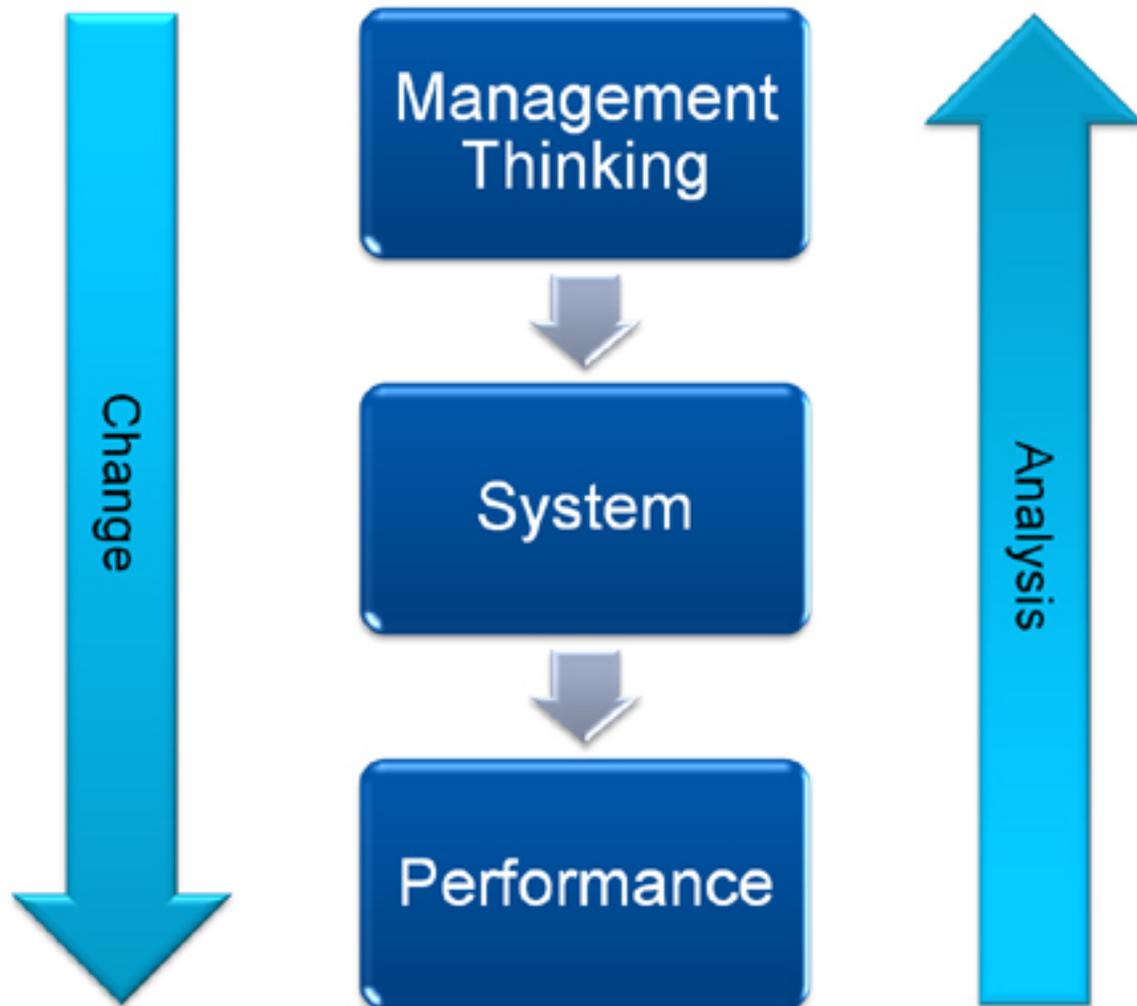


Software Project Management & Agile Software Development



Landon Sewell

Lise Pollack

First Edition, 2012

ISBN 978-81-323-0987-1

© All rights reserved.

Published by:

Academic Studio

4735/22 Prakashdeep Bldg,

Ansari Road, Darya Ganj,

Delhi - 110002

Email: info@wtbooks.com

Table of Contents

Chapter 1 - Software Project Management

Chapter 2 - Application Lifecycle Management and Baseline (Configuration Management)

Chapter 3 - Software Development and Brownfield (Software Development)

Chapter 4 - Dual Vee Model

Chapter 5 - IBM Rational Unified Process

Chapter 6 - Endeavour Software Project Management, Feature Creep and issue log

Chapter 7 - Misuse Case

Chapter 8 - Software Development Effort Estimation and MoSCoW Method

Chapter 9 - Project Triangle and Planning Poker

Chapter 10 - Rapid Application Development and Release Management

Chapter 11 - Agile Software Development

Chapter 12 - Extreme Programming

Chapter 13 - Applied Agile Software Development

Chapter 14 - Agilo for Scrum

Chapter 15 - Feature Driven Development

Chapter 16 - DevOps

Chapter 17 - Lean Software Development

Chapter 18 - Scrum (Development)

Chapter 19 - Presenter First & Planning Poker

Chapter 20 - P-Modeling Framework

Chapter 21 - Test-Driven Development

Chapter 22 - Extreme Programming Practices

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 unfavorably). 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

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.

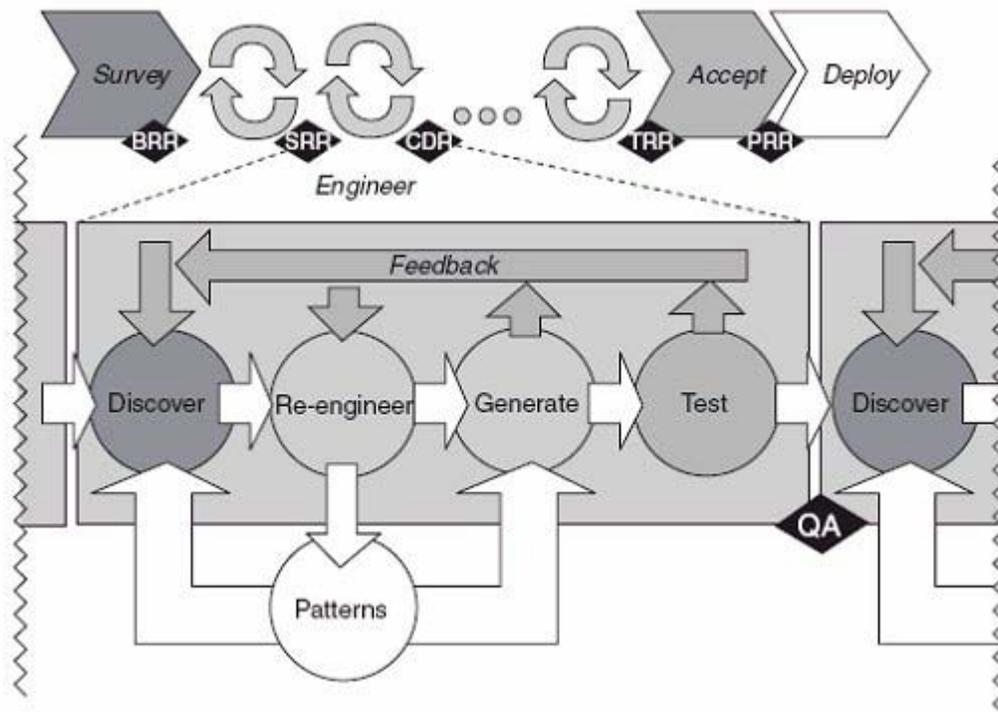
As a result, an increasing proportion of the effort of developing new business capabilities is spent on understanding and integrating with the existing complex system and business landscape rather than delivering value. It has been observed that up to 75% of overall project effort is now spent on software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

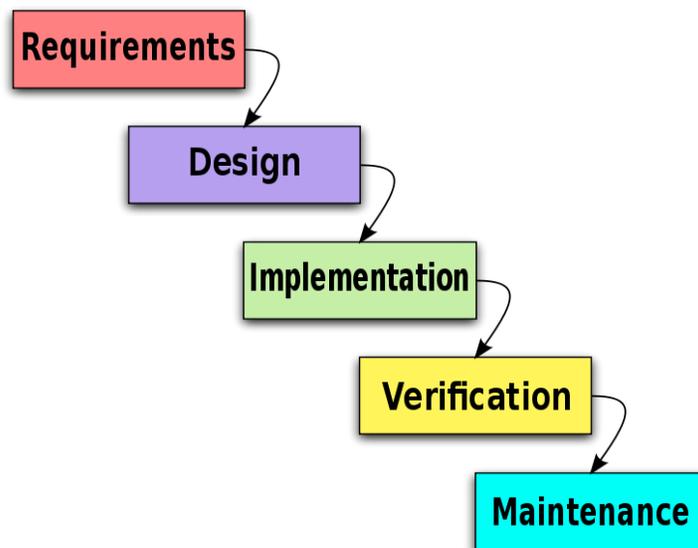
Chapter-4

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

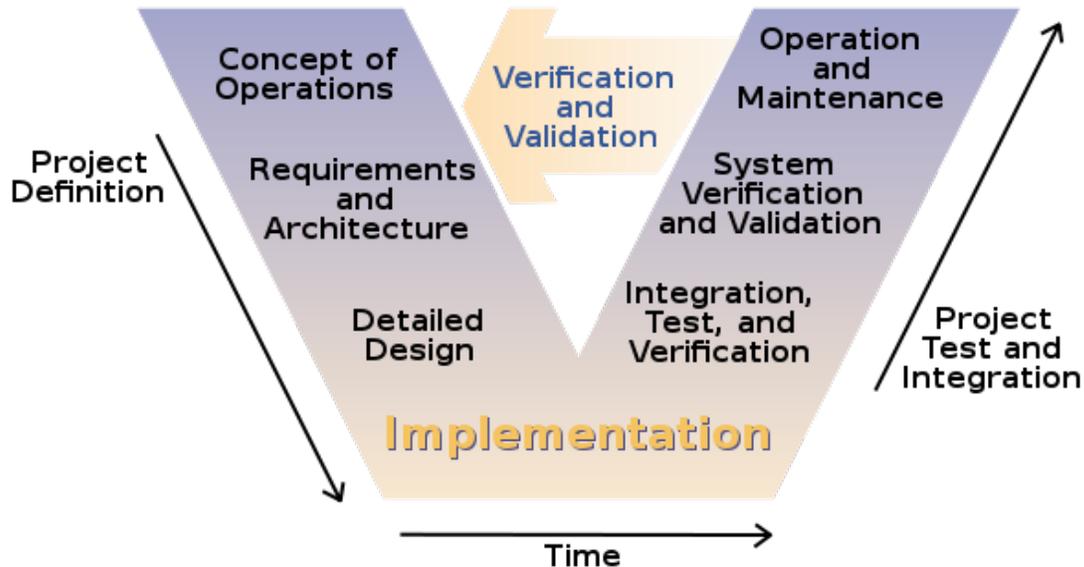


The unmodified "waterfall model". Progress flows from the top to the bottom, like a waterfall.

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

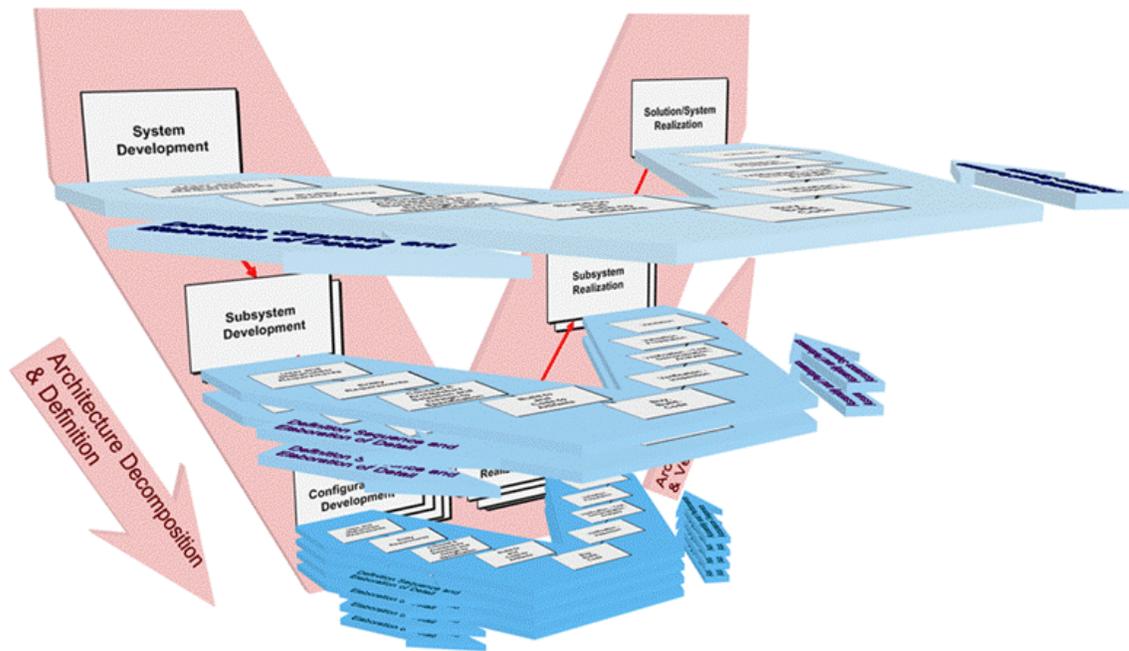


The V-model of the Systems Engineering Process.

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



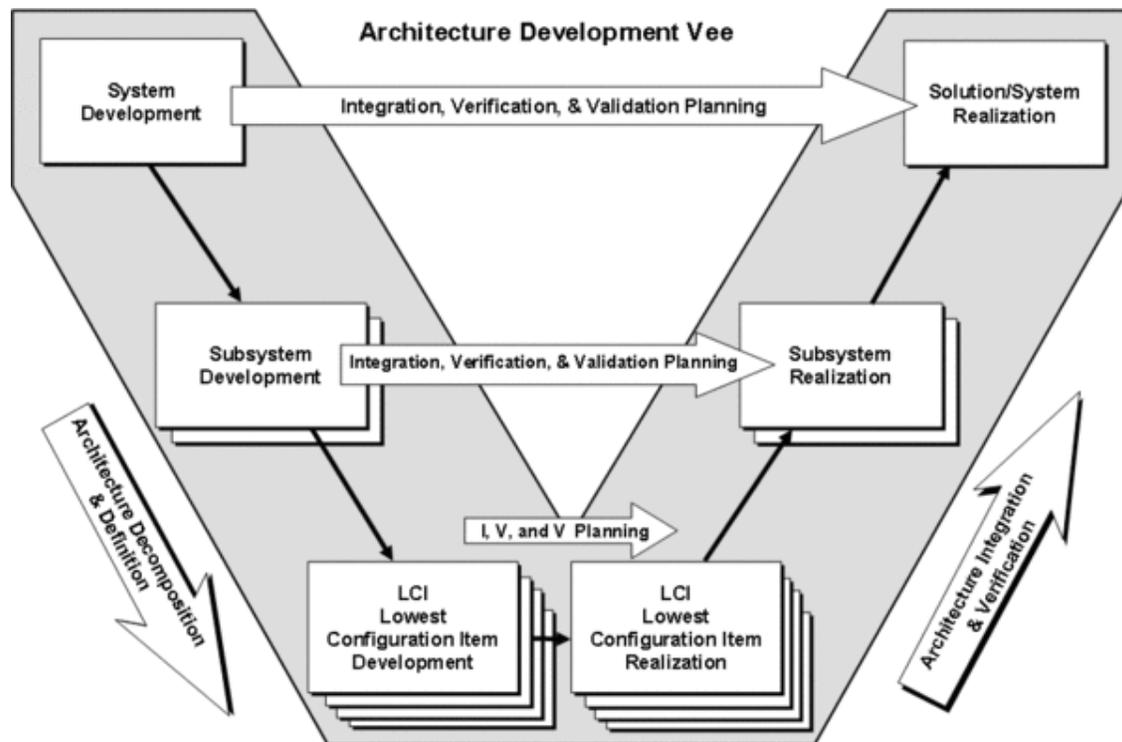
Architecture and Entity Vees Intersecting. Source - Kevin Forsberg and Hal Mooz 2006.

