Software Project Management

Lise Pollack

First Edition, 2012

ISBN 978-81-323-4027-0

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 Project Management

Software project management is the art and science of planning and leading software projects. It is a sub-discipline of project management in which software projects are planned, monitored and controlled.

History

The history of software project management is closely related to the history of software. Software was developed for dedicated purposes for dedicated machines until the concept of object-oriented programming began to become popular in the 1960's, making *repeatable solutions* possible for the software industry. Dedicated systems could be adapted to other uses thanks to component-based software engineering. Companies quickly understood the relative ease of use that software programming had over hardware circuitry, and the software industry grew very quickly in the 1970's and 1980's. To manage new development efforts, companies applied proven project management methods, but project schedules slipped during test runs, especially when confusion occurred in the gray zone between the user specifications and the delivered software. To be able to avoid these problems, software project management methods focused on matching user requirements to delivered products, in a method known now as the waterfall model. Since then, analysis of software project management failures has shown that the following are the most common causes:

1. Unrealistic or unarticulated project goals
2. Inaccurate estimates of needed resources
3. Badly defined system requirements
4. Poor reporting of the project's status
5. Unmanaged risks
6. Poor communication among customers, developers, and users
7. Use of immature technology
8. Inability to handle the project's complexity
9. Sloppy development practices
10. Poor project management
11. Stakeholder politics

12. Commercial pressures

The first three items in the list above show the difficulties articulating the needs of the client in such a way that proper resources can deliver the proper project goals. Specific software project management tools are useful and often necessary, but the true art in software project management is applying the correct method and then using tools to support the method. Without a method, tools are worthless. Since the 1960's, several proprietary software project management methods have been developed by software manufacturers for their own use, while computer consulting firms have also developed similar methods for their clients. Today software project management methods are still evolving, but the current trend leads away from the waterfall model to a more cyclic project delivery model that imitates a Software release life cycle.

Software development process

A software development process is concerned primarily with the production aspect of software development, as opposed to the technical aspect, such as software tools. These processes exist primarily for supporting the management of software development, and are generally skewed toward addressing business concerns. Many software development processes can be run in a similar way to general project management processes. Examples are:

- Risk management is the process of measuring or assessing risk and then developing strategies to manage the risk. In general, the strategies employed include transferring the risk to another party, avoiding the risk, reducing the negative effect of the risk, and accepting some or all of the consequences of a particular risk. Risk management in software project management begins with the business case for starting the project, which includes a cost-benefit analysis as well as a list of fallback options for project failure, called a contingency plan.
    - A subset of risk management that is gaining more and more attention is "Opportunity Management", which means the same thing, except that the potential risk outcome will have a positive, rather than a negative impact. Though theoretically handled in the same way, using the term "opportunity" rather than the somewhat negative term "risk" helps to keep a team focussed on possible positive outcomes of any given risk register in their projects, such as spin-off projects, windfalls, and free extra resources.
- Requirements management is the process of identifying, eliciting, documenting, analyzing, tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders. New or altered computer system Requirements management, which includes Requirements analysis, is an important part of the software engineering process; whereby business analysts or software developers identify the needs or requirements of a client; having identified these requirements they are then in a position to design a solution.
- Change management is the process of identifying, documenting, analyzing, prioritizing and agreeing on changes to scope (project management) and then controlling changes and communicating to relevant stakeholders. Change impact

analysis of new or altered scope, which includes Requirements analysis at the change level, is an important part of the software engineering process; whereby business analysts or software developers identify the altered needs or requirements of a client; having identified these requirements they are then in a position to re-design or modify a solution. Theoretically, each change can impact the timeline and budget of a software project, and therefore by definition must include risk-benefit analysis before approval.

- Software configuration management is the process of identifying, and documenting the scope itself, which is the software product underway, including all sub-products and changes and enabling communication of these to relevant stakeholders. In general, the processes employed include version control, naming convention (programming), and software archival agreements.
- Release management is the process of identifying, documenting, prioritizing and agreeing on releases of software and then controlling the release schedule and communicating to relevant stakeholders. Most software projects have access to three software environments to which software can be released; Development, Test, and Production. In very large projects, where distributed teams need to integrate their work before release to users, there will often be more environments for testing, called unit testing, system testing, or integration testing, before release to User acceptance testing (UAT).
    - A subset of release management that is gaining more and more attention is Data Management, as obviously the users can only test based on data that they know, and "real" data is only in the software environment called "production". In order to test their work, programmers must therefore also often create "dummy data" or "data stubs". Traditionally, older versions of a production system were once used for this purpose, but as companies rely more and more on outside contributors for software development, company data may not be released to development teams. In complex environments, datasets may be created that are then migrated across test environments according to a test release schedule, much like the overall software release schedule.

Project planning, monitoring and control

The purpose of project planning is to identify the scope of the project, estimate the work involved, and create a project schedule. Project planning begins with requirements that define the software to be developed. The project plan is then developed to describe the tasks that will lead to completion.

The purpose of project monitoring and control is to keep the team and management up to date on the project's progress. If the project deviates from the plan, then the project manager can take action to correct the problem. Project monitoring and control involves status meetings to gather status from the team. When changes need to be made, change control is used to keep the products up to date.

Issue

In computing, the term **issue** is a unit of work to accomplish an improvement in a system. An issue could be a bug, a requested feature, task, missing documentation, and so forth. The word "issue" is popularly misused in lieu of "problem." This usage is probably related.

For example, OpenOffice.org used to call their modified version of BugZilla IssueZilla. As of September 2010, they call their system Issue Tracker.

Problems occur from time to time and fixing them in a timely fashion is essential to achieve correctness of a system and avoid delayed deliveries of products.

Severity levels

Issues are often categorized in terms of **severity levels**. Different companies have different definitions of severities, but some of the most common ones are:

- Critical
- High - The bug or issue affects a crucial part of a system, and must be fixed in order for it to resume normal operation.
- Medium - The bug or issue affects a minor part of a system, but has some impact on its operation. This severity level is assigned when a non-central requirement of a system is affected.
- Low - The bug or issue affects a minor part of a system, and has very little impact on its operation. This severity level is assigned when a non-central requirement of a system (and with lower importance) is affected.
- Cosmetic - The system works correctly, but the appearance does not match the expected one. For example: wrong colors, too much or too little spacing between contents, incorrect font sizes, typos, etc. This is the lowest priority issue.

In many software companies, issues are often investigated by Quality Assurance Analysts when they verify a system for correctness, and then assigned to the developer(s) that are responsible for resolving them. They can also be assigned by system users during the User Acceptance Testing (UAT) phase.

Issues are commonly communicated using Issue or Defect Tracking Systems. In some other cases, emails or instant messengers are used.

Philosophy

As a subdiscipline of project management, some regard the management of software development akin to the management of manufacturing, which can be performed by someone with management skills, but no programming skills. John C. Reynolds rebuts this view, and argues that software development is entirely design work, and compares a manager who cannot program to the managing editor of a newspaper who cannot write.

Chapter-2

Application Lifecycle Management and Baseline (Configuration Management)

Application lifecycle management

Application Lifecycle Management (ALM) is a continuous process of managing the life of an application through governance, development and maintenance. ALM is the marriage of business management to software engineering made possible by tools that facilitate and integrate requirements management, architecture, coding, testing, tracking, and release management.

Benefits

Proponents of application lifecycle management claim that it

- Increases productivity, as the team shares best practices for development and deployment, and developers need focus only on current business requirements
- Improves quality, so the final application meets the needs and expectations of users
- Breaks boundaries through collaboration and smooth information flow
- Accelerates development through simplified integration
- Cuts maintenance time by synchronizing application and design
- Maximizes investments in skills, processes, and technologies
- Increases flexibility by reducing the time it takes to build and adapt applications that support new business initiatives

Disadvantages

Opponents of application lifecycle management claim that it

- Increases an application's whole-life cost
- Increases vendor lock-in

Categories of ALM Tools

- Requirements Analysis
- Requirements Management
- Feature management
- Modeling
- Design
- Project Management
- Change management
- Configuration Management
- Software Information Management (for ALM Tool Integration)
- Build management
- Software Testing
- Release Management
- Software Deployment
- Issue management
- Monitoring and reporting
- Workflow

As the Integrated Development Environment (IDE) continues to evolve, tool vendors are increasingly integrating their products to deliver suites. IDEs are giving way to tools that reach outside of pure coding and into the architectural, deployment, and management phases of the application lifecycle, providing full Application Lifecycle Management. The hallmark of these suites is a common user interface, meta model, and process engine that also enable ALM team members to communicate using standards-based architectures and technologies such as Unified Modeling Language (UML).

Notable ALM products

Notable ALM products include:

Name	Vendor
HP Application Lifecycle Management Software	HP Software Division
Visual Studio Application Lifecycle Management	Microsoft
IBM Rational Team Concert	IBM
CollabNet TeamForge	CollabNet

There are other products that can support an ALM model but that only currently deliver a portion of the capabilities required of an ALM product:

Name	Vendor
Atego Workbench	Atego

CodeBeamer	Intland Software
CollabNet TeamForge	CollabNet
Serena Dimensions CM	Serena Software
MKS Integrity	MKS Inc.
Parasoft Concerto	Parasoft
Pulse	Genuitec
SAP Solution Manager	SAP
StarTeam - Change and Configuration Management	Borland
JIRA	Atlassian
FogBugz	Fog Creek Software
IKAN ALM	IKAN
Polarion ALM	Polarion Software
BuildMaster	Inedo

Open Source Alternatives

Name	Sponsor
Endeavour Agile ALM	Community Driven

Baseline (configuration management)

Configuration management is the process of managing change in hardware, software, firmware, documentation, measurements, etc. As change requires an initial state and *next* state, the marking of significant states within a series of several changes becomes important. The identification of significant states within the revision history of a configuration item is the central purpose of **baseline** identification.

Typically, significant states are those that receive a formal approval status, either explicitly or implicitly (approval statuses may be marked individually, when such a marking has been defined, or signified merely by association to a certain baseline). Nevertheless, this approval status is usually recognized publicly. Thus, a baseline may also mark an approved configuration item, e.g. a project plan that has been signed off for execution. In a similar manner, associating multiple configuration items with such a baseline indicates those items as being approved.

Conversely, the configuration of a project often includes one or more baselines, the status of the configuration, and any metrics collected. The current configuration refers to the current status, current audit, current metrics, and latest revision of all configuration items. Similarly, but less frequently, a baseline may refer to all items associated with a specific

project. This may include all revisions of all items, or only the latest revision of all items in the project, depending upon context, e.g. "the baseline of the project is proceeding as planned."

A baseline may be specialized as a specific type of baseline. Some examples include:

- Functional Baseline: initial specifications established; contract, etc.
- Allocated Baseline: state of work products once requirements are approved
- Developmental Baseline: state of work products amid development
- Product Baseline: contains the releasable contents of the project
- others, based upon proprietary business practices

Capabilities of Baselines

While marking approval status covers the majority of uses for a baseline, baselines may also be established merely to signify the progress of work through the passage of time. In this case, a baseline is a visible stake through an endured collective effort, e.g. a developmental baseline. Baselines may also mark milestones; albeit some milestones also signify approval.

Baselines themselves are valued not only for their ability to identify the notable state of work product(s) but also provide particular importance in their ability to be retrieved. Once retrieved, the state of the work product(s) in that subset share the same significance in their history of changes that this significance was observed. The baseline is then regarded with poignant qualities (either favorably or disfavorably). For this reason, baseline identification, monitoring, and retrieval are critical to the success of configuration management. However, the ease of retrieving any given baseline varies according to the system employed for performing configuration management which may use a manual, automated, or hybrid approach. Once retrieved, the baseline may be compared to a particular configuration or another baseline.

Most baselines are established at a fixed point in time and serve to continue to reference that point (identification of state). However, some baselines are established to carry forward as a reference to the item itself regardless of any changes to the item. These latter baselines evolve with the progression of the work effort but continue to identify notable work products in the project.

Baselining Configuration Items

In the process of performing configuration management, configuration items (or work products) may be baselined so as to establish them with a certain status to interested parties. In this sense, to baseline a work product may require certain change(s) to the work product to ensure its conformance to the characteristics associated with the baseline referenced. This varies upon context, but in many cases this has the implication that the work product is "reset" to an initial (possibly inherently approved) state from which work may proceed.

Baseline Control

In many environments, baselines are controlled such that certain subsequent activities against work products in that baseline are either prohibited or permitted. These activities are carefully selected and controlled, and again, depending upon the configuration management system, are also monitored. Consequently, baselines are ordinarily subjected to configuration management audits. Audits may include an examination of specific actions performed against the baseline, identification of individuals involved in any action, an evaluation of change within the baseline, (re-)certification for approval, accounting, metric collection, comparison to another baseline, or all of these.

Application

Though common in software revision control systems as **labels** or **tags**, the existence of baselines is found in several other technology-related domains. Baselines can be found in UML modeling systems and business rule management systems, among others.

In addition to the field of Hardware and Software Engineering, baselines can be found in Medicine (e.g. monitoring health progress), Politics (e.g. statistics), Physics & Chemistry (e.g. observations and changes), Finance (e.g. budgeting), and others.

Chapter-3

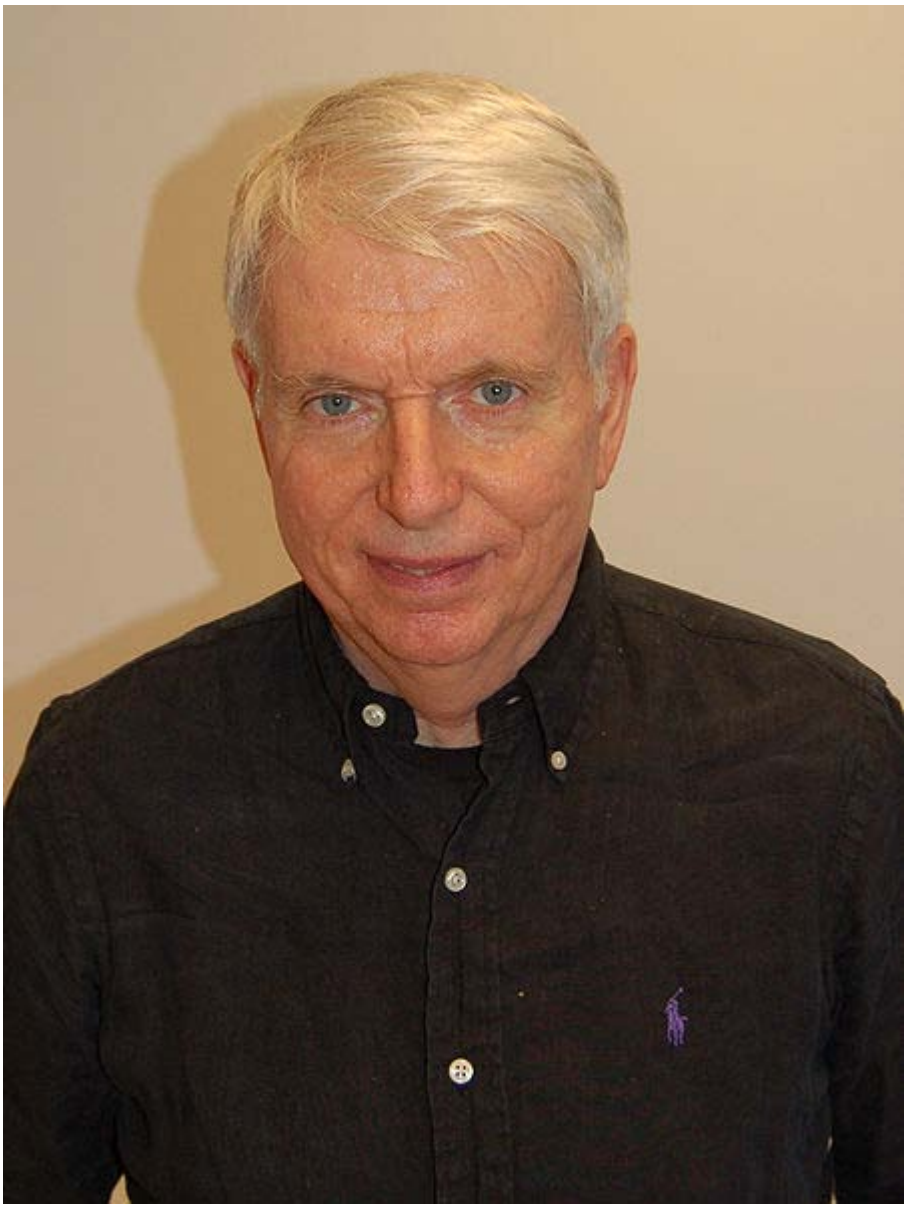
Agile Software Development

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 (often supported through special languages see SPIN model checker) 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 see Wirth's law
- 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-4

Software Development and Brownfield (Software Development)

Software development

Software development (also known as **application development**, **software design**, **designing software**, **software application development**, **enterprise application development**, or **platform development**) is the development of a software product. The term "software development" may be used to refer to the activity of computer programming, which is the process of writing and maintaining the source code, but in a broader sense of the term it includes all that is involved between the conception of the desired software through to the final manifestation of the software, ideally in a planned and structured process. Therefore, software development may include research, new development, modification, reuse, re-engineering, maintenance, or any other activities that result in software products.

Software can be developed for a variety of purposes, the three most common being to meet specific needs of a specific client/business (the case with custom software), to meet a perceived need of some set of potential users (the case with commercial and open source software), or for personal use (e.g. a scientist may write software to automate a mundane task). **Embedded software development**, that is, the development of embedded software such as used for controlling consumer products, requires the development process to be integrated with the development of the controlled physical product.

The need for better quality control of the software development process has given rise to the discipline of software engineering, which aims to apply the systematic approach exemplified in the engineering paradigm to the process of software development.

Overview

There are several different approaches to software development, much like the various views of political parties toward governing a country. Some take a more structured, engineering-based approach to developing business solutions, whereas others may take a more incremental approach, where software evolves as it is developed piece-by-piece.

