Software Maintenance & Software Development

Margorie Broome

First Edition, 2012

ISBN 978-81-323-3104-9

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

Table of Contents

Chapter 1

Software Maintenance

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

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

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

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

The Importance of Software Maintenance

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

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

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

Maintenance specialists 35%

High staff experience 34%

Table-driven variables and data 33%

Low complexity of base code 32%

Y2K and special search engines 30%

Code restructuring tools 29%

Re-engineering tools 27%

High level programming languages 25%

Reverse engineering tools 23%

Complexity analysis tools 20%

Defect tracking tools 20%

Y2K “mass update” specialists 20%

Automated change control tools 18%

Unpaid overtime 18%

Quality measurements 16%

Formal base code inspections 15%

Regression test libraries 15%

Excellent response time 12%

Annual training of > 10 days 12%

High management experience 12%

HELP desk automation 12%

No error prone modules 10%

On-line defect reporting 10%

Productivity measurements 8%

Excellent ease of use 7%

User satisfaction measurements 5%

High team morale 5%

Sum 503%

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

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

Maintenance Factors Minus Range

Error prone modules -50%

Embedded variables and data -45%

Staff inexperience -40%

High complexity of base code -30%

No Y2K of special search engines -28%

Manual change control methods -27%

Low level programming languages -25%

No defect tracking tools -24%

No Y2K “mass update” specialists -22%

Poor ease of use -18%

No quality measurements -18%

No maintenance specialists -18%

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

No regression test libraries -15%

No HELP desk automation -15%

No on-line defect reporting -12%

Management inexperience -15%

No code restructuring tools -10%

No annual training -10%

No reengineering tools -10%

No reverse engineering tools -10%

No complexity analysis tools -10%

No productivity measurements -7%

Poor team morale -6%

No user satisfaction measurements -4%

No unpaid overtime 0%

Sum -500%

Software maintenance planning

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

Software maintenance processes

This section describes the six software maintenance processes as:

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

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

- Transition: a controlled and coordinated sequence of activities during which a system is transferred progressively from the developer to the maintainer;
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts negotiated by maintainers;
- Modification Request and Problem Report Help Desk: a problem-handling process used by maintainers to prioritize, documents and route the requests they receive;
- Modification Request acceptance/rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

Categories of maintenance in ISO/IEC 14764

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

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

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

Chapter 2

Software Evolution

Software evolution is the term used in software engineering (specifically software maintenance) to refer to the process of developing software initially, then repeatedly updating it for various reasons.

General introduction

Fred Brooks, in his key book *The Mythical Man-Month*, states that over 90% of the costs of a typical system arise in the maintenance phase, and that any successful piece of software will inevitably be maintained.

In fact, Agile methods stem from maintenance like activities in and around web based technologies, where the bulk of the capability comes from frameworks and standards.

Software maintenance address bug fixes and minor enhancements and software evolution focus on adaptation and migration.

IMPACT OF SOFTWARE EVOLUTION

The aim of software evolution would be implementing (and revalidate) the possible major changes to the system without being able a priori to predict how user requirements will evolve . The existing larger system is never complete and continues to evolve . As it evolves, the complexity of the system will grow unless there is a better solution available to solve these issues. The main objectives of software evolution are ensuring the reliability and flexibility of the system. During the 20 years pasted, the lifespan of a system could be in average 6-10 years. However, recently found that a system should be evolved once few months to ensure it is compromised to the real-world environment. This is due to the rapid growth of World Wide Web and Internet Resources that make users easier to find related information. The idea of software evolution leads to open source development as anybody could download the source codes and hence modify it.

The positive impact in this case is large amounts of new ideas would be discovered and generated that aims the system to have better improvement in variety choices. However, the negative impact is there is no copyright if a software product has been published as open source.

references 1.Template:Author == K.H.Bennett, V.T Rajlich, R. Mohamad Mazrul,

2.Template:Author == Trung Hung Vo,

Changes in Software Evolution Models and Theories

Over time, software systems, programs as well as applications continue to develop. These changes will require new laws and theories to be created and justified. Some models as well would require additional aspects in developing future programs. Innovations and improvements do increase unexpected form of software development. The maintenance issues also would probably changed as to adapt to the evolution of the future software. Software process and development are an ongoing experience that has a never-ending cycle. After going through learning and refinements, it is always an arguable issue when it comes to matter of efficiency and effectiveness of the programs. [aeddy; ref: Understanding Open Source Software Evolution Walt Scacchi Institute for Software Research]

Types of software maintenance

E.B. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective. Four categories of software were then catalogued by Lientz and Swanson (1980) . These have since been updated and normalized internationally in the ISO/IEC 14764:2006:

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

All of the preceding take place when there is a known requirement for change.

Although these categories were supplemented by many authors like Warren et al. (1999) and Chapin (2001), the ISO/IEC 14764:2006 international standard has kept the basic four categories.

More recently the description of software maintenance and evolution has been done using ontologies (Kitchemham et al. (1999), Derider (2002), Vizcaíno 2003, Dias (2003), and Ruiz (2004)), which enrich the description of the many evolution activities.

STAGE MODEL

Current trends and practices are projected forward using a new model of software evolution called the staged model . Staged model was introduced to replace conventional analysis which is less suitable for modern software development is rapid changing due to its difficulties of hard to contribute in software evolution. There are five distinct stages contribute in simple staged model (Initial development, Evolution, Servicing, Phase-out, and Close-down). • According to K.H.Bennett and V.T Rajlich , the key contribution is to separate the 'maintenance' phase into an evolution stage followed by a servicing and phase out stages. The first version of software system which is lacking some features will be developed during initial development or also known as alpha stage . However, the architecture has already been possessed during this stage will bring for any future changes or amendments. Most references in this stage will base on scenarios or case study. Knowledge has defined as another important outcome of initial development. Such knowledge including the knowledge of application domain, user requirements, business rules, policies, solutions, algorithm, etc. Knowledge also seems as the important factor for the subsequent phase of evolution. • Once the previous stage completed successfully (and must be completed successfully before entering next stage), the next stage would be evolution. Users tend to change their requirements as well as they prefer to see some improvements or changes. Due to this factor, the software industry is facing the challenges of rapid changes environment. Hence the goal of evolution is to adapt the application to the ever-changing user requirements and operating environment . During the previous stage, the first version application created might contain a lot of faults, and those faults will be fixed during evolution stage based on more specified and accurate requirements due to the case study or scenarios. • The software will continuously evolve until it is no longer evolvable and then enter stage of servicing (also known as software maturity). During this stage, only minor changes will be done. • Next stage which is phase-out, there is no more servicing available for that particular software. However, the software still in production. • Lastly, close-down. The software use is disconnected or discontinue and the users are directed towards a replacement..

Lehman's Laws of Software Evolution

Prof. Meir M. Lehman, who worked at Imperial College London from 1972 to 2002, and his colleagues have identified a set of behaviours in the evolution of proprietary software. These behaviours (or observations) are known as *Lehman's Laws*, and there are eight of them:

1. Continuing Change
2. Increasing Complexity
3. Large Program Evolution
4. Invariant Work-Rate

5. Conservation of Familiarity
6. Continuing Growth
7. Declining Quality
8. Feedback System

It is worth mentioning that the laws are believed to apply mainly to monolithic, proprietary software. For example, some empirical observations coming from the study of open source software development appear to challenge some of the laws.

The laws predict that change is inevitable and not a consequence of bad programming and that there are limits to what a software evolution team can achieve in terms of safely implementing changes and new functionality.

Maturity Models specific to software evolution have been developed to help improve processes to ensure continuous rejuvenation of the software evolves iteratively.

The "global process" that is made by the many stakeholders (e.g. developers, users, their managers) has many feedback loops. The evolution speed is a function of the feedback loop structure and other characteristics of the global system. Process simulation techniques, such as system dynamics can be useful in understanding and managing such global process.

Software evolution is not likely to be Darwinian, Lamarckian or Baldwinian, but an important phenomenon on its own. Giving the increasing dependence on software at all levels of society and economy, the successful evolution of software is becoming increasingly critical. This is an important topic of research that hasn't received much attention.

The evolution of software, because of its rapid path in comparison to other man-made entities, was seen by Lehman as the "fruit fly" of the study of the evolution of artificial systems.

Chapter 3

Software Brittleness & Software Archaeology

Software Brittleness

The term ***software brittleness*** refers to the increased difficulty in fixing older software that may appear reliable, but fails badly when presented with unusual data or altered in a seemingly minor way. The term is derived from analogies to brittleness in metalworking.

Causes

When software is new, it is very malleable; it can be formed to be whatever is wanted by the implementers. But as the software in a given project grows larger and larger, and develops a larger base of users with long experience with the software, it becomes less and less malleable. Like a metal that has been work-hardened, the software becomes a legacy system, brittle and unable to be easily maintained without fracturing the entire system.

Brittleness in software can be caused by algorithms that do not work well for the full range of input data. A good example is an algorithm that allows a divide by zero to occur, or a curve-fitting equation that is used to extrapolate beyond the data that it was fitted to. Another cause of brittleness is the use of data structures that restrict values. This was commonly seen in the late 1990s as people realized that their software only had room for a 2 digit year entry; this led to the sudden updating of tremendous quantities of brittle software before the year 2000. Another more commonly encountered form of brittleness is in graphical user interfaces that make invalid assumptions. For example, a user may be running on a low resolution display, and the software will open a window too large to fit the display. Another common problem is expressed when a user uses a color scheme other than the default, causing text to be rendered in the same color as the background, or

a user uses a font other than the default, which won't fit in the allowed space and cuts off instructions and labels.

Very often, an old code base is simply abandoned and a brand-new system (which is intended to be free of many of the burdens of the legacy system) created from scratch, but this can be an expensive and time-consuming process.

Some examples and reasons behind software brittleness:

- Users expect a relatively constant user interface; once a feature has been implemented and exposed to the users, it is very difficult to convince them to accept major changes to that feature, even if the feature was not well designed or the existence of the feature blocks further progress.
- A great deal of documentation may describe the current behavior and would be expensive to change. In addition, it is essentially impossible to recall all copies of the existing documentation, so users are likely to continue to refer to obsolete manuals.
- The original implementers (who knew how things really worked) have moved on and left insufficient documentation of the internal workings of the software. Many small implementation details were only understood through the oral traditions of the design team, and many of these details eventually are irretrievably lost, although some can be rediscovered through the diligent (and expensive) application of software archaeology.
- Patches have probably been issued throughout the years, subtly changing the behavior of the software. In many cases, these patches, while correcting the overt failure for which they were issued, introduce other, more subtle, failures into the system. These subtle failures make subsequent changes to the system more difficult.
- More subtle forms of brittleness commonly occur in artificial intelligence systems. These systems often rely on significant assumptions about the input data. When these assumptions aren't met – and, because they may not be stated, this may easily be the case – then the system will respond in completely unpredictable ways.
- Systems can also be brittle if the component dependencies are too rigid. One example of this is seen in the difficulties transitioning to new versions of dependencies. When one component expects another to output only a given range of values, and that range changes, then it can cause errors to ripple through the system, either during building or at runtime.

