Agile Software Development

Landon Sewell

First Edition, 2012

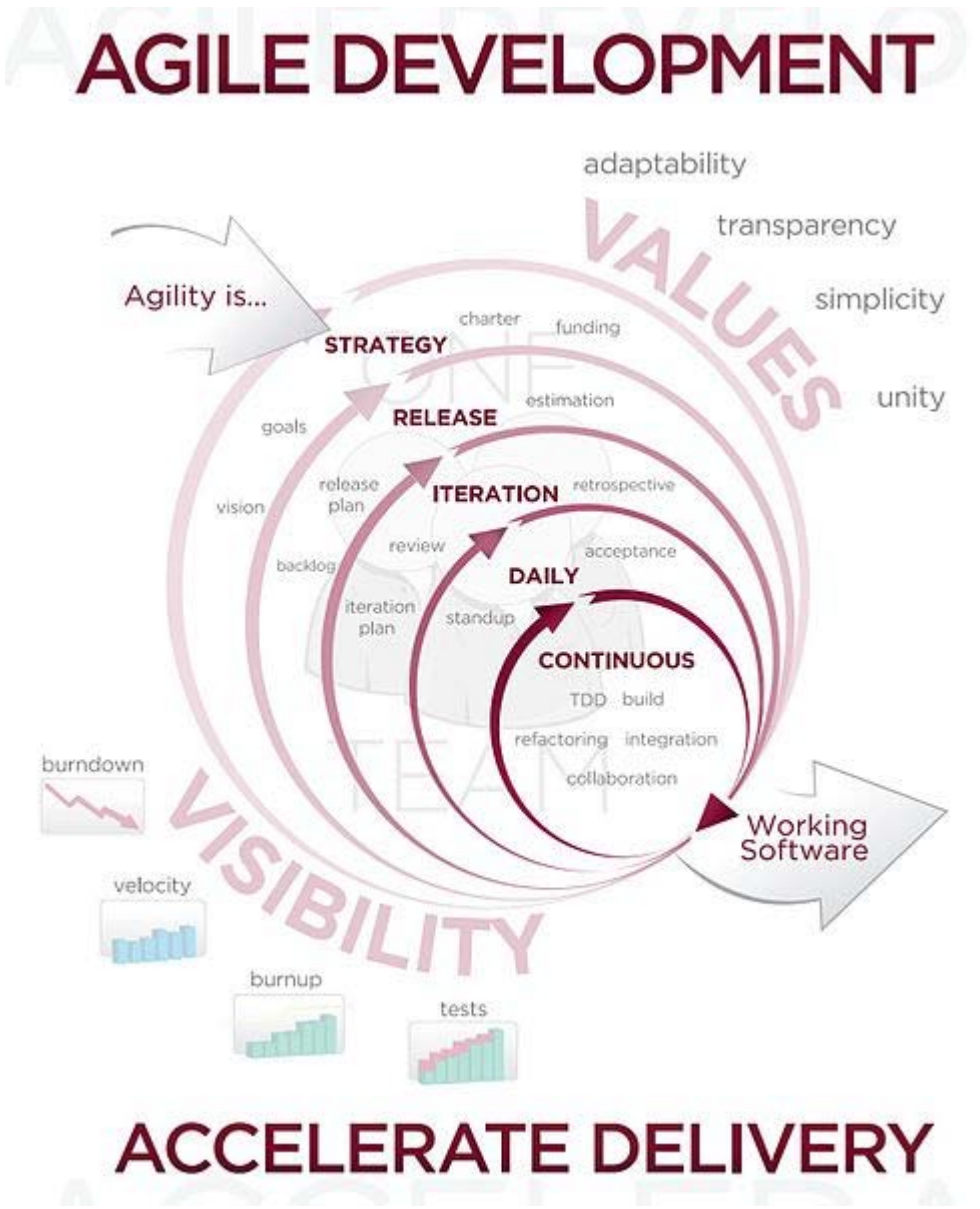
ISBN 978-81-323-3023-3

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

Table of Contents

Chapter 1

Agile Software Development

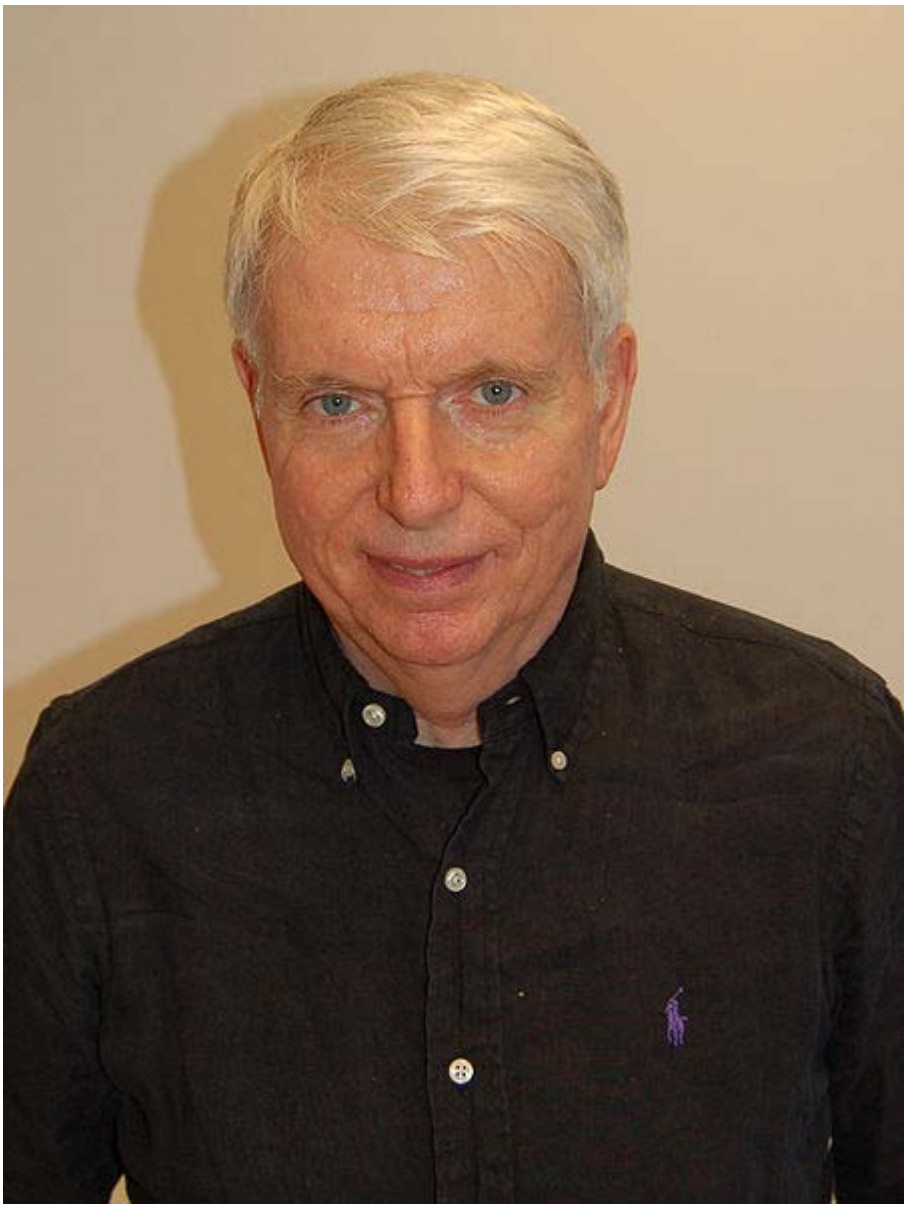


Agile software development poster

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.