Most methodologies share some combination of the following stages of software development:

- Market research
- Gathering requirements for the proposed business solution
- Analyzing the problem
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

These stages are often referred to collectively as the software development lifecycle, or SDLC. Different approaches to software development may carry out these stages in different orders, or devote more or less time to different stages. The level of detail of the documentation produced at each stage of software development may also vary. These stages may also be carried out in turn (a “waterfall” based approach), or they may be repeated over various cycles or iterations (a more "extreme" approach). The more extreme approach usually involves less time spent on planning and documentation, and more time spent on coding and development of automated tests. More “extreme” approaches also promote continuous testing throughout the development lifecycle, as well as having a working (or bug-free) product at all times. More structured or “waterfall” based approaches attempt to assess the majority of risks and develop a detailed plan for the software before implementation (coding) begins, and avoid significant design changes and re-coding in later stages of the software development lifecycle.

There are significant advantages and disadvantages to the various methodologies, and the best approach to solving a problem using software will often depend on the type of problem. If the problem is well understood and a solution can be effectively planned out ahead of time, the more "waterfall" based approach may work the best. If, on the other hand, the problem is unique (at least to the development team) and the structure of the software solution cannot be easily envisioned, then a more "extreme" incremental approach may work best. A software development process is a structure 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.

Consistency in Software

In order to ensure that software can evolve in a way that maintains its inherent multidimensionality, one must ensure that the different dimensions evolve together in a consistent manner.Software has too many dimensions to combine within a single framework, Not only do the different notations but they also do not interact hierarchically.A good mechanism should not be geared to a specific problem such as ensuring the consistency of a UML class diagram with the source code. Instead it should

be flexible enough to handle the broad range of dimensions that are actually involved in software development.

Software development topic

Marketing

The sources of ideas for software products are legion. These ideas can come from market research including the demographics of potential new customers, existing customers, sales prospects who rejected the product, other internal software development staff, or a creative third party. Ideas for software products are usually first evaluated by marketing personnel for economic feasibility, for fit with existing channels distribution, for possible effects on existing product lines, required features, and for fit with the company's marketing objectives. In a marketing evaluation phase, the cost and time assumptions become evaluated. A decision is reached early in the first phase as to whether, based on the more detailed information generated by the marketing and development staff, the project should be pursued further. Pranay.

In the book *"Great Software Debates"*, Alan M. Davis states in the chapter *"Requirements"*, subchapter *"The Missing Piece of Software Development"*

> *“ Students of engineering learn engineering and are rarely exposed to finance or marketing. Students of marketing learn marketing and are rarely exposed to finance or engineering. Most of us become specialists in just one area. To complicate matters, few of us meet interdisciplinary people in the workforce, so there are few roles to mimic. Yet, software product planning is critical to the development success and absolutely requires knowledge of multiple disciplines. ”*

Because software development may involve compromising or going beyond what is required by the client, a software development project may stray into less technical concerns such as human resources, risk management, intellectual property, budgeting, crisis management, etc. These processes may also cause the role of business development to overlap with software development.

Software development methodology

A software development methodology is a framework that is used to structure, plan, and control the process of developing information systems. A wide variety of such frameworks have evolved over the years, each with its own recognized strengths and weaknesses. One system development methodology is not necessarily suitable for use by all projects. Each of the available methodologies is best suited to specific kinds of projects, based on various technical, organizational, project and team considerations.

Brownfield (software development)

Brownfield development is a term commonly used in the IT industry to describe problem spaces needing the development and deployment of new software systems in the immediate presence of existing (legacy) software applications/systems. This implies that any new software architecture must take into account and coexist with live software already in situ. In contemporary civil engineering, Brownfield land means places where new buildings may need to be designed and erected considering the other structures and services already in place.

Brownfield development adds a number of improvements to conventional software engineering practices. These traditionally assume a "clean sheet of paper" or "green field" target environment throughout the design and implementation phases of software development. Brownfield extends such traditions by insisting that the context (local landscape) of the system being created be factored into any development exercise. This requires a detailed knowledge of the systems, services and data in the immediate vicinity of the solution under construction.

Addressing environmental complexity

Reliably reengineering existing business and IT environments into modern competitive, integrated architectures is non-trivial. The complexity of business and IT environments has been accumulating almost unchecked for forty years making changes ever more expensive. This is because:

- Environmental complexity is often expressed in legacy code. Legacy skills shortages are driving up maintenance and integration costs.
- Existing complex environments must be re-engineered in phases that make operational sense to their associated business function. These phases often default to wholesale, risky replacements of systems as ignorance of existing complexity means that potential incremental changes are too difficult to understand and engineer.
- Accelerated development methods have left enterprises with modern legacy systems. Complex Java and .NET applications have many of the same problems as older COBOL applications.

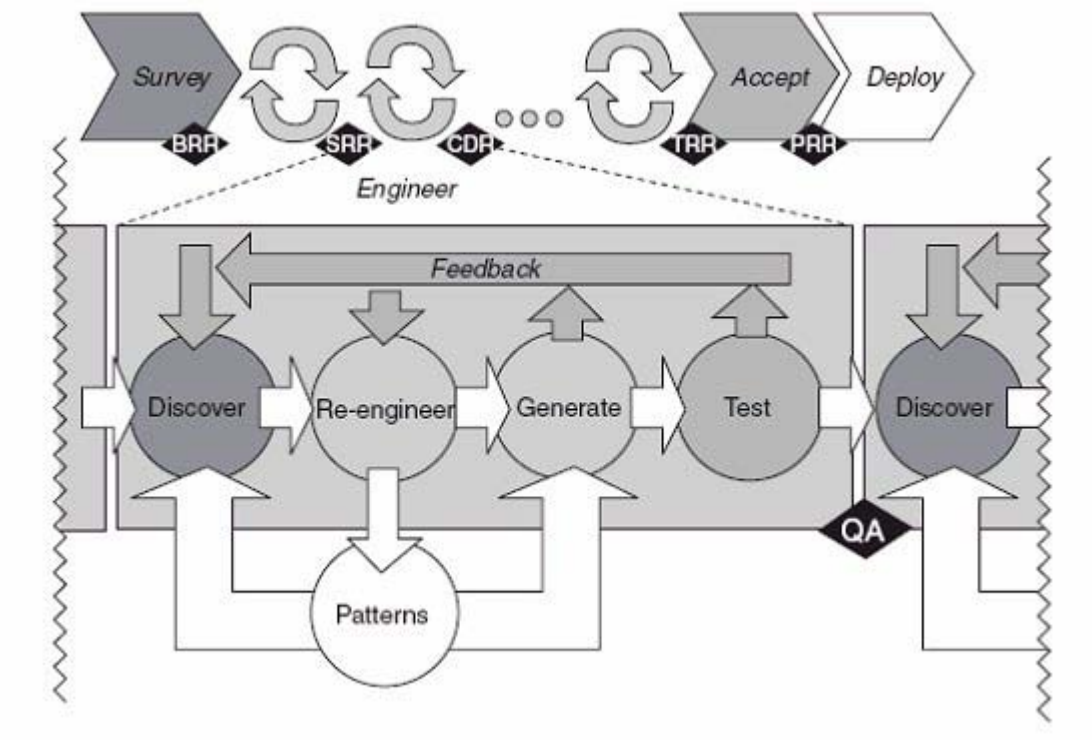
As a result, an increasing proportion of the effort of developing new business capabilities is spent on understanding and integrating with the existing complex system and business landscape rather than delivering value. It has been observed that up to 75% of overall project effort is now spent on software integration and migration rather than new function.

The IT industry as a whole has a poor success rate at delivering such large scale change for its clients. The CHAOS survey from the Standish Group has tracked an overall improvement in IT project delivery success over the last twenty years, but even in 2006 large IT projects still failed more often than succeeded. Engineering changes and in such

environments has many parallels with the concerns of the construction industry in redeveloping industrial or contaminated sites. They are full of hazards, unexpected complexities and tend to be risky and expensive to redevelop. The accumulated complexity of IT environments has made them “Brownfield” sites.

Ironically it is not the complexity of the new function or any new system characteristics that are the root of large project failures – it is our understanding and communication of the overall requirement (as identified in The Mythical Man Month). To succeed the requirements need to include a precise and thorough understanding of the constraints of the existing business and IT. Current “Greenfield” tooling and methods use early, informal and often imprecise abstractions that essentially ignore such complexity. However the devil is always in the detail. Early, poorly informed abstractions are usually wrong and are often detected late in construction, resulting in delays, expensive rework and even failed developments. A Brownfield oriented approach embraces existing complexity, and is used to reliably accelerate the overall solution engineering process, including enabling phased, incremental change wherever possible.

Brownfield takes the standard OMG model/pattern driven approach and turns it on its head. Rather than taking the conventional approach of starting with a Conceptual Model and driving down to Platform Specific Models and code generation, Brownfield starts by harvesting code and other existing artifacts and uses patterns to formally abstract upwards towards the Architecture and Business tier.



Outline of the Brownfield development process

Standard Greenfield techniques are then used in combination to define the preferred business target. This “meet in the middle” technique is familiar from other development methods, but the extensive use of formal abstraction and the use of patterns for both discovery and generation is novel.

The underlying conceptual architecture of all Brownfield tools is known as VITA. VITA stands for Views, Inventory, Transformation and Artifacts. In a VITA architecture, the problem definition of the target space can be maintained as separate (though related) native "headfulls" of knowledge known as Views. The core advantage of a View is that it can be based on pretty much any formal tool. Brownfield does not impose a single tool or language on a problem space – a core tenet is that the headfulls continue to be maintained in their native forms and tools.

Native Views are then brought together and linked into a single Inventory. The Inventory is then used with a series of Transformation capabilities to produce the Artifacts that the solution needs.

Views can currently be imported from a wide variety of sources including UML, XML sources, DDL, spreadsheets etc. The Analysis and Renovation Catalyst tool from IBM has taken this capability even further via the use of formal grammars and Abstract Syntax Trees to enable almost any program to be parsed and tokenized into a View for inclusion into the Inventory.

The rapid cyclic nature of the discovery, re-engineer, generate and test cycle used in this approach means that solutions can be refined iteratively in terms of their logical and physical definitions as more of the constraints become known and the solution architecture is refined.

Iterative Brownfield development can allow the gradual refinement of logical and physical architectures and incremental testing for the whole approach, resulting in development acceleration, improved solution quality and cheaper defect removal. Brownfield can also be used to generate solution documentation, ensuring it is always up to date and consistent across different viewpoints.

The Inventory that is created through Brownfield processed may be highly complex, being an interconnected multi-dimensional semantic network. The level of knowledge in the Inventory can be very fine grained, highly detailed and interrelated. Such things are hard to understand and can provide barriers to communication however. Brownfield solves this problem by abstracting concepts via an artisan’s best guess, using known patterns in its Inventories to extract and infer higher level relationships.

Formal abstractions enable the complexity of the Inventory to be translated into simpler, but inherently accurate, representations for easier consumption by those that need to understand the problem space. These abstracted Inventory models can be used to automatically render multi-layered architecture representations in tools such as Second Life.

Such visualizations enable complex information to be shared and experienced by multiple individuals from around the globe in real time. This enhances both understanding and a sense of a single team.

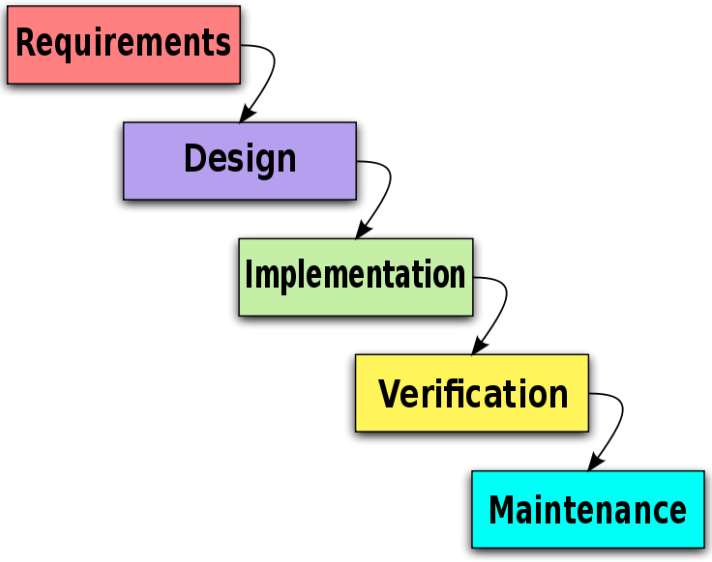
Chapter-5

Dual Vee Model

The **Dual Vee Model** builds on the V-Model to cleanly depict the complexity associated with designing and developing systems. In systems engineering it defines a uniform procedure for product or project development. The model depicts concurrent development of a system's architecture as one Vee with each entity of that architecture as another Vee that intersects the architecture Vee. This clearly shows interactions and sequences in developing a complex system and a system of systems.

Background

Waterfall Model

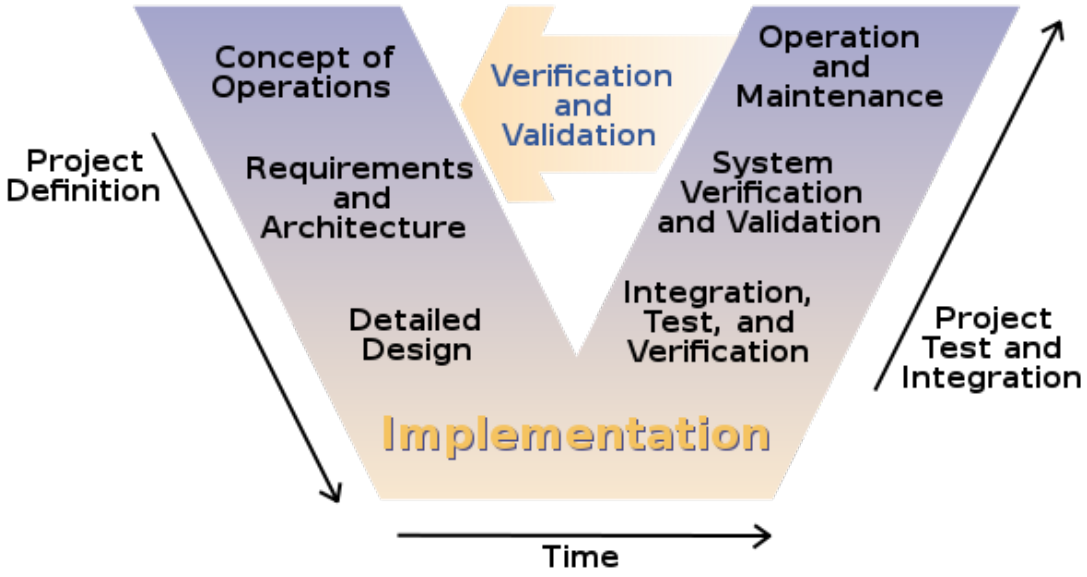


The unmodified "waterfall model". Progress flows from the top to the bottom, like a waterfall.

The Waterfall Model breaks up the development process into development phases. The name implies that design decisions flow from the requirements, implementation decisions

flow from the design, etc. On a large project, many different people only work on each part. So a designer may ask, "What am I trying to design?" and the response would be, "You're trying to design something that will satisfy the product requirements." The builder may ask, "What am I trying to build?" and the response would be, "You're trying to build what was designed." etc.

Vee Model

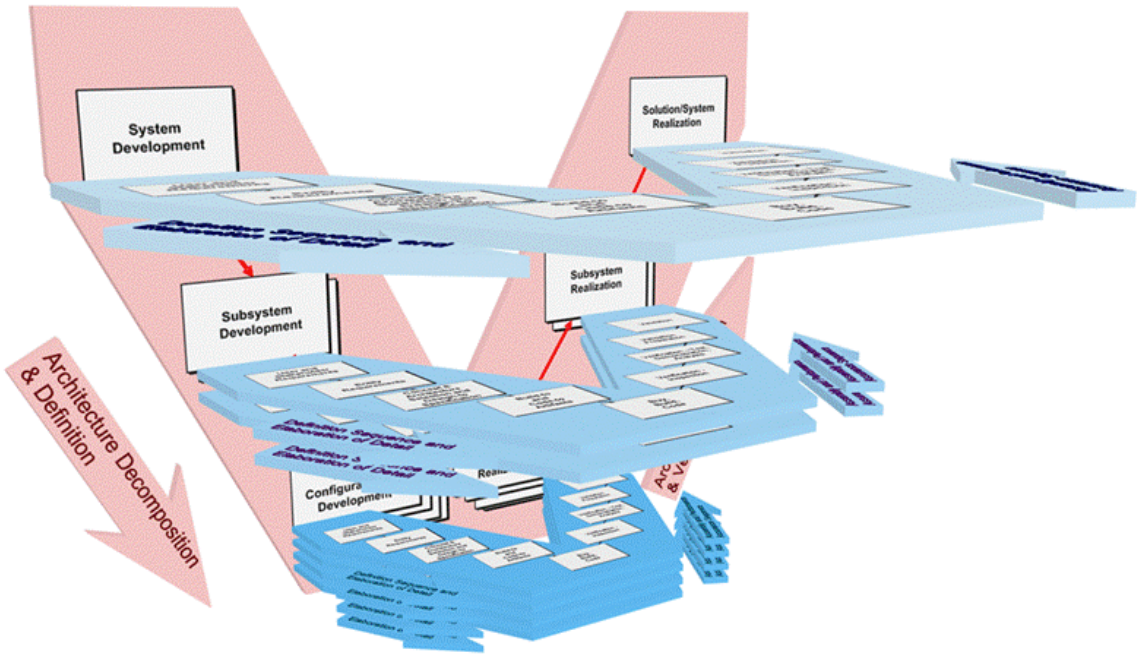


The V-model of the Systems Engineering Process.

The Vee Model organizes development phases into levels of complexity with the most complex item on top and least complex item on bottom (a.k.a. Lowest Configuration Item). This places the requirements at the beginning next to the product's operation at the end, and the design next to verification. This makes sense because when developer delivers a product to the customer, the customer may ask, "Why should I accept this product?" and the developer should answer, "Because it meets your (the customer's) requirements." The requirements are connect to the operation. When performing product testing, the test engineer may ask, "What tests should I conduct?" and the designer should answer, "You should conduct tests to show the product that was built matches the design." Verification is connected to the design.

The Vee Model can be expanded in several ways to meet system requirements. It can include the seven INCOSE layers of system complexity (i.e. system, element, subsystem, assembly, subassembly, component, and part). It can include decomposition, definition, integration, and verification. It may also include user/stakeholder participation, opportunity and risk management, and problem resolution.

