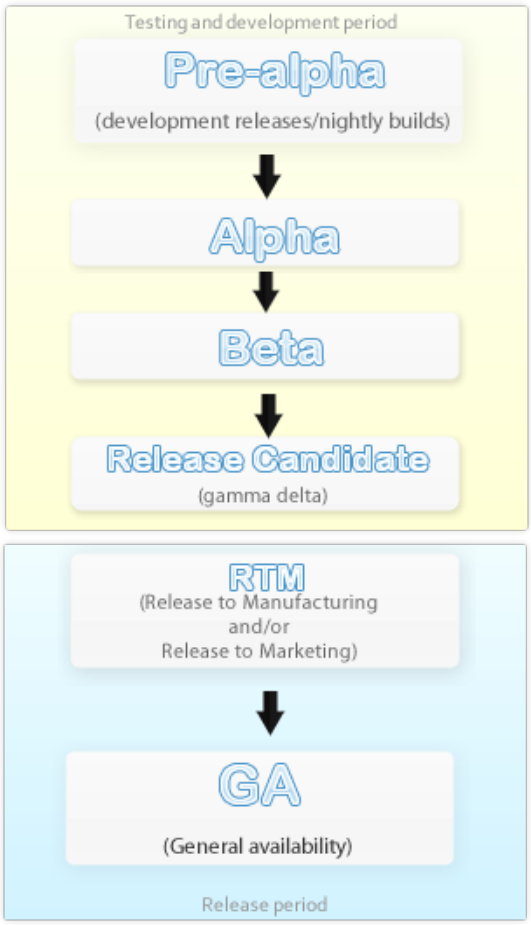
Update
: An **update** (or **sync**) merges changes made in the repository (by other people, for example) into the local **working copy**.

Working copy
: The **working copy** is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy, hence the name. Conceptually, it is a *sandbox*.

Chapter 18

Software Release Life Cycle

Software Development and Release Stages



A **software release** is the distribution of software code, documentation, and support materials. The **software release life cycle** is composed of discrete phases that describe the software's maturity as it advances from planning and development to release and support phases.

Development

Pre-alpha

Pre-alpha refers to all activities performed during the software project prior to testing. These activities can include requirements analysis, software design, software development and unit testing.

In typical open source development, there are several types of pre-alpha versions. **Milestone** versions include specific sets of functions and are released as soon as the functionality is complete.

Alpha

The alpha phase of the release life cycle is the first phase to begin software testing (alpha is the first letter of the ancient Greek alphabet, used as the number 1). In this phase, developers generally test the software using white box techniques. Additional validation is then performed using black box or gray box techniques, by another testing team. Moving to black box testing inside the organization is known as *alpha release*.

Alpha software can be unstable and could cause crashes or data loss. The exception to this is when the alpha is available publicly (such as a pre-order bonus), in which developers normally push for stability so that their testers can test properly. External availability of alpha software is uncommon.

The alpha phase usually ends with a feature freeze, indicating that no more features will be added to the software. At this time, the software is said to be feature complete.

Beta

Beta is the software development phase following alpha (beta is the second letter of the ancient Greek alphabet, used as the number 2. It is not nowadays usual to speak of a later gamma test). It generally begins when the software is feature complete. The focus of beta testing is reducing impacts to users, often incorporating usability testing. The process of delivering a beta version to the users is called **beta release** and this is typically the first time that the software is available outside of the organization that developed it.

The users of a beta version are called *beta testers*. They are usually customers or prospective customers of the organization that develops the software, willing to test the software without charge, often receiving the final software free of charge or for a reduced price.

Beta version software is often useful for demonstrations and previews within an organisation and to prospective customers. Some developers refer to this stage as a **preview**, **prototype**, **technical preview** (**TP**), or **early access**.

Some software is kept in perpetual beta.

Open and closed beta

Developers release either a **closed beta** or an **open beta**; closed beta versions are released to a select group of individuals for a user test, while open betas are to a larger community group, sometimes to anyone interested. The testers report any bugs that they find, and sometimes suggest additional features they think should be available in the final version. Examples of a major public beta test are:

- In September 2000 a *boxed version* of Apple Inc.'s Mac OS X Public Beta operating system was released.
- Microsoft's release of **community technology previews** (**CTPs**) for Windows Vista in January 2005.

Open betas serve the dual purpose of demonstrating a product to potential consumers, and testing among an extremely wide user base likely to bring to light obscure errors that a much smaller testing team may not find.

Though, online Free-to-play (F2P) games have developed a trend to use Open Beta to lure customers with an intent to profit off customers in a same aspect if such F2P were a final product. Many such games can be locked in Open Beta for years before going final for example Conquer Online. Open Beta in F2P is coming a more meaningless concept for testing compared to a powerful business model for profit with Cash shop items either in game or on official website. Such purchases can only be done by exchanging real money for Item mall currency. Such currency exchange varies from F2P to F2P.

Release candidate

The term **release candidate** (**RC**) refers to a version with potential to be a final product, ready to release unless fatal bugs emerge. In this stage of product stabilization, all product features have been designed, coded and tested through one or more beta cycles with no known showstopper-class bug.