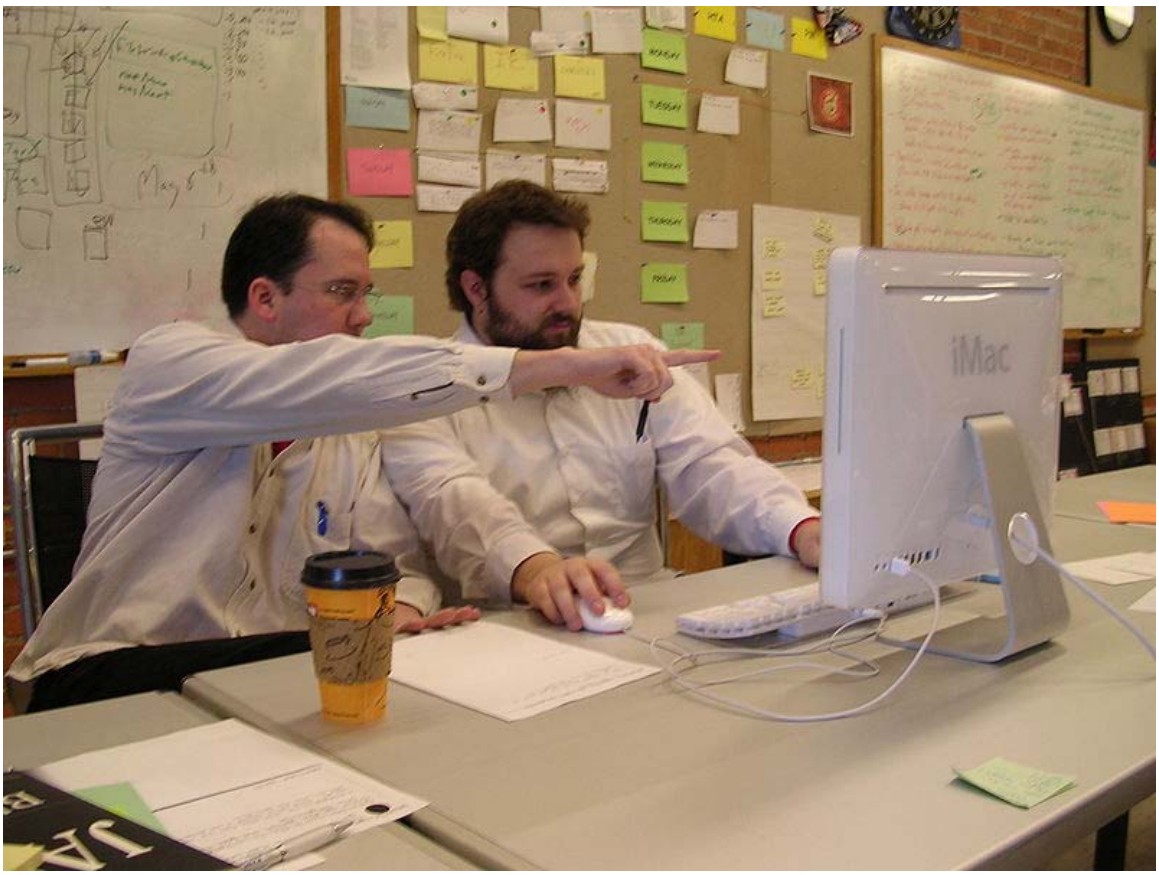
Individuals and Interactions – in agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
Working software – working software will be more useful and welcome than just presenting documents to clients in meetings.
Customer collaboration – 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

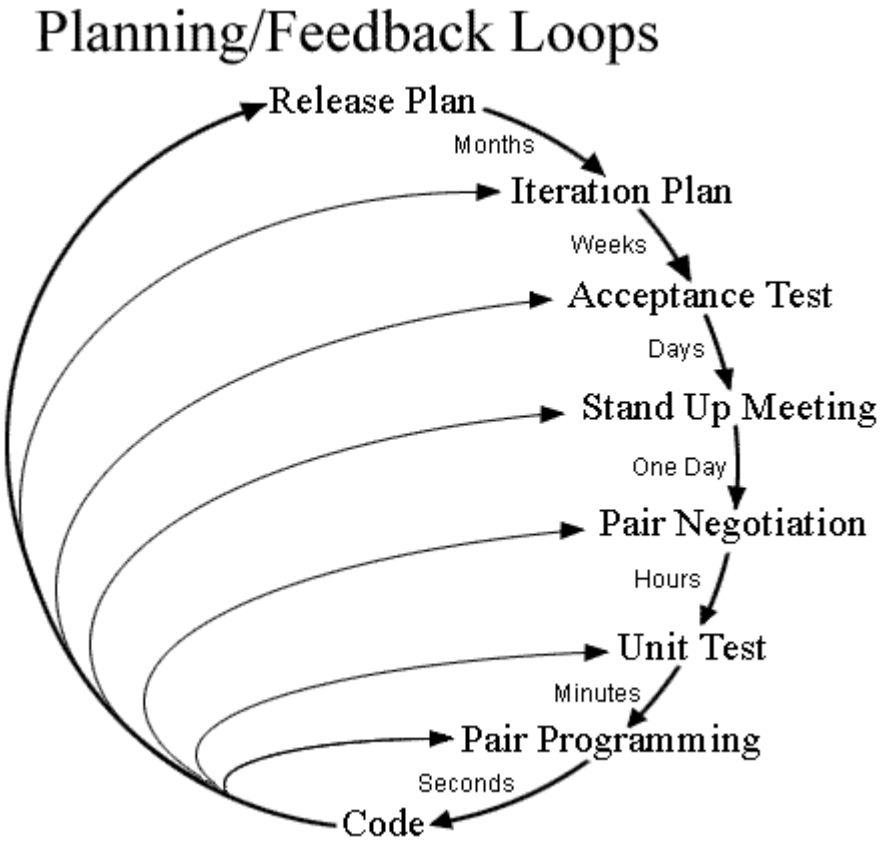
peer-reviewed experience report track. The experience reports share industry experiences with agile software development.

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 2

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.