Dual Vee Model



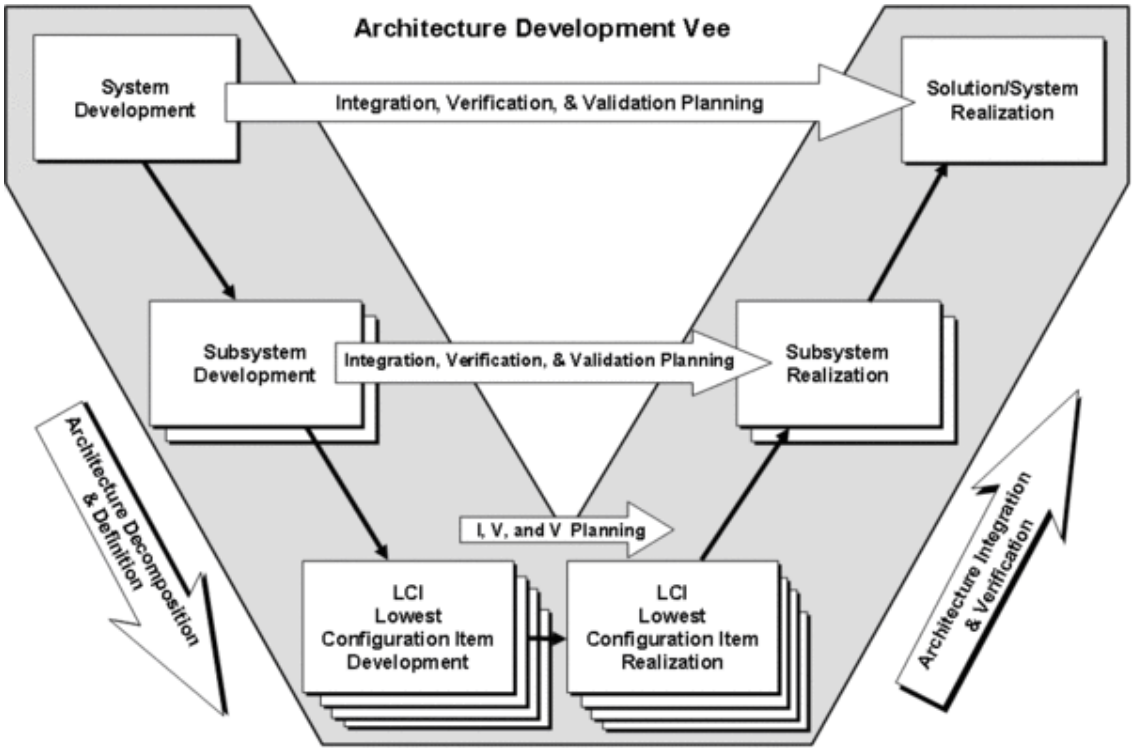
Architecture and Entity Vees Intersecting. Source - Kevin Forsberg and Hal Mooz 2006.

The Dual Vee Model builds on the Vee Model to manage a system of systems. An architecture Vee manages the system and entity Vees branch off the architecture Vee to manage sub-systems.

For example, GPS includes a constellation of satellites, a ground control network, and millions of users worldwide. Each satellite, ground control center, and GPS receiver is a complex system that could be managed by a separate Entity Vee. Development of a satellite can affect the design, manufacturing, or cost of receivers. Similarly, development of a receiver can affect design, manufacturing, or cost of satellites. So everything must be integrated into a system of systems that are developed within a larger Architecture Vee.

The Architecture Vee

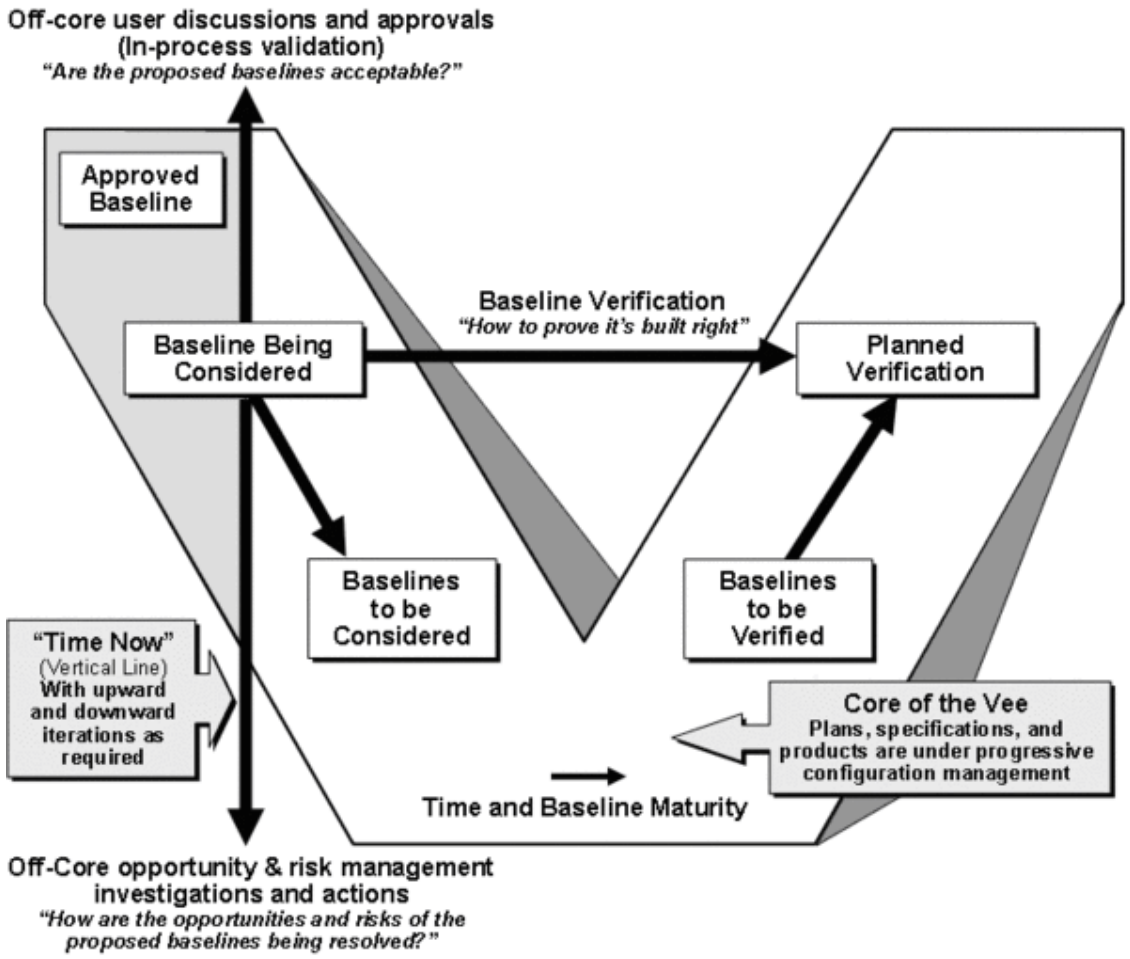
When developing complicated systems, a system engineer must manage a system baseline configuration from start to finish. The baseline can include design documents, user manuals, the product itself, and should answer every What?, Why?, and Who? for a system's architecture. At each development phase, there will be changes to the system, which will change the baseline.



Architecture Development Vee Model (Provides What, Why, and Who). Source - Kevin Forsberg and Hal Mooz 2006.

The core of the Vee is the evolving architecture baseline from initial requirements to a delivered system. The Architecture Vee produces the what, why, and who (which entity level) is responsible for a system's architecture.

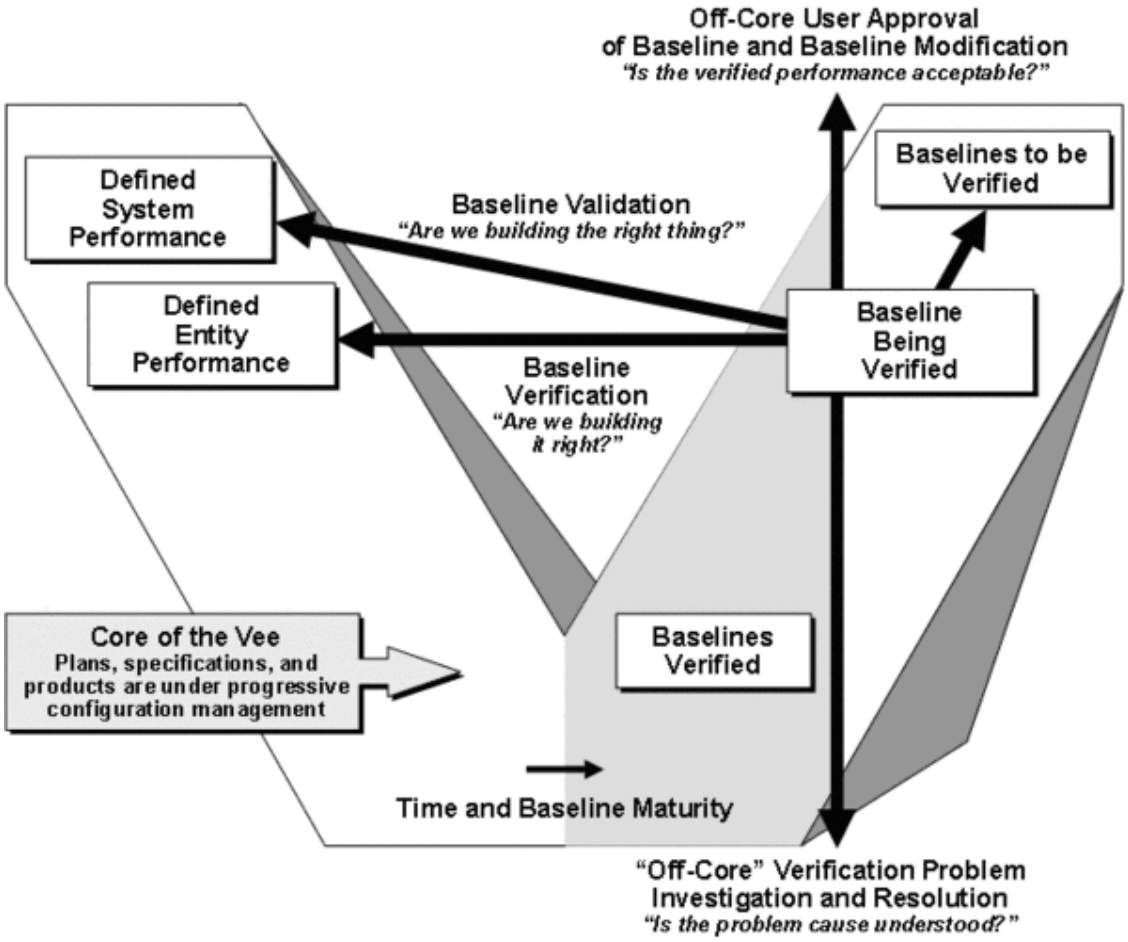
Downward off-Vee core investigations (figure - below) facilitate gaining knowledge to justify architecture baseline decisions made on the Vee core. Upward off core communication with customers and users facilitates in-process validation keeping the stakeholders abreast of and committed to the evolving baseline. Note that in all Vee representations time and maturity move from left to right. Just as we cannot move backward in time, so too one cannot move from right to left in the Vee model representation. Iteration is essential in system development, and all iteration is done vertically off-core, upward to users and customers (which is in-process validation), and downward for opportunity and risk management, as shown in the following figure.



Vee Model - Opportunity and Risk Investigations. Source - Kevin Forsberg and Hal Mooz 2006.

The left leg off-Vee core investigations center around what concept is best and what architecture is best for that concept. For example, commercial products usually face the dilemma as to whether batteries should be standard, unique, replaceable, or not. Downward off-Vee core investigations and analysis can facilitate determining the most desirable approach that would then be baselined on the Vee core if the stakeholders agree. Similar investigations can prove the viability and technical feasibility of candidate concepts.

Right leg off-Vee core downward investigations (figure - below) are directed at investigating integration anomalies to determine their root cause and to correct them. Upward communication with stakeholders determines if they can live with the as-integrated and as-verified performance.



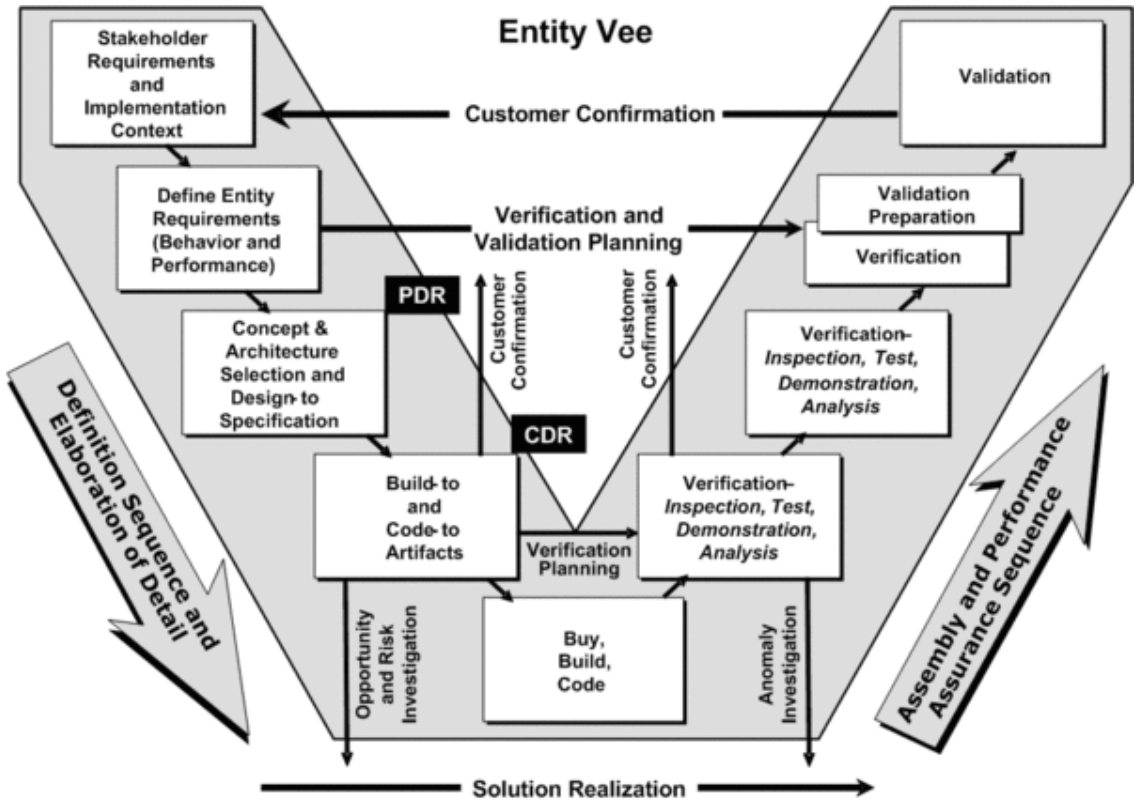
Vee Model - Integration and Verification. Source - Kevin Forsberg and Hal Mooz 2006.

At each decomposition level there is a direct correlation between activities on the left and right sides of the Vee. This is deliberate. For example, the method of integration, verification, and validation to be used on the right must be determined on the left as concepts are defined at each decomposition level. This minimizes the chances that concepts are conceived in a way that cannot be carried out.

The Entity Vee Model

The Entity Vee illustrates the entity development and realization process which describes how each entity will be obtained (development, purchase, reuse, etc.). An Entity Vee (figure - below) exists for every entity of the architecture from the system, down to the lowest configuration items (LCIs), such as computer software units or hardware components. All activities within an Entity Vee reside at the same architecture level (System, Subsystem, LCI). The left Vee leg represents entity definition elaboration from very sketchy user requirements, through concept determination and on to design-to specifications and fully detailed build-to artifacts. The right Vee leg represents the

sequence of entity assembly and performance assurance on through verification and validation of the entity.



Entity Vee Model (Provides How). Source - Kevin Forsberg and Hal Mooz 2006.

At each elaboration, there is a direct correlation between activities on the left and right legs of the Entity Vee. Again this is deliberate. The method of verification to be used on the right Vee leg must be defined as requirements are developed on the left, otherwise requirements might be created that could not be verified. For example “user friendly” is a valid requirement, but it is unverifiable. Instead, a requirement that a computer screen display have “no more than five lines of 14-point text” defines one user's view of “user friendly” in measurable terms. Verification plans should be baselined to ensure verification requirements and methods are known and planned for at the design-to decision gate, commonly called Preliminary Design Review (PDR). Draft verification procedures based on the verification requirements, verification plan, and proposed entity design should be available at the build-to and code-to decision gate, commonly called Critical Design Review (CDR). This reduces the chances that verification as specified cannot in fact be performed.

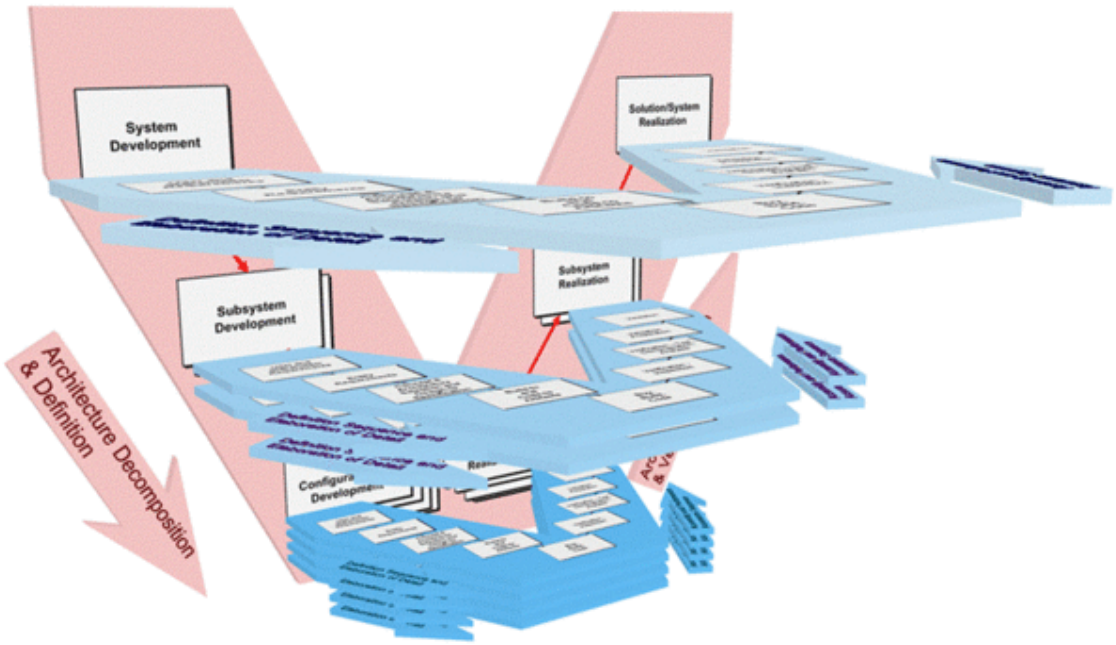
The vertical dimension of the Entity Vee is baseline elaboration at the selected architecture level and the core of the Entity Vee represents entity baseline elaboration progression. Also included (similar to the Architecture Vee) are the activities associated with opportunity and risk management, pursued downward and off-core to the level of detail necessary for issue evaluation and resolution. For example, laboratory test of a

computer chip or of software code may be necessary to confirm technical feasibility. Unlike the commonly held view of the Waterfall Model, there is no prohibition against doing exploratory design and analysis at any point in the project cycle to investigate or prove performance or feasibility. Unlike the Spiral Model, the Vee opportunity and risk investigations may be performed either in series or in parallel with the on-core development work, rather than being conducted sequentially and prior to the design development process. Hardware and software requirements-understanding models or technical feasibility models are encouraged early in the project cycle to pursue opportunities, such as new technologies, and to reduce risk. For instance, to evaluate a concept of a manual override versus full automation, technical feasibility of the two concepts could be modeled with selection based on response time versus cost. Customer confirmation could then provide valuable in-process validation of the preferred approach.