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

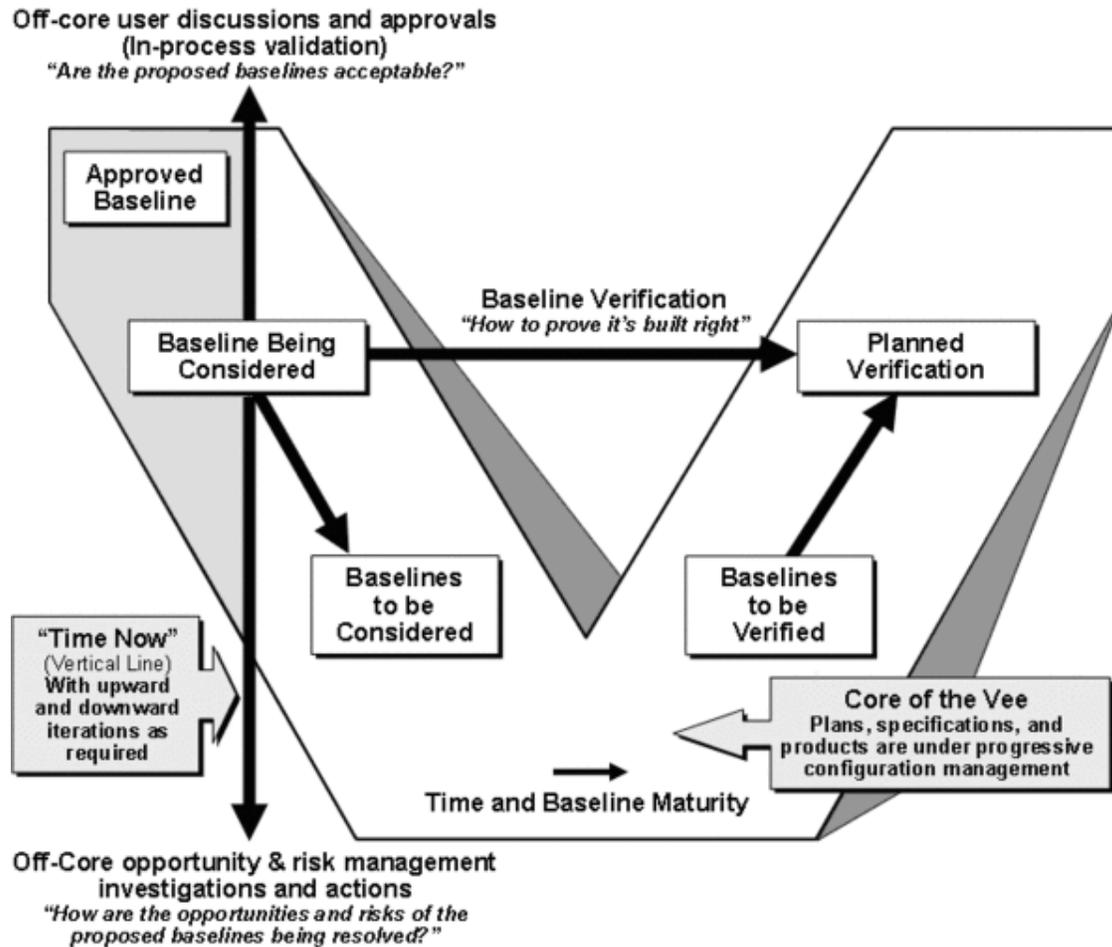
When developing complicated systems, a system engineer must manage a system baseline configuration from start to finish. The baseline can include design documents, user manuals, the product itself, and should answer every What?, Why?, and Who? for a system's architecture. At each development phase, there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

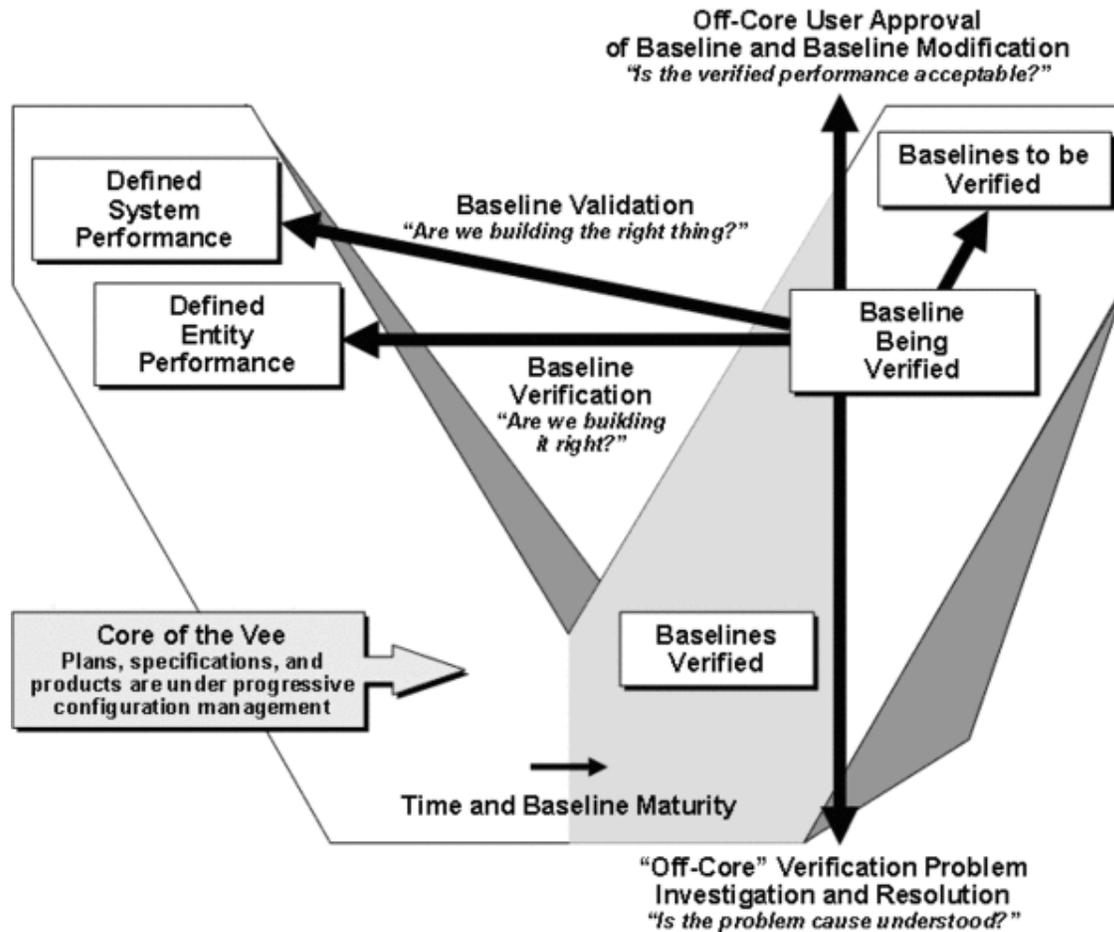
Downward off-Vee core investigations (figure - below) facilitate gaining knowledge to justify architecture baseline decisions made on the Vee core. Upward off core communication with customers and users facilitates in-process validation keeping the stakeholders abreast of and committed to the evolving 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.



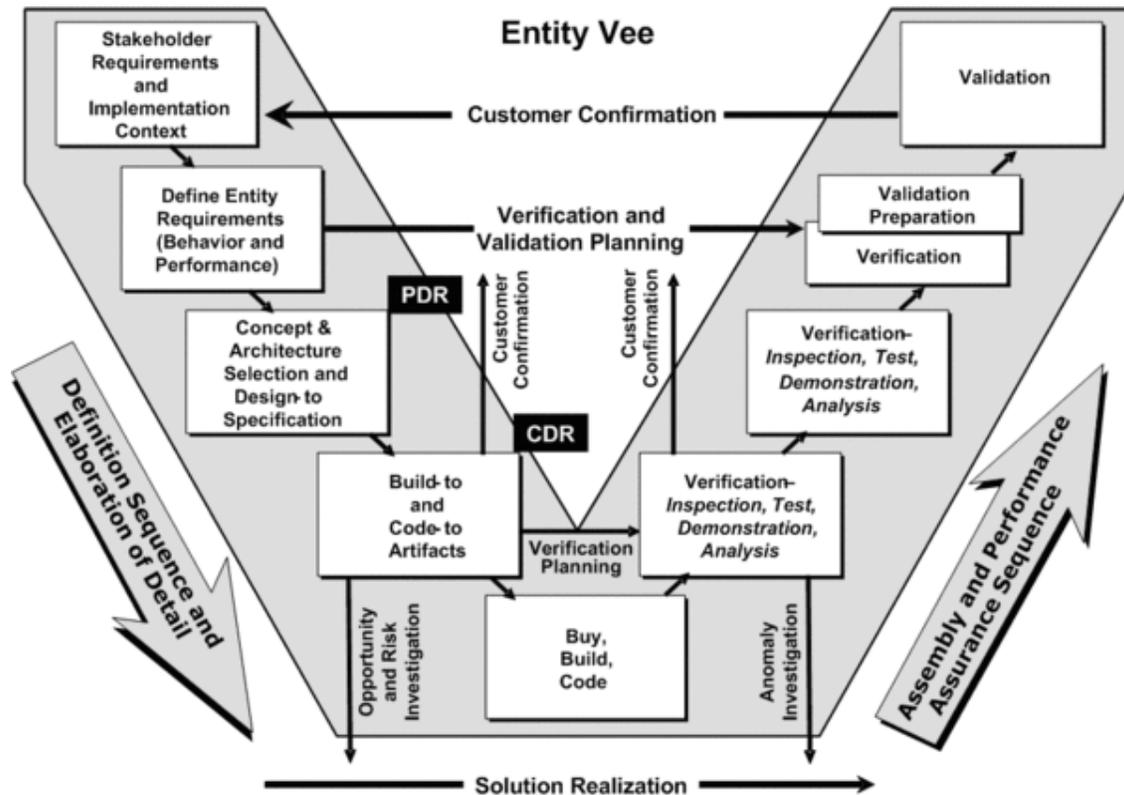
Vee Model - Integration and Verification. Source - Kevin Forsberg and Hal Mooz 2006.

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.

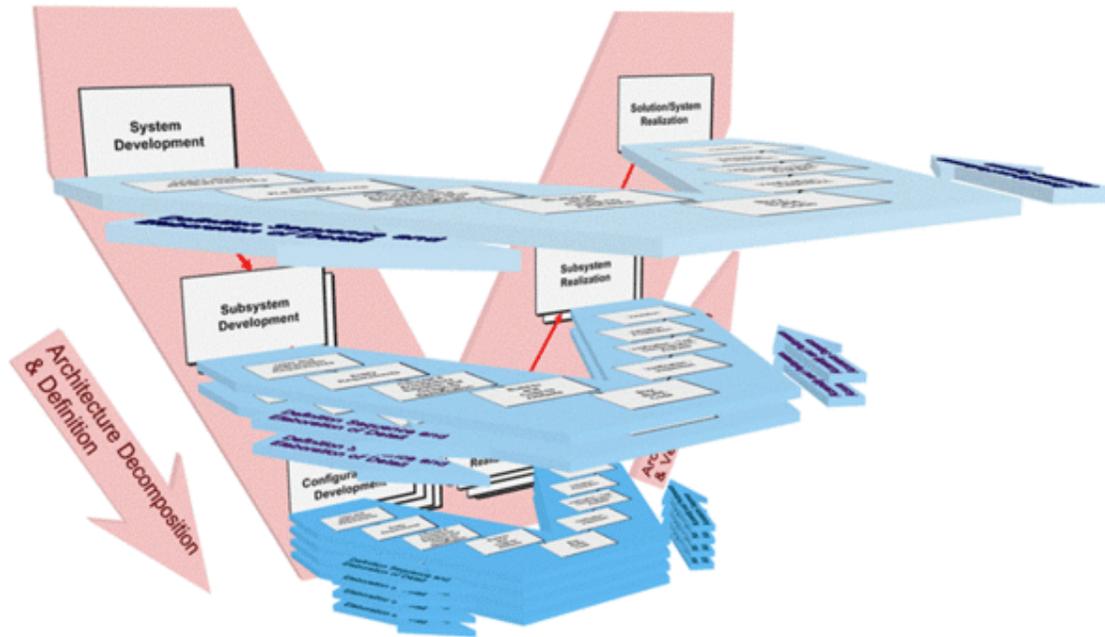
The vertical dimension of the Entity Vee is baseline elaboration at the selected architecture level and the core of the Entity Vee represents entity baseline elaboration progression. Also included (similar to the Architecture Vee) are the activities associated with opportunity and risk management, pursued 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

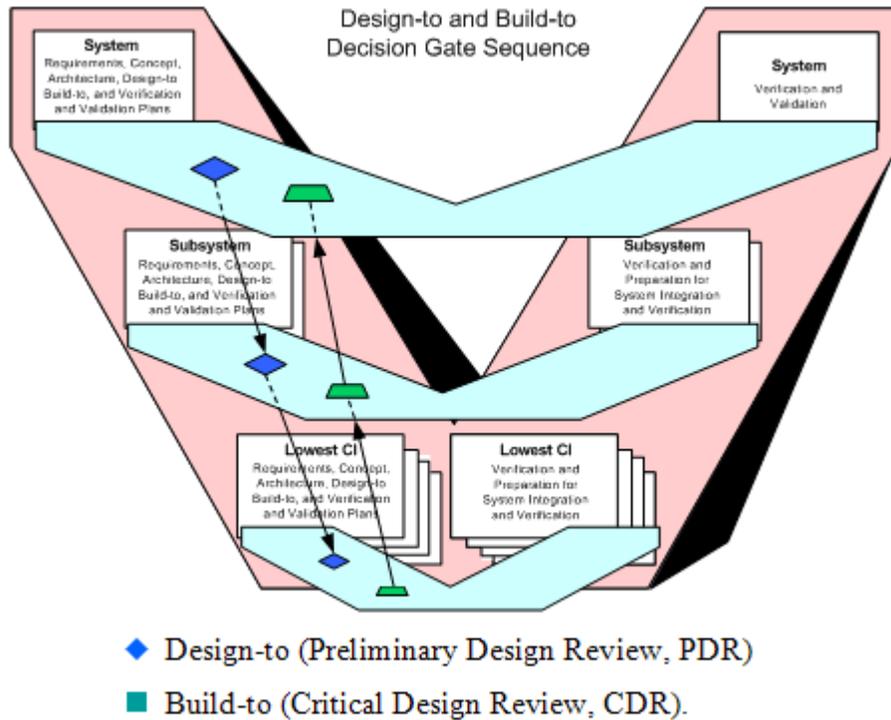
To evolve user needs into a system that satisfies those needs requires a best value solution for every entity of the architecture. This can be visualized by positioning Entity Vees orthogonal to the Architecture Vee as shown in the figure below. For each entity of the Architecture Vee there is a corresponding 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-5

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 Verdex, 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:

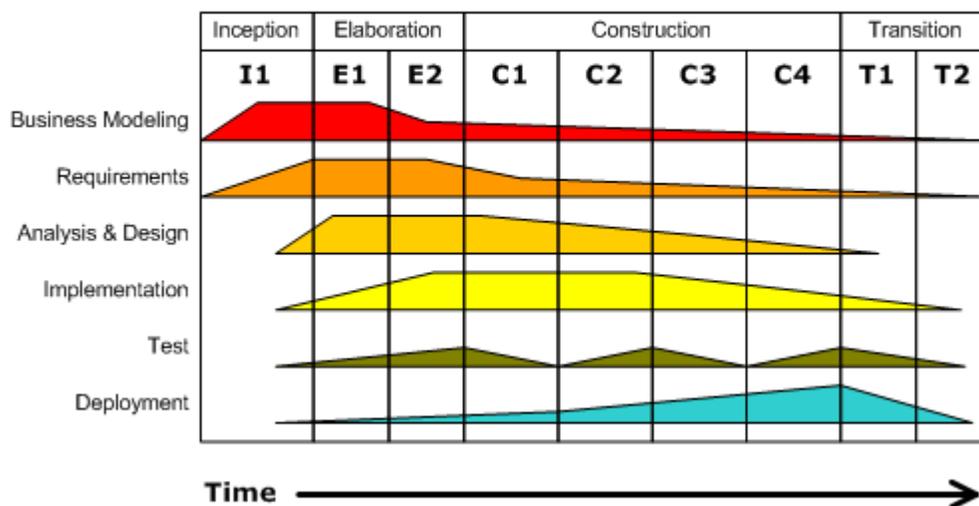
- Roles (who) – A Role defines a set of related skills, competencies and responsibilities.
- Work Products (what) – A Work Product represents something resulting from a task, including all the documents and models produced while working through the process.
- Tasks (how) – A Task describes a unit of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

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



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-6

Endeavour Software Project Management, Feature Creep and Issue Log

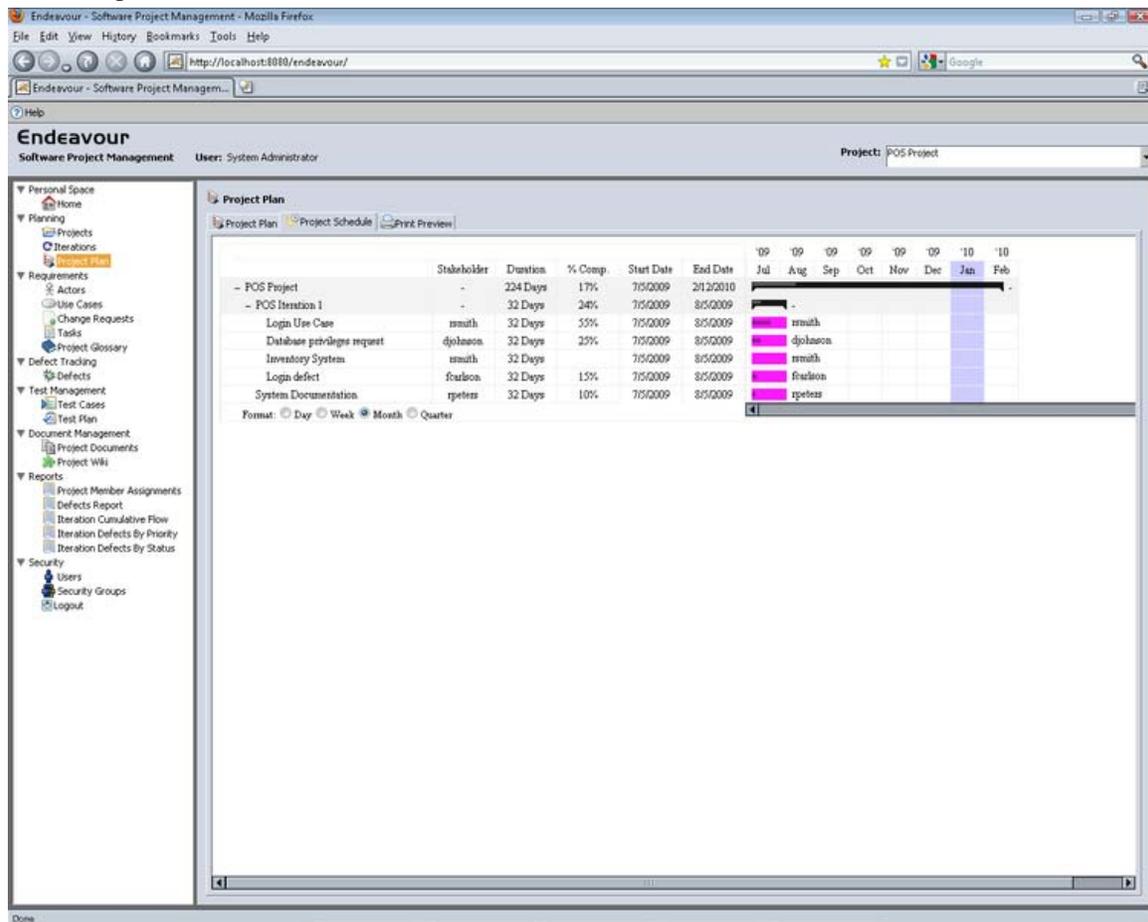
Endeavour Software Project Management

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

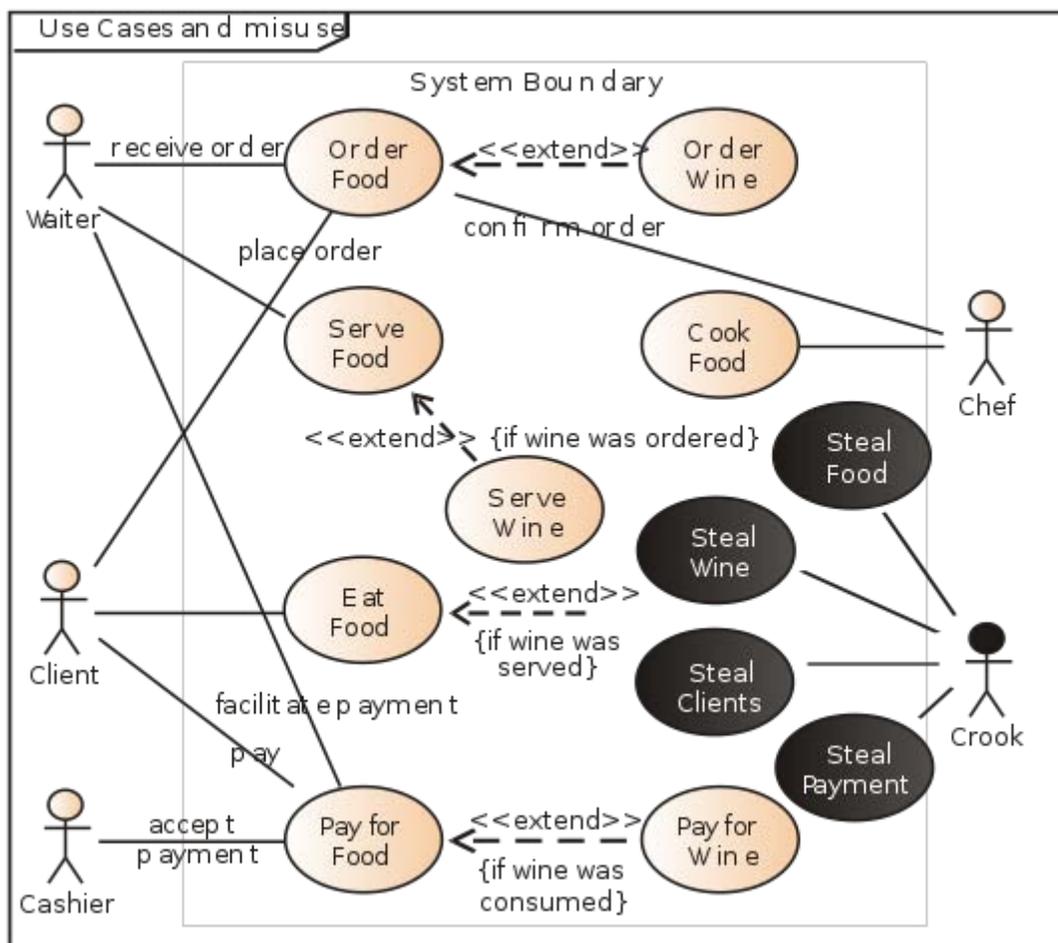
- **Date raised:** when the issue is raised.
- **Date assigned:** when the issue is assigned.
- **Deadline:** when is the final date to get the issue settled.
- **Date resolved:** when the issue is actually solved.

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-7

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 become 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 given 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-8

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:

$$MRE = |\text{actual effort} - \text{estimated effort}| / |\text{actual effort}|$$

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 **Minimum Usable SubseT**.

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 "Would 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.

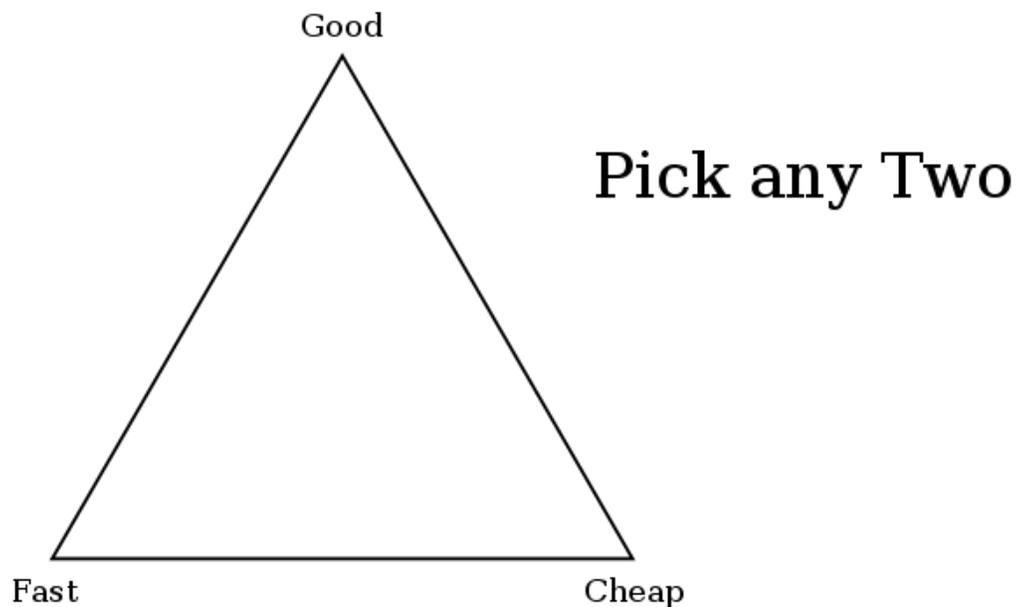
Chapter-9

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.

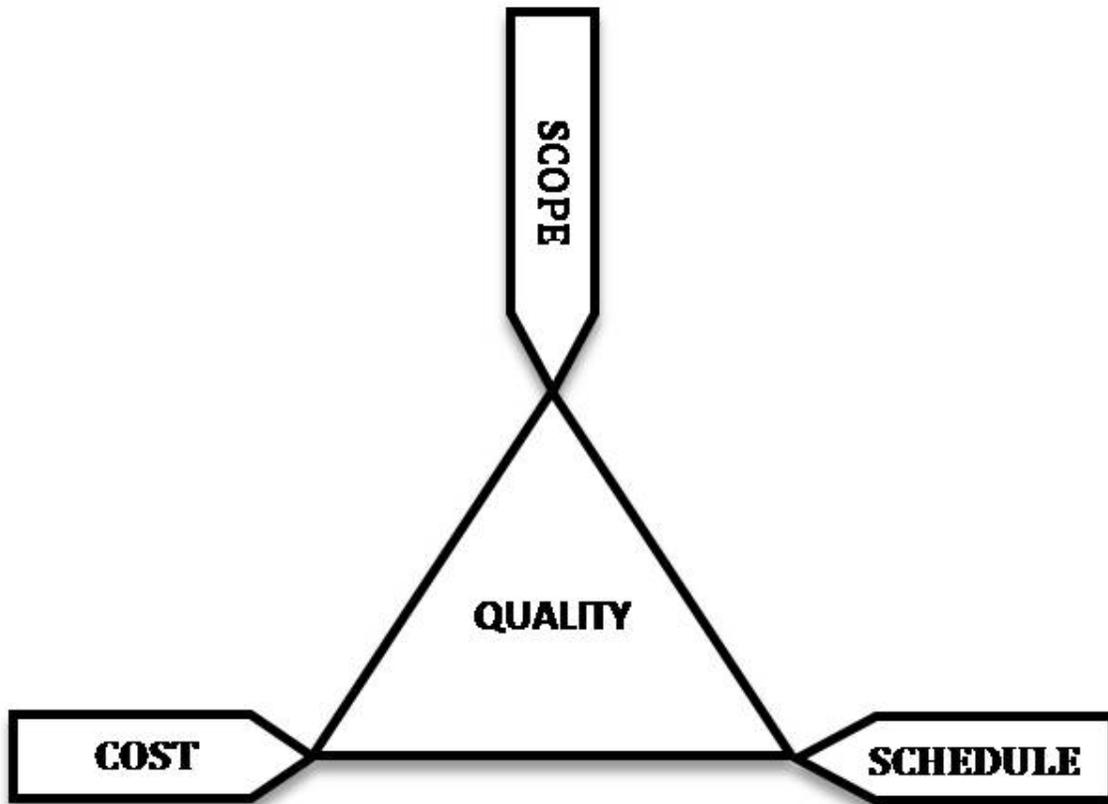
This constraint may apply to creative human activity, such as software, movies, books etc where quality is subjective. This is a false dichotomy when comparing technologies such as digital to analog in media formats.

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-10

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-11

Agile Software Development

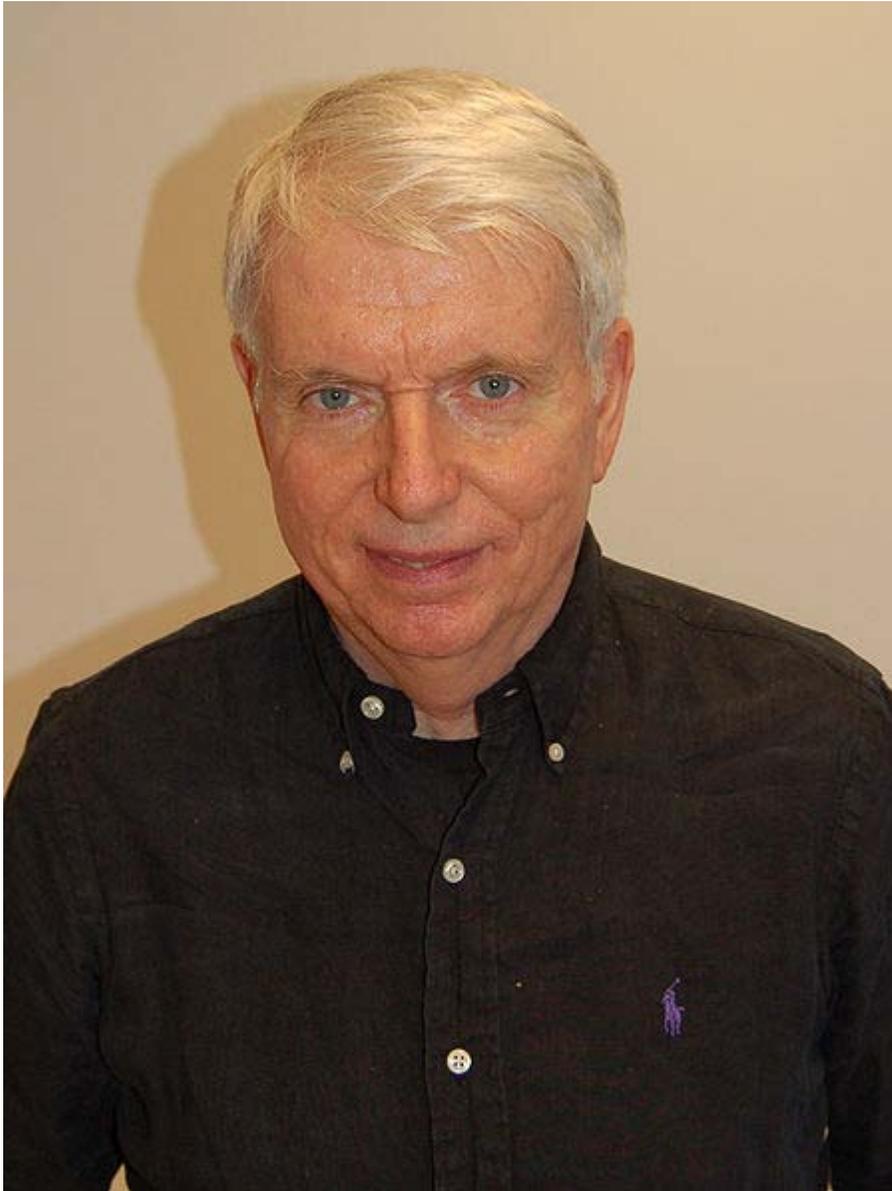


Agile software development poster

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.