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

Current state

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

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

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

Concept

Goals

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

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

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

Activities

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

Coding

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

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

Testing

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

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

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

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

Listening

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

Designing

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

Values

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

Communication

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

Simplicity

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

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

Feedback

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

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

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

Courage

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

Respect

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

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

Rules

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

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

Principles

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

Feedback

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

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

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

Assuming simplicity

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

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

Embracing change

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

Practices

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

Fine scale feedback

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

Continuous process

- Continuous integration
- Refactoring or design improvement
- Small releases

Shared understanding

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

Programmer welfare

- Sustainable pace

Coding

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

Testing

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

Controversial aspects

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

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

Other potentially controversial aspects of extreme programming include:

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

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

Scalability

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

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

Severability and responses

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

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

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

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

Criticism

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

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

Criticisms include:

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

Chapter 3

Applied Agile Software Development

AASD logo

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

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

- AASD procedures and methods may copy good practices from wherever it is legally possible. That's why AASD is similar to Scrum.

- AASD procedures should be very applicable, truly "out of the trenches"; there should never exist a procedure that is "good in theory, but can't be used in practice".
- AASD procedures should mention all needed software tools (such as SubVersion, Eclipse, etc.), and its versions.
- AASD emphasizes the use FOSS tools, but that's not an imposition of the method. That is:

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

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

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

Scope

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

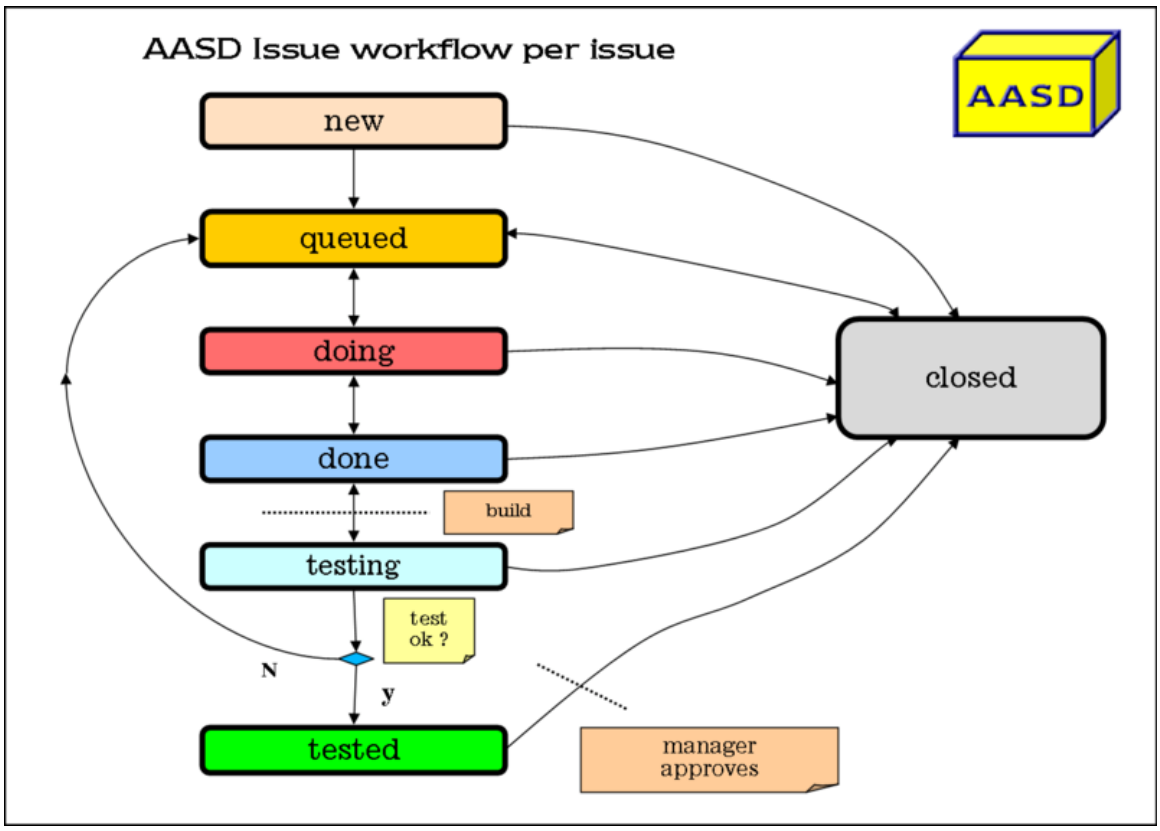
AASD is mainly conceived as a method for software development. But it should be remarked that it is viable as a method for a non-software project management, specially the tool for "Task Workflow".

Task Workflow

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

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

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



Software Tools

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

Implementing Task Workflow with Mantis

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

- Have some computer configured with apache web server and php.

- Download Mantis 1.2.4.
- Create file config_inc.php with contents below.

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

- Create file custom_strings_inc.php with contents below.

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

- Edit file bug_report.php. Find line below

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

and change to line below.

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

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

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

Chapter 4

Agilo for Scrum

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

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

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

Version 0.8 is based on Trac 0.11.

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

Features

- Interactive Whiteboard for distributed teams
- Real Ticket Typing: Agilo for Scrum adds to *Trac* custom ticket attributes that are only available for specific ticket types and can be configured freely. This enables Agilo for Scrum to track tickets, user stories, requirements and tasks on par to each other.
- Traceability: Agilo supports the defined *Scrum* process by linking all artefacts according to their semantic. A user story can be linked with a requirement it fulfills, a set of tasks and the related commits.
- Scrum Sprint artifacts: Agilo can maintain the Product Backlog, Sprint Backlog, Burn down chart and various statistics.

- Scrum Scaling Support: Agilo supports multiple teams working against the same product backlog, each team with their own Sprint.
- Custom Backlogs: Users can define backlogs with different scope (e.g. product, release, sprint)

Special features to support the standard *Scrum* roles:

For Product Owners

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

For Scrum Masters

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

For Team Members

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

Chapter 5

Feature Driven Development

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

History

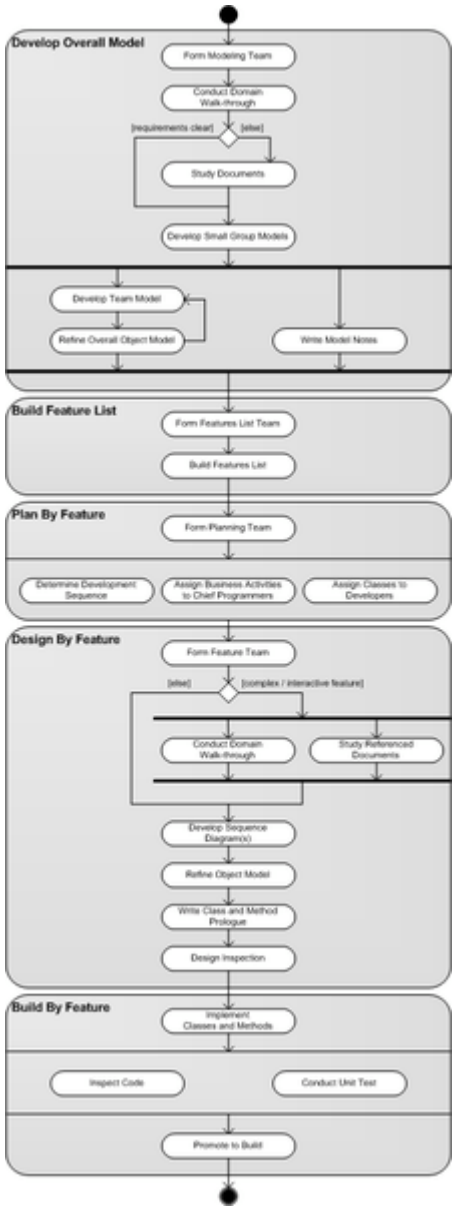
FDD was initially devised by Jeff De Luca to meet the specific needs of a 15 month, 50 person software development project at a large Singapore bank in 1997. Jeff De Luca delivered a set of five processes that covered the development of an overall model and the listing, planning, design and building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

