



The Agile Process

Written by Eamon O'Tuathail

*Values,
Principles,
Practices,
Work Products and
Reviews for*

Agile software development projects

Table of Contents

1: Overview.....	5
1.1: Goals.....	5
1.2: Development Team and Customer Team.....	5
1.3: The Agile Process in a Nutshell.....	6
1.4: References.....	6
2: Values Of The Agile Process.....	8
2.1: The Agile Manifesto.....	8
2.2: Individuals And Interactions.....	8
2.2.1: Sufficient Processes and Tools.....	8
2.3: Working Software.....	8
2.3.1: Sufficient Documentation.....	9
2.4: Customer Collaboration.....	9
2.4.1: Sufficient Contracts.....	9
2.5: Responding to Change.....	9
2.5.1: Sufficient Planning.....	10
3: Principles Of the Agile Process.....	11
3.1: Principle 1.....	11
3.2: Principle 2.....	12
3.3: Principle 3.....	12
3.4: Principle 4.....	13
3.5: Principle 5.....	13
3.6: Principle 6.....	14
3.7: Principle 7.....	15
3.8: Principle 8.....	16
3.9: Principle 9.....	17
3.10: Principle 10.....	17
3.11: Principle 11.....	18
3.12: Principle 12.....	18

4: Practices Of the Agile Process.....	19
4.1: Overview.....	19
4.2: Information Radiators.....	19
4.3: Roles of People in the Agile Process.....	19
4.4: Shared Code Ownership.....	20
4.5: Frequent Integration.....	21
4.6: Value-Driven Development (VDD).....	22
4.7: High Level Vision.....	23
4.8: Gather requirements as they become available.....	23
4.9: Encourage Flexible Scoping.....	24
4.10: Daily Stand-up Meetings.....	24
4.11: Occasional Pair Programming.....	25
4.12: Sustainable Pace.....	25
4.13: Risk Management.....	25
4.14: Avoid Functional Decomposition.....	25
4.15: Responsibilities and Behaviours are Key to Good Architecture.....	25
4.16: Initial Architecture Skeleton.....	25
4.17: Common Workspace For Development Team.....	26
4.18: Iterations & Release Planning.....	26
4.19: Team of Teams.....	26
4.20: Clearly Define Contract Models.....	26
4.21: Understand Stability vs. Flexibility Tradeoffs.....	26
4.22: Travel Light.....	26
4.23: Templates and Coding Standards.....	26
4.24: Avoiding Multiple Independent Representations of the same Information.....	26
4.25: Customer Team	26
4.26: Test-first.....	27
4.27: Waste Elimination.....	27
4.28: Learning Through Exploration and experimentation.....	27
4.29: Set based development.....	27
4.30: Decide at the last responsible moment.....	27

5: Work Products.....	28
5.1: Overview.....	28
5.2: Engineering Group Specifications.....	28
5.2.1: Corporate Software Development Strategy Specification.....	28
5.2.2: Coding Standards and Code Review Specification.....	28
5.2.3: Software Development Process Specification.....	28
5.3: Project Specifications.....	29
5.3.1: Requirements Specification.....	29
5.3.2: Architecture Specification.....	29
5.3.3: System Test Specification.....	29
5.3.4: Security Threat Model Specification.....	29
5.3.5: User Interaction Design Specification.....	29
5.4: Other Communications Techniques.....	30
5.5: Project Lifecycle Documents.....	30
5.6: Additional Documents.....	30
6: Agile Process Review Checklist.....	31
6.1: Overview.....	31
6.2: Review Questions.....	31
6.2.1: The primary review question.....	31
6.2.2: Development Team.....	31
6.2.3: Development Environment.....	32
6.2.4: Work Products.....	32
6.2.5: Customer Team.....	32

1: Overview

1.1: Goals

The agile process is a customer-responsive highly productive software development technique. Software development is a mixture of creativity, engineering and communication and hence it can be challenging to manage. It shares much in common with new product development (where variation, experimentation and learning are encouraged) and little in common with production techniques (where meticulous planning excels) The goal of this document is to provide a succinct description of how software is developed according to the values, principles and practices of the agile process.

Think of the agile concept as a continuum (of tolerance to tuning, amount of ceremony needed, dynamism), with Extreme Programming (XP) at one end and the Unified Process at the other. In the middle are many overlapping techniques (Crystal, Lean, APM, Scrum, AM, FDD) and we use the generic term “agile process” to describe this middle ground - a combination of ideas that work well in many situations. It encourages a low ceremony approach that is tolerant of customisation to suit local needs.

1.2: Development Team and Customer Team

With the agile process, two teams participate in each project, one representing the customer and the other carrying out software development. As we explore the agile process, it is important to realise that most of what we discuss concerns the relationship between these two teams.