The meanings of the Manifesto items on the left within the agile software development context are described below.

Individuals and Interactions – in agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.

Working software – working software will be more useful and welcome than just presenting documents to clients in meetings.

Customer collaboration – requirements cannot be fully collected at the beginning of the software development cycle, therefore continuous customer or stakeholder involvement is very important.

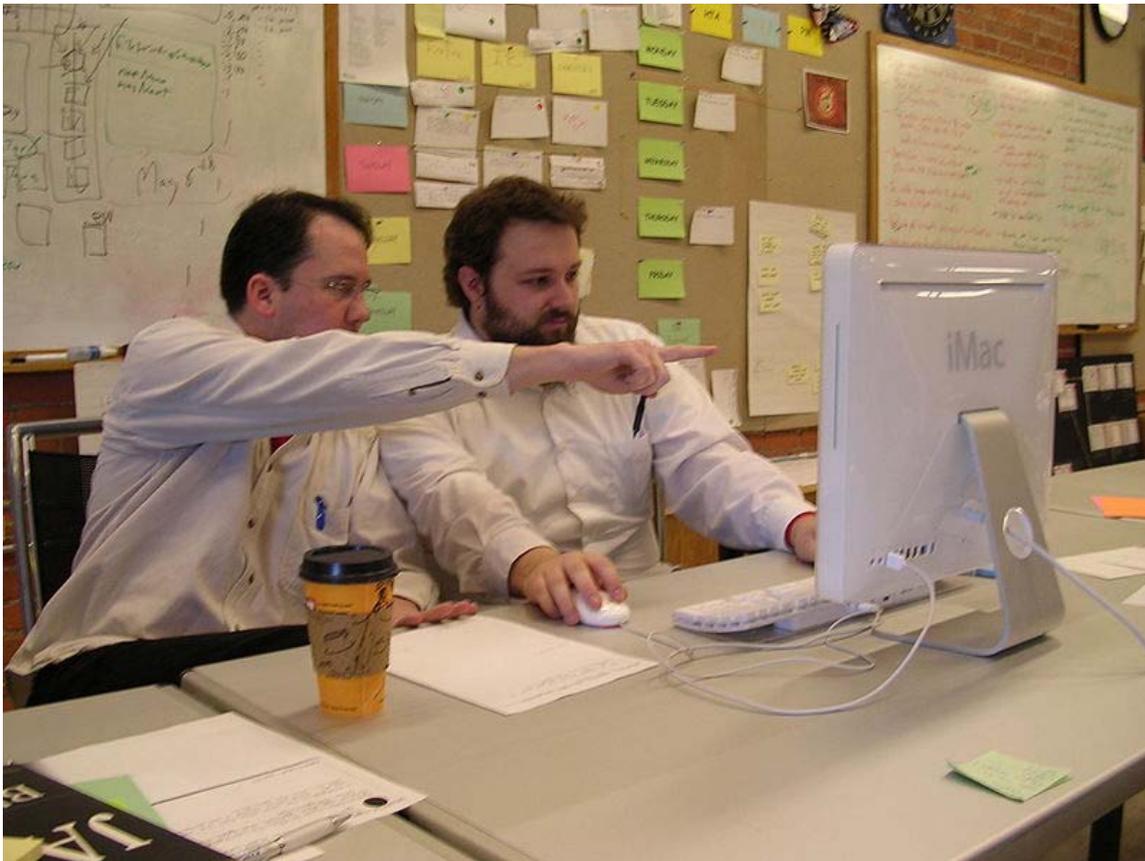
Responding to change – agile development is focused on quick responses to change and continuous development.

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.

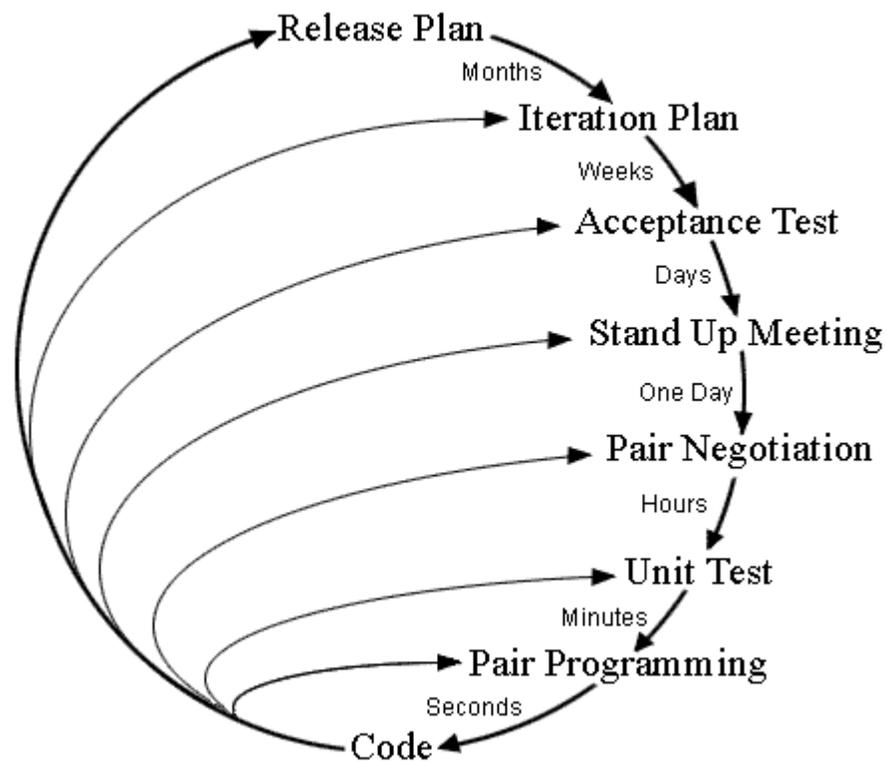
As of 2006, experience reports have been or will be presented at the following conferences:

- 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-12

Extreme Programming

Planning/Feedback Loops



Planning and feedback loops in Extreme Programming.

Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles (timeboxing), which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels, on the theory that if some is good, more is better.

Critics have noted several potential drawbacks, including problems with unstable requirements, no documented compromises of user conflicts, and a lack of an overall design specification or document.

History

Extreme Programming was created by Kent Beck during his work on the Chrysler Comprehensive Compensation System (C3) payroll project. Beck became the C3 project leader in March 1996 and began to refine the development method used in the project and wrote a book on the method (in October 1999, *Extreme Programming Explained* was published). Chrysler cancelled the C3 project in February 2000, after the company was acquired by Daimler-Benz.

Although extreme programming itself is relatively new, many of its practices have been around for some time; the methodology, after all, takes "best practices" to extreme levels. For example, the "practice of test-first development, planning and writing tests before each micro-increment" was used as early as NASA's Project Mercury, in the early 1960s (Larman 2003). To shorten the total development time, some formal test documents (such as for acceptance testing) have been developed in parallel (or shortly before) the software is ready for testing. A NASA independent test group can write the test procedures, based on formal requirements and logical limits, before the software has been written and integrated with the hardware. In XP, this concept is taken to the extreme level by writing automated tests (perhaps inside of software modules) which validate the operation of even small sections of software coding, rather than only testing the larger features. Some other XP practices, such as refactoring, modularity, bottom-up design, and incremental design were described by Leo Brodie in his book published in 1984.

Origins

Software development in the 1990s was shaped by two major influences: internally, object-oriented programming replaced procedural programming as the programming paradigm favored by some in the industry; externally, the rise of the Internet and the dot-com boom emphasized speed-to-market and company-growth as competitive business factors. Rapidly-changing requirements demanded shorter product life-cycles, and were often incompatible with traditional methods of software development.

The Chrysler Comprehensive Compensation System was started in order to determine the best way to use object technologies, using the payroll systems at Chrysler as the object of research, with Smalltalk as the language and GemStone as the data access layer. They brought in Kent Beck, a prominent Smalltalk practitioner, to do performance tuning on the system, but his role expanded as he noted several problems they were having with their development process. He took this opportunity to propose and implement some changes in their practices based on his work with his frequent collaborator, Ward Cunningham. Beck describes the early conception of the methods:

The first time I was asked to lead a team, I asked them to do a little bit of the things I thought were sensible, like testing and reviews. The second time there was a lot more on the line. I thought, "Damn the torpedoes, at least this will make a good article," [and] asked the team to crank up all the knobs to 10 on the things I thought were essential and leave out everything else.

Beck invited Ron Jeffries to the project to help develop and refine these methods. Jeffries thereafter acted as a coach to instill the practices as habits in the C3 team.

Beck edited a series of books on XP, beginning with his own *Extreme Programming Explained*, spreading his ideas to a much larger, yet very receptive, audience. Authors in the series went through various aspects attending XP and its practices. Even a book was written, critical of the practices.

Current state

XP created quite a buzz in the late 1990s and early 2000s, seeing adoption in a number of environments radically different from its origins.

The high discipline required by the original practices often went by the wayside, causing some of these practices, such as those thought too rigid, to be deprecated or reduced, or even left unfinished, on individual sites. For example, the practice of end-of-day integration tests, for a particular project, could be changed to an end-of-week schedule, or simply reduced to mutually agreed dates. Such a more relaxed schedule could avoid people feeling rushed to generate artificial stubs just to pass the end-of-day testing. A less rigid schedule allows, instead, for some complex features to be more fully developed over a several-day period. However, some level of periodic integration testing can detect groups of people working in non-compatible, tangent efforts before too much work is invested in divergent, wrong directions.

Meanwhile, other agile development practices have not stood still, and XP is still evolving, assimilating more lessons from experiences in the field, to use other practices. In the second edition of *Extreme Programming Explained*, Beck added more values and practices and differentiated between primary and corollary practices.

Concept

Goals

Extreme Programming Explained describes Extreme Programming as a software development discipline that organizes people to produce higher quality software more productively.

XP attempts to reduce the cost of changes in requirements by having multiple short development cycles, rather than one long. In this doctrine changes are a natural, inescapable and desirable aspect of software development projects, and should be planned for instead of attempting to define a stable set of requirements.

Extreme programming also introduces a number of basic values, principles and practices on top of the agile programming framework.

Activities

XP describes four basic activities that are performed within the software development process: coding, testing, listening, and designing. Each of those activities is described below.

Coding

The advocates of XP argue that the only truly important product of the system development process is code - software instructions a computer can interpret. Without code, there is no working product.

Coding can also be used to figure out the most suitable solution. Coding can also help to communicate thoughts about programming problems. A programmer dealing with a complex programming problem and finding it hard to explain the solution to fellow programmers might code it and use the code to demonstrate what he or she means. Code, say the proponents of this position, is always clear and concise and cannot be interpreted in more than one way. Other programmers can give feedback on this code by also coding their thoughts.

Testing

Extreme programming's approach is that if a little testing can eliminate a few flaws, a lot of testing can eliminate many more flaws.

- Unit tests determine whether a given feature works as intended. A programmer writes as many automated tests as they can think of that might "break" the code; if all tests run successfully, then the coding is complete. Every piece of code that is written is tested before moving on to the next feature.

- Acceptance tests verify that the requirements as understood by the programmers satisfy the customer's actual requirements. These occur in the exploration phase of release planning.

A "testathon" is an event when programmers meet to do collaborative test writing, a kind of brainstorming relative to software testing.

Listening

Programmers must listen to what the customers need the system to do, what "business logic" is needed. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved. Communication between the customer and programmer is further addressed in the Planning Game.

Designing

From the point of view of simplicity, of course one could say that system development doesn't need more than coding, testing and listening. If those activities are performed well, the result should always be a system that works. In practice, this will not work. One can come a long way without designing but at a given time one will get stuck. The system becomes too complex and the dependencies within the system cease to be clear. One can avoid this by creating a design structure that organizes the logic in the system. Good design will avoid lots of dependencies within a system; this means that changing one part of the system will not affect other parts of the system.

Values

Extreme Programming initially recognized four values in 1999. A new value was added in the second edition of *Extreme Programming Explained*. The five values are:

Communication

Building software systems requires communicating system requirements to the developers of the system. In formal software development methodologies, this task is accomplished through documentation. Extreme programming techniques can be viewed as methods for rapidly building and disseminating institutional knowledge among members of a development team. The goal is to give all developers a shared view of the system which matches the view held by the users of the system. To this end, extreme programming favors simple designs, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.

Simplicity

Extreme Programming encourages starting with the simplest solution. Extra functionality can then be added later. The difference between this approach and more conventional

system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month. This is sometimes summed up as the "you ain't gonna need it" (YAGNI) approach. Proponents of XP acknowledge the disadvantage that this can sometimes entail more effort tomorrow to change the system; their claim is that this is more than compensated for by the advantage of not investing in possible future requirements that might change before they become relevant. Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed. Related to the "communication" value, simplicity in design and coding should improve the quality of communication. A simple design with very simple code could be easily understood by most programmers in the team.

Feedback

Within extreme programming, feedback relates to different dimensions of the system development:

- Feedback from the system: by writing unit tests, or running periodic integration tests, the programmers have direct feedback from the state of the system after implementing changes.
- Feedback from the customer: The functional tests (aka acceptance tests) are written by the customer and the testers. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks so the customer can easily steer the development.
- Feedback from the team: When customers come up with new requirements in the planning game the team directly gives an estimation of the time that it will take to implement.

Feedback is closely related to communication and simplicity. Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will break. The direct feedback from the system tells programmers to recode this part. A customer is able to test the system periodically according to the functional requirements, known as *user stories*. To quote Kent Beck, "Optimism is an occupational hazard of programming, feedback is the treatment."

Courage

Several practices embody courage. One is the commandment to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in design and requiring a lot of effort to implement anything else. Courage enables developers to feel comfortable with refactoring their code when necessary. This means reviewing the existing system and modifying it so that future changes can be implemented more easily. Another example of courage is knowing when to throw code away: courage to remove source code that is obsolete, no matter how much effort was used to create that source code. Also, courage means persistence: A programmer might be stuck on a complex problem for an entire day, then solve the problem quickly the next day, if only they are persistent.

Respect

The respect value includes respect for others as well as self-respect. Programmers should never commit changes that break compilation, that make existing unit-tests fail, or that otherwise delay the work of their peers. Members respect their own work by always striving for high quality and seeking for the best design for the solution at hand through refactoring.

Adopting the four earlier values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored. This ensures a high level of motivation and encourages loyalty toward the team and toward the goal of the project. This value is very dependent upon the other values, and is very much oriented toward people in a team.

Rules

The first version of rules for XP was published in 1999 by Don Wells at the XP website. 29 rules are given in the categories of planning, managing, designing, coding, and testing. Planning, managing and designing are called out explicitly to counter claims that XP doesn't support those activities.

Another version of XP rules was proposed by Ken Auer in XP/Agile Universe 2003. He felt XP was defined by its rules, not its practices (which are subject to more variation and ambiguity). He defined two categories: "Rules of Engagement" which dictate the environment in which software development can take place effectively, and "Rules of Play" which define the minute-by-minute activities and rules within the framework of the Rules of Engagement.

Principles

The principles that form the basis of XP are based on the values just described and are intended to foster decisions in a system development project. The principles are intended to be more concrete than the values and more easily translated to guidance in a practical situation.