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

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

Overview

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



Process model for FDD

Activities

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

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

Develop Overall Model

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

Build Feature List

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

Plan By Feature

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

Design By Feature

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

Build By Feature

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

Milestones

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

Table 1: Milestones

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

Best practices

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

- **Domain Object Modeling**. Domain Object Modeling consists of exploring and explaining the domain of the problem to be solved. The resulting domain object model provides an overall framework in which to add features.

- **Developing by Feature**. Any function that is too complex to be implemented within two weeks is further decomposed into smaller functions until each sub-problem is small enough to be called a feature. This makes it easier to deliver correct functions and to extend or modify the system.

- **Individual Class (Code) Ownership**. Individual class ownership means that distinct pieces or grouping of code are assigned to a single owner. The owner is responsible for the consistency, performance, and conceptual integrity of the class.

- **Feature Teams**. A feature team is a small, dynamically formed team that develops a small activity. By doing so, multiple minds are always applied to each design decision and also multiple design options are always evaluated before one is chosen.

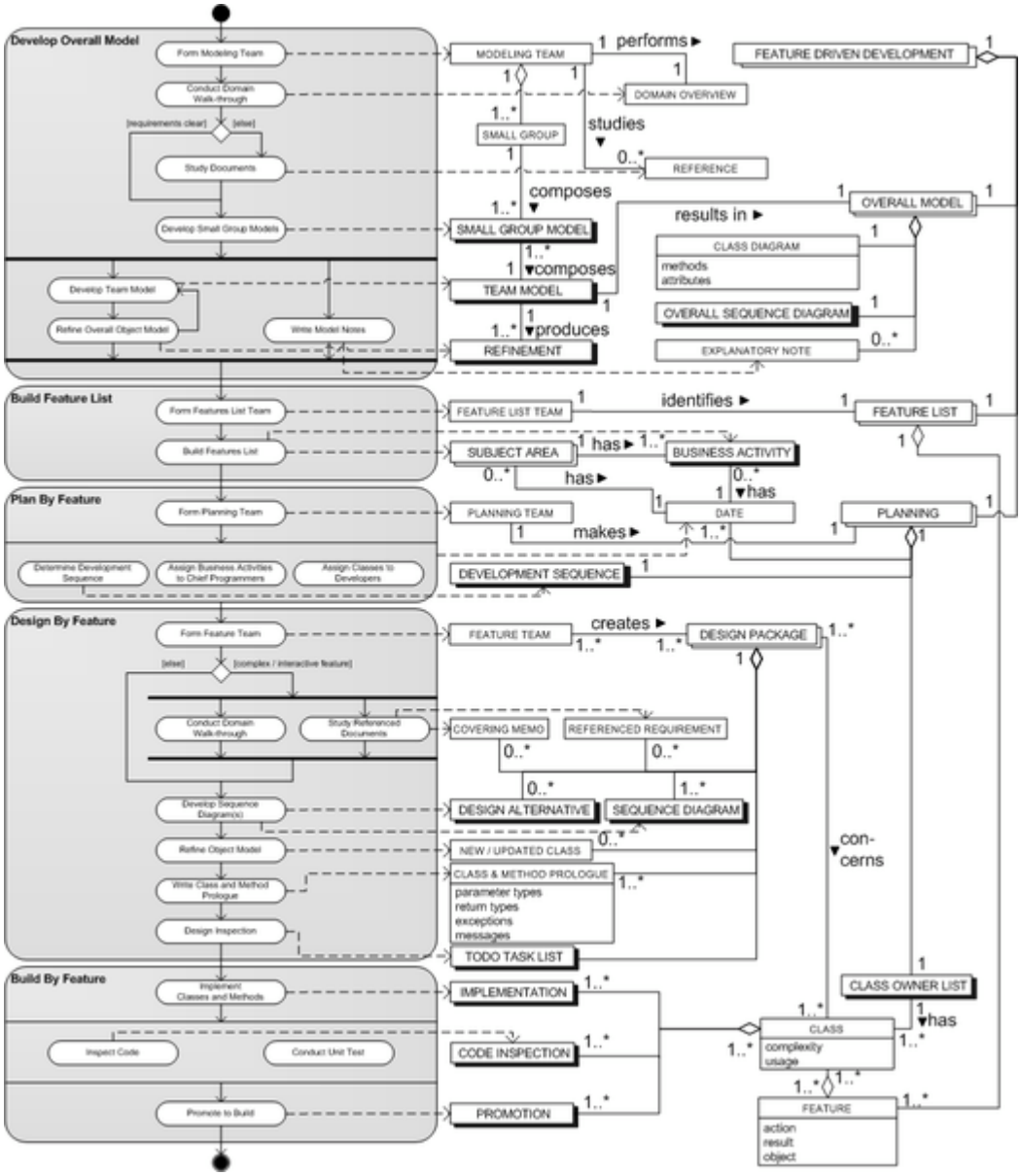
- **Inspections**. Inspections are carried out to ensure good quality design and code, primarily by detection of defects.

- **Configuration Management**. Configuration management helps with identifying the source code for all features that have been completed to date and to maintain a history of changes to classes as feature teams enhance them.

- **Regular Builds**. Regular builds ensure there is always an up to date system that can be demonstrated to the client and helps highlighting integration errors of source code for the features early.

- **Visibility of progress and results**. By frequent, appropriate, and accurate progress reporting at all levels inside and outside the project, based on completed work, managers are helped at steering a project correctly.

Metamodel (MetaModeling)



Process-Data Model for FDD

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

The left side of the metadata model, depicted on the right, shows the five basic activities involved in a software development project using FDD. The activities all contain sub-activities that correspond to the sub-activities in the FDD process description on Jeff De Luca´s website. The right side of the model shows the concepts involved. These concepts

originate from the activities depicted in the left side of the diagram. A definition of the concepts is given in Table 3.