Examples of brittleness in software

- Ariane 5 Flight 501, in which an overflow in a velocity calculation shut down a rocket's navigational system.
- Corner case, a common area in which systems are brittle.

Software Archaeology

Software archaeology or **software archeology** is the study of poorly documented or undocumented legacy software implementations, as part of software maintenance. Software archaeology, named by analogy with archaeology, includes the reverse engineering of software modules, and the application of a variety of tools and processes for extracting and understanding program structure and recovering design information. Software archaeology may reveal dysfunctional team processes which have produced poorly designed or even unused software modules. The term has been in use for several decades, and reflects a fairly natural metaphor: a programmer reading legacy code may feel that he or she is in the same situation as an archaeologist exploring the rubble of an ancient civilization.

Techniques

A workshop on Software Archaeology at the 2001 OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) conference identified the following software archaeology techniques, some of which are specific to object-oriented programming:

- Scripting languages to build static reports and for filtering diagnostic output
- Ongoing documentation in HTML pages
- Synoptic signature analysis, statistical analysis, and software visualization tools
- Reverse-engineering tools
- Operating-system-level tracing via truss or strace
- Web search engines and tools to search for keywords in source files
- IDE file browsing
- Test harnesses such as JUnit and CppUnit
- API documentation generation using tools such as Javadoc and doxygen
- Debuggers

More generally, Andy Hunt and Dave Thomas note the importance of version control, dependency management, text indexing tools such as GLIMPSE and SWISH-E, and "[drawing] a map as you begin exploring."

Like true archaeology, software archaeology involves investigative work to understand the thought processes of one's predecessors. At the OOPSLA workshop, Ward Cunningham suggested a synoptic signature analysis technique which gave an overall "feel" for a program by showing only punctuation, such as semicolons and curly braces. In the same vein, Cunningham has suggested viewing programs in 2 point font in order to understand the overall structure. Another technique identified at the workshop was the use of aspect-oriented programming tools such as AspectJ to systematically introduce tracing code without directly editing the legacy program.

Network and temporal analysis techniques can reveal the patterns of collaborative activity by the developers of legacy software, which in turn may shed light on the strengths and weaknesses of the software artifacts produced.

Michael Rozlog of Embarcadero Technologies has described software archaeology as a six-step process which enables programmers to answer questions such as "What have I just inherited?" and "Where are the scary sections of the code?" These steps, similar to those identified by the OOPSLA workshop, include using visualization to obtain a visual representation of the program's design, using software metrics to look for design and style violations, using unit testing and profiling to look for bugs and performance bottlenecks, and assembling design information recovered by the process. Software archaeology can also be a service provided to programmers by external consultants.

Software archaeology has continued to be a topic of discussion at more recent software engineering conferences.

Chapter 4

Program Slicing & Patch (Computing)

Program Slicing

In computer programming, **program slicing** is the computation of the set of programs statements, the **program slice**, that may affect the values at some point of interest, referred to as a **slicing criterion**. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control.

Slicing techniques have been seeing a rapid development since the original definition by Mark Weiser. At first, slicing was only static, i.e., applied on the source code with no other information than the source code. Bogdan Korel and Janusz Laski introduced *dynamic slicing*, which works on a specific execution of the program (for a given execution trace). Other forms of slicing exist, for instance path slicing.

Static slicing

Based on the original definition of Weiser, informally, a static program slice S consists of all statements in program P that may affect the value of variable v at some point p. The slice is defined for a slicing criterion C=(x,V), where x is a statement in program P and V is a subset of variables in P. A static slice includes all the statements that affect variable v for a set of all possible inputs at the point of interest (i.e., at the statement x). Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

Example

```
int i;
int sum = 0;
int product = 1;
for(i = 0; i < N; ++i) {
  sum = sum + i;
```

```
  product = product *i;
}
write(sum);
write(product);
```

This new program is a valid slicing of the above program with respect to the criterion (`write(sum)`,{sum}):

```
int i;
int sum = 0;

for(i = 0; i < N; ++i) {
  sum = sum + i;

}
write(sum);
```

In fact, most static slicing techniques, including Weiser's own technique, will also remove the `write(sum)` statement. Indeed, at the statement `write(sum)`, the value of `sum` is not dependent on the statement itself.

Dynamic slicing

Makes use of information about a particular execution of a program. A dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program.

An example to clarify the difference between static and dynamic slicing. Consider a small piece of a program unit, in which there is an iteration block containing an if-else block. There are a few statements in both the if and else blocks that have an affect on a variable. In the case of static slicing, since we look at the whole program unit irrespective of a particular execution of the program, the affected statements in both blocks would be included in the slice. But, in the case of dynamic slicing we consider a particular execution of the program, wherein the if block gets executed and the affected statements in the else block do not get executed. So, in this particular execution case, the dynamic slice would contain only the statements in the if block.

Patch (Computing)

A **patch** is a piece of software designed to fix problems with, or update a computer program or its supporting data. This includes fixing security vulnerabilities and other bugs, and improving the usability or performance. Though meant to fix problems, poorly designed patches can sometimes introduce new problems.

Patch management is the process of using a strategy and plan of what patches should be applied to which systems at a specified time.

Types

Programmers publish and apply patches in various forms. Because proprietary software authors withhold their source code, their patches are distributed as binary executables instead of source. This type of patch modifies the program executable—the program the user actually runs—either by modifying the binary file to include the fixes or by completely replacing it.

Patches can also circulate in the form of source code modifications. In these cases, the patches consist of textual differences between two source code files. These types of patches commonly come out of open source projects. In these cases, developers expect users to compile the new or changed files themselves.

Because the word "patch" carries the connotation of a small fix, large fixes may use different nomenclature. Bulky patches or patches that significantly change a program may circulate as "service packs" or as "software updates". Microsoft Windows NT and its successors (including Windows 2000, Windows XP, and later versions) use the "service pack" terminology.

In several Unix-like systems, particularly Linux, updates between releases are delivered as new software packages. These updates are in the same format as the original installation so they can be used either to update an existing package in-place (effectively patching) or be used directly for new installations.

History

Historically, software suppliers distributed patches on paper tape or on punched cards, expecting the recipient to cut out the indicated part of the original tape (or deck), and patch in (hence the name) the replacement segment. Later patch distributions used magnetic tape. Then, after the invention of removable disk drives, patches came from the software developer via a disk or, later, CD-ROM via mail. Today, with almost universal Internet access, end-users must download most patches from the developer's web site.

Today, computer programs can often coordinate patches to update a target program. Automation simplifies the end-users' task — they need only to execute an update program, whereupon that program makes sure that updating the target takes place completely and correctly. Service packs for Microsoft Windows NT and its successors and for many commercial software products adopt such automated strategies.

Some programs can update themselves via the Internet with very little or no intervention on the part of users. The maintenance of server software and of operating systems often takes place in this manner. In situations where system administrators control a number of computers, this sort of automation helps to maintain consistency. The application of security patches commonly occurs in this manner.

Application

The size of patches may vary from a few kilobytes to hundreds of megabytes — mostly more significant changes imply a larger size, though this also depends on whether the patch includes entire files or only the changed portion(s) of files. In particular, patches can become quite large when the changes add or replace non-program data, such as graphics and sounds files. Such situations commonly occur in the patching of computer games. Compared with the initial installation of software, patches usually do not take long to apply.

In the case of operating systems and computer server software, patches have the particularly important role of fixing security holes. To facilitate updates, operating systems often provide automatic or semi-automatic update facilities.

Completely automatic updates have not succeeded in gaining widespread popularity in corporate computing environments, partly because of the aforementioned glitches, but also because administrators fear that software companies may gain unlimited control over their computers. Package management systems can offer various degrees of patch automation.

Usage of completely automatic updates is far more widespread in the consumer market, due largely to the fact that Microsoft Windows added support for them, and Service Pack 2 of Windows XP enabled them by default.

Cautious users, particularly system administrators, tend to put off applying patches until they can verify the stability of the fixes. Microsoft (W)SUS support this. In the cases of large patches or of significant changes, distributors often limit availability of patches to qualified developers as a beta test.

Applying patches to firmware poses special challenges: re-embedding typically small code sets on hardware devices often involves the provision of totally new program code, rather than simply of differences from the previous version. Often the patch consists of bare binary data and a special program that *replaces* the previous version with the new version is provided. A motherboard BIOS update is an example of a common firmware patch. Any unexpected error or interruption during the update, such as a power outage, may render the motherboard unusable. It is possible for motherboard manufacturers to put safeguards in place to prevent serious damage. An example safeguard is to keep a backup of the firmware to use in case the primary copy is determined to be corrupt (usually through the use of a checksum, such as a CRC).

Computer games

Computer games receive patches to fix compatibility problems after their initial release just like any other software, but they can also be applied to change game rules or algorithms. These patches may be prompted by the discovery of exploits in the multiplayer game experience that can be used to gain unfair advantages over other

players. Extra features and game play tweaks can often be added. These kinds of patches are common in first-person shooters with multiplayer capability, and in MMORPGs. MMORPGs, which are typically very complex with large amounts of content, almost always rely heavily on patches following the initial release, where patches sometimes add new content and abilities available to players. Because the balance and fairness for all players of an MMORPG can be severely corrupted within a short amount of time by an exploit, servers of an MMORPG are sometimes taken down with short notice in order to apply a critical patch with a fix.

Tools

There are several tools to aid in the patch application process, such as RTPatch, JUpdater, StableUpdate or Visual Patch. WinZip Self-Extractor can launch a program that can apply a patch

In software development

Patches sometimes become mandatory to fix problems with libraries or with portions of source code for programs in frequent use or in maintenance. This commonly occurs on very large-scale software projects, but rarely in small-scale development.

In open source projects, the authors commonly receive patches or many people publish patches that fix particular problems or add certain functionality, like support for local languages outside the project's locale. In an example from the early development of the Linux operating system (noted for publishing its complete source code), Linus Torvalds, the original author, received hundreds of thousands of patches from many programmers to apply against his original version.

The Apache HTTP Server originally evolved as a number of patches that Brian Behlendorf collated to improve NCSA HTTPd, hence a name that implies that it is a collection of patches ("a patchy server"). The FAQ on the project's official site states that the name 'Apache' was chosen from respect for the Native American Indian tribe of Apache. However, the 'a patchy server' explanation was initially given on the project's website.

Security patches

A security patch is a change applied to an asset to correct the weakness described by a vulnerability. This corrective action will prevent successful exploitation and remove or mitigate a threat’s capability to exploit a specific vulnerability in an asset.

Security patches are the primary method of fixing security vulnerabilities in software. Currently Microsoft releases its security patches once a month, and other operating systems and software projects have security teams dedicated to releasing the most reliable software patches as soon after a vulnerability announcement as possible. Security patches are closely tied to responsible disclosure.

Hot patching

Hot patching is the application of patches without shutting down and restarting the system or the program concerned. This addresses problems related to unavailability of service provided by the system or the program. A patch that can be applied in this way is called a hot patch.