Apple Inc. uses the term "golden master" for its release candidates, and the final golden master is used as the general availability release. Other Greek letters, such as **gamma** and **delta**, are sometimes used to indicate versions that are substantially complete, but still undergoing testing, with **omega** or **zenith** used to indicate final testing versions that are believed to be relatively bug-free, ready for production.

A release is called **code complete** when the development team agrees that no entirely new source code will be added to this release. There may still be source code changes to fix defects. There may still be changes to documentation and data files, and to the code for test cases or utilities. New code may be added in a future release.

Origin of alpha and beta terminology

The term **beta test** comes from an IBM convention, dating back to punched card tabulating and sorting machines. Hardware first went through an **alpha test** for preliminary functionality and small scale manufacturing feasibility. Then came a **beta test**, to verify that the hardware correctly performed the intended functions and could be manufactured at scale. Ultimately, a **gamma test** was performed to verify safety.

When IBM began testing software, it used the same terminology as it had for hardware. As other companies began developing their own software, they adopted and kept the same terminology.

Release

RTM

The term "**release to manufacturing**" or "**release to marketing**" (both abbreviated RTM, initials also commonly used for the quite different "return to manufacturer" of faulty goods)—also known as "going gold"—is a term used when software is ready for or has been delivered or provided to the customer. It is typically used in certain retail mass-production software contexts—as opposed to a specialized software production or project in a commercial or government production and distribution—where the software is sold as part of a bundle in a related computer hardware sale and typically where the software and related hardware is ultimately to be available and sold on mass/public basis at retail stores to indicate that the software has met a defined quality level and is ready for mass retail distribution. RTM could also mean in other contexts that the software has been delivered or released to a client or customer for installation or distribution to the related hardware end user computers or machines. The term does *not* define the delivery mechanism or volume; it only states that the quality is sufficient for mass distribution. The deliverable from the engineering organization is frequently in the form of a **gold** master CD used for duplication or to produce the image for the web.

RTM precedes general availability (GA) when the product is released to the public.

General availability

General availability or **general acceptance** (**GA**) is the point where all necessary commercialization activities have been completed and the software has been made available to the general market either via the web or physical media. Another term with a meaning almost identical to GA is **first customer shipment** (**FCS**). Some companies (such as Sun Microsystems and Cisco) use FCS to describe a software version that has been shipped for revenue.

Commercialization activities could include but are not limited to the availability of media world wide via dispersed distribution centers, marketing collateral is completed and available in as many languages as deemed necessary for the target market, the finishing

of security and compliance tests, etc. The time between RTM and GA can be from a week to months in some cases before a generally available release can be declared because of the time needed to complete all commercialization activities required by GA.

It is also at this stage that the software is considered to have "gone live". The **production**, **live** version is the final version of a particular product. A live release is considered to be very stable and relatively bug-free with a quality suitable for wide distribution and use by end users. In commercial software releases, this version may also be signed (used to allow end-users to verify that code has not been modified since the release). The expression that a software product "has gone live" means that the code has been completed and is ready for distribution. Other terms for the live version include *live master*, *live release*, and *live build*.

In some areas of software development it is at this stage that the release is referred to as a **gold** release; this seems to be confined mainly to game software.

Some release versions might be classified as a long term support (LTS) release, which should guarantee the ability to upgrade to the next LTS release and will be supported/updated/patched for a longer time than a non-LTS release.

Support

Service release

During its supported lifetime, software is sometimes subjected to service releases, or service packs. As a well used example, Microsoft's Windows XP has currently had 3 major Service Packs.

Such service releases contain a collection of updates, fixes and/or enhancements, delivered in the form of a single installable package. They may also contain entirely new features.

Certain software is released with the expectation of regular support. Classes of software that generally involve protracted support as the norm include anti-virus suites and massively multiplayer online games.

End-of-life

When software is no longer sold or supported, the product is said to have reached end-of-life.

Impact of the World Wide Web

As the Internet has allowed for rapid and inexpensive distribution of software, companies have begun to take a more loose approach to use of the word "beta". Netscape Communications was infamous for releasing alpha level versions of its Netscape web

browser to the public and calling them "beta" releases. In February 2005, ZDNet published an article about the recent phenomenon of a beta version often staying for years and being used as if it were in production level. It noted that Gmail and Google News, for example, had been in beta for a long period of time and were not expected to drop the beta status despite the fact that they were widely used; however, Google News did leave beta in January 2006, followed by Google Apps, including Gmail, in July 2009. This technique may also allow a developer to delay offering full support and/or responsibility for remaining issues. In the context of Web 2.0, people even talk of perpetual betas to signify that some software is meant to stay in beta state. Also, "beta" is sometimes used to indicate something more like a release candidate, or as a form of time limited demo, or marketing technique.

Some users disparagingly refer to release candidates and even final "point oh" releases as "gamma test" software, suggesting that the developer has chosen to use its customers to test software that is not truly ready for general release. Beta testers, if privately selected, will often be credited for using the release candidate as though it were a finished product.

Web release

A **web release** is a means of software delivery that utilizes the Internet for distribution. No physical media are produced in this type of release mechanism by the manufacturer. This is sometimes also referred to as *release to web* (RTW).