Feedback

Extreme programming sees feedback as most useful if it is done rapidly and expresses that the time between an action and its feedback is critical to learning and making changes. Unlike traditional system development methods, contact with the customer occurs in more frequent iterations. The customer has clear insight into the system that is being developed. He or she can give feedback and steer the development as needed.

Unit tests also contribute to the rapid feedback principle. When writing code, the unit test provides direct feedback as to how the system reacts to the changes one has made. If, for instance, the changes affect a part of the system that is not in the scope of the

programmer who made them, that programmer will not notice the flaw. There is a large chance that this bug will appear when the system is in production.

Assuming simplicity

This is about treating every problem as if its solution were "extremely simple". Traditional system development methods say to plan for the future and to code for reusability. Extreme programming rejects these ideas.

The advocates of extreme programming say that making big changes all at once does not work. Extreme programming applies incremental changes: for example, a system might have small releases every three weeks. When many little steps are made, the customer has more control over the development process and the system that is being developed.

Embracing change

The principle of embracing change is about not working against changes but embracing them. For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.

Practices

Extreme programming has been described as having 12 practices, grouped into four areas:

Fine scale feedback

- Pair programming
- Planning game
- Test-driven development
- Whole team

Continuous process

- Continuous integration
- Refactoring or design improvement
- Small releases

Shared understanding

- Coding standards
- Collective code ownership
- Simple design
- System metaphor

Programmer welfare

- Sustainable pace

Coding

- The customer is always available
- Code the Unit test first
- Only one pair integrates code at a time
- Leave Optimization till last
- No Overtime

Testing

- All code must have Unit tests
- All code must pass all Unit tests before it can be released.
- When a Bug is found tests are created before the bug is addressed (a bug is not an error in logic, it is a test you forgot to write)
- Acceptance tests are run often and the results are published

Controversial aspects

The practices in XP have been heavily debated. Proponents of extreme programming claim that by having the on-site customer request changes informally, the process becomes flexible, and saves the cost of formal overhead. Critics of XP claim this can lead to costly rework and project scope creep beyond what was previously agreed or funded.

Change control boards are a sign that there are potential conflicts in project objectives and constraints between multiple users. XP's expedited methodology is somewhat dependent on programmers being able to assume a unified client viewpoint so the programmer can concentrate on coding rather than documentation of compromise objectives and constraints. This also applies when multiple programming organizations are involved, particularly organizations which compete for shares of projects.

Other potentially controversial aspects of extreme programming include:

- Requirements are expressed as automated acceptance tests rather than specification documents.
- Requirements are defined incrementally, rather than trying to get them all in advance.
- Software developers are usually required to work in pairs.
- There is no Big Design Up Front. Most of the design activity takes place on the fly and incrementally, starting with "the simplest thing that could possibly work" and adding complexity only when it's required by failing tests. Critics compare this to "debugging a system into appearance" and fear this will result in more re-design effort than only re-designing when requirements change.

- A customer representative is attached to the project. This role can become a single-point-of-failure for the project, and some people have found it to be a source of stress. Also, there is the danger of micro-management by a non-technical representative trying to dictate the use of technical software features and architecture.
- Dependence upon all other aspects of XP: "XP is like a ring of poisonous snakes, daisy-chained together. All it takes is for one of them to wriggle loose, and you've got a very angry, poisonous snake heading your way."

Scalability

Historically, XP only works on teams of twelve or fewer people. One way to circumvent this limitation is to break up the project into smaller pieces and the team into smaller groups. It has been claimed that XP has been used successfully on teams of over a hundred developers. ThoughtWorks has claimed reasonable success on distributed XP projects with up to sixty people.

In 2004 Industrial Extreme Programming (IXP) was introduced as an evolution of XP. It is intended to bring the ability to work in large and distributed teams. It now has 23 practices and flexible values. As it is a new member of the Agile family, there is not enough data to prove its usability, however it claims to be an answer to what it sees as XP's imperfections.

Severability and responses

In 2003, Matt Stephens and Doug Rosenberg published *Extreme Programming Refactored: The Case Against XP* which questioned the value of the XP process and suggested ways in which it could be improved. This triggered a lengthy debate in articles, internet newsgroups, and web-site chat areas. The core argument of the book is that XP's practices are interdependent but that few practical organizations are willing/able to adopt all the practices; therefore the entire process fails. The book also makes other criticisms and it draws a likeness of XP's "collective ownership" model to socialism in a negative manner.

Certain aspects of XP have changed since the book *Extreme Programming Refactored* (2003) was published; in particular, XP now accommodates modifications to the practices as long as the required objectives are still met. XP also uses increasingly generic terms for processes. Some argue that these changes invalidate previous criticisms; others claim that this is simply watering the process down.

RDP Practice is a technique for tailoring extreme programming. This practice was initially proposed as a long research paper in a workshop organized by Philippe Kruchten and Steve Adolph and yet it is the only proposed and applicable method for customizing XP. The valuable concepts behind RDP practice, in a short time provided the rationale for applicability of it in industries. RDP Practice tries to customize XP by relying on technique XP Rules.

Other authors have tried to reconcile XP with the older methods in order to form a unified methodology. Some of these XP sought to replace, such as the waterfall method; example: Project Lifecycles: Waterfall, Rapid Application Development, and All That. JPMorgan Chase & Co. tried combining XP with the computer programming methodologies of Capability Maturity Model Integration (CMMI), and Six Sigma. They found that the three systems reinforced each other well, leading to better development, and did not mutually contradict.

Criticism

Extreme programming's initial buzz and controversial tenets, such as pair programming and continuous design, have attracted particular criticisms, such as the ones coming from McBreen and Boehm and Turner. Many of the criticisms, however, are believed by Agile practitioners to be misunderstandings of agile development.

In particular, extreme programming is reviewed and critiqued by Matt Stephens's and Doug Rosenberg's *Extreme Programming Refactored*.

Criticisms include:

- A methodology is only as effective as the people involved, Agile does not solve this
- Often used as a means to bleed money from customers through lack of defining a deliverable
- Lack of structure and necessary documentation
- Only works with senior-level developers
- Incorporates insufficient software design
- Requires meetings at frequent intervals at enormous expense to customers
- Requires too much cultural change to adopt
- Can lead to more difficult contractual negotiations
- Can be very inefficient—if the requirements for one area of code change through various iterations, the same programming may need to be done several times over. Whereas if a plan were there to be followed, a single area of code is expected to be written once.
- Impossible to develop realistic estimates of work effort needed to provide a quote, because at the beginning of the project no one knows the entire scope/requirements
- Can increase the risk of scope creep due to the lack of detailed requirements documentation
- Agile is feature driven; non-functional quality attributes are hard to be placed as user stories

Chapter-13

Applied Agile Software Development



AASD logo

Agile Software Development (ASD) is a set of principles; **Applied Agile Software Development** (AASD) is one of the choices for making ASD work. AASD is a very tangible set of procedures to develop software in a mature and efficient way, based on the principles of ASD. AASD is strongly based on Scrum. AASD can almost be taken as a method that inherits from Scrum, but there's no formal compromise to use the Scrum procedures the exact way it was registered.

The main principles of Applied Agile Software Development's procedures are

- AASD procedures and methods may copy good practices from wherever it is legally possible. That's why AASD is similar to Scrum.
- AASD procedures should be very applicable, truly "out of the trenches"; there should never exist a procedure that is "good in theory, but can't be used in practice".

- AASD procedures should mention all needed software tools (such as SubVersion, Eclipse, etc.), and its versions.
- AASD emphasizes the use FOSS tools, but that's not an imposition of the method. That is:

COTS tools are not forbidden. The most important goal is to have the method to be really applicable, useful and productive, that is actually used in real firms and produce successful results.

- Since the technology market changes constantly, so is AASD.

The goal is to pursue the excellence in Software Development, proven with practice in real world.

Scope

AASD can be applied theoretically to any computer language. Most examples here will be given to Java, and C++.

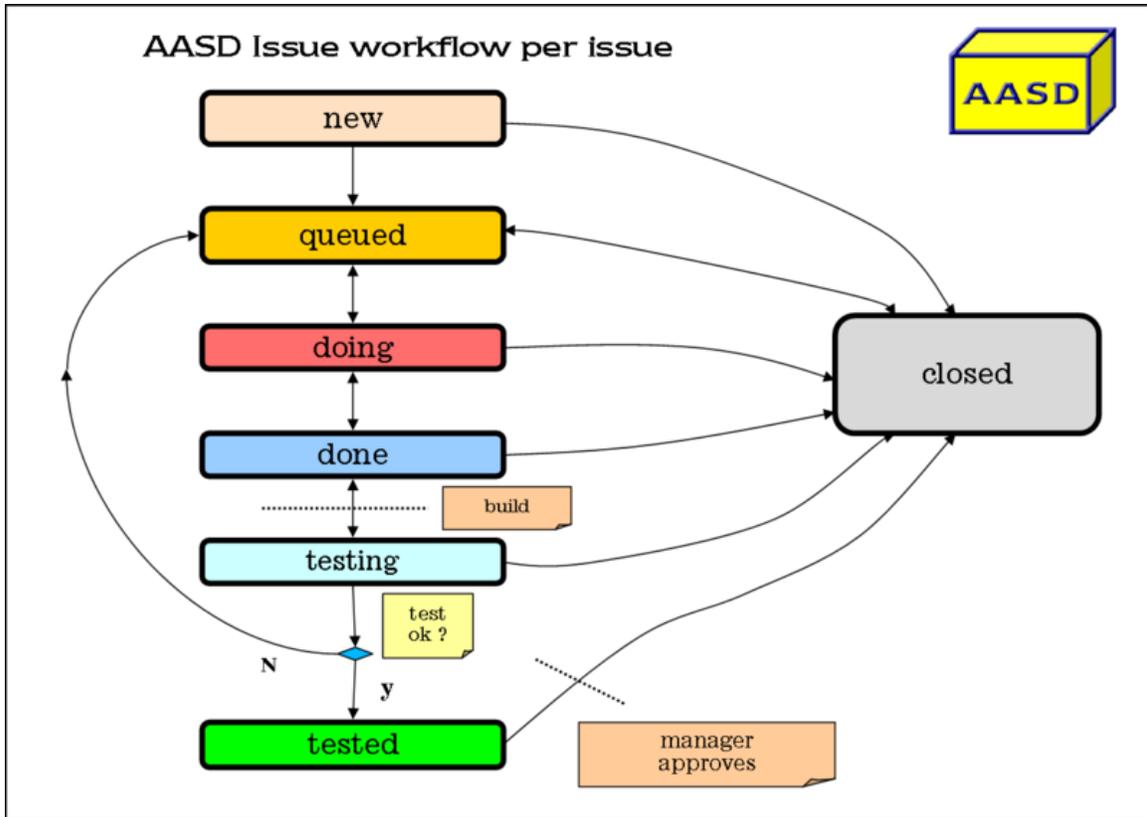
AASD is mainly conceived as a method for software development. But it should be remarked that it is viable as a method for a non-software project management, specially the tool for "Task Workflow".

Task Workflow

The concept of "task" and "issue" are equivalent. The description of the task workflow is described below.

- An issue is created in state "new".
- The manager makes the issue to go to "queued" state.
- When the developer actually starts to code, he/she should change the state to "doing".
- When the developer finishes the issue, the state should be changed to "done" and assign the issue to the tester; this state has also the meaning of "test queue".
- The tester sees the issues in "done" state as work assigned to them, much like the developers look to states in "queue" assigned to them. The testers do the test and accept or reject. Accepted tests go to "tested" state, while non accepted states go back to "queued".
- The "tested" issues remain in this state until the manager changes them to "closed".

"closed" issues are the ones out of focus, that remain existing for documentation purposes; "closed" issues can be reopened by the manager if some new demand happens related to the issue.



Software Tools

- Source Code Server (repository) SubVersion. Other software of the same category: CVS, Git, Mercurial, Darcs, Bazaar, StarTeam, Perforce, BitKeeper, Visual SourceSafe.
- Software Development State Workflow (also known as bugtrack \ system) Mantis. Other software of the same category: Bugzilla, Redmine, Trac, DotProject.
- IDE (Recommended: Eclipse, other options: NetBeans)
- Coding Standards Auto-Critic
- BuildID Stamper

Implementing Task Workflow with Mantis

You can use Mantis and customize it to easily implement the AASD workflow. Mantis is a bug track software, open source, written for web, using php language. The example below uses Mantis version 1.2.4. Follow the steps below.

- Have some computer configured with apache web server and php.
- Download Mantis 1.2.4.
- Create file config_inc.php with contents below.

<?php

```
$g_hostname = 'localhost';
$g_db_type = 'mysql';
$g_database_name = '...';
$g_db_username = '...';
$g_db_password = '...';
$g_bug_submit_status = 1;
$g_status_colors['new'] = '#ffE0C1';
$g_status_colors['feedback'] = '#ffCC00'; // queued
$g_status_colors['acknowledged'] = '#ff6d6d'; // doing
$g_status_colors['confirmed'] = '#99ccff'; // done
$g_status_colors['assigned'] = '#ccffff'; // testing
$g_status_colors['resolved'] = '#00ff00'; // tested
$g_status_colors['closed'] = '#d9d9d9';
?>
```

- Create file `custom_strings_inc.php` with contents below.

```
<?php
$s_status_enum_string =
'10:new,20:queued,30:doing,40:done,50:testing,80:tested,90:closed';
?>
```

- Edit file `bug_report.php`. Find line below

```
$t_bug_data->status = config_get( 'bug_submit_status' );
```

and change to line below.

```
$t_bug_data->status = NEW_;
```

If needed, change the default timezone by adding line below to `core.php`

```
# added to fix default time zone
date_default_timezone_set('America/Noronha');
```

Chapter-14

Agilo for Scrum

Agilo for Scrum is an open source, web-based Scrum tool to support the agile Scrum software development process. Agilo is based on Trac, a widely used Issue tracking system. It is programmed in Python and is distributed under the Apache Software License 2.0.

Its development was started in January 2007 by Andrea Tomasini, the first public version was released in January 2008.

Agilo is used in agile software development projects in all economic sectors who use the *Scrum* methodology. The python application can be downloaded and used either as source tarball, python-egg, SaaS, a VMWare Virtual appliance or a Windows Installer.

Version 0.8 is based on Trac 0.11.

Agilo supports Scrum-Teams, ScrumMasters and Product Owners in running and coordinating agile software development projects.

Features

- Interactive Whiteboard for distributed teams
- Real Ticket Typing: Agilo for Scrum adds to *Trac* custom ticket attributes that are only available for specific ticket types and can be configured freely. This enables Agilo for Scrum to track tickets, user stories, requirements and tasks on par to each other.
- Traceability: Agilo supports the defined *Scrum* process by linking all artefacts according to their semantic. A user story can be linked with a requirement it fulfills, a set of tasks and the related commits.
- Scrum Sprint artifacts: Agilo can maintain the Product Backlog, Sprint Backlog, Burn down chart and various statistics.
- Scrum Scaling Support: Agilo supports multiple teams working against the same product backlog, each team with their own Sprint.
- Custom Backlogs: Users can define backlogs with different scope (e.g. product, release, sprint)

Special features to support the standard *Scrum* roles:

For Product Owners

- Requirement Prioritization: Can use Business Value points to weight requirements and support the benefit/cost analysis.
- Traceability: Bi-directional links between *requirements*, *user stories*, *tasks*, *bugs* and *checkins* make the work transparent and support accurate estimation in the *Burn down chart* for all stakeholders.
- User Stories: Explicit support for *user stories* help keeping *Product Owners* focused on the outcome of features and away from implementation details which belong to the team to decide.