Chapter 5

Software Rot & Backporting

Software Rot

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

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

Causes

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

Environment change

When changes occur in the program's environment, particularly changes which the designer of the program did not anticipate, the software may no longer operate as originally intended. For example, many early computer game designers made

assumptions about processing speed of the CPU which the games were designed for. A consequence of this was that when the CPU's clock speed was used as a timer in the game, the gameplay speed would increase with that of the CPU, making the software less usable over time.

Onceability

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

Unused code

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

Rarely updated code

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

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

Classification

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

Dormant rot

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

Active rot

Software that is being continuously modified may lose its integrity over time if proper mitigating processes are not consistently applied. However, much software requires continuous changes to meet new requirements and correct bugs, and re-engineering software each time a change is made is rarely practical. This creates what is essentially an evolution process for the program, causing it to depart from the original engineered design. As a consequence of this and a changing environment, assumptions made by the original designers may be invalidated, introducing bugs.

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

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

Example

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

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

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

- The administrator can accidentally make changes which conflict with each other or the original software, causing the forum to behave unexpectedly or break down altogether. This leaves him in a very bad position: as he has deviated so greatly from the original code, technical support and assistance in reviving the forum will be difficult to obtain.
- A security hole may be discovered in the original forum source code, requiring a security patch. However, because the administrator has modified the code so extensively, the patch may not be directly applicable to his code, requiring the administrator to effectively rewrite the update.

- The administrator who made the modifications could vacate his position, leaving the new administrator with a convoluted and heavily-modified forum that lacks full documentation. Without fully understanding the modifications, it is difficult for the new administrator to make changes without introducing conflicts and bugs. (Furthermore, documentation of the original system might no longer be available; it might have been abandoned, become proprietary closed software, or, with the passage of enough time, been lost.)

Refactoring

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

Backporting

Backporting is the action of taking a certain software modification (patch) and applying it to an older version of the software than it was initially created for. It is part of the maintenance step in a software development process.

The simplest and probably most common situation of backporting is a fixed security hole in a newer version of a piece of software. Consider this simplified example:

- Software v2.0 had a security vulnerability that is fixed by changing the text 'is_unsecured' to 'is_secured'.
- The same security hole exists in Software v1.0, from which the codebase for the newer version is derived, but there the text is called 'is_notsecure'.

By taking the modification that fixes Software v2.0 and changing it so that it applies to Software v1.0, one has effectively backported the fix.

In real life situations, the modifications that a single aspect of the software has undergone may be simple (only a few lines of code have changed) up to heavy and massive (many modifications spread across multiple files of the code). In the latter case, backporting is tedious and inefficient and should only be undergone if the older version of the software is really needed in favour of the newer (if, for example, the newer version still suffers stability problems that prevent it from being used in mission-critical situations).

The process of backporting can roughly be divided into these steps:

- Identification of the problem in the older version of the software that needs to be fixed by a backport

- Finding out which (usually recent) modification of the code fixed the problem
- Adapting the modification to the old code situation (the proper backporting)
- One or several levels of quality control—testing whether the backported version maintains previous functionality as well as if it properly implements the new functionality

Usually, multiple such modifications are bundled in a patch set.

Backports can be provided by the core developer group of the software. Since backporting needs access to the source code of a piece of software, this is the only way that backporting is done for closed source software—the backports will usually be incorporated in binary upgrades along the old version line of the software. With open-source software, backports are sometimes created by software distributors and later sent upstream (that is, submitted to the core developers of the afflicted software).

Chapter 6

Software Modernization

Legacy Modernization, or Software modernization, refers to the conversion, rewriting or porting of a legacy system to a modern computer programming language, software libraries, protocols, or hardware platform. Legacy transformation aims to retain and extend the value of the legacy investment through migration to new platforms.

Strategies

Legacy system modernization is often a large, multiyear project. Because these legacy systems are often critical in the operations of most enterprises, deploying the modernized system all at once introduces an unacceptable level of operational risk. As a result, legacy systems are typically modernized incrementally. Initially, the system consists completely of legacy code. As each increment is completed, the percentage of legacy code decreases. Eventually, the system is completely modernized. A migration strategy must ensure that the system remains fully functional during the modernization effort.

Making of software modernization decisions is a process within some organizational context. “Real world” decision making in business organizations often have to be made based on “bounded rationality” (Simon, 1983). Besides that there exists multiple (and possibly conflicting) decision criteria, the certainty, completeness, and availability of useful information (as a basis for the decision) is often limited.

MODERNIZATION STRATEGIES AND BENEFITS:

WMU (Warrants, Maintenance, Upgrade, Sahin & Zahedi, 2001b) is a model for choosing appropriate maintenance strategies based on aspired customer satisfaction level and their effects on it.

SABA (Bennett et al., 1999), is a high-level framework for planning the evolution and migration of legacy systems taking into account both organizational and technical issues.

MODERNIZATION RISK MANAGEMENT:

RPFA (Reengineering Project Failure Analysis, Bergey et al., 1999) is basically a check-list of potential problems related to reengineering projects, and of the corresponding appropriate technical and other means to react to the situation.

RMM (Risk-Managed Modernization, Seacord et al., 2003) is a new, general software modernization management approach taking risks (and both technological and business objectives) explicitly into account.

MODERNIZATION COSTS:

Softcalc (Sneed, 1995a) is a model and tool for estimating costs of incoming maintenance requests, developed based on COCOMO and FPA.

EMEE (Early Maintenance Effort Estimation, De Lucia et al., 2002;2001) is a new approach for quick maintenance effort estimation before starting the actual maintenance.

RENAISSANCE: A method to support system evolution by first recovering a stable basis using reengineering, and subsequently continuously improving the system by a stream of incremental changes. The approach integrates successfully with different project management processes

CHALLENGES IN LEGACY MODERNIZATION

Typical legacy systems have been in existence for more than two decades. Migrating is fraught with challenges:

```
Organizational change management – Users must be re-trained and
equipped to use and   understand the new applications and platforms
effectively
```

Coexistence of legacy and new systems – Organizations with a large footprint of legacy systems cannot migrate at once. A phased modernization approach needs to be adopted. However, this brings its own set of challenges like providing complete business coverage with well understood and implemented overlapping functionality, data duplication; throw away systems to bridge legacy and new systems needed during the interim phases.

MODERNIZATION OPTIONS

Over the years, several different options have come into being for legacy modernization – each of them met with varying success and adoption. Even now, there are a range of possibilities, as explained below, and there is no “the option” for all legacy transformation initiatives.

Migration: Migration of languages (3GL or 4GL), databases (legacy to RDBMS, and one RDBMS to another), platform (from one OS to another OS), often using automated

parsers and converters for high efficiency. This is quick and cost-effective way of transforming legacy systems.

Re-engineering: A technique to rebuild legacy applications in a new technology or platform, with same or enhanced functionality – usually by adopting Services Oriented Architecture (SOA). This is the most efficient and agile way of transforming legacy applications.

Re-hosting: Running the legacy applications, with no major changes, on a different platform. This is often used as an intermediate step to eliminate legacy and expensive hardware. Most common examples include mainframe applications being rehosted on UNIX or Wintel platform.

Package Implementation: Replacement of legacy applications, in whole or part, with off-the-shelf software (COTS) such as ERP, CRM, SCM, Billing software etc.

A legacy code is any application based on older technologies and hardware, such as mainframes, that continues to provide core services to an organization. Legacy applications are frequently large and difficult to modify, and scrapping or replacing them often means re-engineering an organization's business processes as well. However, more and more applications that were written in so called modern languages like java are becoming legacy. Whereas 'legacy' languages such as Cobol are top on the list for what would be considered legacy, newer languages can be just as monolithic, hard to modify, and thus, be candidates of modernization projects.

Re-implementing applications on new platforms in this way can reduce operational costs, and the additional capabilities of new technologies can provide access to functions such as web services and integrated development environments. Once transformation is complete and functional equivalence has been reached the applications can be aligned more closely to current and future business needs through the addition of new functionality to the transformed application. The recent development of new technologies such as program transformation by software modernization enterprises have made the legacy transformation process a cost-effective and accurate way to preserve legacy investments and thereby avoid the costs and business impact of migration to entirely new software.

The goal of legacy transformation is to retain the value of the legacy asset on the new platform. In practice this transformation can take several forms. For example, it might involve translation of the source code, or some level of re-use of existing code plus a Web-to-host capability to provide the customer access required by the business. If a rewrite is necessary, then the existing business rules can be extracted to form part of the statement of requirements for a rewrite.

Software Migration

Software migration is the process of moving from the use of one operating environment to another operating environment that is, in most cases, is thought to be a better one. For example, moving from Windows NT Server to Windows 2000 Server would usually be considered a migration because it involves making sure that new features are exploited, old settings do not require changing, and taking steps to ensure that current applications continue to work in the new environment. Migration could also mean moving from Windows NT to a UNIX-based operating system (or the reverse). Migration can involve moving to new hardware, new software, or both. Migration can be small-scale, such as migrating a single system, or large-scale, involving many systems, new applications, or a redesigned network.

One can migrate data from one kind of database to another kind of database. This usually requires the data into some common format that can be output from the old database and input into the new database. Since the new database may be organized differently, it may be necessary to write a program that can process the migrating files.

When a software migration reaches functional equivalence, the migrated application can be aligned more closely to current and future business needs through the addition of new functionality to the transformed application.

The migration of installed software from an old PC to a new PC can be done with a software migration tool. Migration is also used to refer simply to the process of moving data from one storage device to another.

Commercial Application Modernization Software

The following companies provide software that facilitates legacy application modernization--generally from hosted or client/server architectures to the Internet.

- Alchemy Solutions
- actifsource
- ATERAS
- ATX Technologies Ltd
- carrara engineering
- Datatek, Inc.
- DBBest
- HP HP Application Lifecycle Management
- IBM Rational Application Migration Solutions
- Innobec Technologies inc
- LANSA, Inc
- looksoftware
- Metex Inc.
- Micro_Focus_International
- michaels, ross & cole ltd. (mrc)

- Nexaweb
- Seagull Software
- Semantic Designs
- Software Revolution, Inc
- Software_AG
- Surround Tech
- Synchrony Systems, Inc
- Visionet Systems
- Legacy Application Modernization
- The Modernization Experts
- Zylog Systems (Europe) Ltd

Articles, papers and books

Creating Reusable Software

Due to the evolution of technology today some companies or group of people don't know the importance of legacy system. Because some of their functions are too important to be left unused and too expensive to reproduce again. The software industry and researchers have recently paid more attention towards the component based software development to enhance the productivity and accelerate time to market

Risk-Managed Modernization

In general, three classes of information system technology are of interest in legacy system modernization: Technologies used to construct the legacy systems, including the languages and database systems. Modern technologies, which often represent nirvana to those mired in decades-old technology and which hold (the often unfulfilled) promise of powerful, effective, easily maintained enterprise information systems. Technologies offered by the legacy system vendors. These technologies provide an upgrade path for those too timid or wise to jump head-first into the latest wave of IT offerings. Legacy system vendors offer these technologies for one simple reason: to provide an upgrade path for system modernization that does not necessitate leaving the comfort of the "mainframe womb." Although these technologies can provide a smoother road toward a modern system, they often result in an acceptable solution that falls short of the ideal.