In the right leg, downward off-core investigations are applied to resolve assembly and verification anomalies. This may require descending to design errors, a cold solder joint, or operator error and the like. Upward off-core user interactions obtain user and customer confirmation or rejection of the realized performance. Note that in the entity Vee these interactions address individual entity solutions and not the integration of the architecture which is conducted on the Architecture Vee. At any level of decomposition, the customer of an entity is the manager of the next higher level of decomposition. For example, the power subsystem manager is the customer of the battery and is responsible for battery validation.

Dual Vees: Intersecting Architecture and Entity Vees

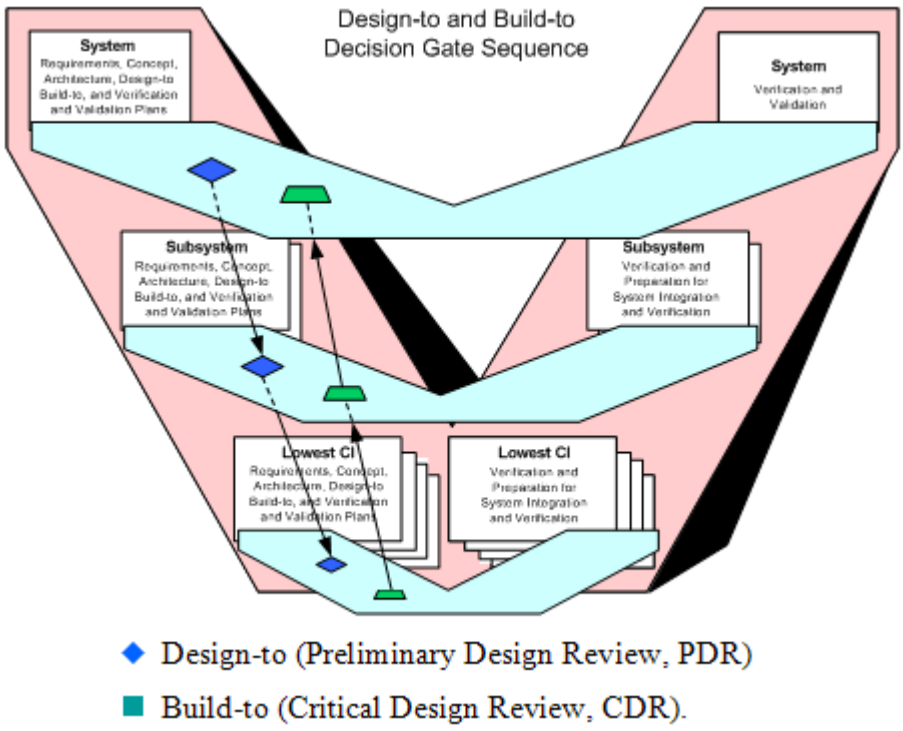
To evolve user needs into a system that satisfies those needs requires a best value solution for every entity of the architecture. This can be visualized by positioning Entity Vees orthogonal to the Architecture Vee as shown in the figure below. For each entity of the Architecture Vee there is a corresponding Entity Vee that addresses the entity development and realization. For example, the Architecture Vee of the figure below contains two subsystems hence there are two Entity Vees to represent the concurrent development of those two subsystems.



Architecture and Entity Vees Intersecting. Source - Kevin Forsberg and Hal Mooz 2006.

Phasing of decision gates

Architecture entities are developed and integrated into the system architecture in a phased sequence consistent with systems engineering best practices. The figure below provides a three dimensional view of Design-to and Build-to Decision Gate phasing



Design-to and Build-to Decision Gate Sequence. Source - Kevin Forsberg and Hal Mooz 2006.

For simplicity of illustration, only one Entity Vee is shown intersecting the Architecture Vee at each decomposition level. Note that the design-to sequence is top down, starting at the system level and proceeding consistent with decomposition to the lowest configuration item level (LCI). This sequence ensures that there is proper top down requirements flowdown and traceability.

When build-to and code-to artifacts, including draft verification procedures, are ready for baselining, the build-to decision gate sequence is conducted bottom up to prove the feasibility of building or coding the designs. The decision gate also confirms that, if the solution is built according to the build-to artifacts, the required performance will be achieved. This sequence ensures that, if the entity designs satisfy their design-to requirements, the entities will integrate into the next higher level entity, will perform as expected, and will meet user requirements.

Chapter-6

IBM Rational Unified Process

The **Rational Unified Process** (**RUP**) is an iterative software development process framework created by the Rational Software Corporation, a division of IBM since 2003. RUP is not a single concrete prescriptive process, but rather an adaptable process framework, intended to be tailored by the development organizations and software project teams that will select the elements of the process that are appropriate for their needs.

History

The Rational Unified Process (RUP) is a software process product, originally developed by Rational Software, which was acquired by IBM in February 2003. The product includes a hyperlinked knowledge base with sample artifacts and detailed descriptions for many different types of activities. RUP is included in the IBM Rational Method Composer (RMC) product which allows customization of the process.

By 1997, Rational had acquired Verdix, Objectory, Requisite, SQA, Performance Awareness, and Pure-Atria. Combining the experience base of companies led to the articulation of six *best practices* for modern software engineering:

1. Develop iteratively, with risk as the primary iteration driver
2. Manage requirements
3. Employ a component-based architecture
4. Model software visually
5. Continuously verify quality
6. Control changes

These best practices both drove the development of Rational's products, and were used by Rational's field teams to help customers improve the quality and predictability of their software development efforts. To make this knowledge more accessible, Philippe Kruchten, a Rational techrep, was tasked with the assembly of an explicit process

framework for modern software engineering. This effort employed the HTML-based process delivery mechanism developed by Objectory. The resulting "Rational Unified Process" (RUP) completed a strategic tripod for Rational:

- a *tailorable process* that guided development
- *tools* that automated the application of that process
- *services* that accelerated adoption of both the process and the tools.

Rational Unified Process topics

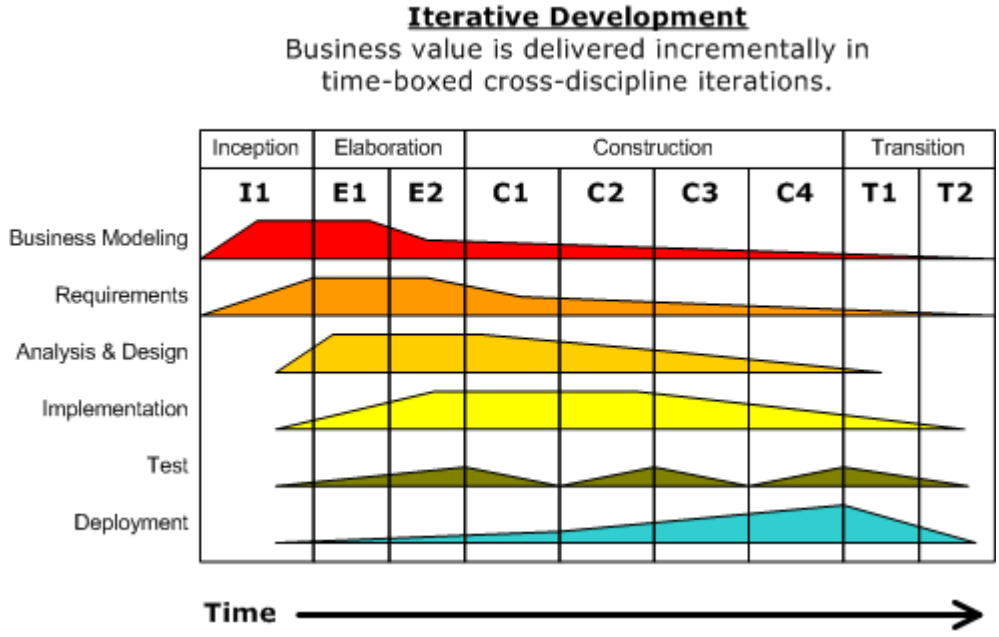
RUP building blocks

RUP is based on a set of building blocks, or content elements, describing what is to be produced, the necessary skills required and the step-by-step explanation describing how specific development goals are to be achieved. The main building blocks, or content elements, are the following:

- Roles (who) – A Role defines a set of related skills, competencies and responsibilities.
- Work Products (what) – A Work Product represents something resulting from a task, including all the documents and models produced while working through the process.
- Tasks (how) – A Task describes a unit of work assigned to a Role that provides a meaningful result.

Within each iteration, the tasks are categorized into nine disciplines: six "engineering disciplines" (Business Modeling, Requirements, Analysis and Design, Implementation, Test, Deployment) and three supporting disciplines (Configuration and Change Management, Project Management, Environment).

Four Project Life cycle Phases



RUP phases and disciplines.

The RUP has determined a project life cycle consisting of four phases. These phases allow the process to be presented at a high level in a similar way to how a 'waterfall'-styled project might be presented, although in essence the key to the process lies in the iterations of development that lie within all of the phases. Also, each phase has one key objective and milestone at the end that denotes the objective being accomplished. The visualization of RUP phases and disciplines over time is referred to as the RUP hump chart.

Inception Phase

The primary objective is to scope the system adequately as a basis for validating initial costing and budgets. In this phase the business case which includes business context, success factors (expected revenue, market recognition, etc.), and financial forecast is established. To complement the business case, a basic use case model, project plan, initial risk assessment and project description (the core project requirements, constraints and key features) are generated. After these are completed, the project is checked against the following criteria:

- Stakeholder concurrence on scope definition and cost/schedule estimates.
- Requirements understanding as evidenced by the fidelity of the primary use cases.
- Credibility of the cost/schedule estimates, priorities, risks, and development process.
- Depth and breadth of any architectural prototype that was developed.
- Establishing a baseline by which to compare actual expenditures versus planned expenditures.

If the project does not pass this milestone, called the Lifecycle Objective Milestone, it either can be canceled or repeated after being redesigned to better meet the criteria.

Elaboration Phase

The primary objective is to mitigate the key risk items identified by analysis up to the end of this phase. The elaboration phase is where the project starts to take shape. In this phase the problem domain analysis is made and the architecture of the project gets its basic form.

This phase must pass the Lifecycle Architecture Milestone by meeting the following deliverables:

- A use-case model in which the use-cases and the actors have been identified and most of the use-case descriptions are developed. The use-case model should be 80% complete.
- A description of the software architecture in a software system development process.
- An executable architecture that realizes architecturally significant use cases.
- Business case and risk list which are revised.
- A development plan for the overall project.
- Prototypes that demonstrably mitigate each identified technical risk.

If the project cannot pass this milestone, there is still time for it to be canceled or redesigned. However, after leaving this phase, the project transitions into a high-risk operation where changes are much more difficult and detrimental when made.

The key domain analysis for the elaboration is the system architecture.

Construction Phase

The primary objective is to build the software system. In this phase, the main focus is on the development of components and other features of the system. This is the phase when the bulk of the coding takes place. In larger projects, several construction iterations may be developed in an effort to divide the use cases into manageable segments that produce demonstrable prototypes.

This phase produces the first external release of the software. Its conclusion is marked by the Initial Operational Capability Milestone.

Transition Phase

The primary objective is to 'transit' the system from development into production, making it available to and understood by the end user. The activities of this phase include training the end users and maintainers and beta testing the system to validate it against the end

users' expectations. The product is also checked against the quality level set in the Inception phase.

If all objectives are met, the Product Release Milestone is reached and the development cycle ends.

The IBM Rational Method Composer product

The IBM Rational Method Composer product is a tool for authoring, configuring, viewing, and publishing processes.

Certification

In January 2007, the new RUP certification examination for *IBM Certified Solution Designer - Rational Unified Process 7.0* was released which replaces the previous version of the course called *IBM Rational Certified Specialist - Rational Unified Process*. The new examination will not only test knowledge related to the RUP content but also to the process structure elements.

To pass the new RUP certification examination, a person must take IBM's *Test 839: Rational Unified Process v7.0*. You are given 75 minutes to take the 52 question exam. The passing score is 62%.

Six Best Practices

Six Best Practices as described in the Rational Unified Process is a paradigm in software engineering, that lists six ideas to follow when designing any software project to minimize faults and increase productivity . These practices are:

Develop iteratively
: It is best to know all requirements in advance; however, often this is not the case. Several software development processes exist that deal with providing solution on how to minimize cost in terms of development phases.

Manage requirements
: Always keep in mind the requirements set by users.

Use components
: Breaking down an advanced project is not only suggested but in fact unavoidable. This promotes ability to test individual components before they are integrated into a larger system. Also, code reuse is a big plus and can be accomplished more easily through the use of object-oriented programming.

Model visually
: Use diagrams to represent all major components, users, and their interaction. "UML", short for Unified Modeling Language, is one tool that can be used to make this task more feasible.

Verify quality

Always make testing a major part of the project at any point of time. Testing becomes heavier as the project progresses but should be a constant factor in any software product creation.

Control changes

Many projects are created by many teams, sometimes in various locations, different platforms may be used, etc. As a result it is essential to make sure that changes made to a system are synchronized and verified constantly.

Chapter-7

Endeavour Software Project Management, Feature Creep and Issue Log

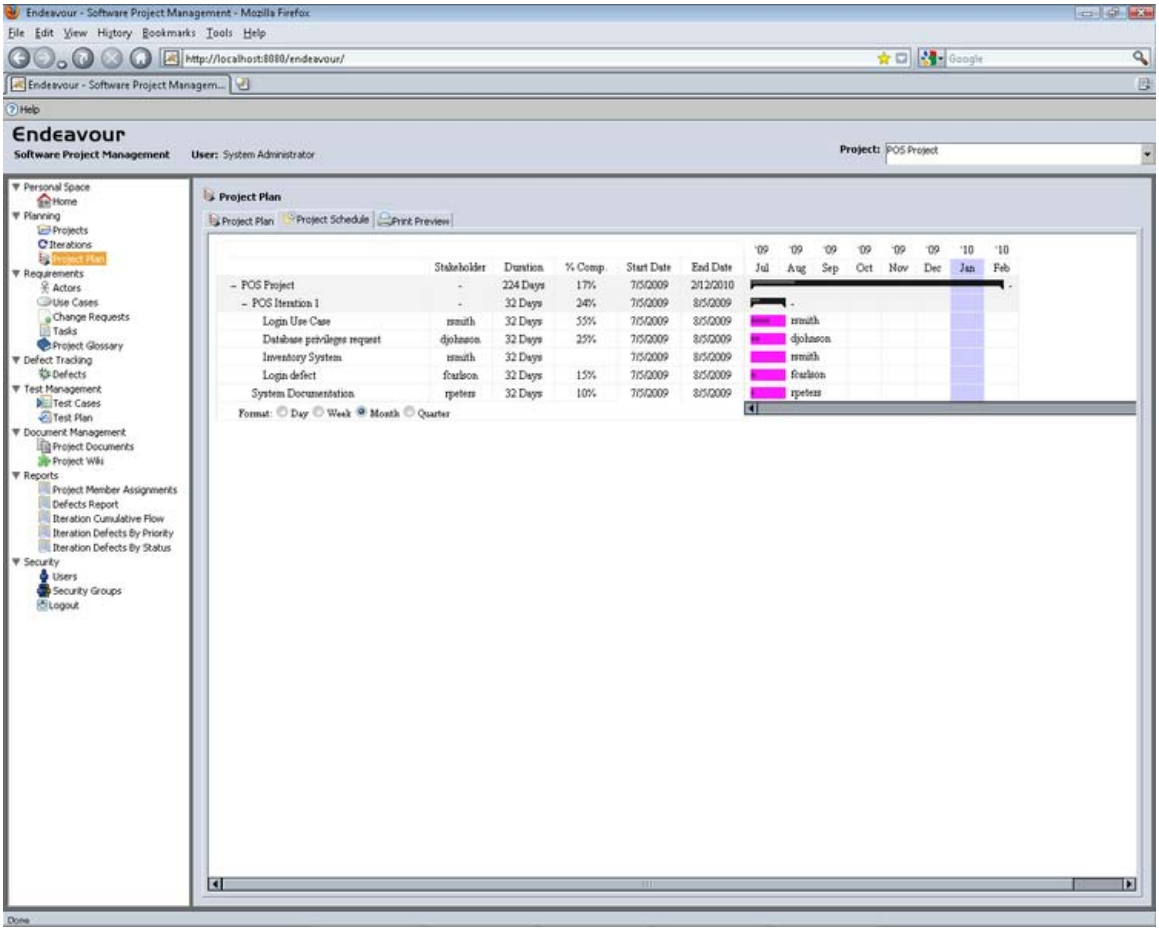
Endeavour Software Project Management

Endeavour Software Project Management is an open source solution to manage large-scale enterprise software projects in an iterative and incremental development process.