Tools used for Feature Driven Development

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

Chapter 6

DevOps

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

History

Predecessors

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

DevOpsDays

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

DevOps Methodologies

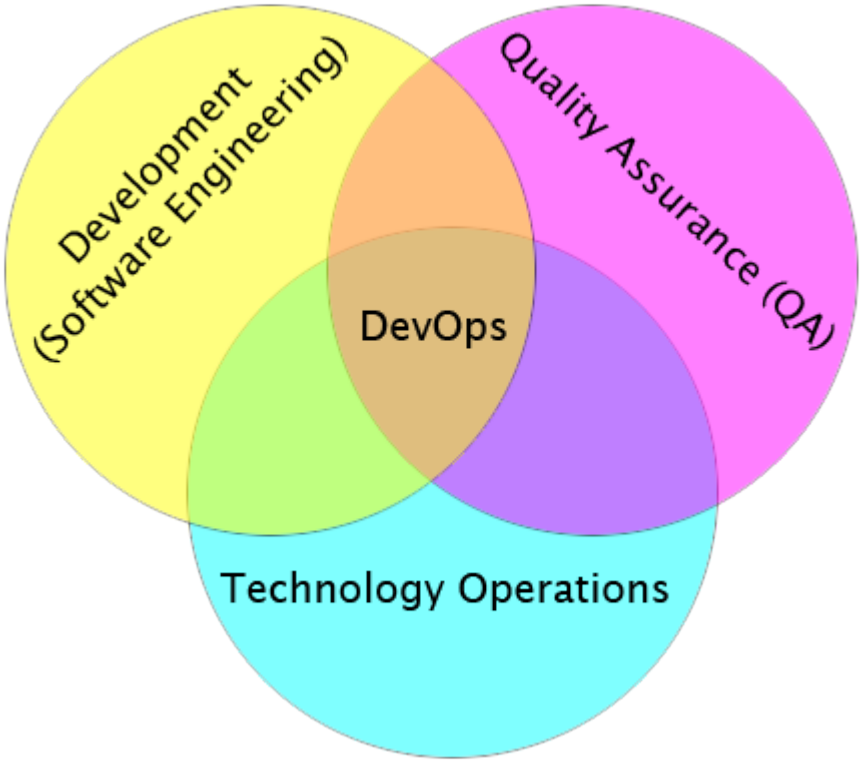


Illustration showing DevOps as the intersection of development (software engineering), technology operations and quality assurance (QA)

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

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

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

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

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

DevOps Impact on Application Releases

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

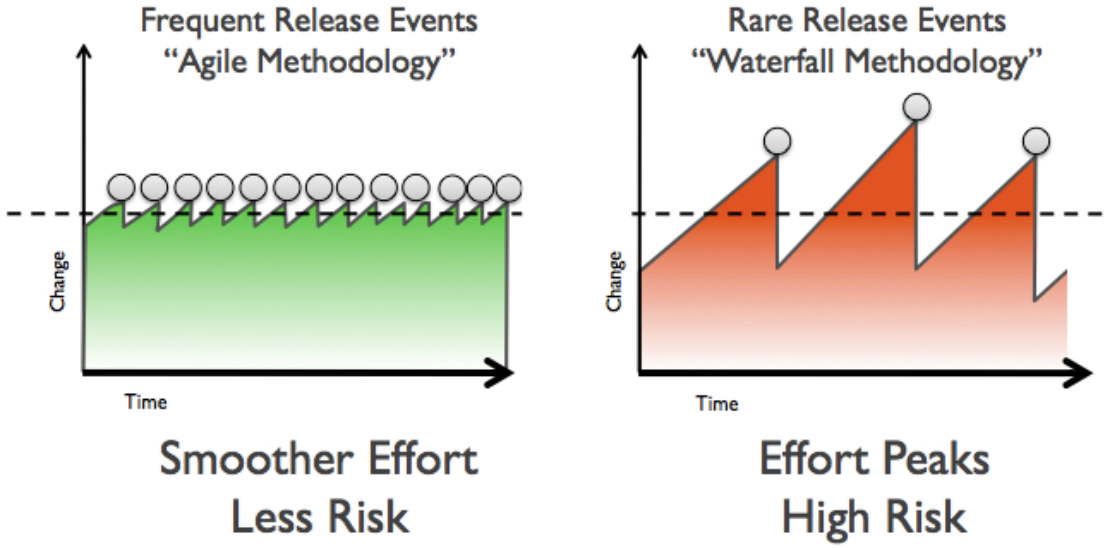


Illustration showing effect of Agile methodology's dramatic increase in frequency of release events -- often measured in days or weeks -- in contrast to large, infrequent releases, often measured in quarters or years, using traditional development methodologies.

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

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

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

Adoption of DevOps Methodologies

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

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

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

Needs

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

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

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

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

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

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

Release coordinator

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

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

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

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

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

Criticisms of DevOps

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

Chapter 7

Lean Software Development

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

Origin

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

Lean principles

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

Eliminate waste

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

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

In order to be able to eliminate waste, one should be able to recognize and see it. If some activity could be bypassed or the result could be achieved without it, it is waste. Partially

done coding eventually abandoned during the development process is waste. Extra processes and features not often used by customers are waste. Waiting for other activities, teams, processes is waste. Defects and lower quality are waste. Managerial overhead not producing real value is waste. A value stream mapping technique is used to distinguish and recognize waste. The second step is to point out sources of waste and eliminate them. The same should be done iteratively until even essential-seeming processes and procedures are liquidated.

Amplify learning

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

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

Decide as late as possible

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

An agile software development approach can move the building of options earlier for customers, thus delaying certain crucial decisions until customers have realized their needs better. This also allows later adaptation to changes and the prevention of costly

earlier technology-bounded decisions. This does not mean that no planning should be involved – on the contrary, planning activities should be concentrated on the different options and adapting to the current situation, as well as clarifying confusing situations by establishing patterns for rapid action. Evaluating different options is effective as soon as it is realized that they are not free, but provide the needed flexibility for late decision making.

Deliver as fast as possible

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

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

Empower the team

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

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

Build integrity in

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

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

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

See the whole

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

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

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

Lean software practices

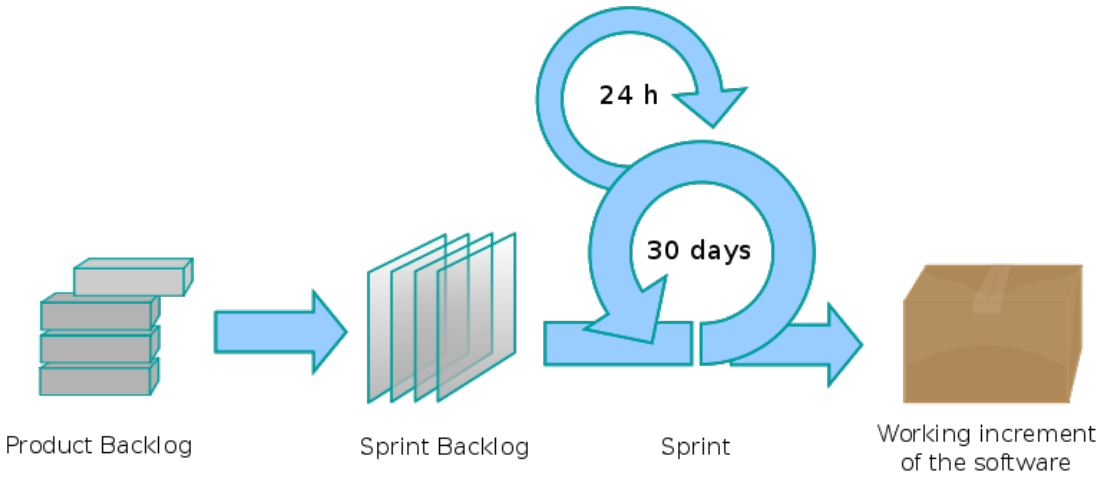
Lean software development practices, or what the Poppendiecks call "tools" are expressed slightly differently from their equivalents in agile software development, but there are parallels. Examples of such practices include:

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

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

Chapter 8

Scrum (Development)



The Scrum process.