Chapter 7

Legacy System

A **legacy system** is an old method, technology, computer system, or application program that continues to be used, typically because it still functions for the users' needs, even though newer technology or more efficient methods of performing a task are now available. A legacy system may include procedures or terminology which are no longer relevant in the current context, and may hinder or confuse understanding of the methods or technologies used.

The term "legacy" may have little to do with the size or age of the system — mainframes run 64-bit Linux and Java alongside 1960s vintage code.

Although the term is most commonly used to describe computers and software, it may also be used to describe human behaviors, methods, and tools. For example, timber framing using wattle and daub is a legacy building construction method.

Overview

Organizations can have compelling reasons for keeping a legacy system, such as:

- The system works satisfactorily, and the owner sees no reason for changing it.
- The costs of redesigning or replacing the system are prohibitive because it is large, monolithic, and/or complex.
- Retraining on a new system would be costly in lost time and money, compared to the anticipated appreciable benefits of replacing it (which may be zero).
- The system requires close to 100 percent availability, so it cannot be taken out of service, and the cost of designing a new system with a similar availability level is high. Examples include systems to handle customers' accounts in banks, computer reservation systems, air traffic control, energy distribution (power grids), nuclear power plants, military defense installations, and systems such as the TOPS database.

- The way that the system works is not well understood. Such a situation can occur when the designers of the system have left the organization, and the system has either not been fully documented or documentation has been lost.
- The user expects that the system can easily be replaced when this becomes necessary.

NASA example

NASA's Space Shuttle program still uses a large amount of 1970s-era technology. Replacement is cost-prohibitive because of the expensive requirement for flight certification; the legacy hardware currently being used has completed the expensive integration and certification requirement for flight, but any new equipment would have to go through that entire process – requiring extensive tests of the new components in their new configurations – before a single unit could be used in the Space Shuttle program. This would make any new system that started the certification process a *de facto* legacy system by the time of completion.

Additionally, because the entire Space Shuttle system, including ground and launch vehicle assets, was designed to work together as a closed system, and the specifications have not changed, all of the certified systems and components still serve well in the roles for which they were designed. It was advantageous for NASA – even before the Shuttle was scheduled to be retired in 2010 – to keep using many pieces of 1970s technology rather than to upgrade those systems.

Potential problems

Legacy systems are considered to be potentially problematic by many software engineers for several reasons.

- Legacy systems often run on obsolete (and usually slow) hardware, and spare parts for such computers may become increasingly difficult to obtain.
- If legacy software runs on only antiquated hardware, the cost of maintaining the system may eventually outweigh the cost of replacing both the software and hardware unless some form of emulation or backward compatibility allows the software to run on new hardware.
- These systems can be hard to maintain, improve, and expand because there is a general lack of understanding of the system; the staff who were experts on it have retired or forgotten what they knew about it, and staff who entered the field after it became "legacy" never learned about it in the first place. This can be worsened by lack or loss of documentation. A regional airline fired its CEO in 2004 due to the failure of an antiquated legacy crew scheduling system that ran into a limitation not known to anyone in the company.
- Integration with newer systems may also be difficult because new software may use completely different technologies. The kind of bridge hardware and software that becomes available for different technologies that are popular at the same time are often not developed for differing technologies in different times, because of

the lack of a large demand for it and the lack of associated reward of a large market economies of scale, though some of this "glue" does get developed by vendors and enthusiasts of particular legacy technologies (often called "retrocomputing" communities).

Improvements on legacy software systems

Where it is impossible to replace legacy systems through the practice of application retirement, it is still possible to enhance them. Most development often goes into adding new interfaces to a legacy system. The most prominent technique is to provide a Web-based interface to a terminal-based mainframe application. This may reduce staff productivity due to slower response times and slower mouse-based operator actions, yet it is often seen as an "upgrade", because the interface style is familiar to unskilled users and is easy for them to use. John McCormick discusses such strategies that involve middleware.

Printing improvements are problematic because legacy software systems often add no formatting instructions, or they use protocols that are not usable in modern PC/Windows printers. A print server can be used to intercept the data and translate it to a more modern code. Rich Text Format (RTF) or PostScript documents may be created in the legacy application and then interpreted at a PC before being printed.

Biometric security measures are difficult to implement on legacy systems. A workable solution is to use a telnet or http proxy server to sit between users and the mainframe to implement secure access to the legacy application.

The change being undertaken in some organizations is to switch to Automated Business Process (ABP) software which generates complete systems. These systems can then interface to the organizations' legacy systems and use them as data repositories. This approach can provide a number of significant benefits: the users are insulated from the inefficiencies of their legacy systems, and the changes can be incorporated quickly and easily in the ABP software .

Legacy support

The term *legacy support* is often used with reference to obsolete or legacy computer hardware, whether peripherals or core components. Operating systems with "legacy support" can detect and use legacy hardware.

It is also used as a verb for what vendors do for products in legacy mode – they "support", or provide software maintenance, for older products. A "legacy" product may have some advantage over a modern product, even if not one that causes a majority of the market to favor it over the newer offering. A product is only truly "obsolete" if it has an advantage to nobody – if no person making a rational decision would choose to acquire it new.

In some cases, "legacy mode" refers more specifically to backward compatibility.

The computer mainframe era saw many applications running in legacy mode. In the modern business computing environment, n-tier, or 3-tier architectures are more difficult to place into legacy mode as they include many components making up a single system. Government regulatory changes must also be considered in a system running in legacy mode.

Virtualization technology allows for a resurgence of modern software applications entering legacy mode.

Brownfield architecture

IT has borrowed the term *brownfield* from the building industry, where undeveloped land (and especially unpolluted land) is described as *greenfield* and previously developed land – which is often polluted and abandoned – is described as *brownfield*.

- A *brownfield architecture* is an IT network design that incorporates legacy systems.
- A *brownfield deployment* is an upgrade or addition to an existing IT network and uses some legacy components.

Alternative view

There is an alternate point of view — growing since the "Dot Com" bubble burst in 1999 — that legacy systems are simply computer systems that are both installed and working. In other words, the term is not pejorative, but the opposite. Bjarne Stroustrup, creator of the C++ language, addressed this issue succinctly:

"Legacy code" often differs from its suggested alternative by actually working and scaling.
—Bjarne Stroustrup

IT analysts estimate that the cost to replace business logic is about five times that of reuse, and that's not counting the risks involved in wholesale replacement. Ideally, businesses would never have to rewrite most core business logic; debits must equal credits — they always have, and they always will. New software may increase the risk of system failures and security breaches.

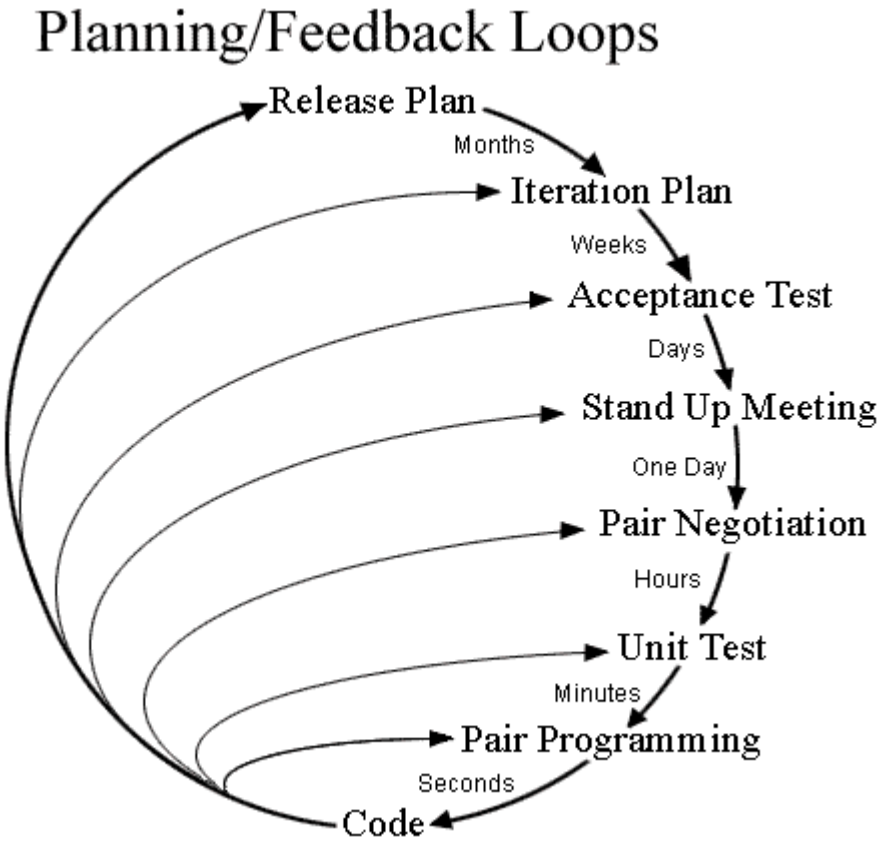
The IT industry is responding to these concerns. "Legacy modernization" and "legacy transformation" refer to the act of reusing and refactoring existing, core business logic by providing new user interfaces (typically Web interfaces), sometimes through the use of techniques such as screen scraping and service-enabled access (e.g., through Web services). These techniques allow organisations to understand their existing code assets (using discovery tools), provide new user and application interfaces to existing code,

improve workflow, contain costs, minimize risk, and enjoy classic qualities of service (near 100% uptime, security, scalability, etc.).

The reexamination of attitudes toward legacy systems is also inviting more reflection on what makes legacy systems as durable as they are. Technologists are relearning the fact that sound architecture, practiced up front, helps businesses avoid costly and risky rewrites in the first place. The most common legacy systems tend to be those which embraced well-known IT architectural principles, with careful planning and strict methodology during implementation. Poorly designed systems often don't last, both because they wear out and because their reliability or usability are low enough that no one is inclined to make an effort to extend their term of service when replacement is an option. Thus, many organizations are rediscovering the value of both their legacy systems themselves and those systems' philosophical underpinnings.

Chapter 8

Extreme Programming



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.