For Scrum Masters

- Autogenerated Scrum Artifacts: *Burn down chart* and project statistics don't need to be generated and updated by hand or in custom made spreadsheets.
- Configurable Capacity: Configurable default capacity (e.g. Friday always off) helps showing accurate prognostics and statistics.
- Custom Backlogs: If needed an impediment backlog can easily make problems visible and allows everybody to see when and how they are removed.
- Scrum Terminology Reference: Integrated knowledge base to explain the concepts and spread scrum-knowledge in the team.

For Team Members

- Progress Tracking: The Sprint Dashboard allows team members to get information about the project status, progress and health quickly on a day-to-day basis
- Self-Organization: Agilo helps breaking down *user stories* into work items (tasks) that can be assigned to team members and accurately estimated (usually two hours to two days) which gives very accurate project sprint estimations
- Information Radiation: Team members can update the state of their tasks before the daily standup so they can concentrate there on exchanging information and not about who did what.
- Subversion Integration: Day-to-day tasks such as updating task status can be done direct in the svn commit message and without context switches to different tools
- Adaptable: Agilo is flexible; you can adapt it to fit your unique team workflow

Chapter-15

Feature Driven Development

Feature Driven Development (FDD) is an iterative and incremental software development process. It is one of a number of Agile methods for developing software and forms part of the Agile Alliance. FDD blends a number of industry-recognized best practices into a cohesive whole. These practices are all driven from a client-valued functionality (feature) perspective. Its main purpose is to deliver tangible, working software repeatedly in a timely manner.

History

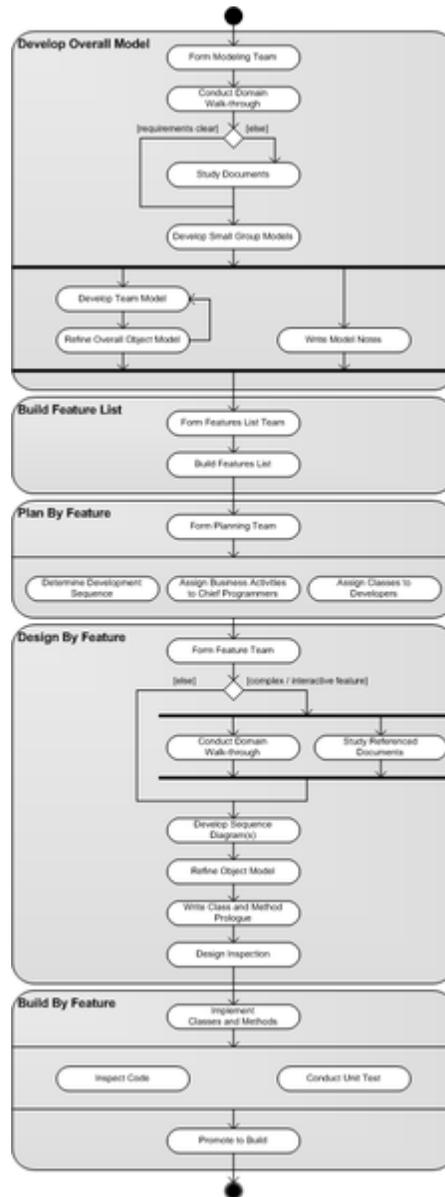
FDD was initially devised by Jeff De Luca to meet the specific needs of a 15 month, 50 person software development project at a large Singapore bank in 1997. Jeff De Luca delivered a set of five processes that covered the development of an overall model and the listing, planning, design and building of features. The first process is heavily influenced by Peter Coad's approach to object modeling. The second process incorporates Peter Coad's ideas of using a feature list to manage functional requirements and development tasks. The other processes and the blending of the processes into a cohesive whole is a result of Jeff De Luca's experience. Since its successful use on the Singapore project there have been several implementations of FDD.

The description of FDD was first introduced to the world in Chapter 6 of the book *Java Modeling in Color with UML* by Peter Coad, Eric Lefebvre and Jeff De Luca in 1999. In Stephen Palmer and Mac Felsing's book *A Practical Guide to Feature-Driven Development* (published in 2002) a more general description of FDD, decoupled from java modeling in color, is given.

The original and latest FDD processes can be found on Jeff De Luca's website under the 'Article' area. There is also a Community website available at which people can learn more about FDD, questions can be asked, and experiences and the processes themselves are discussed.

Overview

FDD is a model-driven short-iteration process that consists of five basic activities. For accurate state reporting and keeping track of the software development project, milestones that mark the progress made on each feature are defined. This section gives a high level overview of the activities.



Process model for FDD

Activities

FDD describes five basic activities that are within the software development process. In the figure on the right the meta-process model for these activities is displayed. During the

first three sequential activities an overall model shape is established. The final two activities are iterated for each feature. For more detailed information about the individual sub-activities have a look at Table 2 (derived from the process description in the 'Article' section of Jeff De Luca's website). The concepts involved in these activities are explained in Table 3.

Develop Overall Model

The project starts with a high-level walkthrough of the scope of the system and its context. Next, detailed domain walkthroughs are held for each modeling area. In support of each domain, walkthrough models are then composed by small groups which are presented for peer review and discussion. One of the proposed models or a merge of them is selected which becomes the model for that particular domain area. Domain area models are merged into an overall model, the overall model shape being adjusted along the way.

Build Feature List

The knowledge that is gathered during the initial modeling is used to identify a list of features. This is done by functionally decomposing the domain into subject areas. Subject areas each contain business activities, the steps within each business activity form the categorized feature list. Features in this respect are small pieces of client-valued functions expressed in the form <action> <result> <object>, for example: 'Calculate the total of a sale' or 'Validate the password of a user'. Features should not take more than two weeks to complete, else they should be broken down into smaller pieces.

Plan By Feature

Now that the feature list is complete, the next step is to produce the development plan. Class ownership is done by ordering and assigning features (or feature sets) as classes to chief programmers.

Design By Feature

A design package is produced for each feature. A chief programmer selects a small group of features that are to be developed within two weeks. Together with the corresponding class owners, the chief programmer works out detailed sequence diagrams for each feature and refines the overall model. Next, the class and method prologues are written and finally a design inspection is held.

Build By Feature

After a successful design inspection a per feature activity to produce a completed client-valued function (feature) is being produced. The class owners develop the actual code for their classes. After a unit test and a successful code inspection, the completed feature is promoted to the main build.

Milestones

Since features are small, completing a feature is a relatively small task. For accurate state reporting and keeping track of the software development project it is however important to mark the progress made on each feature. FDD therefore defines six milestones per feature that are to be completed sequentially. The first three milestones are completed during the Design By Feature activity, the last three are completed during the Build By Feature activity. To help with tracking progress, a percentage complete is assigned to each milestone. In the table below the milestones (and their completion percentage) are shown. A feature that is still being coded is 44% complete (Domain Walkthrough 1%, Design 40% and Design Inspection 3% = 44%).

Table 1: Milestones

| Domain Walkthrough | Design | Design Inspection | Code | Code Inspection | Promote To Build |
|---------------------------|---------------|--------------------------|-------------|------------------------|-------------------------|
| 1% | 40% | 3% | 45% | 10% | 1% |

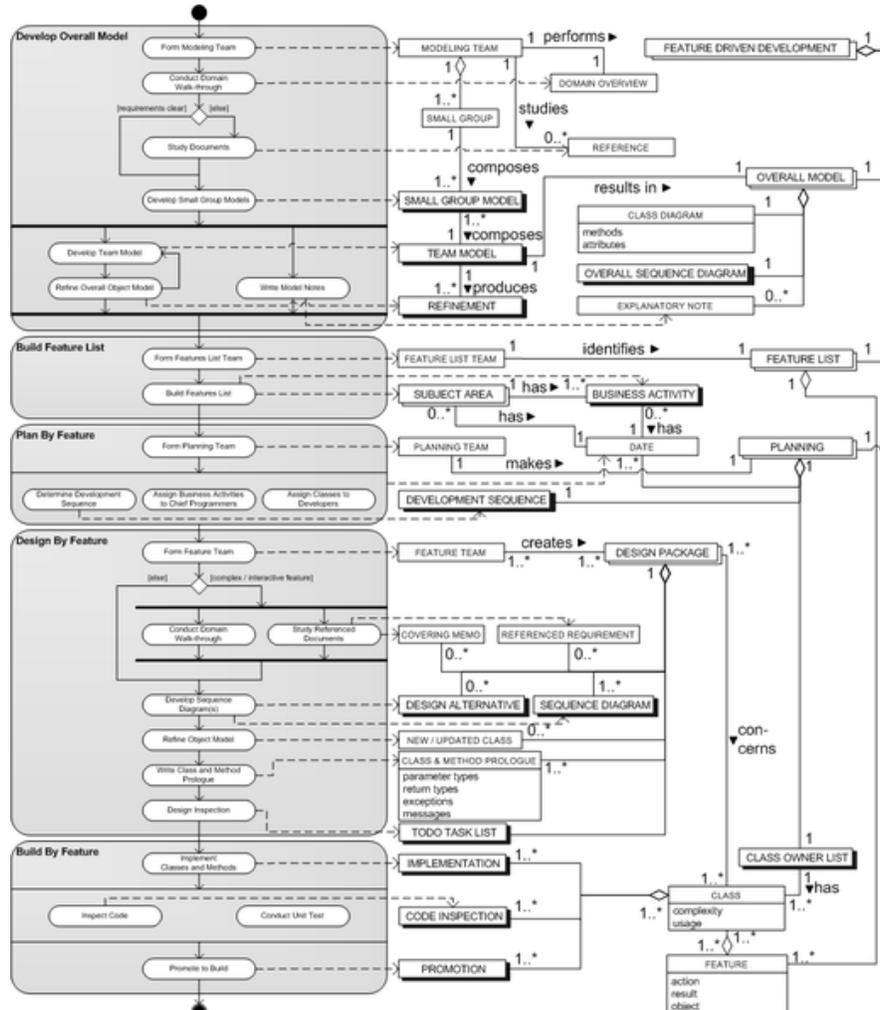
Best practices

Feature-Driven Development is built around a core set of industry-recognized best practices, derived from software engineering. These practices are all driven from a client-valued feature perspective. It is the combination of these practices and techniques that makes FDD so compelling. The best practices that make up FDD are shortly described below. For each best practice a short description will be given.

- **Domain Object Modeling.** Domain Object Modeling consists of exploring and explaining the domain of the problem to be solved. The resulting domain object model provides an overall framework in which to add features.
- **Developing by Feature.** Any function that is too complex to be implemented within two weeks is further decomposed into smaller functions until each sub-problem is small enough to be called a feature. This makes it easier to deliver correct functions and to extend or modify the system.
- **Individual Class (Code) Ownership.** Individual class ownership means that distinct pieces or grouping of code are assigned to a single owner. The owner is responsible for the consistency, performance, and conceptual integrity of the class.
- **Feature Teams.** A feature team is a small, dynamically formed team that develops a small activity. By doing so, multiple minds are always applied to each design decision and also multiple design options are always evaluated before one is chosen.

- **Inspections.** Inspections are carried out to ensure good quality design and code, primarily by detection of defects.
- **Configuration Management.** Configuration management helps with identifying the source code for all features that have been completed to date and to maintain a history of changes to classes as feature teams enhance them.
- **Regular Builds.** Regular builds ensure there is always an up to date system that can be demonstrated to the client and helps highlighting integration errors of source code for the features early.
- **Visibility of progress and results.** By frequent, appropriate, and accurate progress reporting at all levels inside and outside the project, based on completed work, managers are helped at steering a project correctly.

Metamodel (MetaModeling)



Process-Data Model for FDD

Metamodeling helps visualizing both the processes and the data of a method, such that methods can be compared and method fragments in the method engineering process can easily be reused. The advantage of the technique is that it is clear, compact, and consistent with UML standards.

The left side of the metadata model, depicted on the right, shows the five basic activities involved in a software development project using FDD. The activities all contain sub-activities that correspond to the sub-activities in the FDD process description on Jeff De Luca's website. The right side of the model shows the concepts involved. These concepts originate from the activities depicted in the left side of the diagram. A definition of the concepts is given in Table 3.

Tools used for Feature Driven Development

- CASE Spec. CASE Spec is a commercial enterprise tool for Feature-Driven development.
- TechExcel DevSuite. TechExcel DevSuite is a commercial suite of applications to enable Feature-Driven development.
- FDD Project Manager Application. FDDPMA is a web-based effort that aims to facilitate iterative development by reducing FDD management overhead, producing graphical progress reports, and providing a workplace where all the FDD related documentation is collected.
- FDD Tools. The FDD Tools project aims to produce an open source, cross-platform toolkit supporting the Feature Driven Development methodology.
- FDD Viewer. FDD Viewer is a utility to display and print parking lots.

Chapter-16

DevOps

DevOps is an emerging set of principles, methods and practices for communication, collaboration and integration between software development (application/software engineering) and IT operations (systems administration/infrastructure) professionals. It has developed in response to the emerging understanding of the interdependence and importance of both the development and operations disciplines in meeting an organization's goal of rapidly producing software products and services.

History

Predecessors

Many of the ideas (and people) involved in DevOps came from the Enterprise Systems Management and Agile Infrastructure/Agile Operations movements.

DevOpsDays

The term "DevOps" was coined by Patrick Debois from Belgium and spread initially through a conference in Ghent called DevOpsDays held in late 2009 . Since then, there have been seven DevOpsDays conferences held in the US, Brazil, Australia, and Germany.

DevOps Methodologies

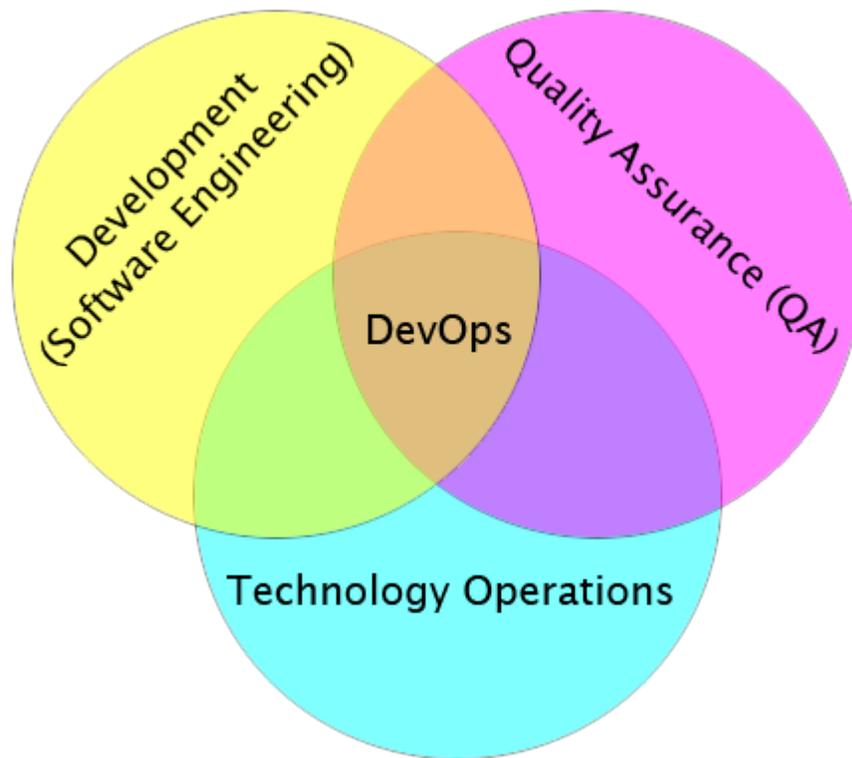


Illustration showing DevOps as the intersection of development (software engineering), technology operations and quality assurance (QA)

When new development methodologies (such as agile software development) are adopted in a traditional organization with separate departments for Dev, IT operations and QA, development and deployment activities that did not previously need deep cross-departmental integration with IT support or QA now require intimate multi-departmental collaboration. DevOps concerns more than just software deployments, however. It is a set of processes and methods for thinking about communication and collaboration between departments. Companies with very frequent releases may require a DevOps awareness or orientation. Flickr developed DevOps capability to support a business requirement of ten deployments per day; this daily deployment cycle would be much higher at organizations producing multi-focus or multi-function applications. This is referred to as continuous deployment and is frequently associated with the lean startup methodology. Working groups, professional associations and blogs have sprouted on the topic since 2009.