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

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

History

In 1986, Hirotaka Takeuchi and Ikujiro Nonaka described a new approach to commercial product development that would increase speed and flexibility, based on case studies from manufacturing firms in the automotive, computer, photocopier, and printer industries. They called this the *holistic or rugby approach*, as the whole process is

performed by one cross-functional team across multiple overlapping phases, where the *scrum* (or whole team) "tries to go the distance as a unit, passing the ball back and forth".

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

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

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

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

Characteristics

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

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

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

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

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

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

Roles

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

Core Scrum roles

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

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

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

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

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

Ancillary Scrum roles

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

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

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

Agile Project Management with Scrum

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

Meetings

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

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

  During the meeting, each team member answers three questions:

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

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

Sprint Planning Meeting

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

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

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

Sprint Review Meeting

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

Sprint Retrospective

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

Artifacts

Product backlog

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

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

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

Sprint backlog

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

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

Burn down

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

It should not be confused with an earned value chart.

Adaptive project management

The following are some general practices of Scrum:

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

Terminology

The following terminology is used in Scrum:

Roles

Scrum Team
: Product Owner, ScrumMaster and Team

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

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

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

Artifacts

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

Product backlog
: A prioritized list of high level requirements.

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

Others

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

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

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

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

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

Point Scale

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

Definition of Done (DoD)

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

Scrum modifications

Scrum-ban

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

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

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

- Unstarted
- Ongoing
- Completed

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

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

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

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

Product development

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

Chapter 9

Presenter First & Planning Poker

Presenter First

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

Approach

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

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

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

Implementation

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

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

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

Planning Poker

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

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

Process

Equipment

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

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

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

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

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

Procedure

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

The meeting proceeds as follows:

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

discussion must cease and another round of poker is played. The structure in the conversation is re-introduced by the soap boxes.

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

Planning Poker benefits

Planning Poker is a tool for estimating software development projects. It is a technique that minimizes anchoring by asking each team member to play their estimate card such that it cannot be seen by the other players. After each player has selected a card, all cards are exposed at once.

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

Avoid anchoring

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

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

Planning poker exposes the potentially influential team member as being isolated in his or her opinion among the group. It then demands that she or he argue the case against the prevailing opinion. If a group is able to express its unity in this manner they are more likely to have faith in their original estimates. If the influential person has a good case to

argue everyone will see sense and follow, but at least the rest of the team won't have been anchored; instead they will have listened to reason.

Chapter 10

P-Modeling Framework

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

History

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

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

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

Subsequently, Vladimir L. Pavlov conducted a number of additional experiments intended to reveal whether the “silent” modeling sessions are more productive than the

traditional ones. In these experiments, silent teams appeared to be at least as efficient as the others, and in some cases the silent teams outperformed the traditional ones.

Some of the interpretations of these results are the following:

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

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

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

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

Basic principles

Reverse Semantic Traceability

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

original and restored versions of a text are to be compared semantically, with a focus on the "meaning" of the text, not on particular "words" used in it.

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

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

Speechless modeling

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

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

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

Requirements and limitations

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

Criticism

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

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

Chapter 11

Test-Driven Development

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

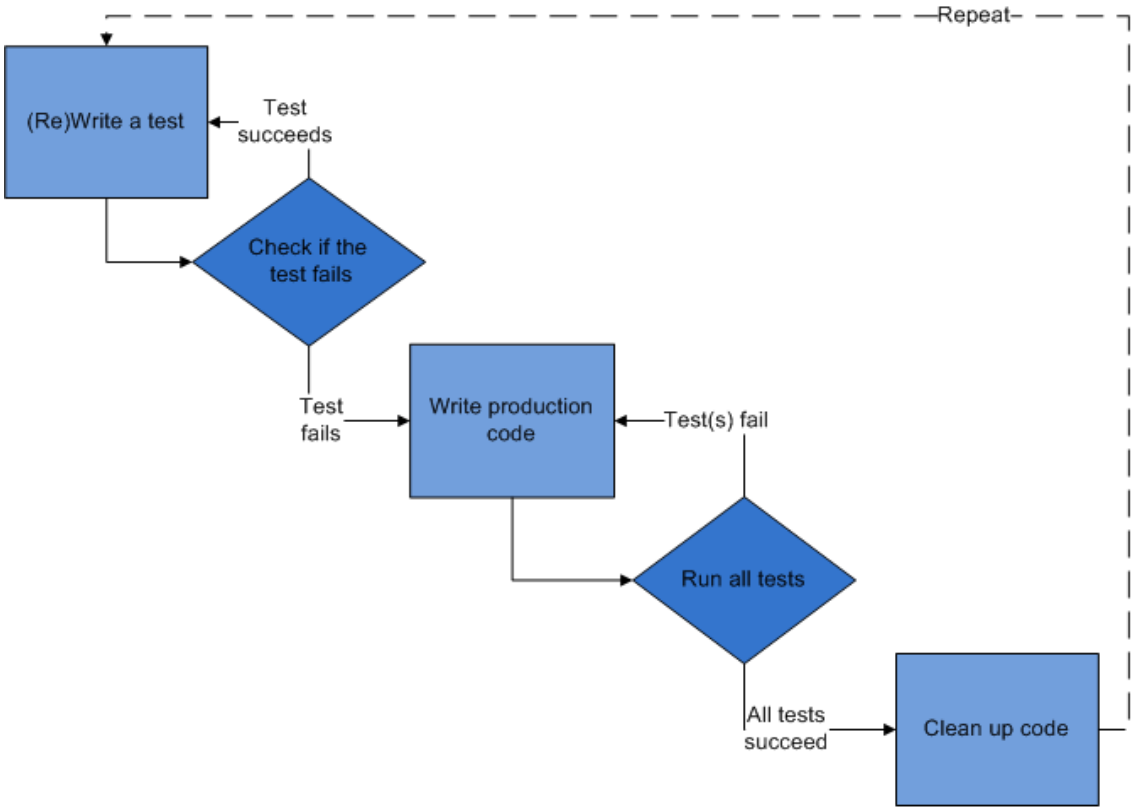
Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

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

Requirements

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

Test-driven development cycle



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

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

Add a test

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

Run all tests and see if the new one fails

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

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

Write some code

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

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

Run the automated tests and see them succeed

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

Refactor code

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

Repeat

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

Development style

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

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

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

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

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

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

Benefits

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

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

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

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

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

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

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

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

Vulnerabilities

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

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

Code visibility

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

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

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

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

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

Fakes, mocks and integration tests

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

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

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

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

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

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

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

Chapter 12

Extreme Programming Practices

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

Fine scale feedback

Pair programming

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

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

Planning game

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

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

    - Commitment Phase: Within the commitment phase business and developers will commit themselves to the functionality that will be included and the date of the next release.
    - Steering Phase: In the steering phase the plan can be adjusted, new requirements can be added and/or existing requirements can be changed or removed.