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 9

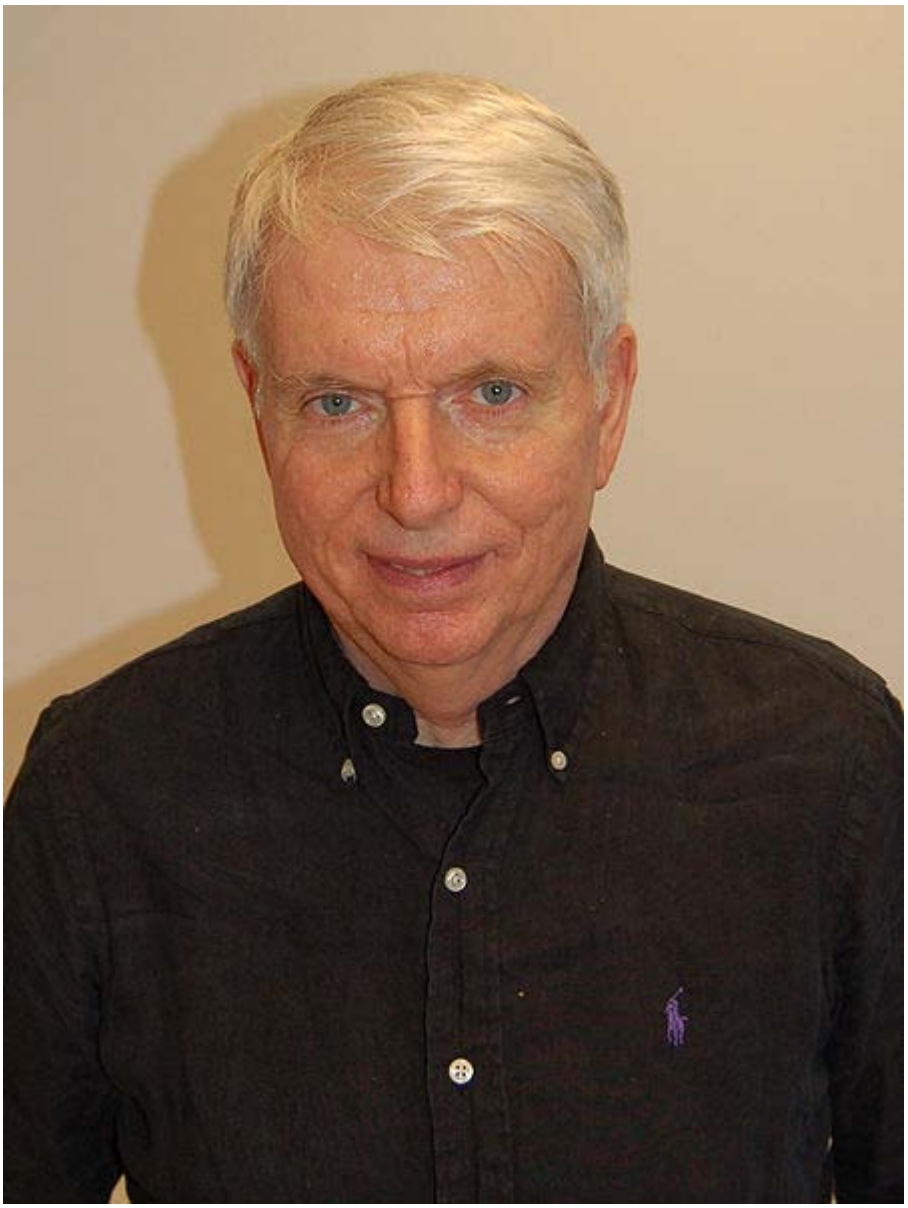
Agile Software Development

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

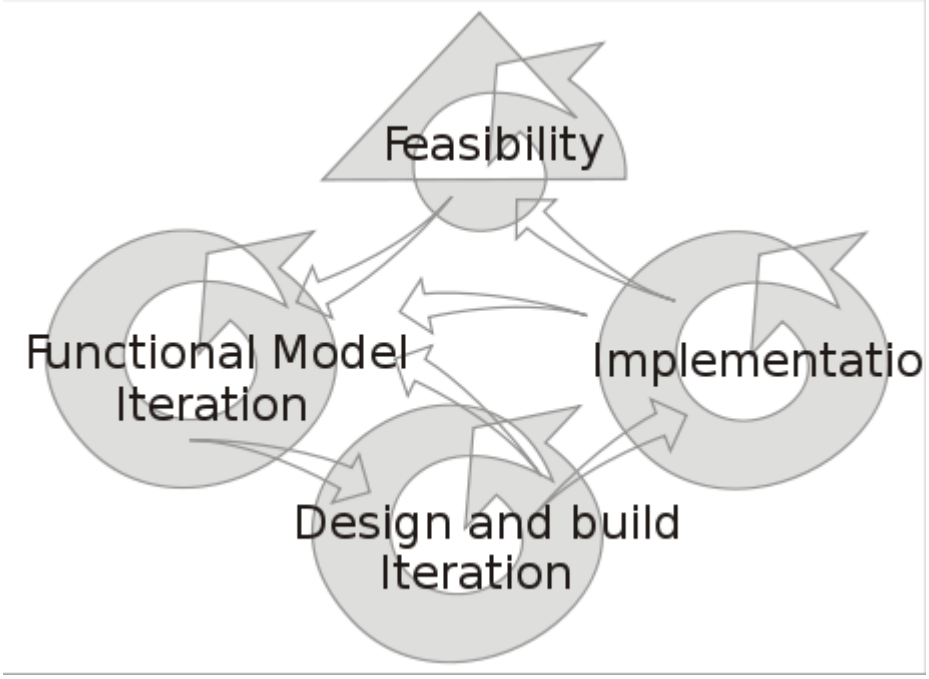
peer-reviewed experience report track. The experience reports share industry experiences with agile software development.

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 10

Dynamic Systems Development Method



Model of the DSDM software development process.

Dynamic Systems Development Method (**DSDM**) is primarily a software development methodology originally based upon the Rapid Application Development methodology. In 2007 DSDM became a generic approach to project management and solution delivery. DSDM is an iterative and incremental approach that emphasizes continuous user/customer involvement.

Its goal is to deliver projects on time and on budget while adjusting for changing requirements along the way. DSDM is one of a number of Agile methods for developing software and non-I.T. solutions, and it forms a part of the Agile Alliance.

The most recent version of DSDM is called DSDM Atern. The name Atern is a shortening of Arctic Tern - a collaborative bird that can travel vast distances and epitomises many facets of the method which are natural ways of working e.g. prioritisation and collaboration.

The previous version of DSDM (released in May 2003) which is still widely used and is still valid is DSDM 4.2 which is a slightly extended version of DSDM version 4. The extended version contains guidance on how to use DSDM with Extreme Programming.

Overview of DSDM Atern

In 2007 a team set up by the DSDM Consortium looked into the content of DSDM and decided that the underlying mechanics and structure were completely sound but that the terminology and the focus purely on I.T. applications should be updated to meet the needs of projects in the new millenium. Some of the content of the method had been there since 1995. The new version was launched at the Cafe Royale in London on 24 April 2007.

Overview of DSDM version 4.2

As an extension of rapid application development, DSDM focuses on Information Systems projects that are characterised by tight schedules and budgets. DSDM addresses the most common failures of information systems projects, including exceeding budgets, missing deadlines, and lack of user involvement and top-management commitment. By encouraging the use of RAD, however, careless adoption of DSDM may increase the risk of cutting too many corners. DSDM consists of

- Three phases: pre-project phase, project life-cycle phase, and post-project phase.
- A project life-cycle phase is subdivided into 5 stages: feasibility study, business study, functional model iteration, design and build iteration, and implementation.

In some circumstances, there are possibilities to integrate practices from other methodologies, such as Rational Unified Process (RUP), Extreme Programming (XP), and PRINCE2, as complements to DSDM. Another agile method that has some similarity in process and concept to DSDM is Scrum.

DSDM was developed in the United Kingdom in the 1990s by the DSDM Consortium, an association of vendors and experts in the field of software engineering created with the objective of "jointly developing and promoting an independent RAD framework" by combining their best practice experiences. The DSDM Consortium is a not-for-profit, vendor-independent organisation which owns and administers the DSDM framework. The first version was completed in January 1995 and was published in February 1995. In

July 2006, DSDM Public Version 4.2 was made available for individuals to view and use; however, anyone reselling DSDM must still be a member of the not-for-profit consortium.

The DSDM approach

Principles

There are nine underlying principles consisting of four foundations and five starting-points.

- User involvement is the main key in running an efficient and effective project, where both users and developers share a workplace, so that the decisions can be made accurately.
- The project team must be empowered to make decisions that are important to the progress of the project without waiting for higher-level approval.
- A focus on frequent delivery of products, with assumption that to deliver something "good enough" earlier is always better than to deliver everything "perfectly" in the end. By delivering product frequently from an early stage of the project, the product can be tested and reviewed where the test record and review document can be taken into account at the next iteration or phase.
- The main criteria for acceptance of a "deliverable" is delivering a system that addresses the current business needs. Delivering a perfect system which addresses all possible business needs is less important than focusing on critical functionalities.
- Development is iterative and incremental and driven by users' feedback to converge on an effective business solution.
- All changes during the development are reversible.
- The high level scope and requirements should be base-lined before the project starts.
- Testing is carried out throughout the project life-cycle.
- Communication and cooperation among all project stakeholders is required to be efficient and effective.

Prerequisites for using DSDM

In order for DSDM to be a success, a number of prerequisites need to be realised. First, there needs to be interactivity between the project team, future end users and higher management. This addresses well known failures of IS development projects due to lack of top management motivation and/or user involvement. The second prerequisite for DSDM projects is that the project can be decomposed in to smaller parts enabling the use of an iterative approach.

Two examples of types of projects for which DSDM is not considered well-suited are:

- Safety-critical projects - the extensive testing and validation found in safety-critical projects conflict with DSDM goals of being on time and on budget.
- Projects that aim to produce re-usable components - the demands on perfection are often too high and conflict with the 80%/20% principle described earlier.

DSDM Project Life-cycle

Overview : three phases of DSDM

The DSDM framework consists of three sequential phases, namely the pre-project, project life-cycle and post-project phases. The project phase of DSDM is the most elaborate of the three phases. The project life-cycle phase consists of 5 stages that form an iterative step-by-step approach in developing an IS. The three phases and corresponding stages are explained extensively in the subsequent sections. For each stage/phase, the most important activities are addressed and the deliverables are mentioned.

Phase 1 - The Pre-project

In the pre-project phase candidate projects are identified, project funding is realised and project commitment is ensured. Handling these issues at an early stage avoids problems at later stages of the project like cows.