The major features include support for the following software artifacts:

- Projects
- Use Cases
- Iterations
- Project Plans
- Change Requests
- Defect Tracking
- Test Cases
- Test Plans
- Task
- Actors
- Document Management
- Project Glossary
- Developer Management
- Reports (Assignments, Defects, Cumulative Flow)
- SVN Browser Integration with svenson
- CI Integration with Hudson
- Email notifications
- Fully Internationalizable

History



Endeavour's Project Plan Gantt Chart

Endeavour Software Project Management was founded in September 2008 with the intention to develop a solution for replacing expensive and complex project management systems that is easy to use, intuitive, and realistic by eliminating features considered unnecessary.

In September 2009 the project was registered in SourceForge, and in April 2010 the project was included in SourceForge's blog with an average of 210 weekly downloads.

License

Endeavour Software Project Management is available via the GNU General Public License.

System requirements

Endeavour Software Project Management can be deployed in any Java EE-compliant application server and any relational database running under a variety of different

operating systems. Its cross-browser capability allows it to run in most popular web browsers.

Usage

- Software Project Management
- Iterative and Incremental development
- Use Case Driven
- Issue tracking
- Test Case Management Software

Feature creep

Feature creep (or "**creeping featurism**") is the ongoing expansion or addition of new features in a product, such as in computer software. Extra features go beyond the basic function of the product and so can result in over-complication, or "featuritis", rather than simple design. Viewed over a longer time period, extra or unnecessary features seem to "creep into" the system, beyond the initial goals.

Causes

The most common cause of feature creep is the desire to provide the consumer with a more useful or desirable product, in order to increase sales or distribution. However, once the product reaches the point at which it does everything that it is designed to do, the manufacturer is left with the choice of adding unneeded functions, sometimes at the cost of efficiency, or sticking with the old version, at the cost of a perceived lack of improvement.

Another major cause of feature creep might be a compromise from a committee which decides to implement multiple, different viewpoints in the same product. Then, as more features are added to support each viewpoint, it might be necessary to have cross-conversion features between the multiple viewpoints, further complicating the total features.

Characteristics

Feature creep is the most common source of cost and schedule overruns. It thus endangers and can even kill products and projects. Apple's abandoned Copland operating system is an example of this.

Control

There are several methods to control feature creep, including: strict limits for allowable features; multiple variations, and pruning excess features.

Temptation of later feature creep may be avoided to some degree by basing initial design on strong software fundamentals, such as logical separation of functionality and data access. It can be actively controlled with rigorous change management and by delaying changes to later delivery phases of a project.

Another method of controlling feature creep is to maintain multiple variations of products, where features are kept limited in some variations. Because the ever-growing, ever-expanding addition of new features might exceed available resources, a minimal core "basic" version of a product can be maintained separately, to ensure operation in smaller operating environments. Using the "80/20 Rule" the more basic product variations might support the needs of about "80%" of the users, so they would not be subjected to the complexity (or extra expense) of features requested by the other 20% of users. The extra features are still available, but they have not crept into all versions of the products.

At some point, the cost of maintaining a particular subset of features might become prohibitive, and pruning can be used. A new product version could simply omit the extra features, or perhaps a transition period would be used, where old features were deprecated before eventual removal from the system. If there are multiple variations of products, then some of them might be phased out of use.

Consequences

Occasionally, uncontrolled feature creep can lead to products far beyond the scope of what was originally intended. For example, the video game The Elder Scrolls: Arena was originally intended to be a "medieval style gladiator game", however due to feature creep the game quickly expanded into a very large open role-playing game, spawning several sequels. Another example of this is the game Shogun: Total War, which was originally intended to be simply a "B-grade" combat simulation game, but also expanded to produce multiple sequels. Microsoft's Windows Vista was planned to be a minor release between Windows XP and then the codenamed Windows "Blackcomb"(Windows 7), but it turned out to become a major release which took 5 years of development. However, a more common consequence of feature creep is to produce the cancellation of the product, which almost invariably becomes more expensive than was originally intended.

Issue Log

Issue Log is a documentation element of software project management. An issue log contains a list of ongoing and closed issues of the project. While issue logs can be viewed as a way to track errors in the project, the role it plays often extends further. Issue logs can be used to order and organize the current issues by type and severity in order to prioritize issues associated with the current milestone or iteration. Issue logs may also contain customer requests and remarks about the various problems that can be found in current code.

Issue Management

An issue log is usually blank at the beginning of the project, but this is not always true for subsequent releases. In some projects, the issue log is actually used as a guideline for the release schedule; in that case the issue log can be populated with issues that are specifically tagged for completion in the upcoming release. As a result, issue log-guided projects may be easier to manage in terms of completion time and progress estimation. In large projects, issues are usually managed by issue tracking software that can provide different ways and tools to help the project manager and the development team handle thousands of issues for one or several of their projects. Some issue tracking systems also provide a way for the community to contribute new ideas and/or code to the project; this type of collaboration is widely used in [Open source|open-source] programming.

Release Issues : Known Issues

In a case when the project issues can not be fully resolved (such as in pre-release development stages), a Known Issues document is supplied with the software. That document contains a list of issues that are known to exist and, in some cases, instructions on how to overcome the problems caused by these issues.

Template

In a typical issue log, the document must be a table containing multiple rows in which each row describes a separate issue. The various attributes of the issue are listed in different columns. An example of a typical issue log is shown below.

Basic Issue Information

- **Issue reference number (ID)**: Typical number to identify different issues.
- **Issue name**: Issue's name.
- **Description**: Briefly describe what the issue concerns.
- **Issue author**: The person who raise this issue.
- **Parties**: all the people involved in solving the issue.

Issue Categories

- **Issue type**: what knowledge domain the issue belongs to. (E.g. IT infrastructure, IT application, etc.)
- **Issue priority**: it determines which issue is the most urgent and should be solved first. (E.g. the priorities may encompass Immediate, Soon, Later, etc.)
- **Issue severity**: how bad the consequence would be if the issue is left unsolved. (E.g. the severity may encompass Vital, Major, Medium, Minor, etc.)

Issue Date Information

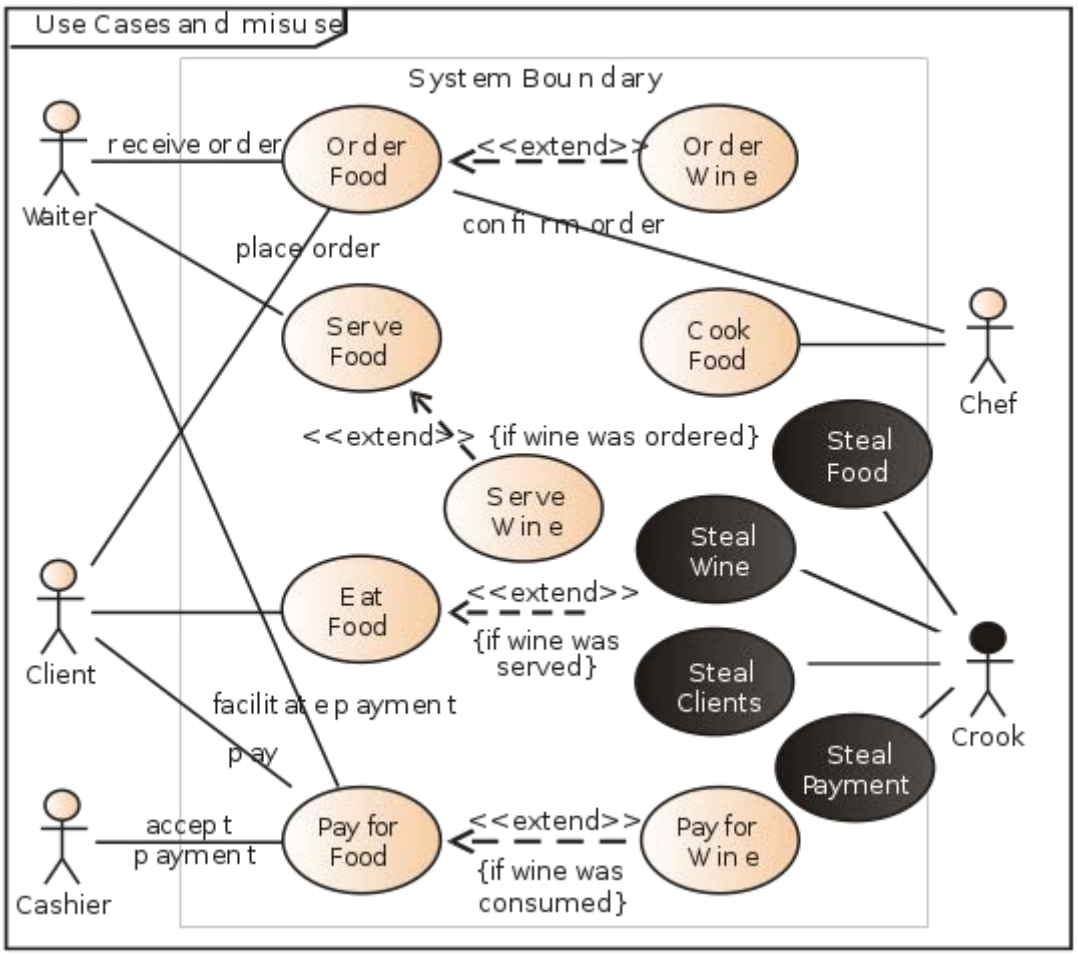
- **Date raised**: when the issue is raised.
- **Date assigned**: when the issue is assigned.
- **Deadline**: when is the final date to get the issue settled.
- **Date resolved**: when the issue is actually solved.

Issue Status

- **Current status**: the current status the issue is within. (E.g. Investigating, escalated, resolved, etc.)
- **Actions updating**: Actions performed before issue is resolved (List all the actions according to dates.)
- **Resolution**: The final resolution to settle the issue.

Chapter-8

Misuse Case



Example of the Misuse case principle, which could be used in thinking about capturing security requirements.

Misuse Case is a business process modeling tool used in the software development business. The term "Misuse case" or "mis-use case" has derived from use case, meaning it is the inverse of a use case. The concept was created in the 1990s by Guttorm Sindre of

the Norwegian University of Science and Technology, and Andreas L. Opdahl of the University of Bergen, Norway. The basic concept is describing the steps of performing a malicious act against a system, just as you would describe an act that the system is supposed to perform in a use case.

Overview

The required behaviour of software and other products under development is often specified in Use cases, which are essentially structured stories or scenarios describing what should happen when the software or product is used. In contrast, a Misuse Case describes or at least names something that should not happen, a Negative Scenario. The threat thus posed often leads to new requirements, which again may be expressed in Use Cases.

It is argued that this modeling tool has several strengths:

- Using misuse cases as a modeling tool lets you treat non-functional requirements (e.g., typical security requirements, platform requirements, etc.) in a way more similar to functional requirements than with many other tools.
- It makes you focus on security from the very beginning, and at the same time, it helps you avoid taking premature design decisions.
- It is a very good tool for enhancing the communication between the developers and the stakeholders and is a valuable tool to use for verifying that the developer(s) and the stakeholder(s) agree on critical system solutions. This is in particular true regarding Trade-off analysis.
- Creating misuse cases will often trigger a chain reaction which makes it easier to identify both functional and non-functional requirements. The discovery of a misuse case will often lead to the creation of a new use case as a counter measure (which in turn might be subject for yet a new misuse case).
- It relates very well to both use cases and UML making employing the model more seamless than what would be the situation with many other tools.

It also has some weaknesses. Its simplicity is one of them; it often needs to be combined with more powerful tools in order to establish an adequate plan for the accomplishment of a project. One other weakness is its lack of structure and semantics.

From use to misuse case

In an industry it's important to describe a system's behavior when it responds to a request that originates from outside : the use cases have become popular for requirements between the engineers thanks to its features like the visual modeling technique, they describe a system from an actor's viewpoint and it format explicitly conveys each actor's goals and the flows the system must implement to accomplish them .

The level of abstraction of an use case model makes it an appropriate starting point for design activities, thanks to the use of UML use case diagrams and the end user's or

domain expert's language. But for software security analyses, the developers should pay attention to negative scenarios and understand them. That is why, in the 1990s, the concept of "inverse of an use case" is born in Norway.

The contrast between the misuse case and the use case is the goal: the misuse case describes potential system behaviors that a system's stakeholders consider unacceptable or, as Guttorm Sindre and Andreas L. Opdahl said, "a function that the system should not allow". This difference is also in the scenarios: a "positive" scenario is a sequence of actions leading to a Goal desired by a person or organization, while a "negative" one is a scenario whose goal is desired not to occur by the organization in question or desired by a hostile agent (not necessarily human).

Another description of the difference is by that defines a use case as a completed sequence of actions which gives increased value to the user, one could define a misuse case as a completed sequence of actions which results in loss for the organization or some specific stakeholder.

Between the "good" and the "bad" case the language to represent the scenario is common: the use case diagrams are formally included in two modeling languages defined by the OMG: the Unified Modeling Language (UML) and the Systems Modeling Language (SysML), and this use of drawing the agents and misuse cases of the scenario explicitly helps focus attention on it .

Area of use

The area where misuse case are the most used is security. Nowadays with the ever growing importance of IT system in our modern economy, the capability to protect your data has became vital for almost every company.

When it comes to application security the "common sense" can't be used anymore. Because hackers use most of the time black hole already present in software to gain access to something they should not have had access to. So the best way to avoid leaving some "black holes" for malicious persons, is to take the same point of view as them and to describe what they might be able to do.

The idea is to "draw a usecase but from a hacker's point of view". So that during the implementation phase all the behavior related to the application are known (good and bad) and can be taken in consideration to choose the good way to go.

Basic concepts

A misuse case diagram is created together with a corresponding use case diagram. The model introduces 2 new important entities (in addition to those from the traditional use case model, *use case* and *actor*:

- *Misuse case* : A sequence of actions that can be performed by any person or entity in order to harm the system.
- *Misuser* : The actor that initiates the misuse case. This can either be done intentionally or inadvertently.

Diagrams

The misuse case model makes use of those relation types found in the use case model; *include*, *extend*, *generalize* and *association*. In addition, it introduces 2 new relations to be used in the diagram:

mitigates
: A use case can mitigate the chance that a misuse case will complete successfully.

threatens
: A misuse case can threaten a use case, e.g. by exploiting it or hinder it to achieve its goals.

These new concepts together with the existing ones from use case gives the following meta model, which is also found as fig. 2 in Sindre and Opdahl (2004).

Descriptions

There are two different way of describing a misuse case textual; one is embedded in a use case description template - where you add an extra description field called **Threats**. This is the field where you fill in your misuse case steps (and alternate steps). This is referred to as the *lightweight* mode of describing a misuse case.

The other way of describing a misuse case, is by using a separate template for this purpose only. It is suggested to inherit some of the field from use case description (**Name**, **Summary**, **Author** and **Date**). It also adapts the fields **Basic path** and **Alternative path**, where they now describe the paths of the misuse cases instead of the use cases. In addition to there, it is proposed to use several other fields too:

- Misuse case name
- Summary
- Author
- Date
- Basic path
- Alternative paths
- Mitigation points
- Extension points
- Triggers
- Preconditions
- Assumptions
- Mitigation guarantee
- Related business rules

- Potential misuser profile
- Stakeholders and threats
- Terminology and explanations
- Scope
- Abstraction level
- Precision level

As one might understand, the list above is too comprehensive to be completely filled out every time. Not all the fields are required to be filled in at the beginning, and it should thus be viewed as a living document. There has also been some debating whether to start with diagrams or to start with descriptions. The recommendation gives by Sindre and Opdahl on that matter is that it should be done as with use cases. Do it the way you feel most familiar with, since both variants each have their strengths and their weaknesses.

Sindre and Opdahl proposes the following 5 steps for using misuse cases to identify security requirements:

1. *Identify critical assets* in the system
2. *Define security goals* for each assets
3. *Identify threats* to each of these security goals, by identifying the stakeholders that may want to cause harm to the system
4. *Identify and analyze* risks for the threats, using techniques like Risk Assessment
5. *Define security requirements* for the risks.

It is suggested to use a repository of reusable misuse cases as a support in this 5-step process.

Research

Current field of research

Current research on misuse cases are primarily focused on the security improvements they can bring to a project, software projects in particular. Ways to increase the widespread adoption of the practice of misuse case development during earlier phases of application development are being considered: the sooner you find a flaw, the easier it is to find a patch and the lower the impact is on the final cost of the project.

Other research focuses on improving the misuse case to achieve its final goal: for "there is a lack on the application process, and the results are too general and can cause a under-definition or misinterpretation of their concepts". They suggest furthermore "to see the misuse case in the light of a reference model for *information system security risk management* (ISSRM)" to obtain a security risk management process.

Future improvement

The misuse cases are well known by the population of researchers. The body of research on the subject demonstrate the knowledge, but beyond the academic world, the misuse case has not been broadly adopted.

As Sindre and Opdahl (the parents of the misuse case concept) suggest: "Another important goal for further work is to facilitate broader industrial adoption of misuse cases". They propose, in the same article, to embedd the misuse case in a usecase modeling tool and to create a "database" of standard misuse cases to assist software architects. System stakeholders should create their own misuse case charts for requirements that are specific to their own problem domains. Once developed, a knowledge database can reduce the amount of standard security flaws used by lambda hackers.

Other research focused on possible missing concrete solutions of the misuse case: as wrote "While this approach can help in a high level elicitation of security requirements, it does not show how to associate the misuse cases to legitimate behavior and concrete assets; therefore, it is not clear what misuse case should be considered, nor in what context". These criticisms might be addressed with the suggestions and improvements presented in the precedent section.

Standardization of the misuse case as part of the UML notation might allow it to become a mandatory part of project development. "It might be useful to create a specific notation for security functionality, or countermeasures that have been added to mitigate vulnerabilities and threats."

Chapter-9

Software Development Effort Estimation and MoSCoW Method

Software development effort estimation

Software development efforts estimation is the process of predicting the most realistic use of effort required to develop or maintain software based on incomplete, uncertain and/or noisy input. Effort estimates may be used as input to project plans, iteration plans, budgets, investment analyses, pricing processes and bidding rounds.

State-of-practice

Published surveys on estimation practice suggest that expert estimation is the dominant strategy when estimating software development effort.

Typically, effort estimates are over-optimistic and there is a strong over-confidence in their accuracy. The mean effort overrun seems to be about 30% and not decreasing over time. However, the measurement of estimation error is not unproblematic. The strong over-confidence in the accuracy of the effort estimates is illustrated by the finding that, on average, if a software professional is 90% confident or "almost sure" to include the actual effort in a minimum-maximum interval, the observed frequency of including the actual effort is only 60-70% .