- Iteration Planning: This plans the activities and tasks of the developers. In this process the customer is not involved. Iteration Planning also consists of three phases:
    - Exploration Phase: Within this phase the requirement will be translated to different tasks. The tasks are recorded on task cards.
    - Commitment Phase: The tasks will be assigned to the programmers and the time it takes to complete will be estimated.
    - Steering Phase: The tasks are performed and the end result is matched with the original user story.

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

Release planning

Exploration phase

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

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

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

Commitment phase

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

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

Sort by value

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

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

Sort by risk

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

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

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

Steering phase

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

Iteration planning

Exploration phase

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

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

Commitment phase

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

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

Steering phase

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

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

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

Test driven development

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

Whole team

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

Continuous process

Continuous integration

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

Design improvement

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

Small releases

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

Shared understanding

Coding standard

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

Collective code ownership

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

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

Simple design

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

System metaphor

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

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

Programmer welfare

Sustainable pace

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

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

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

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

Chapter 13

Continuous Integration

In software engineering, **continuous integration** (**CI**) implements *continuous* processes of applying quality control — small pieces of effort, applied frequently. Continuous integration aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control *after* completing all development.

Theory

When embarking on a change, a developer takes a copy of the current code base on which to work. As other developers submit changed code to the code repository, this copy gradually ceases to reflect the repository code. When developers submit code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes the repository contains, the more work developers must do before submitting their own changes.

Eventually, the repository may become so different from the developers' baselines that they enter what is sometimes called "integration hell", where the time it takes to integrate exceeds the time it took to make their original changes. In a worst-case scenario, developers may have to discard their changes and completely redo the work.

Continuous integration involves integrating early and often, so as to avoid the pitfalls of "integration hell". The practice aims to reduce rework and thus reduce cost and time.

Recommended practices

Continuous integration - as the practice of frequently integrating one's new or changed code with the existing code repository - should occur frequently enough that no intervening window remains between commit and build, and such that no errors can arise without developers noticing them and correcting them immediately. Normal practice is to trigger these builds by every commit to a repository, rather than a periodically scheduled build. The practicalities of doing this in a multi-developer environment of rapid commits

are such that it's usual to trigger a short timer after each commit, then to start a build when either this timer expires, or after a rather longer interval since the last build. Automated tools such as CruiseControl or Jenkins or Hudson offer this scheduling automatically.

Another factor is the need for a version control system that supports atomic commits, i.e. all of a developer's changes may be seen as a single commit operation. There is no point in trying to build from only half of the changed files.

Maintain a code repository

This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository. In this practice and in the revision control community, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. Extreme Programming advocate Martin Fowler also mentions that where branching is supported by tools, its use should be minimised. Instead, it is preferred that changes are integrated rather than creating multiple versions of the software that are maintained simultaneously. The mainline (or trunk) should be the place for the working version of the software.

Automate the build

A single command should have the capability of building the system. Many build-tools, such as make, have existed for many years. Other more recent tools like Ant, Maven, MSBuild or IBM Rational Build Forge are frequently used in continuous integration environments. Automation of the build should include automating the integration, which often includes deployment into a production-like environment. In many cases, the build script not only compiles binaries, but also generates documentation, website pages, statistics and distribution media (such as Windows MSI files, RPM or DEB files).

Make the build self-testing

Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave.

Everyone commits to the baseline every day

By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making.

Many programmers recommend committing all changes at least once a day (once per feature built), and in addition performing a nightly build.

Every commit (to baseline) should be built

The system should build commits to the current working version in order to verify that they integrate correctly. A common practice is to use Automated Continuous Integration, although this may be done manually. For many, continuous integration is synonymous with using Automated Continuous Integration where a continuous integration server or daemon monitors the version control system for changes, then automatically runs the build process.

Keep the build fast

The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.

Test in a clone of the production environment

Having a test environment can lead to failures in tested systems when they deploy in the production environment, because the production environment may differ from the test environment in a significant way. However, building a replica of a production environment is cost prohibitive. Instead, the pre-production environment should be built to be a scalable version of the actual production environment to both alleviate costs while maintaining technology stack composition and nuances.

Make it easy to get the latest deliverables

Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when rebuilding a feature that doesn't meet requirements. Additionally, early testing reduces the chances that defects survive until deployment. Finding errors earlier also, in some cases, reduces the amount of work necessary to resolve them.

Everyone can see the results of the latest build

It should be easy to find out where/whether the build breaks and who made the relevant change.

Automate deployment

Most CI systems allow the running of scripts after a build finishes. In most situations, it is possible to write a script to deploy the application to a live test server that everyone can look at. A further advance in this way of thinking is Continuous Deployment, which calls for the software to be deployed directly into production, often with additional automation to prevent defects or regressions.

History

Continuous Integration emerged in the Extreme Programming (XP) community, and XP advocates Martin Fowler and Kent Beck first wrote about continuous integration circa 1999. Fowler's paper is a popular source of information on the subject. Beck's book *Extreme Programming Explained*, the original reference for Extreme Programming, also describes the term.

Advantages and disadvantages

Advantages

Continuous integration has many advantages:

- when unit tests fail or a bug emerges, developers might revert the codebase back to a bug-free state, without wasting time debugging
- developers detect and fix integration problems continuously - avoiding last-minute chaos at release dates, (when everyone tries to check in their slightly incompatible versions).
- early warning of broken/incompatible code
- early warning of conflicting changes
- immediate unit testing of all changes
- constant availability of a "current" build for testing, demo, or release purposes
- immediate feedback to developers on the quality, functionality, or system-wide impact of code they are writing
- frequent code check-in pushes developers to create modular, less complex code
- metrics generated from automated testing and CI (such as metrics for code coverage, code complexity, and features complete) focus developers on developing functional, quality code, and help develop momentum in a team

Disadvantages

- initial setup time required
- well-developed test-suite required to achieve automated testing advantages
- large-scale refactoring can be troublesome due to continuously changing code base
- hardware costs for build machines can be significant

Many teams using CI report that the advantages of CI well outweigh the disadvantages. The effect of finding and fixing integration bugs early in the development process saves both time and money over the lifespan of a project.

Software

To support continuous integration, software tools such as automated build software can be employed.