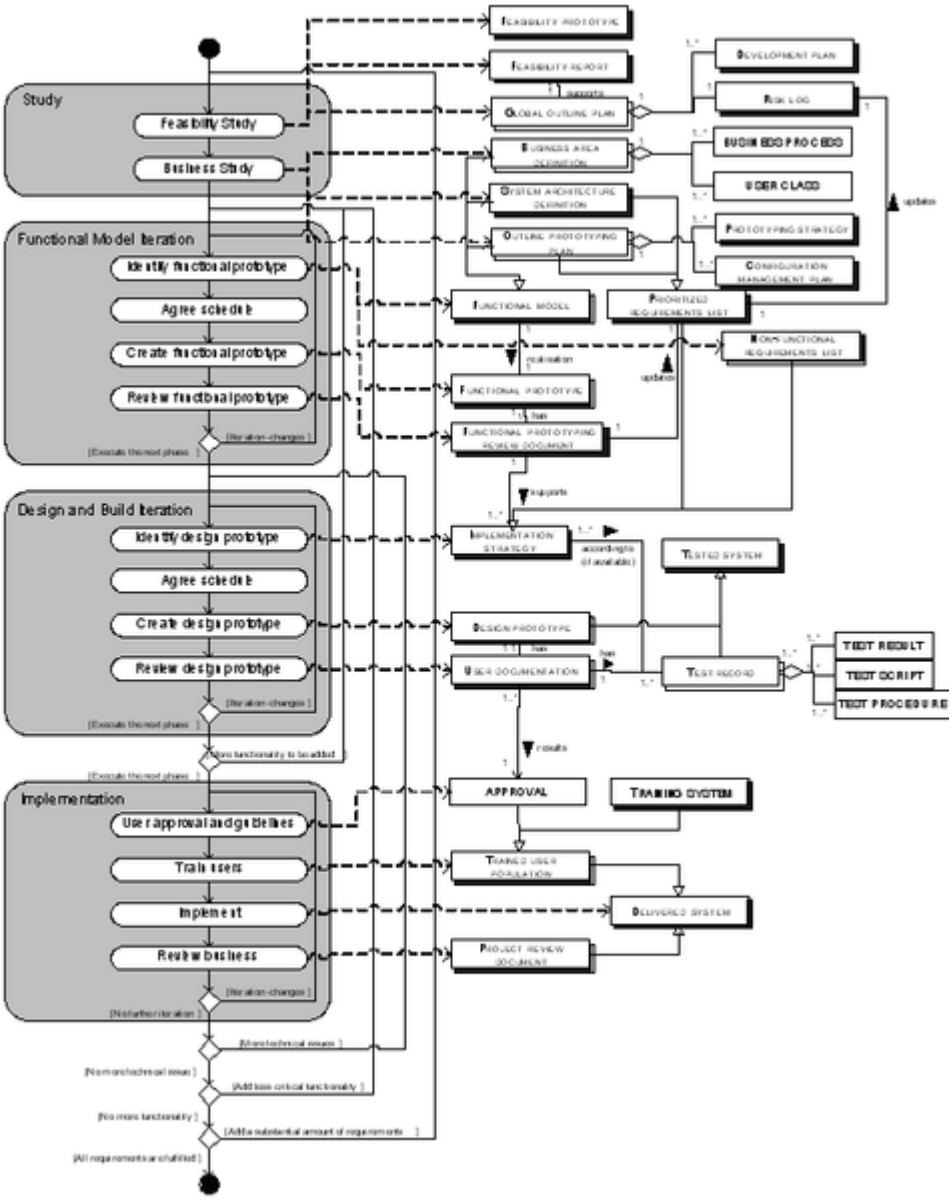
Phase 2 - The Project life-cycle

The process overview in the figure below shows the project life-cycle of this phase of DSDM. It depicts the 5 stages a project will have to go through to create an IS. The first two stages, the Feasibility Study and Business Study are sequential phases that complement to each other. After these phases have been concluded, the system is developed iteratively and incrementally in the Functional Model Iteration, Design & Build Iteration and Implementation stages. The iterative and incremental nature of DSDM will be addressed further in a later section.

Phase 3 - Post-project

The post-project phase ensures the system operates effectively and efficiently. This is realised by maintenance, enhancements and fixes according to DSDM principles. The maintenance can be viewed as continuing development based on the iterative and incremental nature of DSDM. Instead of finishing the project in one cycle usually the project can return to the previous phases or stages so that the previous step and the deliverable products can be refined.

Below is the process-data diagram of DSDM as a whole Project life-cycle with all of its four steps. This diagram depicts the DSDM iterative development, started on functional model iteration, design and build iteration, and implementation phase. The description of each stage will be explained later in this entry.



The process-data diagram of DSDM Project Life-cycle

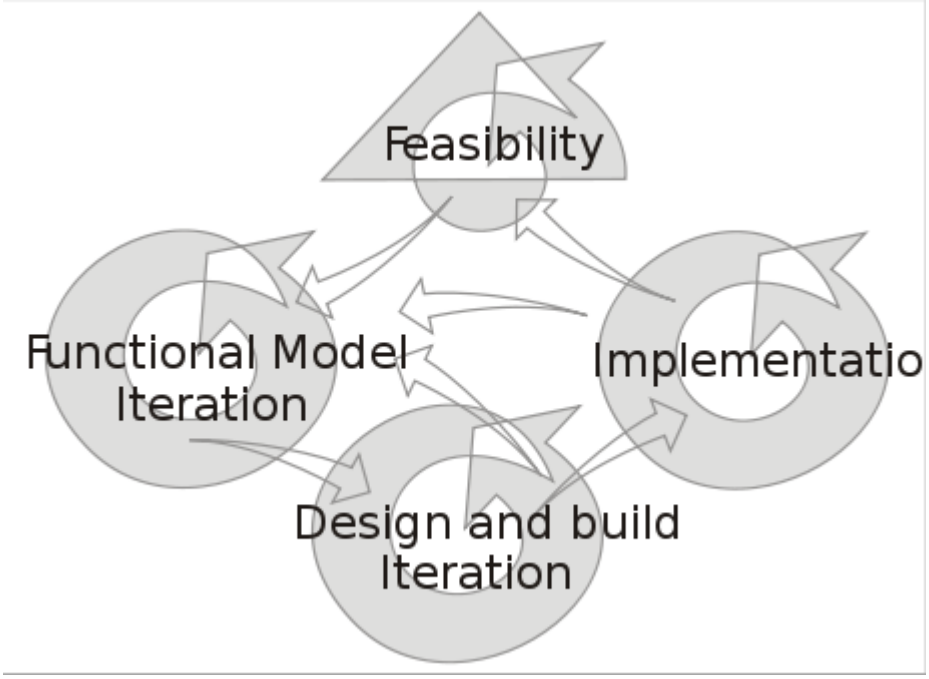
The four steps of the DSDM Project Life-cycle

Activity	Sub activity	Description
Study	Feasibility Study	Stage where the suitability of DSDM is assessed. Judging by the type of project, organisational and people issues, the decision is made, whether to use DSDM or not. Therefore it will generate a FEASIBILITY REPORT, a FEASIBILITY PROTOTYPE, and a GLOBAL

		OUTLINE PLAN which includes a DEVELOPMENT PLAN and a RISK LOG.
	Business Study	Stage where the essential characteristics of business and technology are analysed. Approach to organise workshops, where a sufficient number of the customer's experts are gathered to be able to consider all relevant facets of the system, and to be able to agree on development priorities. In this stage, a PRIORITISED REQUIREMENTS LIST, a BUSINESS AREA DEFINITION, a SYSTEM ARCHITECTURE DEFINITION, and an OUTLINE PROTOTYPING PLAN are developed.
Functional Model Iteration	Identify functional prototype	Determine the functionalities to be implemented in the prototype that results from this iteration. In this sub-stage, a FUNCTIONAL MODEL is developed according to the deliverables result of business study stage.
	Agree schedule	Agree on how and when to develop these functionalities.
	Create functional prototype	Develop the FUNCTIONAL PROTOTYPE, according to the agreed schedule and FUNCTIONAL MODEL.
	Review functional prototype	Check correctness of the developed prototype. This can be done via testing by end-user and/or reviewing documentation. The deliverable is a FUNCTIONAL PROTOTYPING REVIEW DOCUMENT.
Design and Build Iteration	Identify design prototype	Identify functional and non-functional requirements that need to be in the tested system. And based on these identifications, an IMPLEMENTATION STRATEGY is involved. If there is a TEST RECORD from the previous iteration, then it will be also used to determine the IMPLEMENTATION STRATEGY.
	Agree schedule	Agree on how and when to realise these requirements.
	Create design prototype	Create a system (DESIGN PROTOTYPE) that can safely be handed to end-users for daily use, also for testing purposes.

	Review design prototype	Check the correctness of the designed system. Again testing and reviewing are the main techniques used. An USER DOCUMENTATION and a TEST RECORD will be developed.
Implementation	User approval and guidelines	End users approve the tested system (APPROVAL) for implementation and guidelines with respect to the implementation and use of the system are created.
	Train users	Train future end user in the use of the system. TRAINED USER POPULATION is the deliverable of this sub-stage.
	Implement	Implement the tested system at the location of the end users, called as DELIVERED SYSTEM.
	Review business	Review the impact of the implemented system on the business, a central issue will be whether the system meets the goals set at the beginning of the project. Depending on this the project goes to the next stage, the post-project or loops back to one of the preceding stages for further development. This review is will be documented in a PROJECT REVIEW DOCUMENT.

Four stages of the Project life-cycle



Model of the DSDM software development process.

Stage 1A: The Feasibility Study

During this stage of the project, the feasibility of the project for the use of DSDM is examined. Prerequisites for the use of DSDM are addressed by answering questions like: 'Can this project meet the required business needs?', 'Is this project suited for the use of DSDM?' and 'What are the most important risks involved?'. The most important techniques used in this phase are the Workshops.

The deliverables for this stage are the Feasibility Report and the Feasibility Prototype that address the feasibility of the project at hand. It is extended with a Global Outline Plan for the rest of the project and a Risk Log that identifies the most important risks for the project.

Stage 1B: The Business Study

The business study extends the feasibility study. After the project has been deemed feasible for the use of DSDM, this stage examines the influenced business processes, user groups involved and their respective needs and wishes. Again the workshops are one of the most valuable techniques, workshops in which the different stakeholders come together to discuss the proposed system. The information from these sessions is combined into a requirements list. An important property of the requirements list is the fact that the requirements are (can be) prioritised. These requirements are prioritised using the MoSCoW approach. Based on this prioritisation, a development plan is constructed as a guideline for the rest of the **project.**

An important project technique used in the development of this plan is timeboxing. This technique is essential in realising the goals of DSDM, namely being on time and on budget, guaranteeing the desired quality. A system architecture is another aid to guide the development of the IS.The deliverables for this stage are a business area definition that describes the context of the project within the company, a system architecture definition that provides an initial global architecture of the IS under development together with a development plan that outlines the most important steps in the development process. At the base of these last two documents there is the prioritised requirements list. This list states all the requirements for the system, organised according to the MoSCoW principle. And last the Risk Log is updated with the facts that have been identified during this phase of DSDM.

Stage 2: Functional Model Iteration

The requirements that have been identified in the previous stages are converted to a functional model. This model consists of both a functioning prototype and models. Prototyping is one of the key project techniques within this stage that helps to realise good user involvement throughout the project. The developed prototype is reviewed by different user groups. In order to assure quality, testing is implemented throughout every iteration of DSDM. An important part of testing is realised in the Functional Model Iteration. The Functional Model can be subdivided into four sub-stages:

- Identify Functional Prototype: Determine the functionalities to be implemented in the prototype that results from this iteration.
- Agree Schedule: Agree on how and when to develop these functionalities.
- Create Functional Prototype: Develop the prototype. Investigate, refine, and consolidate it with the combined Functional prototype of previous iterations.
- Review Prototype: Check the correctness of the developed prototype. This can be done via testing by end-user, then use the test records and user's feedbacks to generate the functional prototyping review document.

The deliverables for this stage are a Functional Model and a Functional Prototype that together represent the functionalities that could be realised in this iteration, ready for testing by users. Next to this, the Requirements List is updated, deleting the items that have been realised and rethinking the prioritisation of the remaining requirements. The Risk Log is also updated by having risk analysis of further development after reviewing the prototyping document.