Currently the term "effort estimate" is used to denote as different concepts as most likely use of effort (modal value), the effort that corresponds to a probability of 50% of not exceeding (median), the planned effort, the budgeted effort or the effort used to propose a bid or price to the client. This is believed to be unfortunate, because communication problems may occur and because the concepts serve different goals.

History

Software researchers and practitioners have been addressing the problems of effort estimation for software development projects since at least the 1960s; see, e.g., work by Farr and Nelson .

Most of the research has focused on the construction of formal software effort estimation models. The early models were typically based on regression analysis or mathematically derived from theories from other domains. Since then a high number of model building approaches have been evaluated, such as approaches founded on case-based reasoning, classification and regression trees, simulation, neural networks, Bayesian statistics, lexical analysis of requirement specifications, genetic programming, linear programming, economic production models, soft computing, fuzzy logic modeling, statistical bootstrapping, and combinations of two or more of these models. The perhaps most common estimation products today, e.g., the formal estimation models COCOMO and SLIM have their basis in estimation research conducted in the 1970s and 1980s. The estimation approaches based on functionality-based size measures, e.g., function points, is also based on research conducted in the 1970s and 1980s, but are re-appearing with modified size measures under different labels, such as "use case points" in the 1990s and COSMIC in the 2000s.

Estimation approaches

There are many ways of categorizing estimation approaches, see for example . The top level categories are the following:

- Expert estimation: The quantification step, i.e., the step where the estimate is produced based on judgmental processes.
- Formal estimation model: The quantification step is based on mechanical processes, e.g., the use of a formula derived from historical data.
- Combination-based estimation: The quantification step is based on a judgmental or mechanical combination of estimates from different sources.

Below are examples of estimation approaches within each category.

Estimation approach	Category	Examples of support of implementation of estimation approach
Analogy-based estimation	Formal estimation model	ANGEL, Weighted Micro Function Points
WBS-based (bottom up) estimation	Expert estimation	Project management software, company specific activity templates
Parametric models	Formal estimation model	COCOMO, SLIM, SEER-SEM

Size-based estimation models	Formal estimation model	Function Point Analysis, Use Case Analysis, Story points-based estimation in Agile software development
Group estimation	Expert estimation	Planning poker, Wideband Delphi
Mechanical combination	Combination-based estimation	Average of an analogy-based and a Work breakdown structure-based effort estimate
Judgmental combination	Combination-based estimation	Expert judgment based on estimates from a parametric model and group estimation

Selection of estimation approach

The evidence on differences in estimation accuracy of different estimation approaches and models suggest that there is no "best approach" and that the relative accuracy of one approach or model in comparison to another depends strongly on the context . This implies that different organizations benefit from different estimation approaches. Findings, summarized in , that may support the selection of estimation approach based on the expected accuracy of an approach include:

- Expert estimation is on average at least as accurate as model-based effort estimation. In particular, situations with unstable relationships and information of high importance not included in the model may suggest use of expert estimation. This assumes, of course, that experts with relevant experience are available.

- Formal estimation models not tailored to a particular organization's own context, may be very inaccurate. Use of own historical data is consequently crucial if one cannot be sure that the estimation model's core relationships (e.g., formula parameters) are based on similar project contexts.

- Formal estimation models may be particularly useful in situations where the model is tailored to the organization's context (either through use of own historical data or that the model is derived from similar projects and contexts), and/or it is likely that the experts' estimates will be subject to a strong degree of wishful thinking.

The most robust finding, in many forecasting domains, is that combination of estimates from independent sources, preferable applying different approaches, will on average improve the estimation accuracy.

In addition, other factors such as ease of understanding and communicating the results of an approach, ease of use of an approach, cost of introduction of an approach should be considered in a selection process.

Uncertainty assessment approaches

The uncertainty of an effort estimate can be described through a prediction interval (PI). An effort PI is based on a stated certainty level and contains a minimum and a maximum effort value. For example, a project leader may estimate that the most likely effort of a project is 1000 work-hours and that it is 90% certain that the actual effort will be between 500 and 2000 work-hours. Then, the interval [500, 2000] work-hours is the 90% PI of the effort estimate of 1000 work-hours. Frequently, other terms are used instead of PI, e.g., prediction bounds, prediction limits, interval prediction, prediction region and, unfortunately, confidence interval. An important difference between confidence interval and PI is that PI refers to the uncertainty of an estimate, while confidence interval usually refers to the uncertainty associated with the parameters of an estimation model or distribution, e.g., the uncertainty of the mean value of a distribution of effort values. The confidence level of a PI refers to the expected (or subjective) probability that the real value is within the predicted interval.

There are several possible approaches to calculate effort PIs, e.g., formal approaches based on regression or bootstrapping , formal or judgmental approaches based on the distribution of previous estimation error , and pure expert judgment of minimum-maximum effort for a given level of confidence. Expert judgments based on the distribution of previous estimation error has been found to systematically lead to more realistic uncertainty assessment than the traditional minimum-maximum effort intervals in several studies, see for example .

Assessing and interpreting the accuracy of effort estimates

The most common measures of the average estimation accuracy is the MMRE (Mean Magnitude of Relative Error), where MRE is defined as:

MRE = |actual effort − estimated effort| / |actual effort|

This measure has been criticized and there are several alternative measures, such as more symmetric measures , Weighted Mean of Quartiles of relative errors (WMQ) and Mean Variation from Estimate (MVFE) .

A high estimation error cannot automatically be interpreted as an indicator of low estimation ability. Alternative, competing or complementing, reasons include low cost control of project, high complexity of development work, and more delivered functionality than originally estimated. A framework for improved use and interpretation of estimation error measurement is included in .

Psychological issues related to effort estimation

There are many psychological factors potentially explaining the strong tendency towards over-optimistic effort estimates that need to be dealt with to increase accuracy of effort estimates. These factors are essential even when using formal estimation models, because

much of the input to these models is judgment-based. Factors that have been demonstrated to be important are: Wishful thinking, anchoring, planning fallacy and cognitive dissonance. A discussion on these and other factors can be found in work by Jørgensen and Grimstad .

- It's easy to estimate what you know.
- It's hard to estimate what you know you don't know.
- It's very hard to estimate things that you don't know you don't know.

MoSCoW Method

MoSCoW is a prioritisation technique used in business analysis and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement - also known as *MoSCoW prioritisation* or *MoSCoW analysis*.

The capital letters in *MoSCoW* stand for:

- **M** - MUST have this.
- **S** - SHOULD have this if at all possible.
- **C** - COULD have this if it does not affect anything else.
- **W** - WON'T have this time but WOULD like in the future. Alternatively WANT.

The o's in MoSCoW are added simply to make the word pronounceable, and are often left lower case to indicate that they don't stand for anything. While some suggest it should be *MuSCoW* (to more accurately reflect the words behind the acronym), MoSCoW is preferred as it is more easily remembered as the capital city of the Russian Federation.

MOSCOW is an acceptable variant, with the 'o's in upper case

Background

This use of *MoSCoW* was first developed by Dai Clegg of Oracle UK Consulting; in *CASE Method Fast-Track: A RAD Approach* ; although he subsequently donated the Intellectual Property Rights to the Dynamic Systems Development Method (DSDM) Consortium.

MoSCoW is often used with timeboxing, where a deadline is fixed so that the focus can be on the most important requirements, and as such is seen as a core aspect of rapid application development (RAD) software development processes, such as Dynamic Systems Development Method (DSDM) and agile software development techniques.

Explanation

All requirements are important, but they are prioritised to deliver the greatest and most immediate business benefits early. Developers will initially try to deliver all the *M*, *S* and *C* requirements but the *S* and *C* requirements will be the first to go if the delivery timescale looks threatened.

The plain English meaning of the MoSCoW words has value in getting customers to understand what they are doing during prioritisation in a way that other ways of attaching priority, like high, medium and low, do not.

Must have
: requirements labeled as *MUST* have to be included in the current delivery timebox in order for it to be a success. If even one *MUST* requirement is not included, the project delivery should be considered a failure (note: requirements can be downgraded from *MUST*, by agreement with all relevant stakeholders; for example, when new requirements are deemed more important). *MUST* can also be considered a backronym for the **M**inimum **U**sable **S**ubse**T**.

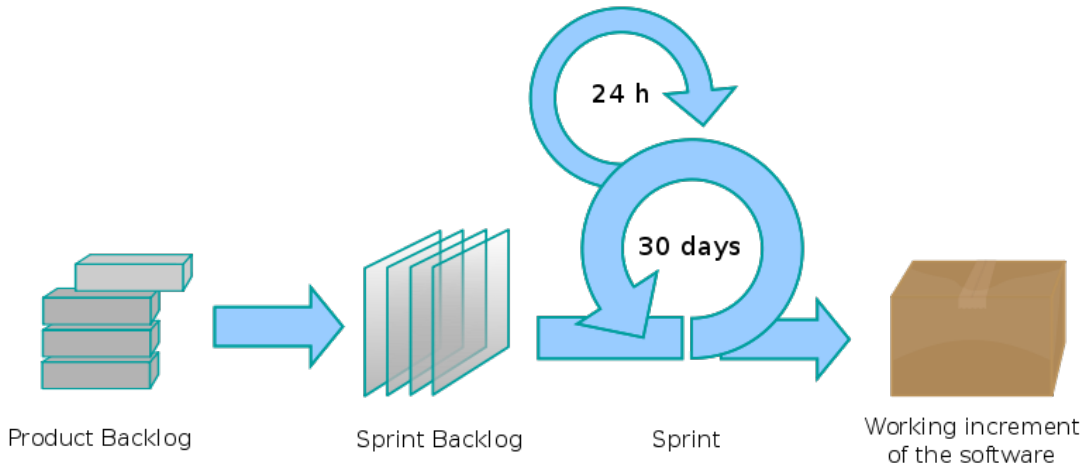
Should have
: *SHOULD* requirements are also critical to the success of the project, but are not necessary for delivery in the current delivery timebox. *SHOULD* requirements are as important as *MUST*, although *SHOULD* requirements are often not as time-critical or have workarounds, allowing another way of satisfying the requirement, so can be held back until a future delivery timebox.

Could have
: requirements labelled as *COULD* are less critical and often seen as *nice to have*. A few easily satisfied *COULD* requirements in a delivery can increase customer satisfaction for little development cost.

Won't have (but Would like)
: *WON'T* requirements are either the least-critical, lowest-payback items, or not appropriate at that time. As a result, *WON'T* requirements are not planned into the schedule for the delivery timebox. *WON'T* requirements are either dropped or reconsidered for inclusion in later timeboxes. This, however doesn't make them any less important.
: Sometimes this is described simply as "**W**ould like to have" in the future, this however leaves some ambiguity in the minds of the users as to its priority compared to the other marks.

Chapter-10

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.

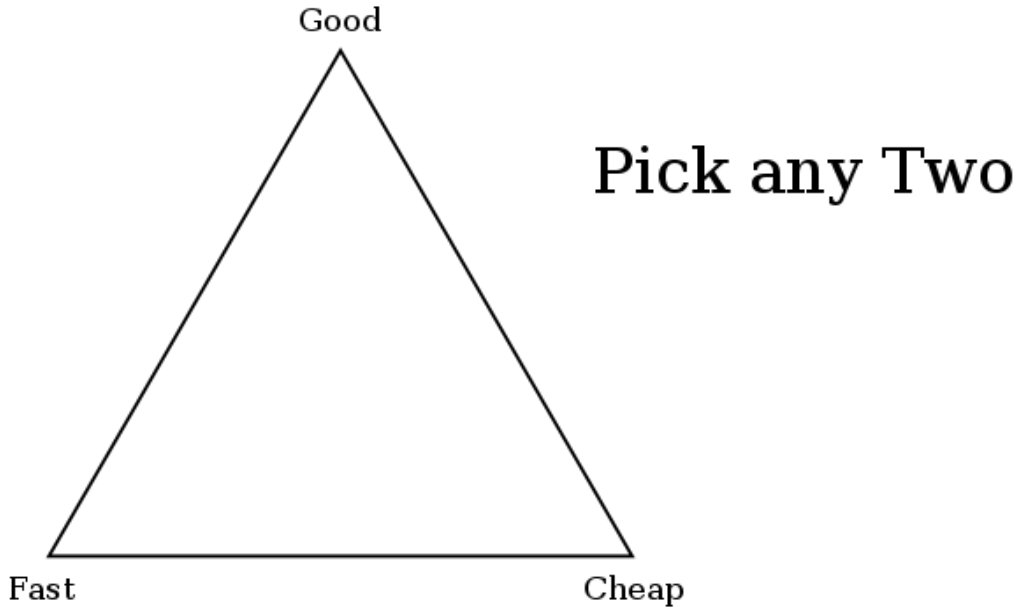
Chapter-11

Project Triangle and Planning Poker

Project triangle

The **Project Triangle** in engineering is a triangle model of project. It is a graphic aid where the three attributes show on the corners of the triangle to show opposition. It is useful to help with intentionally choosing project biases, or analyzing the goals of your project. The constraints are analogous with the project management triangle - in fact, it may be seen as an Evolutionary pressure.

Overview



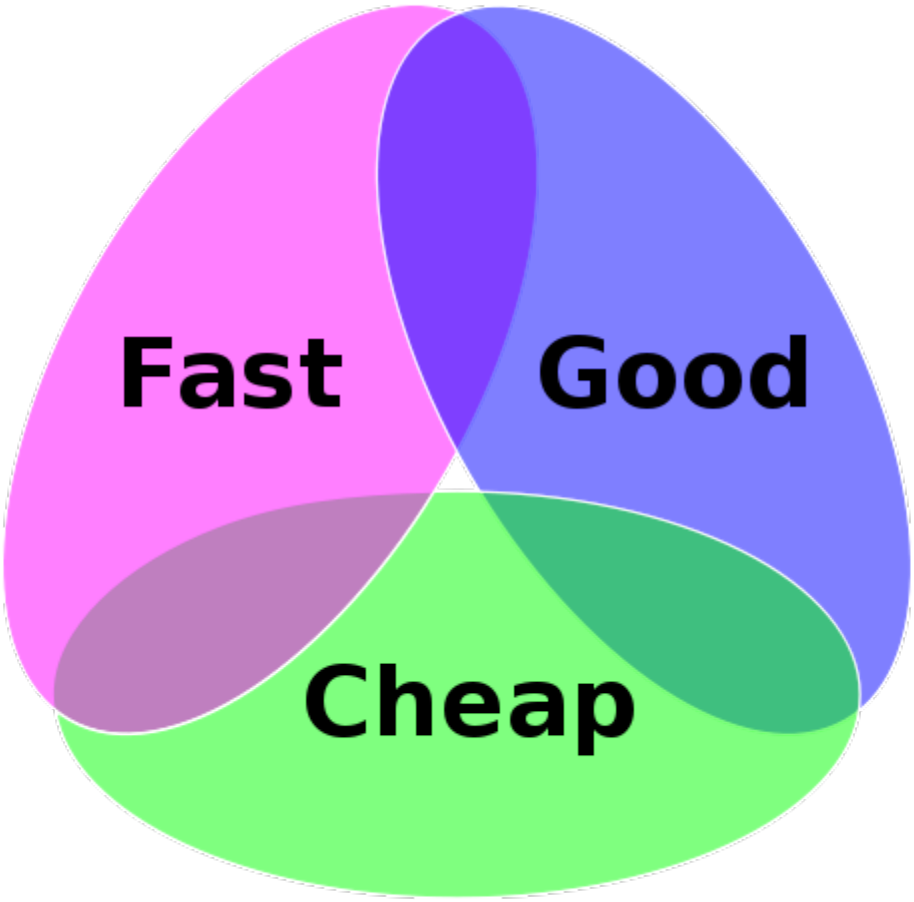
The project triangle.

As a project management graphic aid, a triangle can show time, resources, and technical objective as the **sides** of a triangle, instead of the corners. John Storck, a former instructor of the American Management Association's "Basic Project Management" course, used a pair of triangles called triangle outer and triangle inner to represent the concept that the intent of a project is to complete on or before the allowed time, on or under budget, and to meet or exceed the required scope. The distance between the inner and outer triangles illustrated the hedge or contingency for each of the three elements. Bias could be shown by the distance. His example of a project with a strong time bias was the Alaska pipeline which essentially had to be done on time no matter the cost. After years of development, oil flowed out the end of the pipe within four minutes of schedule. In this illustration, the time side of triangle inner was effectively on top of the triangle outer line. This was true of the technical objective line also. The cost line of triangle inner, however, was outside since the project ran significantly over budget.

James P. Lewis suggests that **project scope** represents the area of the triangle, and can be chosen as a variable to achieve project success. He calls this relationship **PCTS** (Performance, Cost, Time, Scope), and suggests that a project can pick any three.

The real value of the project triangle is to show the complexity that is present in any project. The plane area of the triangle represents the near infinite variations of priorities that could exist between the three competing values. By acknowledging the limitless variety, possible within the triangle, using this graphic aid can facilitate better project decisions and planning and ensure alignment among team members and the project owners.

Example



The project triangle as a "pick any two" Euler diagram.

You are given the options of *Fast*, *Good* and *Cheap*, and told to pick any two. Here *Fast* refers to the time required to deliver the product, *Good* is the quality of the final product, and *Cheap* refers to the total cost of designing and building the product. This triangle reflects the fact that the three properties of a project are interrelated, and it is not possible to optimize all three – one will always suffer. In other words you have three options:

- Design something quickly and to a high standard, but then it will not be cheap.
- Design something quickly and cheaply, but it will not be of high quality.
- Design something with high quality and cheaply, but it will take a long time.