For small projects, the customer “team” may be a single person and for larger projects it will probably involve multiple people. Agile encourages one member of the customer team to be co-located with the development team for the entire duration of the project. This on-site presence encourages instantaneous answers for any developer questions and the customer always has an up to date picture of the state of the project. For the larger projects, it is unlikely that one person will have all the answers, so it is expected that the onsite customer representative refer to expertise within the customer team as needed. Within ISVs, a Product Manager is often in charge of the Customer Team and a Project Manager is in charge of the Development Team.

1.3: The Agile Process in a Nutshell

The author Alistair Cockburn has been debriefing successful teams for ten years, states:

“Most of them repeated the same message:

- Seat people close together, communicating frequently and with good will;
- Get most of the bureaucracy out of their way and let them design;
- Get a real user directly involved;
- Have a good automated regression test suite available;
- Produce shippable functionality early and often.

Do all that, and most of the process details will take care of themselves.”

1.4: References

The Agile Manifesto (<http://www.agilemanifesto.org>) provides the values and principles that we follow. The practices are selected from the following:

- [Cockburn] “Crystal Clear: A Human-Powered Methodology for Small Teams ”, Alistair Cockburn, Addison Wesley, ISBN: 0-201-69947-8, 2004
- [Poppendiecks] “Lean Software Development – An Agile Toolkit”, Mary & Tom Poppendieck, ISBN: 0-321-15078-3, Addison Wesley, 2003
- [Highsmith] “Agile Project Management”, Jim Highsmith, ISBN: 0-321-21977-5, Addison Wesley, 2004
- [Beck] “Extreme Programming Explained”, Kent Beck, ISBN: 0-201-61641-6, Addison Wesley, 2000
- [Ambler] “Agile Modeling”, Scott Ambler, ISBN: 0-471-20282-7, Wiley, 2002
- [Cockburn-01] “Writing Effective Use Cases”, Alistair Cockburn, ISBN: 0-201-70225-8, Addison Wesley, 2001
- [BoehmTurner] “Balancing Agility and Discipline”, Barry Boehm & Richard Turner, ISBN: 0-321-18612-5, Addison Wesley, 2004
- [Schwaber] “Agile Project Management with Scrum”, Ken Schwaber, ISBN: 0-7356-1993-X, Microsoft Press, 2004
- [Ruping] “Agile Documentation”, Andreas Ruping, ISBN: 0-470-85617-3, Wiley, 2003

The development team and customer team are encouraged to read these books to become more familiar with agile software development.

2: Values Of The Agile Process

2.1: The Agile Manifesto

The Agile Manifesto states “... 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.”

This simple collection of statements is leading a sea change in how we develop software. A significant number of software projects based around the waterfall model and high-ceremony approaches are not successful, and the root cause is how we go about them. Agility consists of values, principles and practices that aim to consistently deliver projects that satisfy customers.

The mistake some people new to agile make is to ignore what is on the right. Yes, what is on the left is significantly more important, but what is on the right has some value. Processes and tools are useful, but they play a supporting role, rather than a dominant role. Sufficient documentation, contracts and planning are needed to manage projects.

2.2: Individuals And Interactions

Agile is an attitude that believes teams of educated motivated well-resourced professionals will come up with appropriate solutions to the issues they face. By empowering the development teams and enriching the interactions among them, better software will be produced faster.

2.2.1: Sufficient Processes and Tools

Over emphasis on processes and tools in other methodologies caused significant time wasting. Developers need simple tools and simple processes to complete development tasks.

2.3: Working Software

Seeing is believing. Running code is reality – everything else is just guesswork. Most customers do not know exactly what they want or what they get when they received

developer documentation (architecture, requirements, test plans). When customers see running code that they can deploy on their own machines, there is a clear sense of where the project is and how it should progress.

2.3.1: Sufficient Documentation

There is some controversy over how much developer documentation is needed on an agile project. Each development team must get the current version of the software completed as soon as possible, but also must set up subsequent project iterations. This short-term and long-term project investments in work needs to be managed wisely. ON the one hand some teams spend so long on developer documentation they miss deadlines for the delivery of the working software. On the other hand, some developers only deliver the code and no other artefacts. For prototype-style projects that will not progress, this will suffice, but for most production software systems, some documentation is needed. It is likely that some members of the development team will change over time, and some core documentation is needed.

2.4: Customer Collaboration

“Partnership” precisely defines the optimum relationship between the customer team and the development team. When both teams are working together they will achieve most. When they are constantly battling each other, or when such a deep sense of mistrust exists between them that everything is referred back to written contracts, time will be wasted.