Stage 3: Design and Build Iteration

The main focus of this DSDM iteration is to integrate the functional components from the previous phase into one system that satisfies user needs. It also addresses the non-functional requirements that have been set for the IS. Again testing is an important ongoing activity in this stage. The Design and Build Iteration can be subdivided into four sub-stages:

- Identify Design Prototype: Identify functional and non-functional requirements that need to be in the tested system.
- Agree Schedule: Agree on how and when to realise these requirements.
- Create Design Prototype: Create a system that can safely be handed to end-users for daily use. They investigate, refine, and consolidate the prototype of current iteration within prototyping process are also important in this sub-stage.
- Review Design Prototype: Check the correctness of the designed system. Again testing and reviewing are the main techniques used, since the test records and user's feedbacks are important to generate the user documentation.

The deliverables for this stage are a Design Prototype during the phase that end users get to test and at the end of the Design and Build Iteration the Tested System is handed over to the next phase. In this stage, the system is mainly built where the design and functions are consolidated and integrated in a prototype. Another deliverable for this stage is a User Documentation.

Stage 4: Implementation

In the Implementation stage, the tested system including user documentation is delivered to the users and training of future users is realised. The system to be delivered has been reviewed to include the requirements that have been set in the beginning stages of the project. The Implementation stage can be subdivided into four sub-stages:

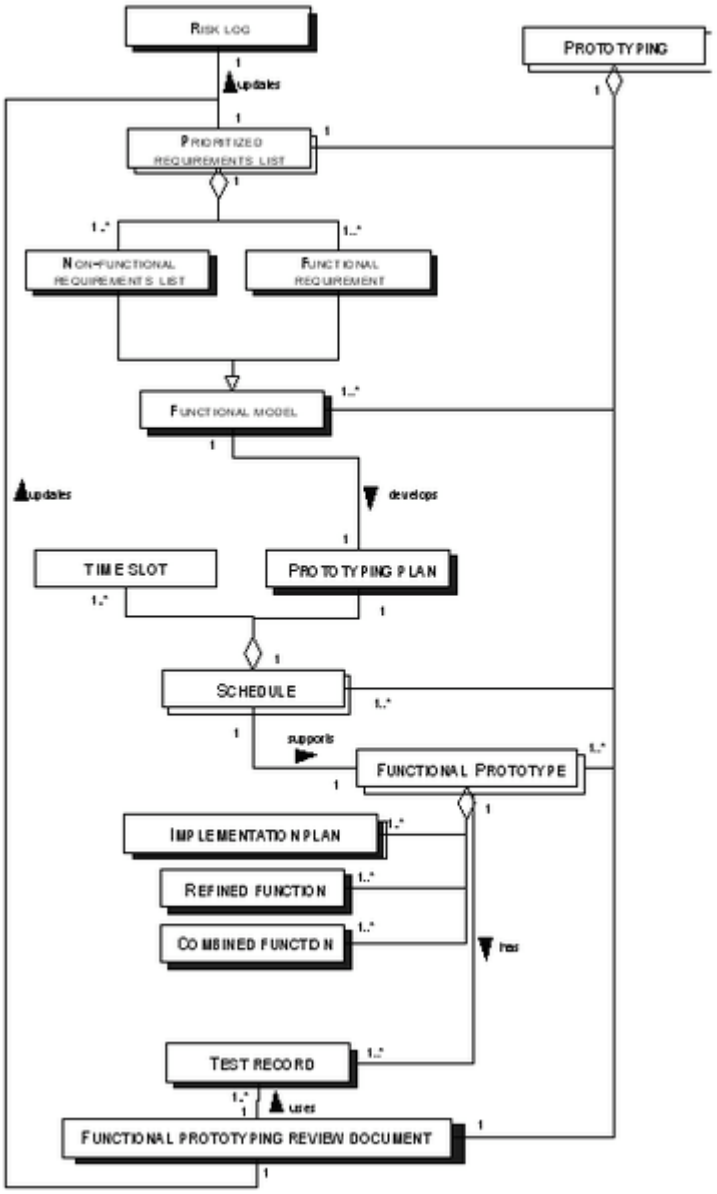
- User Approval and Guidelines: End users approve the tested system for implementation and guidelines with respect to the implementation and use of the system are created.
- Train Users: Train future end user in the use of the system.
- Implement: Implement the tested system at the location of the end users.
- Review Business: Review the impact of the implemented system on the business, a central issue will be whether the system meets the goals set at the beginning of the project. Depending on this the project goes to the next phase, the post-project or loops back to one of the preceding phases for further development.

The deliverables for this stage are a Delivered System on location, ready for use by the end users, Trained Users and detailed Project Review Document of the system.

DSDM Functional Model Iteration

Meta-data model

The associations between concepts of deliverables in Functional Model Iteration stage are depicted in the meta-data model below. This meta-data model will be combined with the meta-process diagram of Functional Model Iteration phase in the next part.



Meta-data model of Functional Model Iteration

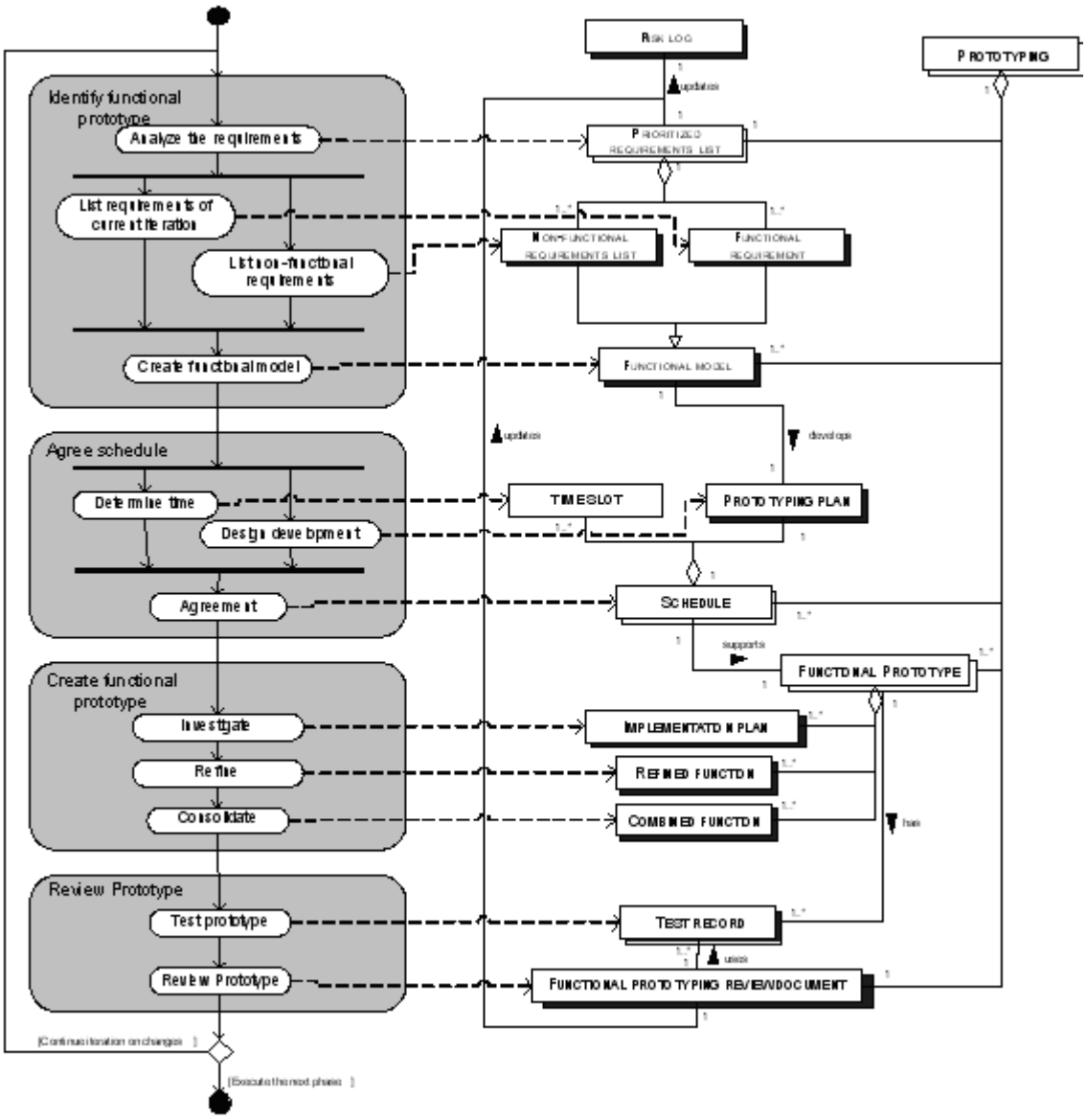
Concept	Definition
RISK LOG	Log of identified risk. Important since the next stage onward, encountered problem will be more difficult to address. This risk log will need to be updated continuously. (VTT Publication 478)
PRIORITISED REQUIREMENTS LIST	List of requirements based on its prioritisation. The prioritisation process is based on MoSCoW technique, to determine which requirements must be implemented first into the system (the ones that

	meet the business needs), and so on.
NON-FUNCTIONAL REQUIREMENTS LIST	List of non-functional requirements is mainly to be dealt in the next stage. (VTT Publication 478)
FUNCTIONAL REQUIREMENT	Function that is used to build the model and prototyping according to its priority.
FUNCTIONAL MODEL	Model that is built according to the functional requirements. It will be used in order to develop the functional prototype in the next sub-stage. This concept will be used to develop a PROTOTYPING PLAN.
PROTOTYPING	The process of quickly putting together a working model (a prototype) in order to test various aspects of the design, illustrate ideas or features and gather early user feedback.
TIME SLOT	The list of available times to do certain activities in order to perform the plan according to the schedule.
PROTOTYPING PLAN	Activities plan within prototyping process that will be performed in available time slots according to the schedule.
SCHEDULE	A set of activities plan with the related time agreed by the developers. This concept will be used to support the implementation of FUNCTIONAL PROTOTYPE.
FUNCTIONAL PROTOTYPE	A prototype of the functions the system should perform and how it should perform them.
IMPLEMENTATION PLAN	A preparation of activities to implement the functional prototyping according to the schedule and the prioritised requirements list.
REFINED FUNCTION	Function of prototype that is being refined within current iteration before it is combined to the others and tested.
COMBINED FUNCTION	Function of prototyped that is combined with the other functional prototypes of previous iteration. The new combination functional prototype will be tested in the next stage.

TEST RECORD	Record set of testing where the test script, test procedure, and test result are included. This test record is used to develop the FUNCTIONAL PROTOTYPING REVIEW DOCUMENT, and is also used indirectly to update the PRIORITISED REQUIREMENTS LIST.
FUNCTIONAL PROTOTYPING REVIEW DOCUMENT	It collects the users' comments about the current increment, working as input for subsequent iterations (VTT Publication 478). This review document will be used to update the RISK LOG and PRIORITISED REQUIREMENTS LIST.