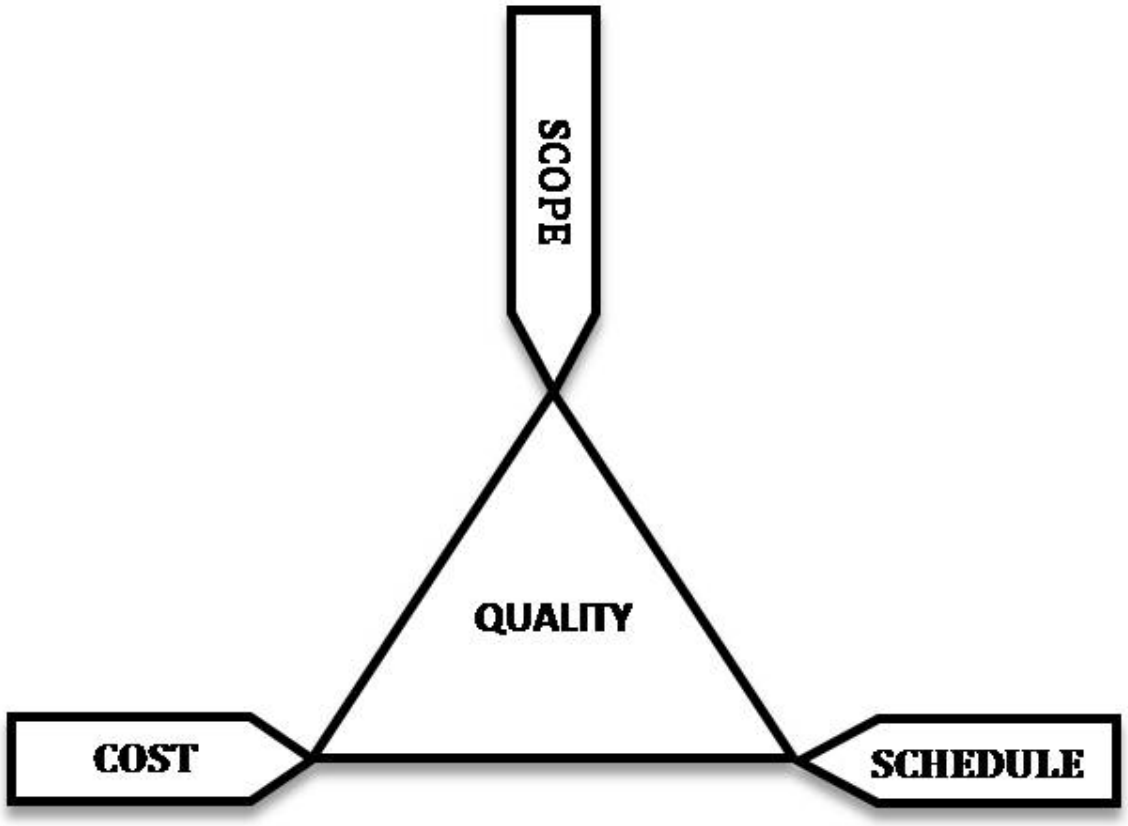
This constraint may apply to creative human activity, such as software, movies, books etc where quality is subjective. This is a false dichotomy when comparing technologies such as digital to analog in media formats.

There are also numerous spinoffs to this triangle, the most common including:

- College: Work, Sleep, Play – Pick two.
- Men: Handsome, High-Earner, Faithful – Pick two.

- Women: Single, Sane, Sexy, Smart – Pick any three. (also called *The four S's of dating*)
- Operating System: Fast, Efficient, Stable - Choose two.
- Bicycle Parts: Strong, Light, Cheap - Pick any two.
- Photographic lenses: Fast, Sharp, Cheap - Pick two.
- Open-source software development: Speed/Time, Inclusiveness/Openness, Quality
- Schedule, Scope, Resources – Pick two.

Project Management Triangle



The Project Management Triangle

Like any human undertaking, projects need to be performed and delivered under certain constraints. Traditionally, these constraints have been listed as "scope," "time," and "cost". These are also referred to as the "Project Management Triangle," where each side represents a constraint. One side of the triangle cannot be changed without affecting the others. A further refinement of the constraints separates product "quality" or "performance" from scope, and turns quality into a fourth constraint.

The time constraint refers to the amount of time available to complete a project. The cost constraint refers to the budgeted amount available for the project. The scope constraint refers to what must be done to produce the project's end result. These three constraints

are often competing constraints: increased scope typically means increased time and increased cost, a tight time constraint could mean increased costs and reduced scope, and a tight budget could mean increased time and reduced scope.

The discipline of Project Management is about providing the tools and techniques that enable the project team (not just the project manager) to organize their work to meet these constraints.

It is worthy to note that in the latest version of the PMBOK, PMI has done away with the project triangle, the reason for this is that a project has many more constraints to be observed other than the scope, the time, and the cost.

Planning poker

Planning Poker, also called Scrum poker, is a consensus-based technique for estimating, mostly used to estimate effort or relative size of tasks in software development. It is a variation of the Wideband Delphi method. It is most commonly used in agile software development, in particular the Extreme Programming methodology.

The method was first described by James Grenning in 2002 and later popularized by Mike Cohn in the book *Agile Estimating and Planning*.

Process

Equipment

Planning Poker is based on a list of features to be delivered and several copies of a deck of numbered cards. The feature list, often a list of user stories, describes some software that needs to be developed.

The cards in the deck have numbers on them. A typical deck has cards showing the Fibonacci sequence including a zero: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89; other decks use similar progressions.

The reason for using the Fibonacci sequence is to reflect the inherent uncertainty in estimating larger items.

One commercially-available deck uses the sequence: 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, 100, and optionally a ? (unsure) and a coffee cup (I need a break). Some organizations use standard playing cards of Ace, 2 ,3 5, 8, and King. Where King means: "this item is too big or too complicated to estimate." "Throwing a King" ends discussion of the item for the current sprint.

Optionally, an egg timer can be used to limit time spent in discussion of each item.

Procedure

At the estimation meeting, each estimator is given one deck of the cards. All decks have identical sets of cards in them.

The meeting proceeds as follows:

- A Moderator, who will not play, chairs the meeting, supported and advised by the Project Manager.
- The most knowledgeable developer for a given feature provides a short overview. The team is given an opportunity to ask questions and discuss to clarify assumptions and risks. A summary of the discussion is recorded by the Project Manager.
- Each individual lays a card face down representing their estimate. Units used vary - they can be days duration, ideal days or story points. During discussion, numbers must not be mentioned at all in relation to feature size to avoid anchoring.
- Everyone calls their cards simultaneously by turning them over.
- People with high estimates and low estimates are given a soap box to offer their justification for their estimate and then discussion continues.
- Repeat the estimation process until a consensus is reached. The developer who was likely to own the deliverable has a large portion of the "consensus vote", although the Moderator can negotiate the consensus.
- An egg timer is used to ensure that discussion is structured; the Moderator or the Project Manager may at any point turn over the egg timer and when it runs out all discussion must cease and another round of poker is played. The structure in the conversation is re-introduced by the soap boxes.

The cards are numbered as they are to account for the fact that the longer an estimate is, the more uncertainty it contains. Thus, if a developer wants to play a 6 he is forced to reconsider and either work through that some of the perceived uncertainty does not exist and play a 5, or accept a conservative estimate accounting for the uncertainty and play an 8.

Planning Poker benefits

Planning Poker is a tool for estimating software development projects. It is a technique that minimizes anchoring by asking each team member to play their estimate card such

that it cannot be seen by the other players. After each player has selected a card, all cards are exposed at once.

A study by K. Molokken-Ostvold and N.C. Haugen found that estimates obtained through the Planning Poker process were less optimistic and more accurate than estimates obtained through mechanical combination of individual estimates for the same tasks.

Avoid anchoring

Anchoring occurs when a team openly discuss their estimates. A team normally has a mix of conservative and impulsive estimators and there may be people who have agendas; developers are likely to want as much time as they can to do the job and the product owner or customer is likely to want it as quickly as possible.

The estimate becomes anchored when the product owner says something like, "I think this is an easy job, I can't see it taking longer than a couple of weeks", or when the developer says something like, "I think we need to be very careful, clearing up the issues we've had in the back end could take months". Whoever starts the estimating conversation with, "I think it's 50 days" immediately has an impact on the thinking of the other team members; their estimates have been anchored, i.e. they are all now likely to make at least a subconscious reference to the number 50 in their own estimates. Those who were thinking 100 days are likely to reduce and those who thought 10 are likely to raise. This becomes a particular problem if the 50 is spoken by an influential member of the team when the rest of the team are predominantly thinking higher or lower. Because the remainder of the team have been anchored they may consciously or otherwise fail to express their original unity; in fact they may fail to even discover that they were thinking the same thing. This can be dangerous, resulting in estimates that are influenced by agendas or individual opinions that are not focussed on getting the job done right.

Planning poker exposes the potentially influential team member as being isolated in his or her opinion among the group. It then demands that she or he argue the case against the prevailing opinion. If a group is able to express its unity in this manner they are more likely to have faith in their original estimates. If the influential person has a good case to argue everyone will see sense and follow, but at least the rest of the team won't have been anchored; instead they will have listened to reason.

Chapter-12

Rapid Application Development and Release Management

Rapid application development

Rapid application development (**RAD**) refers to a type of software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

Overview

Rapid application development is a software development methodology that involves methods like interactive development and software prototyping. According to Whitten (2004), it is a merger of various structured techniques, especially data-driven Information Engineering, with prototyping techniques to accelerate software systems development.

In rapid application development, structured techniques and prototyping are especially used to define users' requirements and to design the final system. The development process starts with the development of preliminary data models and business process models using structured techniques. In the next stage, requirements are verified using prototyping, eventually to refine the data and process models. These stages are repeated iteratively; further development results in "a combined business requirements and technical design statement to be used for constructing new systems".

RAD approaches may entail compromises in functionality and performance in exchange for enabling faster development and facilitating application maintenance.

History

Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991. Martin's methodology involves iterative development and the construction of prototypes. More recently, the

term and its acronym have come to be used in a broader, general sense that encompasses a variety of methods aimed at speeding application development, such as the use of software frameworks of varied types, such as web application frameworks.

Rapid application development was a response to non-agile processes developed in the 1970s and 1980s, such as the Structured Systems Analysis and Design Method and other Waterfall models. One problem with previous methodologies was that applications took so long to build that requirements had changed before the system was complete, resulting in inadequate or even unusable systems. Another problem was the assumption that a methodical requirements analysis phase alone would identify all the critical requirements. Ample evidence attests to the fact that this is seldom the case, even for projects with highly experienced professionals at all levels.

Starting with the ideas of Brian Gallagher, Alex Balchin, Barry Boehm and Scott Shultz, James Martin developed the rapid application development approach during the 1980s at IBM and finally formalized it by publishing a book in 1991, *Rapid Application Development*.

Relative effectiveness

The shift from traditional session-based client/server development to open sessionless and collaborative development like Web 2.0 has increased the need for faster iterations through the phases of the SDLC. This, coupled with the growing use of open source frameworks and products in core commercial development, has, for many developers, rekindled interest in finding a silver bullet RAD methodology.

Although most RAD methodologies foster software re-use, small team structure and distributed system development, most RAD practitioners recognize that, ultimately, no one “rapid” methodology can provide an order of magnitude improvement over any other development methodology.

All types of RAD have the potential for providing a good framework for faster product development with improved software quality, but successful implementation and benefits often hinge on project type, schedule, software release cycle and corporate culture. It may also be of interest that some of the largest software vendors such as Microsoft and IBM do not extensively use RAD in the development of their flagship products and for the most part, they still primarily rely on traditional waterfall methodologies with some degree of spiraling.

This table contains a high-level summary of some of the major types of RAD and their relative strengths and weaknesses.

Agile software development (Agile)	
Pros	Minimizes feature creep by developing in short intervals resulting in miniature software projects and releasing the product in mini-increments.

Cons	Short iteration may add too little functionality, leading to significant delays in final iterations. Since Agile emphasizes real-time communication (preferably face-to-face), using it is problematic for large multi-team distributed system development. Agile methods produce very little written documentation and require a significant amount of post-project documentation.
Extreme Programming (XP)	
Pros	Lowers the cost of changes through quick spirals of new requirements. Most design activity occurs incrementally and on the fly.
Cons	Programmers must work in pairs, which is difficult for some people. No up-front "detailed design" occurs, which can result in more redesign effort in the long term. The business champion attached to the project full time can potentially become a single point of failure for the project and a major source of stress for a team.
Joint application design (JAD)	
Pros	Captures the voice of the customer by involving them in the design and development of the application through a series of collaborative workshops called JAD sessions.
Cons	The client may create an unrealistic product vision and request extensive gold-plating, leading a team to over- or under-develop functionality.
Lean software development (LD)	
Pros	Creates minimalist solutions (i.e., needs determine technology) and delivers less functionality earlier; per the policy that 80% today is better than 100% tomorrow.
Cons	Product may lose its competitive edge because of insufficient core functionality and may exhibit poor overall quality.
Rapid application development (RAD)	
Pros	Promotes strong collaborative atmosphere and dynamic gathering of requirements. Business owner actively participates in prototyping, writing test cases and performing unit testing.
Cons	Dependence on strong cohesive teams and individual commitment to the project. Decision making relies on the feature functionality team and a communal decision-making process with lesser degree of centralized PM and engineering authority.
Scrum	
Pros	Improved productivity in teams previously paralyzed by heavy "process", ability to prioritize work, use of backlog for completing items in a series of short iterations or sprints, daily measured progress and communications.
Cons	Reliance on facilitation by a master who may lack the political skills to remove impediments and deliver the sprint goal. Due to relying on self-organizing teams and rejecting traditional centralized "process control", internal power struggles can paralyze a team.

Table 1: Pros and Cons of various RAD types

Criticism

Since rapid application development is an iterative and incremental process, it can lead to a succession of prototypes that never culminate in a satisfactory production application. Such failures may be avoided if the application development tools are robust, flexible, and put to proper use. This is addressed in methods such as the 2080 Development method or other post-agile variants.

Practical implications

When organizations adopt rapid development methodologies, care must be taken to avoid role and responsibility confusion and communication breakdown within a development team, and between team and client. In addition, especially in cases where the client is absent or not able to participate with authority in the development process, the system analyst should be endowed with this authority on behalf of the client to ensure appropriate prioritisation of non-functional requirements. Furthermore, no increment of

the system should be developed without a thorough and formally documented design phase.

Release management

The **release management** process is a relatively new but rapidly growing discipline within software engineering of managing software releases.

As software systems, software development processes, and resources become more distributed, they invariably become more specialized and complex. Furthermore, software products (especially web applications) are typically in an ongoing cycle of development, testing, and release. Add to this an evolution and growing complexity of the platforms on which these systems run, and it becomes clear there are a lot of moving pieces that must fit together seamlessly to guarantee the success and long-term value of a product or project.

The need therefore exists for dedicated resources to oversee the integration and flow of development, testing, deployment, and support of these systems. Although project managers have done this in the past, they generally are more concerned with high-level, "grand design" aspects of a project or application, and so often do not have time to oversee some of the more technical or day-to-day aspects. Release managers (aka "RMs") address this need. They must have a general knowledge of every aspect of the software development process, various applicable operating systems and software application or platforms, as well as various business functions and perspectives.

A release manager is:

- Facilitator: serves as a liaison between varying business units to guarantee smooth and timely delivery of software products or updates.
- Gatekeeper: “holds the keys” to production systems/applications and takes responsibility for their implementations.
- Architect: helps to identify, create and/or implement processes or products to efficiently manage the release of code.
- Server application support engineer: help troubleshoot problems with an application (although not typically at a code level).
- Coordinator: utilized to coordinate disparate source trees, projects, teams and components.

Some of the challenges facing a software release manager include the management of:

- Software defects
- Issues
- Risks
- Software change requests

- New development requests (additional features and functions)
- Deployment and packaging
- New development tasks

Impact of agile software development on release management

Agile software development methodologies have driven radically higher numbers of release events in organizations where it has been adopted. More release events have corresponded to increased pressure on release management teams and their colleagues in IT Operations to track and execute complex application release processes. Operations teams have used methodologies—such as Information Technology Infrastructure Library ITIL v3 Book: *Service Transition* (which contains a section on release management) to improve their release management capabilities as they relate to both business applications and internal IT services. Agile has also driven development and operations teams to collaborate more closely during production release events—this trend is referred to as DevOps.

Notable release management software

Notable release management software include:

Name	Vendor
Q	VaraLogix
Go	ThoughtWorks
Nolio ASAP	Nolio
BuildMaster	Inedo

Chapter-13

Use Case

A **use case** in software engineering and systems engineering is a description of steps or actions between a user (or "actor") and a software system which lead the user towards something useful. The user or actor might be a person or something more abstract, such as an external software system or manual process.

Use cases are a software modeling technique that helps developers determine which features to implement and how to gracefully resolve errors.

Within systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals. The detailed requirements may then be captured in SysML requirement diagrams or similar mechanisms.

History

In 1986 Ivar Jacobson, later an important contributor to both the Unified Modeling Language (UML) and the Rational Unified Process (RUP), first formulated the textual, structural and visual modeling techniques for specifying use cases. The use case technique became popular through Jacobson's 1992 book Object-Oriented Software Engineering - A Use Case Driven Approach, co-authored with Magnus Christerson, Patrik Jonsson and Gunnar Overgaard. Originally he used the terms *usage scenarios* and *usage case*, which were the more correct translations of the Swedish word "användningsfall" he used, but found that neither of these terms sounded natural in English, and eventually he settled on the term *use case*. Since Jacobson originated use case modeling many others have contributed to improving this technique, including Kurt Bittner, Ian Spence, Alistair Cockburn, Gunnar Overgaard, Karin Palmquist, Patrik Jonsson, Magnus Christerson and Geri Schneider.

During the 1990s use cases became one of the most common practices for capturing functional requirements. This is especially the case within the object-oriented community where they originated, but their applicability is not restricted to object-oriented systems.

Use cases and the development process

The specific way use cases are used within the development process will depend on which development methodology is being used. In certain development methodologies, a brief use case survey is all that is required. In other development methodologies, use cases evolve in complexity and change in character as the development process proceeds. Use cases can be a valuable source of usage information and usage testing ideas. In some methodologies, they may begin as brief business use cases, evolve into more detailed system use cases, and then eventually develop into highly detailed and exhaustive test cases.

Use case focus

"Each use case focuses on describing how to achieve a goal or a task. For most software projects, this means that multiple, perhaps dozens of use cases are needed to define the scope of the new system. The degree of formality of a particular software project and the stage of the project will influence the level of detail required in each use case.

Use cases should not be confused with the features of the system. One or more features (a.k.a. "system requirements") describe the functionality needed to meet a stakeholder request or user need (a.k.a. "user requirement"). Each feature can be analyzed into one or more use cases, which detail cases where an actor uses the system. Each use case should be traceable to its originating feature, which in turn should be traceable to its originating stakeholder/user request.

Use cases treat the system as a black box, and the interactions with the system, including system responses, are perceived as from outside the system. This is a deliberate policy, because it forces the author to focus on what the system must do, not how it is to be done, and avoids making assumptions about how the functionality will be accomplished.