The use of DevOps integration can have profound results in product delivery, quality testing, feature development and maintenance releases (including the once special but now ubiquitous "hot fix"). Organizations without DevOps capabilities can see problems emerge from the "gap" of information shared between development and operations. This occurs as operations request greater reliability and security, developers ask for faster infrastructure responsiveness, while business users ask for more application enhancements and releases made available faster.

The adoption of DevOps is being driven by factors such as:

1. Use of agile and other development processes and methodologies
2. Demand for an increased rate of production releases from application and business unit stakeholders
3. Wide availability of virtualized and cloud infrastructure from internal and external providers
4. Increased usage of data center automation and configuration management tools
5. It has also been suggested a side-effect of the dominant, traditional US management style ("the Sloan model" vs "the Toyoda model") guarantees the development of silos of automation, thus creating "the DevOps gap" that then requires DevOps capability to address.

DevOps is frequently described as a more collaborative and productive relationship between development teams and operations teams. This improved relationship and collaboration increases efficiency and reduces the production risk associated with frequent changes. There are emerging ROI measurements and potential metrics for DevOps.

DevOps Impact on Application Releases

In many firms, application release events are high stress and high-risk activities involving multiple teams. In a DevOps organization, application releases are lower risk for the following reasons:

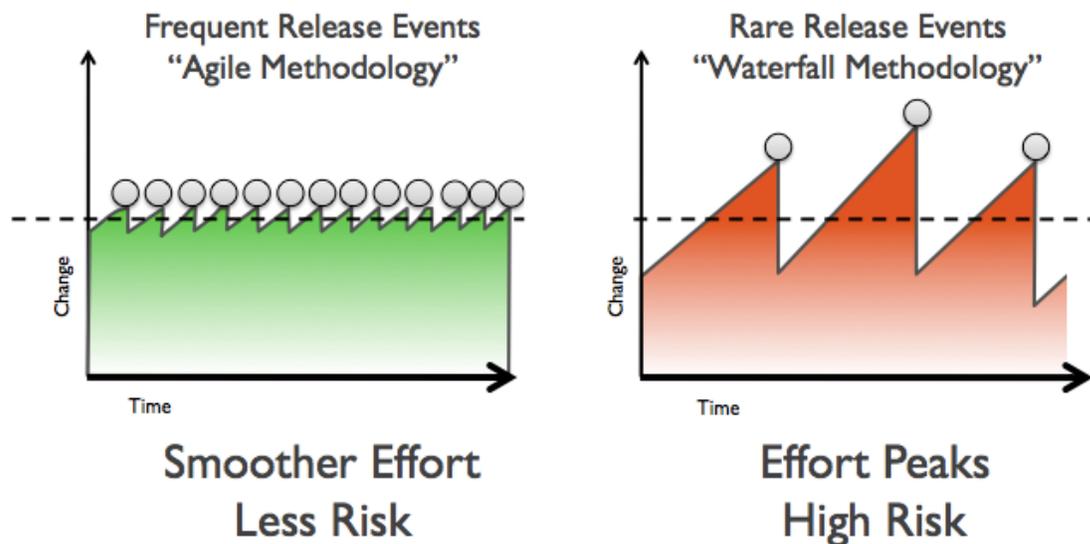


Illustration showing effect of Agile methodology's dramatic increase in frequency of release events -- often measured in days or weeks -- in contrast to large, infrequent releases, often measured in quarters or years, using traditional development methodologies.

Reduced change scope

Adoption of agile or iterative development, in contrast to the traditional waterfall model, means each release has less change but occurs more often. This creates a smooth rate of progressive application change vs. the large impact effect of rare deployments—each of which contains a large number of changes.

Increased release co-ordination

The use of a strong release coordinator to bridge the expertise and communications gap between development and operations; employment of collaboration tools such as spreadsheets, conference calls, instant messaging and corporate portals (sharepoint) to ensure thorough understanding of the change and full cooperation of all involved.

Automation

Comprehensive deployment automation ensure repeatability of deployment tasks and reduces deployment errors.

Adoption of DevOps Methodologies

Many organizations divide Development and System Administration into different departments. While Development departments are usually driven by user needs for frequent delivery of new features, Operations departments focus more on availability, stability of IT services and IT cost efficiency. These two contradicting goals create a "gap" between Development and Operations, which slows down IT's delivery of business value.

- Developers are often not concerned about the impact of their code on Operations. They deliver their code without involving Operations into architectural decisions or code reviews.
- Developers lack communicating configuration or environment changes necessary to run the updated code base.
 - Developers apply configuration changes manually to their workstations and do not document each necessary step. Often, coming up with the necessary configuration parameters for software involves experimentation with various parameters. After reaching a working state it is often difficult to identify the minimal steps to reach the working state.
 - Developers tend to use a tool set optimized for rapid development: Fast feedback on code changes, low memory consumption of runtime environment, etc. This tool set is very different from the target runtime environment in Operations where stability and performance trump flexibility requirements.
 - As Developers work on desktop computers they tend to use Operating Systems optimized for desktop use. The runtime environment is usually running a server operating system.
 - In development, the systems run locally on the developers workstation. In Operations, the system is often distributed amongst various servers like web server, application server, database server, etc.

- Development is driven by functional requirements usually directly related to business needs
- Operations is driven by non-functional requirements like availability, stability, performance, etc.
 - Operations tries to minimize risk for delivering on non-functional requirements by avoiding change
 - If frequent change is avoided, but the amount of necessary change stays constant, every change will be bigger
 - Bigger changes involve more risk, as more areas are affected by any change
- In trying to avoid change, Operations slows down the flow of new features to production and therefore slowing down Developments ability to deliver features
- Operations might not be fully aware of the application's internals making it hard to correctly define the runtime environment and update procedures
- Development might not be fully aware of the runtime environment making it hard to correctly adapt the code accordingly

Needs

- More and smaller changes—mean less risk
- Giving developers more environment control
- Giving infrastructure more application-centric understanding
- Clearly articulating simple processes
- Automating as much as possible
- Collaboration between dev and ops

In summary, as companies seek to streamline the cumbersome transition between development and operations, they typically confront 3 different types of problems:

Release Management Problems

Companies with release management problems are looking for better release planning than spreadsheets. They want an easy way to understand release risks, dependencies, stage gates adherence and an ensure compliance.

Release/Deployment Coordination Problems

Teams with release/deployment coordination problems are focused on better execution of release/deployment events. They want better tracking of discrete activities, faster escalation of issues, documented process control and granular reporting.

Release/Deployment Automation Problems

Companies with release/deployment automation problems usually have existing automation but want to more flexibly manage and drive this automation - without needing to enter everything manually at the command-line. Ideally, this automation can be invoked by non-operations resources in specific non-production environments.

One way to start streamlining release process is to identify which of the above problems is the overall team's highest priority.

Release coordinator

A relatively new role in enterprise IT which is primarily tasked with coordinating deployments of enterprise software to pre-production environments. The need for the release coordinator has been driven by:

1. The need to fill the DevOps "gap"
2. Increased infrastructure complexity - multiple layers and platforms which form web applications
3. Growth in rate of releases - due to agile and iterative development
4. Distributed teams - globally deployed, outsourced and hybrid development, testing and infrastructure teams

The release co-ordinator role (also referred to as a deployment or integration co-ordinator) has emerged from the release management or release engineering teams. This role is similar to an air traffic controller—performing real time co-ordination activities across diverse teams to achieve a group goal (safe landing and take-off) using shared resources (airspace, flight paths, airport runways, and terminal gates).

Release co-ordination contrasts with release management, which is often focused on planning and reporting on software changes, in order to control the release of specific application changes into production. Release engineering is concerned with the systematic and technical work related to building and deploying code into environments.

Change management is the infrastructure discipline for tracking all types of changes in the enterprise IT environment—including both application and infrastructure changes. Change management is a core part of ITIL v3.

Criticisms of DevOps

Some in the IT blogging field have cited criticisms of the "DevOps" label as just a primarily elitist sysadmin club to rebrand an existing problem or a marketing scheme to sell already well-understood methodologies.

Chapter-17

Lean Software Development

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

Origin

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

Lean principles

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

Eliminate waste

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

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

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

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

Amplify learning

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

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

Decide as late as possible

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

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

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

Deliver as fast as possible

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

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

Empower the team

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

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

Build integrity in

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

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

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

See the whole

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

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

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

Lean software practices

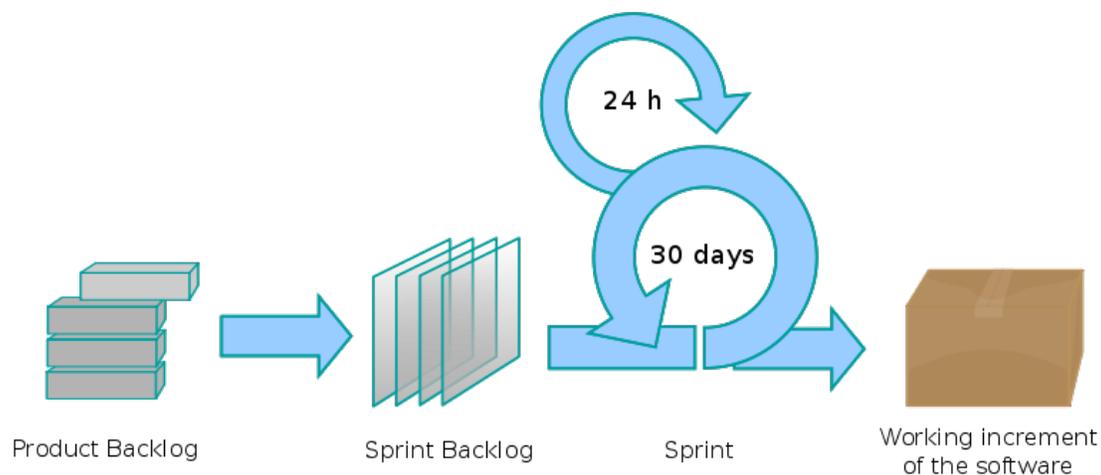
Lean software development practices, or what the Poppendiecks call "tools" are expressed slightly differently from their equivalents in agile software development, but there are parallels. Examples of such practices include:

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

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

Chapter-18

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-19

Presenter First & Planning Poker

Presenter First

Presenter First is a software development approach that combines the ideas of the Model View Presenter (MVP) design pattern, Test-Driven Development, and Feature-Driven Development.

Approach

Presenter First concentrates on transforming each of a customer's requirements into a well tested, working feature as quickly and with as much correlation to the customer's story language (requirement) as possible. The language of the story or requirement is used to directly guide development of the feature - even naming the modules and function calls. As a consequence, the feature implementation tends to closely represent the customer's desire with little extraneous or unneeded functionality. The language of the source code also corresponds closely to the customer's stories.

Presenter First is often applied in Graphical user interface applications. It is equally well applied to the development of command-line interfaces. Further, a slight variation of the approach has been used effectively in Embedded software; here the integral design pattern is known as Model-Conductor-Hardware and the approach is termed Conductor First.

When used in GUI applications, this approach allows the presentation logic and business logic of the application to be developed in a test first manner decoupled from on-screen widgets. Thus, the vast majority of the application programming can be tested via unit tests in an automated test suite. In so doing, the reliance on GUI testing tools to perform extensive system testing can be reduced to verifying basic GUI operation or eliminated entirely.

Implementation

The MVP design pattern decouples on-screen widgets, presentation logic, and business logic. Presenter First starts the development process with the Presenter component of an MVP axis. Test Driven Development is accomplished by mocking the View and Model and writing unit tests for the Presenter. Production code for the Presenter is then written and revised until the Presenter unit tests pass. The cycle is repeated for the Model. Unit testing the View is usually impractical or impossible; thus, view code is left as "thin" and devoid of logic as possible (i.e. the View is a wrapper around widget library calls and presentation logic is contained in the Presenter). The Presenter First approach applied to the MVP pattern allows the vast majority of application logic to be tested under automation leaving only simple on-screen verification testing of the View and its widgets.

The test cases for the Presenter are determined from the customer requirements or stories. A customer will generally explain features in terms of 'when' statements - for example, "When I click the 'save' button then the file should be saved and the unsaved file warning should disappear." Unit tests and Presenter code follow the flow of the 'when' statements. The Presenter expects View events to be fired (e.g. the click of the save button), and in turn it will make calls on the View (e.g. hide the warning message) and the Model (e.g. initiate file save operation) in response.

The many features of an application can make a single monolithic MVP axis unwieldy. Presenter First advocates breaking an application into multiple MVP axes. In a GUI application, each screen, dialog box, and complex widget is represented by an MVP axis (its functional design dictated by a customer story). Communication among the aggregated axis is accomplished through programmatic connections between Models.

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-Ostfold 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-20

P-Modeling Framework

P-Modeling Framework is a package of guidelines, methods, tools and templates for the development process improvement. P-Modeling framework can be integrated into any other SDLC in use, e.g., MSF Agile, MSF CMMI, RUP, etc.

History

The origins of P-Modeling Framework come from "The Babel Experiment" designed by Vladimir L. Pavlov in 2001 as a training program for software engineering students that was aimed at making students go through a “condensed” version of communication problems typical for software development and gain the experience of applying UML to overcome these problems.

This experiment was done in the following manner. A team of students was assigned the task of designing a software system with the following restriction factor: UML had to be the only language allowed for communication while working on the project. The premise was intended to make students go through a “condensed” version of communication problems typical for software development and gain the experience of applying UML to overcome these problems. As the result of this experiment, students developed quite clear and concise models.

A little later, during a design session, there were two independent teams working on the same task. The communication means of the first team was restricted to UML as described above, while the other team was allowed to communicate verbally using a natural language. It turned out that the first, more restricted team, performed the task more efficiently than the other one. The UML diagrams created by the first team were more sound, detailed, readable, and elaborated.

Subsequently, Vladimir L. Pavlov conducted a number of additional experiments intended to reveal whether the “silent” modeling sessions are more productive than the traditional ones. In these experiments, silent teams appeared to be at least as efficient as the others, and in some cases the silent teams outperformed the traditional ones.

Some of the interpretations of these results are the following:

- The restriction on using a natural language may stimulate creativity of the designers as well as force them to stay focused on their job;
- Work in speechless mode may force designers to explicitly uncover all underlying assumptions at the very early stages of the design process;
- UML is not treated as a superfluous burden irrelevant to real-life needs (as a “write-only” language) — instead, the designers may begin to demonstrate greater concern about the quality and readability of their models.

Afterwards, ideas were constructed for conducting additional new experiments with the intention of finding a method to compare UML to natural languages. The premise in these experiments was to set up forward (from a natural language to UML) and backward (from UML to the natural language) "translation" tasks for two teams of professional software designers. This would be done with one team performing the forward translation and the other one performing the backward translation. The intention was to observe how closely the outcome of the backward translation resembled the original text, thus providing verification of correctness of UML model.

The experiments showed that, for information describing software systems, UML has sufficient power of expression required to maintain the model's content. Texts obtained after the backward translation from UML were semantically equivalent to the original.

The experiments suggested the model of the entire software development cycle existed as a series of translations. In subsequent experiments backward translation verification has been demonstrated as a method to help guarantee deliverables of each development step do not lose, or have misinterpreted, anything that was produced at the previous step. This method has been named "Reverse Semantic Traceability." It has proven to be a solid second part completion to the P-Modeling Framework.

Basic principles

Reverse Semantic Traceability

Reverse Semantic Traceability is a quality control method that allows testing outputs of every translation step. Before proceeding to the next phase, the current artifacts are “reverse engineered”, and the restored text is compared to the original. If there is a difference between these two texts – the tested artifacts are corrected to eliminate the problem (or initial text is corrected.) Consequently, every step is confirmed by stepping back and making sure that development stays on the correct track. In this way, issues may be discovered and fixed without delays, so they do not accumulate, and do not cascade to subsequent phases of the development cycle.