Software tools for continuous integration include:

- AnthillPro — continuous integration server by Urbancode
- Apache Continuum — continuous integration server supporting Apache Maven and Apache Ant. Supports CVS, Subversion, Ant, Maven, and shell scripts
- Apache Gump — continuous integration tool by Apache
- Automated Build Studio — proprietary automated build, continuous integration and release management system by AutomatedQA
- Bamboo — proprietary continuous integration server by Atlassian Software Systems
- BuildBot — Python/Twisted-based continuous build system
- BuildMaster — proprietary application lifecycle management and continuous integration tool by Inedo
- CABIE - Continuous Automated Build and Integration Environment — open source, written in Perl; works with CVS, Subversion, AccuRev, Bazaar and Perforce
- Cascade — proprietary continuous integration tool; provides a checkpointing facility to build and test changes before they are committed
- codeBeamer — proprietary collaboration software with built-in continuous integration features
- CruiseControl — Java-based framework for a continuous build process
- CruiseControl.NET — .NET-based automated continuous integration server
- CruiseControl.rb - Lightweight, Ruby-based continuous integration server that can build any codebase, not only Ruby, released under Apache Licence 2.0
- ElectricCommander — proprietary continuous integration and release management solution from Electric Cloud
- FinalBuilder Server — proprietary automated build and continuous integration server by VSoft Technologies
- Go — proprietary agile build and release management software by Thoughtworks
- Hudson — MIT-licensed, written in Java, runs in servlet container, supports CVS, Subversion, Mercurial, Git, StarTeam, Clearcase, Ant, NAnt, Maven, and shell scripts
- Jenkins — MIT-licensed, written in Java, runs in servlet container, supports CVS, Subversion, Mercurial, Git, StarTeam, Clearcase, Ant, NAnt, Maven, and shell scripts
- Software Configuration and Library Manager — software configuration management system for z/OS by IBM Rational Software
- QuickBuild - proprietary continuous integration server with free community edition featuring build life cycle management and pre-commit verification.
- TeamCity — proprietary continuous-integration server by JetBrains with free professional edition
- Team Foundation Server — proprietary continuous integration server and source code repository by Microsoft
- Tinderbox — Mozilla-based product written in Perl
- Rational Team Concert — proprietary software development collaboration platform with built-in build engine by IBM including Rational Build Forge

Chapter 14

Pair Programming

Pair programming

Pair programming is an agile software development technique in which two programmers work together at one work station. One types in code while the other reviews each line of code as it is typed in. The person typing is called the **driver**. The person reviewing the code is called the **observer** (or **navigator**). The two programmers switch roles frequently.

While reviewing, the observer also considers the strategic direction of the work, coming up with ideas for improvements and likely future problems to address. This frees the driver to focus all of his or her attention on the "tactical" aspects of completing the current task, using the observer as a safety net and guide.

Costs and benefits

Some studies have found that programmers working in pairs produce shorter programs, with better designs and fewer bugs, than programmers working alone. Studies have found reduction in defect rates of 15% to 50%, varying depending on programmer experience and task complexity. Pairs typically consider more design alternatives than programmers working solo, and arrive at simpler, more-maintainable designs; they also catch design defects early. Pairs usually complete work faster than one programmer assigned to the same task. Pairs often find that seemingly "impossible" problems become easy or even quick, or at least possible to solve when they work together.

However, a 2007 meta-analysis concluded that "pair programming is not uniformly beneficial or effective" because many other factors besides the choice of whether to use pair programming have large effects on the outcome of a programming task. The meta-study found that pair programming tends to reduce development time somewhat and produces marginal positive effects on code quality, but that pair programming requires significantly more developer effort; that is, it is significantly more expensive than solo programming. The authors suggest that studies of pair programming suffer from publication bias whereby studies that would not show that pair programming is beneficial were either not undertaken, not submitted for publication, or not accepted for publication. They conclude that "you cannot expect faster and better and cheaper."

Even though coding is often completed faster than when one programmer works alone, total programmer time (number of programmers × time spent) increases. A manager needs to balance faster completion of the work and reduced testing and debugging time against the higher cost of coding. The relative weight of these factors can vary from project to project and task to task. The benefit is strongest on tasks that are not yet understood by the programmers, calling for more creativity, challenge, and sophistication. On simple tasks, which the pair already fully understands, pairing results in a net drop in productivity.

Knowledge passes between pair programmers as they work. They share knowledge of the specifics of the system, and they pick up programming techniques from each other. New hires quickly pick up the practices of the team and learn the specifics of the system. With "promiscuous pairing" – each programmer cycling through all the other programmers on the team rather than pairing only with one partner – knowledge of the system spreads throughout the whole team, reducing risk to management if one programmer leaves the team.

Pairing usually brings improved discipline and time management. Programmers are less likely to skip writing unit tests, spend time web-surfing or on personal email, or cut

corners when they are working with a pair partner. The pair partner "keeps them honest". People are more reluctant to interrupt a pair than they are to interrupt someone working alone.

Additional benefits reported include increased morale and greater confidence in the correctness of the code.

Scientific studies

According to *The Economist*,

"Laurie Williams of the University of Utah in Salt Lake City has shown that paired programmers are only 15% slower than two independent individual programmers, but produce 15% fewer bugs. (N.B.: The original study showed that 'error-free' code went from 70% to 85%; it may be more intuitive to call this a 50% decrease of errors, from 30% to 15%.) Since testing and debugging are often many times more costly than initial programming, this is an impressive result."

The Williams et al. 2000 study showed an improvement in correctness of around 15% and a 20%–40% decrease in time, but between a 15% and 60% increase in effort—that is, total programmer-hours. Williams et al. 2000 also cites an earlier study (Nosek 1998) which also had a 40% decrease in time for a 60% increase in effort.

A study (Lui 2006) presents a rigorous scientific experiment in which novice–novice pairs against novice solos experience significantly greater productivity gains than expert–expert pairs against expert solos.

A larger recent study (Arisholm et al. 2007) had 48% increase in correctness for complex systems, but no significant difference in time, whilst simple systems had 20% decrease in time, but no significant difference in correctness. Overall there was no general reduction in time or increase in correctness, but an overall 84% increase in effort.

Lui, Chan, and Nosek (2008) shows that pair programming outperforms for design tasks.

A full-scale meta-analysis of pair programming experimental studies, from before or during 2007, (Hannay et al. 2009) confirms "that you cannot expect faster and better and cheaper". Higher quality for complex tasks costs higher effort, reduced duration for simpler tasks comes with noticeably lower quality – the meta-analysis "suggests that pair programming is not uniformly beneficial or effective".

Variants

Remote pair programming

Remote pair programming, also known as **virtual pair programming** or **distributed pair programming**, is pair programming where the two programmers are in different

locations, working via a collaborative real-time editor, shared desktop, or a remote pair programming IDE plugin. Remote pairing introduces difficulties not present in face-to-face pairing, such as extra delays for coordination, depending more on "heavyweight" task-tracking tools instead of "lightweight" ones like index cards, and loss of non-verbal communication resulting in confusion and conflicts over such things as who "has the keyboard".

Numerous tools, such as Eclipse plug-ins are available to support remote pairing. Some teams have tried VNC and RealVNC with each programmer using their own computer. Others use the multi-display mode (-x) of the text-based GNU screen. Apple Inc. OSX has a built-in Screen Sharing application.

Ping pong pair programming

In **ping pong pair programming**, the observer writes a failing unit test, the driver modifies the code to pass the test, the observer writes a new unit test, and so on. This loop continues as long as the observer is able to write failing unit tests. But it takes more time than the estimated plan.