A use case should:

- Describe what the system shall do for the actor to achieve a particular goal.
- Include no implementation-specific language.
- Be at the appropriate level of detail.
- Not include detail regarding user interfaces and screens. This is done in user-interface design, which references the use case and its business rules.

Degree of detail

Alistair Cockburn, in *Writing Effective Use Cases*, identified three levels of detail in writing use cases:

- Brief use case -- consists of a few sentences summarizing the use case. It can be easily inserted in a spreadsheet cell, and allows the other columns in the

spreadsheet to record priority, duration, a method of estimating duration, technical complexity, release number, and so on.
- Casual use case -- consists of a few paragraphs of text, summarizing the use case.
- Fully dressed use case -- a formal document based on a detailed template with fields for various sections; and it is the most common understanding of the meaning of a use case. Fully dressed use cases are discussed in detail in the next section on use case templates.

Some software development processes do not require anything more than a simple use case to define requirements. However, some other development processes require detailed use cases to define requirements. The larger and more complex the project, the more likely that it will be necessary to use detailed use cases.

The level of detail in a use case often differs according to the progress of the project. The initial use cases may be brief, but as the development process unfolds the use cases become even more detailed. This reflects the different requirements of the use case. Initially they need only be brief, because they are used to summarize the business requirement from the point of view of users. However, later in the process, software developers need far more specific and detailed guidance.

The Rational Unified Process invites developers to write a brief use case description in the use case diagram, with a casual description as comments and a detailed description of the flow of events in a textual analysis. All those can usually be input into the use case tool (e.g., a UML Tool, SysML Tool), or can be written separately in a text editor.

Actors

A use case defines the interactions between external actors and the system under consideration to accomplish a goal. An actor specifies a role played by a person or thing when interacting with the system. The same person using the system may be represented as different actors because they are playing different roles. For example, user "Joe" could be playing the role of a Customer when using an Automated Teller Machine to withdraw cash, or playing the role of a Bank Teller when using the system to restock the cash drawer.

Business vs. System Use Cases

Use cases may be described at the abstract level (business use case, sometimes called essential use case), or at the system level (system use case). The differences between these is the scope.

- A **business use case** is described in technology-free terminology which treats system as a black box and describes the business process that is used by its business actors (people or systems external to the process) to achieve their goals (e.g., manual payment processing, expense report approval, manage corporate real estate). The business use case will describe a process that provides value to the

business actor, and it describes *what* the process does. Business Process Mapping is another method for this level of business description.
- A **system use case** describes a system that automates a business use case or process. It is normally described at the system functionality level (for example, "create voucher") and specifies the function or the service that the system provides for the actor. The system use case details *what* the system will do in response to an actor's actions. For this reason it is recommended that system use case specification begin with a verb (e.g., *create* voucher, *select* payments, *exclude* payment, *cancel* voucher). An actor can be a human user or another system/subsystem interacting with the system being defined.

Use case notation

In Unified Modeling Language, the relationships between all (or a set of) the use cases and actors are represented in a use case diagram or diagrams, originally based upon Ivar Jacobson's Objectory notation. SysML, a UML profile, uses the same notation at the system block level.

Limitations

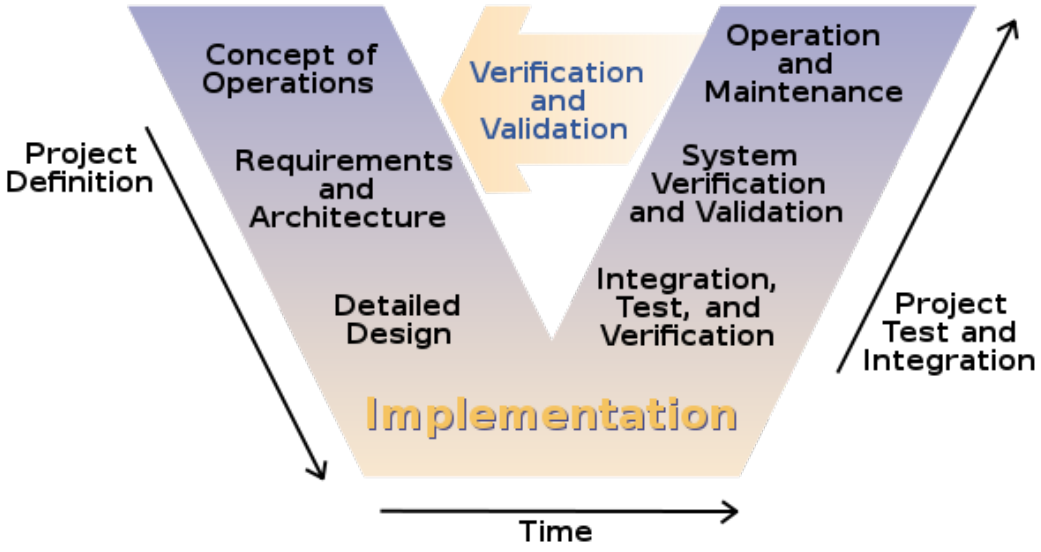
Use cases have limitations:

- Use case flows are not well suited to easily capturing non-interaction based requirements of a system (such as algorithm or mathematical requirements) or non-functional requirements (such as platform, performance, timing, or safety-critical aspects). These are better specified declaratively elsewhere.
- Use case templates do not automatically ensure clarity. Clarity depends on the skill of the writer(s).
- There is a learning curve involved in interpreting use cases correctly, for both end users and developers. As there are no fully standard definitions of use cases, each group must gradually evolve its own interpretation. Some of the relations, such as *extends*, are ambiguous in interpretation and can be difficult for stakeholders to understand.
- Proponents of Extreme Programming often consider use cases too needlessly document-centric, preferring to use the simpler approach of a user story.
- Use case developers often find it difficult to determine the level of user interface (UI) dependency to incorporate in a use case. While use case theory suggests that UI not be reflected in use cases, it can be awkward to abstract out this aspect of design, as it makes the use cases difficult to visualize. Within software engineering, this difficulty is resolved by applying requirements traceability through the use of a traceability matrix.
- Use cases can be over-emphasized. In *Object Oriented Software Construction (2nd edition)*, Bertrand Meyer discusses issues such as driving the system design too literally from use cases and using use cases to the exclusion of other potentially valuable requirements analysis techniques.

- Use cases have received some interest as a starting point for test design. Some use case literature, however, states that use case pre- and postconditions should apply to all scenarios of a use case (i.e., to all possible paths through a use case) which is limiting from a test design standpoint. If the postconditions of a use case are so general as to be valid for all possible use case scenarios, they are likely not to be useful as a basis for specifying expected behavior in test design. For example, the outputs and final state of a failed attempt to withdraw cash from an ATM are not the same as a successful withdrawal: if the postconditions reflect this, they too will differ; if the postconditions don't reflect this, then they can't be used to specify the expected behavior of tests. An alternative perspective on use case pre- and postconditions more suitable for test design based on model-based specification is discussed in.
- Some systems are better described in an information/data-driven approach than in a the functionality-driven approach of use cases. A good example of this kind of system is data-mining systems used for Business Intelligence. If you were to describe this kind of system in a use case model, it would be quite small and uninteresting (there are not many different functions here) but the set of data that the system handles may nevertheless be large and rich in details.

Chapter-14

V-Model



The V-model of the Systems Engineering Process.

The **V-Model** is a systems development model designed to simplify the understanding of the complexity associated with developing systems. In systems engineering it is used to define a uniform procedure for product or project development.

Overview

The V-model is a graphical representation of the systems development lifecycle. It summarizes the main steps to be taken in conjunction with the corresponding deliverables within computerized system validation framework.

The *VEE* represents the sequence of steps in a project life cycle development. It describes the activities and results that have to be produced during product development. The left side of the "V" represents the decomposition of requirements, and creation of system

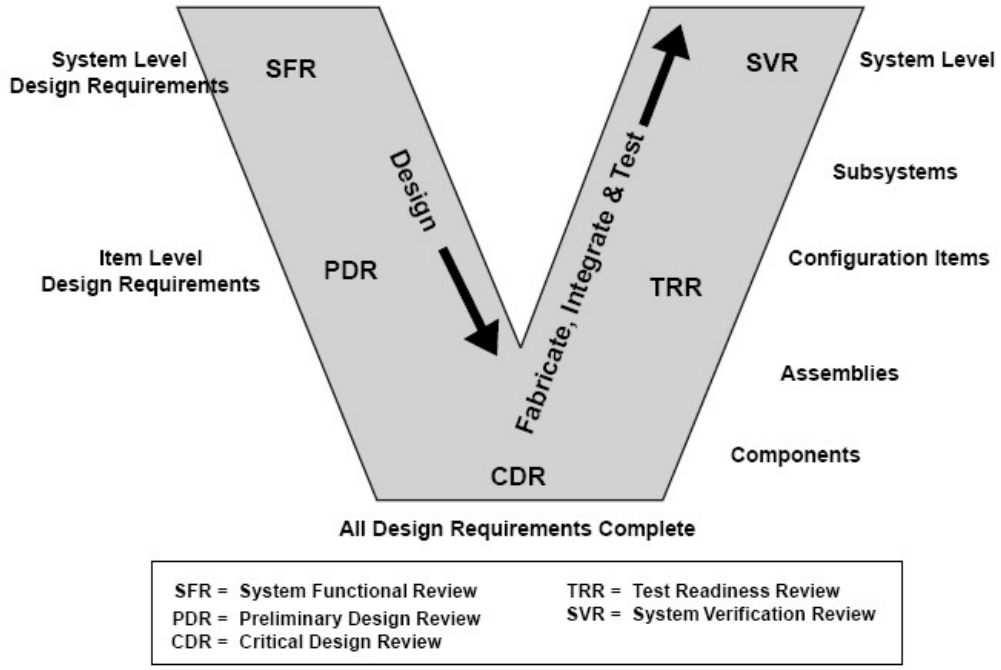
specifications. The right side of the *VEE* represents integration of parts and their verification. V stands for "Verification and Validation".

Objectives

The V-Model provides guidance for the planning and realization of projects. The following objectives are intended to be achieved by a project execution:

- **Minimization of Project Risks**: The V-Model improves project transparency and project control by specifying standardized approaches and describing the corresponding results and responsible roles. It permits an early recognition of planning deviations and risks and improves process management, thus reducing the project risk.
- **Improvement and Guarantee of Quality**: As a standardized process model, the V-Model ensures that the results to be provided are complete and have the desired quality. Defined interim results can be checked at an early stage. Uniform product contents will improve readability, understandability and verifiability.
- **Reduction of Total Cost over the Entire Project and System Life Cycle**: The effort for the development, production, operation and maintenance of a system can be calculated, estimated and controlled in a transparent manner by applying a standardized process model. The results obtained are uniform and easily retraced. This reduces the acquirers dependency on the supplier and the effort for subsequent activities and projects.
- **Improvement of Communication between all Stakeholders**: The standardized and uniform description of all relevant elements and terms is the basis for the mutual understanding between all stakeholders. Thus, the frictional loss between user, acquirer, supplier and developer is reduced.

V Model topics



Systems engineering and verification.

Systems Engineering and verification

The Systems Engineering Process (SEP) provides a path for improving the cost effectiveness of complex systems as experienced by the system owner over the entire life of the system, from conception to retirement.

It involved early and comprehensive identification of goals, a concept of operations that describes user needs and the operating environment, thorough and testable system requirements, detailed design, implementation, rigorous acceptance testing of the implemented system to ensure it meets the stated requirements (system verification), measuring its effectiveness in addressing goals (system validation), on-going operation and maintenance, system upgrades over time, and eventual retirement.

The process emphasizes requirements-driven design and testing. All design elements and acceptance tests must be traceable to one or more system requirements and every requirement must be addressed by at least one design element and acceptance test. Such rigor ensures nothing is done unnecessarily and everything that is necessary is accomplished.

The specification stream

The specification stream mainly consists of:

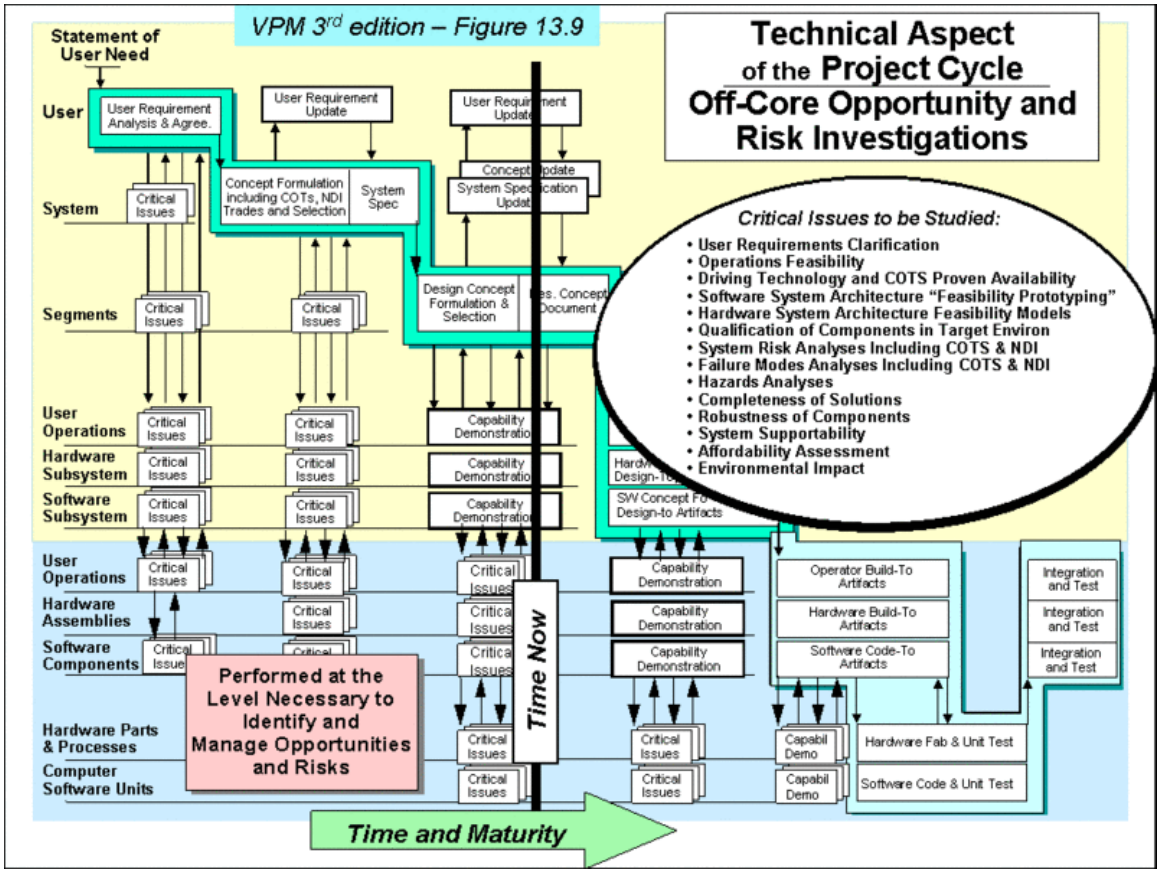
- User Requirement Specifications
- Functional Requirement Specifications
- Design Specifications

The testing stream generally consists of:

- Installation Qualification (IQ)
- Operational Qualification (OQ)
- Performance Qualification (PQ)

The development stream can consist (depending on the system type and the development scope) of customization, configuration or coding.

Applications



Off-Core alternatives (illustrating upward and downward iterations and Time and Maturity dimension).

The V-model is used to regulate the software development process within the German federal administration. Nowadays it is still the standard for German federal administration and defense projects, as well as software developers within in the region.

The concept of the V-Model was developed simultaneously, but independently, in Germany and in the United States in the late 1980s:

- The German V-Model was originally developed by IABG in Ottobrunn, near Munich, in cooperation with the Federal Office for Defence Technology and Procurement in Koblenz, for the Federal Ministry of Defence. It was taken over by the Federal Ministry of the Interior for the civilian public authorities domain in summer 1992.
- The US V-Model, as documented in the 1991 proceedings for the National Council on Systems Engineering (NCOSE; now INCOSE as of 1995), was developed for satellite systems involving hardware, software, and human interaction.
- The V-Model first appeared at Hughes Aircraft circa 1982 as part of the pre-proposal effort for the FAA Advanced Automation System (AAS) program. It eventually formed the test strategy for the Hughes AAS Design Competition Phase (DCP) proposal. It was created to show the test and integration approach which was driven by new challenges to surface latent defects in the software. The need for this new level of latent defect detection was driven by the goal to start automating the thinking and planning processes of the air traffic controller as envisioned by the Automated Enroute Air Traffic Control (AERA) program. The reason the V is so powerful comes from the Hughes culture of coupling all text and analysis to multi dimensional images. It was the foundation of Sequential Thematic Organization of Publications (STOP) created by Hughes in 1963 and used until Hughes was divested by the Howard Hughes Medical Institute in 1985.

It has now found widespread application in commercial as well as defence programs. Its primary use is in Project Management and throughout the project lifecycle.

One fundamental characteristic of the US V-Model is that time and maturity move from left to right and one cannot move back in time. All iteration is along a vertical line to higher or lower levels in the system hierarchy, as shown in the figure. This has proven to be an important aspect of the model. The expansion of the model to a dual-Vee concept is treated in reference.

As the V-model is publicly available many companies also use it. In project management it is a method comparable to PRINCE2 and describes methods for project management as well as methods for system development. The V-Model while rigid in process, can be very flexible in application, especially as it pertains to the scope outside of the realm of the System Development Lifecycle normal parameters.

Advantages

These are the advantages V-Model offers in front of other systems development models:

- The users of The V-Model participate in the development and maintenance of The V-Model. A change control board publicly maintains the V-Model. The change control board meets once a year and processes all received change requests on The V-Model.
- At each project start, the V-Model can be tailored into a specific project V-Model, this being possible because the V-Model is organization and project independent.
- The V-Model provides concrete assistance on how to implement an activity and its work steps, defining explicitly the events needed to complete a work step: each activity schema contains instructions, recommendations and detailed explanations of the activity.

Limits

The following aspects are not covered by the V-Model, they must be regulated in addition, or the V-Model must be adapted accordingly :

- The placing of contracts for services is not regulated.
- The organization and execution of operation, maintenance, repair and disposal of the system are not covered by the V-Model. However, planning and preparation of a concept for these tasks are regulated in the V-Model.
- The V-Model addresses software development within a project rather than a whole organization.