The key word in the name of this method is “Semantic.” It is based on the fact the original and restored versions of a text are to be compared semantically, with a focus on the “meaning” of the text, not on particular “words” used in it.

The highest usage scenarios reported by early adopters of Reverse Semantic Traceability method are:

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

Speechless modeling

Being originally invented as an advanced training to teach Object-Oriented Analysis and Design with UML to students, the Speechless Modeling, in essence, is a restriction on using communication means directly or indirectly involving a natural language. In this way, a team of designers is forced to use the modeling language as the only language available for communication during a design session.

Incorporating P-Modeling Framework into Software Development Life Cycle (SDLC)

Regardless of what type of development process is used in an organization; waterfall, spiral, various iterative-incremental or some others, there are certain processes, such as software design, quality control, human resources management, risk management, communication management, etc. to which can P-Modeling Framework principles can be applied, especially in the earlier stages of a project when quality control activities are either minor or (virtually) absent.

Requirements and limitations

1. All the P-Modeling Session members should speak some graphical modeling language fluently.
2. Minimum of 8 qualified people required for full-blown P-Modeling Session.
3. Minimum of 3 qualified people required for an efficient RST Session.
4. P-modeling Framework doesn't provide the possibility to detect ambiguous, contradicting, and incomplete aspects in requirements or client requests.
5. Speechless Modeling Session requires large amount of energy and efforts from participants.

Criticism

P-Modeling Framework obviously has some room for further improvement. For example:

- P-Modeling Sessions require additional resources without knowledge of the original artifact and add extra workload for programmers.
- While doing RST, texts should be compared manually which means that the framework lacks automation.
- One of the possible outcomes in RST is the situation when people "design for RST" — they create artifacts in a way they can be easily reconstructed, without adding new value.
- There's no reliable statistical evidence of the P-Modeling Framework effectiveness.
- The "silent design sessions" have quite a narrow applicability: only to systems and organizations that can and need document the system in graphical modeling language. This is not the case when:
 - Company doesn't have enough developers "speaking any graphical modeling language" fluently and knowing when and how to apply it, which means very highly-qualified.
 - Company doesn't use any graphical modeling language extensively.
- P-Modeling Sessions can't help to differentiate between good design and bad design.

Chapter-21

Test-Driven Development

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

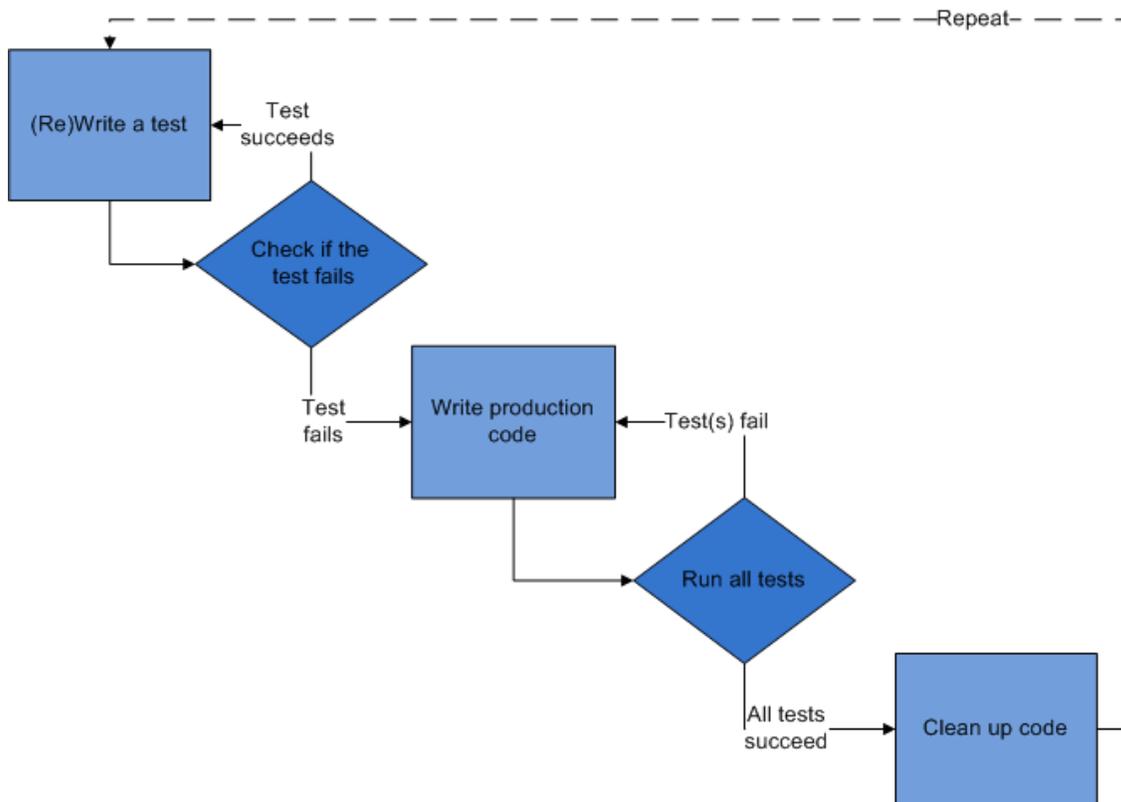
Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

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

Requirements

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

Test-driven development cycle



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

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

Add a test

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

Run all tests and see if the new one fails

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

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

Write some code

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

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

Run the automated tests and see them succeed

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

Refactor code

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

Repeat

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

Development style

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

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

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

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

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

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

Benefits

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

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

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

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

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

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

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

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

Vulnerabilities

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

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

Code visibility

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

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

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

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

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

Fakes, mocks and integration tests

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

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

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

A corollary of such dependency injection is that the actual database or other external-access code is never tested by the TDD process itself. To avoid errors that may arise from this, other tests are needed that instantiate the test-driven code with the “real” implementations of the interfaces discussed above. These tests are quite separate from the TDD unit tests, and are really integration tests. There will be fewer of them, and they need to be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, such as xUnit.

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

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

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

Chapter-22

Extreme Programming Practices

Extreme programming (XP) is a popular agile software development methodology used to implement software projects. Extreme programming has 12 practices, grouped into four areas, derived from the best practices of software engineering.

Fine scale feedback

Pair programming

Pair programming means that all code is produced by two people programming on one task on one workstation. One programmer has control over the workstation and is thinking mostly about the coding in detail. The other programmer is more focused on the big picture, and is continually reviewing the code that is being produced by the first programmer. Programmers trade roles regularly.

The pairs are not fixed: it's recommended that programmers try to mix as much as possible, so that everyone knows what everyone is doing, and everybody can become familiar with the whole system. This way, pair programming also can enhance team-wide communication. (This also goes hand-in-hand with the concept of Collective Ownership).

Planning game

The main planning process within extreme programming is called the Planning Game. The game is a meeting that occurs once per iteration, typically once a week. The planning process is divided into two parts:

- **Release Planning:** This is focused on determining what requirements are included in which near-term releases, and when they should be delivered. The customers and developers are both part of this. Release Planning consists of three phases:
 - **Exploration Phase:** In this phase the customer will provide a shortlist of high-value requirements for the system. These will be written down on user story cards.

- Commitment Phase: Within the commitment phase business and developers will commit themselves to the functionality that will be included and the date of the next release.
- Steering Phase: In the steering phase the plan can be adjusted, new requirements can be added and/or existing requirements can be changed or removed.
- Iteration Planning: This plans the activities and tasks of the developers. In this process the customer is not involved. Iteration Planning also consists of three phases:
 - Exploration Phase: Within this phase the requirement will be translated to different tasks. The tasks are recorded on task cards.
 - Commitment Phase: The tasks will be assigned to the programmers and the time it takes to complete will be estimated.
 - Steering Phase: The tasks are performed and the end result is matched with the original user story.

The purpose of the Planning Game is to guide the product into delivery. Instead of predicting the exact dates of when deliverables will be needed and produced, which is difficult to do, it aims to "steer the project" into delivery using a straightforward approach.

Release planning

Exploration phase

This is an iterative process of gathering requirements and estimating the work impact of each of those requirements.

- Write a Story: Business has come with a problem; during a meeting, development will try to define this problem and get requirements. Based on the business problem, a story (user story) has to be written. This is done by business, where they point out what they want a part of the system to do. It is important that development has no influence on this story. The story is written on a user story card.
- Estimate a Story: Development estimates how long it will take to implement the work implied by the story card. Development can also create spike solutions to analyze or solve the problem. These solutions are used for estimation and discarded once everyone gets clear visualization of the problem. Again, this may not influence the business requirements.
- Split a Story: Every design critical complexity has to be addressed before starting the iteration planning. If development isn't able to estimate the story, it needs to be split up and written again.

When business cannot come up with any more requirements, one proceeds to the commitment phase.

Commitment phase

This phase involves the determination of costs, benefits, and schedule impact. It has four components:

- Sort by Value: Business sorts the user stories by Business Value.
- Sort by Risk: Development sorts the stories by risk.
- Set Velocity: Development determines at what speed they can perform the project.
- Choose scope: The user stories that will be finished in the next release will be picked. Based on the user stories the release date is determined.

Sort by value

The business side sorts the user stories by business value. They will arrange them into three piles:

- Critical: stories without which the system cannot function or has no meaning.
- Significant Business Value: Non-critical user stories that have significant business value.
- Nice to have: User stories that do not have significant business value.

Sort by risk

The developers sort the user stories by risk. They also categorize into three piles: low, medium and high risk user stories. The following is an example of an approach to this:

- Determine Risk Index: Give each user story an index from 0 to 2 on each of the following factors:
 - Completeness (do we know all of the story details?)
 - Complete (0)
 - Incomplete (1)
 - Unknown (2)
 - Volatility (is it likely to change?)
 - low (0)
 - medium (1)
 - high (2)
 - Complexity (how hard is it to build?)
 - simple (0)
 - standard (1)
 - complex (2)

All indexes for a user story are added, assigning the user stories a risk index of low (0–1), medium (2–4), or high (5–6).

Steering phase

Within the steering phase the programmers and business people can "steer" the process. That is to say, they can make changes. Individual user stories, or relative priorities of different user stories, might change; estimates might prove wrong. This is the chance to adjust the plan accordingly.

Iteration planning

Exploration phase

The exploration phase of the iteration planning is about creating tasks and estimating their implementation time.

- Translate the requirement to tasks: Place on task cards.
- Combine/Split task: If the programmer cannot estimate the task because it is too small or too big, the programmer will need to combine or split the task.
- Estimate task: Estimate the time it will take to implement the task.

Commitment phase

Within the commitment phase of the iteration planning programmers are assigned tasks that reference the different user stories.

- A programmer accepts a task: Each programmer picks a task for which he or she takes responsibility.
- Programmer estimates the task: Because the programmer is now responsible for the task, he or she should give the eventual estimation of the task.
- Set load factor: The load factor represents the ideal amount of hands-on development time per programmer within one iteration. For example, in a 40-hour week, with 5 hours dedicated to meetings, this would be no more than 35 hours.
- Balancing: When all programmers within the team have been assigned tasks, a comparison is made between the estimated time of the tasks and the load factor. Then the tasks are balanced out among the programmers. If a programmer is overcommitted, other programmers must take over some of his or her tasks and vice versa.

Steering phase

The implementation of the tasks is done during the steering phase of the iteration planning.

- Get a task card: The programmer gets the task card for one of the tasks to which he or she has committed.
- Find a Partner: The programmer will implement this task along with another programmer. This is further discussed in the practice Pair Programming.

- Design the task: If needed, the programmers will design the functionality of the task.
- Write unit test: Before the programmers start coding the functionality they first write automated tests. This is further discussed in the practice Unit Testing.
- Write code: The programmers start to code.
- Run test: The unit tests are run to test the code.
- Refactor: Remove any code smells from the code.
- Run Functional test: Functional tests (based on the requirements in the associated user story and task card) are run.

Test driven development

Unit tests are automated tests that test the functionality of pieces of the code (e.g. classes, methods). Within XP, unit tests are written before the eventual code is coded. This approach is intended to stimulate the programmer to think about conditions in which his or her code could fail. XP says that the programmer is finished with a certain piece of code when he or she cannot come up with any further condition on which the code may fail.

Whole team

Within XP, the "customer" is not the one who pays the bill, but the one who really uses the system. XP says that the customer should be on hand at all times and available for questions. For instance, the team developing a financial administration system should include a financial administrator.

Continuous process

Continuous integration

The development team should always be working on the latest version of the software. Since different team members may have versions saved locally with various changes and improvements, they should try to upload their current version to the code repository every few hours, or when a significant break presents itself. Continuous integration will avoid delays later on in the project cycle, caused by integration problems.

Design improvement

Because XP doctrine advocates programming only what is needed today, and implementing it as simply as possible, at times this may result in a system that is stuck. One of the symptoms of this is the need for dual (or multiple) maintenance: functional changes start requiring changes to multiple copies of the same (or similar) code. Another symptom is that changes in one part of the code affect lots of other parts. XP doctrine says that when this occurs, the system is telling you to refactor your code by changing the architecture, making it simpler and more generic.

Small releases

The delivery of the software is done via frequent releases of live functionality creating concrete value. The small releases help the customer to gain confidence in the progress of the project. This helps maintain the concept of the whole team as the customer can now come up with his suggestions on the project based on real experience.

Shared understanding

Coding standard

Coding standard is an agreed upon set of rules that the entire development team agree to adhere to throughout the project. The standard specifies a consistent style and format for source code, within the chosen programming language, as well as various programming constructs and patterns that should be avoided in order to reduce the probability of defects. The coding standard may be a standard conventions specified by the language vendor (e.g. The Code Conventions for the Java Programming Language, recommended by Sun), or custom defined by the development team.

Collective code ownership

Collective code ownership means that everyone is responsible for all the code; this, in turn, means that everybody is allowed to change any part of the code. Pair programming contributes to this practice: by working in different pairs, all the programmers get to see all the parts of the code. A major advantage claimed for collective ownership is that it speeds up the development process, because if an error occurs in the code any programmer may fix it.

By giving every programmer the right to change the code, there is risk of errors being introduced by programmers who think they know what they are doing, but do not foresee certain dependencies. Sufficiently well defined unit tests address this problem: if unforeseen dependencies create errors, then when unit tests are run, they will show failures.

Simple design

Programmers should take a "simple is best" approach to software design. Whenever a new piece of code is written, the author should ask themselves 'is there a simpler way to introduce the same functionality?'. If the answer is yes, the simpler course should be chosen. Refactoring should also be used, to make complex code simpler.

System metaphor

The system metaphor is a story that everyone - customers, programmers, and managers - can tell about how the system works. It's a naming concept for classes and methods that should make it easy for a team member to guess the functionality of a particular

class/method, from its name only. For example a library system may create `loan_records(class)` for `borrowers(class)`, and if the item were to become overdue it may perform a `make_overdue` operation on a `catalogue (class)`. For each class or operation the functionality is obvious to the entire team.

Programmer welfare

Sustainable pace

The concept is that programmers or software developers should not work more than 40 hour weeks, and if there is overtime one week, that the next week should not include more overtime. Since the development cycles are short cycles of continuous integration, and full development (release) cycles are more frequent, the projects in XP do not follow the typical crunch time that other projects require (requiring overtime).

Also, included in this concept is that people perform best and most creatively if they are rested.

A key enabler to achieve sustainable pace is frequent code-merge and always executable & test covered high quality code. The constant refactoring way of working enforces team members with fresh and alert minds. The intense collaborative way of working within the team drives a need to recharge over weekends.

Well-tested, continuously integrated, frequently deployed code and environments also minimize the frequency of unexpected production problems and outages, and the associated after-hours nights and weekends work that is required.