Process-data model

Identify functional prototype activity is to identify the functionalities that would be in the prototype of current iteration. Recall that both, analysis and coding are done; prototypes are built, and the experiences gained from them are used in improving the analysis models (based also on updated prioritised requirements list and updated risk log). The built prototypes are not to be entirely discarded, but gradually steered towards such quality that they can be included in the final system. Agree schedule is to determine when and how the prototyping will be implemented; it extends the scope to the available timetable and prototyping plan. And since testing is implemented throughout the whole process, it's also an essential part of this phase, and therefore it is included in the Review Prototype activity right after the functional prototype is built and the test record will eventually be used in the review prototype process and generates the review document. Below is the process-data diagram of Functional Model Iteration stage.



Model of the Functional Model Iteration.

Activity	Sub activity	Description
Identify functional prototype	Analyse the requirements	The requirements of current prototype are analysed according to the prioritised requirements list that is previously developed (in previous iteration and/or in previous phase which is business study phase).
	List requirements of current iteration	Select the functional requirements that would be implemented in the current iteration's prototype, and list them in the FUNCTIONAL REQUIREMENT.

	List non-functional requirements	List the non-functional requirements of the system that is being developed in NON-FUNCTIONAL REQUIREMENTS LIST.
	Create functional model	Analysis model and prototype codes are included in this sub-activity to develop the FUNCTIONAL MODEL.
Agree schedule	Determine time	Identify possible timetable (TIME SLOT) to perform the prototyping activities according to the prototyping plan and prototyping strategy.
	Design development	The PROTOTYPING PLAN, including all prototyping activities that will be performed on available TIME SLOT.
	Agreement	The agreement SCHEDULE of when and how the prototyping activities should be performed.
Create functional prototype	Investigate	Investigate the requirements; analyse the FUNCTIONAL MODEL that has been built in earlier activity, and set the IMPLEMENTATION PLAN according to the analysis model, and will be used to build the prototype in the next sub-activity.
	Refine	Implement the FUNCTIONAL MODEL and IMPLEMENTATION PLAN to build a FUNCTIONAL PROTOTYPE. This prototype will be then refined before it is combined to the other functions. This prototype will be gradually steered towards such quality that it can be included in the final system (through refining process).
	Consolidate	Consolidate the refined FUNCTIONAL PROTOTYPE with the other prototype of previous iteration. The new combination FUNCTIONAL PROTOTYPE will be tested in the next activity.
Review prototype	Test prototype	The essential part of DSDM that needs to be included throughout the whole process of DSDM. The TEST RECORD will be used together with users' comments to develop the PROTOTYPING REVIEW DOCUMENT in the next activity of FMI phase.
	Review prototype	Collects the user comments and documentation. The test records will play an important role to develop this review report. Based on this FUNCTIONAL PROTOTYPING REVIEW DOCUMENT, the

		prioritised requirements list and risk log will be updated, and decide to set the next course whether or not to do another iteration of FMI phase.

Further DSDM topics

Core Techniques of DSDM

- Timeboxing

  Timeboxing is one of the project techniques of DSDM. It is used to support the main goals of DSDM to realise the development of an IS on time, within budget and with the desired quality. The main idea behind timeboxing is to split up the project in portions, each with a fixed budget and a delivery date. For each portion a number of requirements are selected that are prioritised according to the MoSCoW principle. Because time and budget are fixed, the only remaining variables are the requirements. So if a project is running out of time or money the requirements with the lowest priority are omitted. This does not mean that an unfinished product is delivered, because of the pareto principle that 80% of the project comes from 20% of the system requirements, so as long as those most important 20% of requirements are implemented into the system, the system therefore meets the business needs and that no system is built perfectly in the first try.

- MoSCoW

  MoSCoW represents a way of prioritising items. In the context of DSDM the MoSCoW technique is used to prioritise requirements. It is an acronym that stands for:

-

  MUST have this requirement to meet the business needs.
  SHOULD have this requirement if at all possible, but the project success does not rely on this.
  COULD have this requirement if it does not affect the fitness of business needs of the project.
  WOULD have this requirement at later date if there is some time left (or in the future development of the system).

- Prototyping

  This technique refers to the creation of prototypes of the system under development at an early stage of the project. It enables the early discovery of shortcomings in the system and allows future users to ‘test-drive’ the system. This

way good user involvement is realised, one of the key success factors of DSDM, or any System Development project for that matter.

- Testing

  A third important aspect of the goal of DSDM is the creation of an IS with good quality. In order to realise a solution of good quality, DSDM advocates testing throughout each iteration. Since DSDM is a tool and technique independent method, the project team is free to choose its own test management method, for example TMap.

- Workshop

  One of DSDM's project techniques that aims at bringing the different stakeholders of the project together to discuss requirements, functionalities and mutual understanding. In a workshop the stakeholders come together and discuss the project.

- Modeling

  This technique is essential and purposely used to visualise the diagrammatic representation of a specific aspect of the system or business area that is being developed. Modelling gives a better understanding for DSDM project team over a business domain.

- Configuration Management

  A good implementation of this configuration management technique is important for the dynamic nature of DSDM. Since there is more than one thing being handled at once during the development process of the system, and the products are being delivered frequently at a very fast rate, the products therefore need to be controlled strictly as they achieve (partial) completion.

Roles in DSDM

There are some roles introduced within DSDM environment. It is important that the project members need to be appointed to different roles before they start to run the project. Each role has its own responsibility. The roles are:

- **Executive Sponsor** So called the "Project Champion". An important role from the user organisation who has the ability and responsibility to commit appropriate funds and resources. This role has an ultimate power to make decisions.
-
- **Visionary** The one who has the responsibility to initialise the project by ensuring that essential requirements are found early on. Visionary has the most accurate

perception of the business objectives of the system and the project. Another task is to supervise and keep the development process in the right track.

- **Ambassador User** Brings the knowledge of user community into the project, ensures that the developers receive enough amount of user's feedbacks during the development process.
- **Advisor User** Can be any user that represents an important viewpoint and brings the daily knowledge of the project.
- **Project Manager** Can be anyone from user community or IT staff who manages the project in general.
- **Technical Co-ordinator** Responsible in designing the system architecture and control the technical quality in the project.
- **Team Leader** Leads his team and ensures that the team works effectively as a whole.
- **Developer** Interpret the system requirements and model it including developing the deliverable codes and build the prototypes.
- **Tester** Checks the correctness in a technical extents by performing some testings. Tester will have to give some comments and documentation.
- **Scribe** Responsible to gather and record the requirements, agreements, and decisions made in every workshop.
- **Facilitator** Responsible in managing the workshops progress, acts as a motor for preparation and communication.
- **Specialist Roles** Business Architect, Quality Manager, System Integrator, etc.

Iterative and Incremental Nature

Next to timeboxing and prioritising of requirements, the DSDM also provides an iterative and incremental approach to IS development. This can be seen in the figure depicting the Process Overview above.

The Functional Model Iteration, Design & Build Iteration and Implementation stages can go over their sub stages several times before entering the next stage. Each iteration addresses a set of new functionalities, and every iteration builds on a working predecessor. Each iteration can be undone if needed.

The Process Overview figure also shows arrows going back to previous stages. For example, there is an arrow from Implementation to the Business Study. If a big functionality has been discovered during development that couldn't be implemented, it might be possible to start all over by defining new requirements in a Business Study. Similarly, there is an arrow from Implementation to the Functional Model Iteration. Functionality might be omitted during a previous Functional Model Iteration because of time or budget constraints. The project should proceed into the post-project phase only when it meets all the requirements defined by the project and business goals.

Because of the iterative nature of DSDM, it is essential to maintain good requirements management and configuration management throughout the entire project. This ensures that the project does implement the requirements that were decided in the early phases of the project.

Meta-model (Meta-Modeling)

Meta-Modeling takes a higher level look at methods and techniques. In doing so it offers possibilities for comparing similar methods and techniques and engineering new methods from existing ones.

The Meta data model, depicted above, identifies the concepts and associations between these concepts within DSDM. As can be seen from the figure, two main concepts can be identified, namely the *Phase* and the *Flow* concept. Each *Flow* originates from a *Phase* within DSDM. Flows can be divided up in the sub concepts *Data* and *Product*. This subdivision is denoted with a *C*, which means that the subdivision is disjoint and complete. In other words, a *Flow* is always either a *Data Flow* or a *Product Flow*, but never both. In the situation of DSDM a *Data Flow* can be an arc returning to one of the preceding phases. *Product Flows are tangible goods that result from one of the* Phases *and are the input of the next* Phase*, for example reports and prototypes.*

Then there is the second concept *Phase* that is also be divided two sub concepts with a complete and disjoint ordering. These sub concepts are the *Sequential* and the *Iterative Phases*. As was explained in an earlier section, DSDM starts with two sequential phases, The Feasibility and Business Study. Next a number of Iterative phases follow, i.e. Functional Model, Design & Build and Implementation phases. The picture also mentions a number of rules and issues that are not included in the model, but that are important for this meta-model. First there are the rules that concerns the behavior of the ''Flows''. These rules restrict the freedom of the flows so that they correspond to the ''Phase'' transitions within DSDM. Next to the rules a number of important issues are addressed that ensure that the DSDM project life-cycle is guaranteed.

Critical Success Factors of DSDM

Within DSDM a number of factors are identified as being of great importance to ensure successful projects.

- Factor 1: First there is the acceptance of DSDM by senior management and other employees. This ensures that the different actors of the project are motivated from the start and remain involved throughout the project.

- Factor 2: The second factor follows directly from this and that is the commitment of management to ensure end-user involvement. The prototyping approach requires a strong and dedicated involvement by end user to test and judge the functional prototypes.

- Factor 3: Then there is the project team. This team has to be composed of skillful members that form a stable union. An important issue is the empowerment of the project team. This means that the team (or one or more of its members) has to possess the power and possibility to make important decisions regarding the project without having to write formal proposals to higher management, which can be very time-consuming. In order for the project team to be able to run a successful project, they also need the right technology to conduct the project. This means a development environment, project management tools, etc.

- Factor 4: Finally DSDM also states that a supportive relationship between customer and vendor is required. This goes for both projects that are realised internally within companies or by outside contractors. An aid in ensuring a supporting relationship could be ISPL.

Comparison to other IS Development Methods

Over the years a great number of Information System Development methods have been developed and applied, divided in Structured Methods, RAD methods and Object-Oriented Methods. Many of these methods show similarities to each other and also to DSDM. For example Extreme Programming(XP) also has an iterative approach to IS development with extensive user involvement.

The Rational Unified Process is a method that probably has the most in common with DSDM in that it is also a dynamic form of Information System Development. Again the iterative approach is used in this development method.

Like XP and RUP there are many other development methods that show similarities to DSDM, but DSDM does distinguish itself from these methods in a number of ways. First there is the fact that it provides a tool and technique independent framework. This allows users to fill in the specific steps of the process with their own techniques and software aids of choice. Another unique feature is the fact that the variables in the development are not time/resources, but the requirements. This approach ensures the main goals of DSDM, namely to stay within the deadline and the budget. And last there is the strong focus on communication between and the involvement of all the stakeholders in the system. Although this is addressed in other methods, DSDM strongly believes in commitment to the project to ensure a successful outcome.