2.4.1: Sufficient Contracts

Where there is a high level of trust of the customer team and development team genuinely strive to look after the other, contracts take a back seat. Obviously contracts are important from a business viewpoint - each team has its own concerns- the development team will want to get paid, the customer team will want to get working software.

2.5: Responding to Change

The modern business world is in a constant state of flux. The waterfall model essentially told customers they could define requirements roughly once a year, which is much too limiting in a competitive market. Agile encourages customers to make changes whenever it makes business sense, and the changed requirements will be efficiently incorporated into the ongoing development effort.

An additional cause of change is the evolution of the customers' true understanding of their needs. Early in a project, they seldom have a detailed picture of what is really

required. Over time, as they see early versions of the software deployed they gain a greater insight into their true requirements.

2.5.1: Sufficient Planning

Writing software like going into battle. After the first bullet is fired, all military plans change. After the first line of code is written, all software plans change. Creating very detailed plans for software development projects is a questionable activity, as they are subject to significant change over the lifetime of the project.

Software developers should have a high level vision of where their project is heading, some broad understanding of requirements that will be coming up in future iterations, and a more detailed picture – created with the help of the on-site customer – of the requirements for the current iteration. Planning can be created with the level of precision in mind.

3: Principles Of the Agile Process

The Agile Manifesto (<http://www.agilemanifesto.org>) defines the following agile principles that guide our software development activities.

3.1: Principle 1

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”

Agile developers have a single highest priority – of satisfying the customer – and everything else is secondary (this does not mean they are unimportant, but they are not as important as our highest priority). The twin techniques we plan to use to satisfy the customer are:

- Early delivery of valuable software
- Continuous delivery of valuable software

What do we mean by valuable? It is clear from considering what our highest priority is, that only the customer team can decide what is valuable. It is what brings business value to the customer, as defined by the customer.

Software that is currently being developed but has yet to be delivered to the customer is similar to product inventory in a warehouse. It is a sunk cost, which may deliver benefit in future or may be scrapped (the longer it takes for delivery the greater the chance of the latter). It is a form of waste, since it is not adding value to the customer. In the manufacturing world, companies with the smallest amount of inventory (e.g. DELL, Toyota) are the most efficient; agile brings a similar approach to software development.

Early delivery is important because customers see an early return on their investment. Assuming customers have prioritised high-value features to be completed initially, they will very soon have a software delivery that can be deployed in a production environment and become revenue-generating. Customers who see early returns on their software development investments are far more likely to invest further in subsequent releases - hence the development team also benefits. (Early delivery also brings additional benefits in the form of learning and optimising project direction, which we will discuss later).

Some developers new to agile software development focus solely on early delivery and forget about continuous delivery. Their projects tend to appear successful initially but do

not fare well over time. There is plenty of discussion about how to optimise achievement of continuous delivery – each development team is empowered to come up with the right solution for their specific needs. Continuous delivery is important because customer needs change over time and the software needs to evolve to continue satisfying the customer. What kept the customer happy six months ago will not suffice now. Agile software development has been likened to a subscription service, with regular deliveries (e.g. monthly) of new material that brings real value to the customer.

3.2: Principle 2

“Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.”

Change is good. Change causes disruption in the marketplace. Change is how innovative customers can dislodge competitors with entrenched positions. The customer's business is continuously changing, and hence the software needed is likewise in flux. To be able to more quickly respond to new software requirements can bring the customer significant and sustainable competitive advantage over its rivals.

In the waterfall model, requirements were collected first and then “frozen” for the rest of the project, indicating that no or few new/changed requirements would be acceptable. In the context of the modern business environment, this is ridiculous, as the one certainty is that requirements will be changing (and sometimes quite substantially).

3.3: Principle 3

“Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.”

Defining requirements for a software project is analogous to a game of golf on a hazy morning. You tee off by hitting the ball in the general direction of the hole some way off in the distance. You have a rough idea where it is at first, but as you examine where the ball has landed you have an improved picture. After more shots you get closer and closer, until eventually you reach the target.

At the start of a project, customers seldom know exactly what they want from new software systems. They generally have some vision of what is needed, but cannot deliver precise requirements early on. Within software circles, that there are so many phrases (and these are used so often) for non-production versions of software projects - “pre-

production”, “throwaway”, “prototype”, “demo”, “trial version”, “proof of concept” – is an indication of how widespread is the need to see working code before deciding on the real requirements. Working software has a much richer educational value compared to paper specifications. By actually using the software, customers begin to get a feel for how it should really behave. By delivering working software frequently, the development team is facilitating the customer's own learning process about how the software can truly deliver business value.

3.4: Principle 4

“Business people and developers must work together daily throughout the project.”

In traditional development processes, the business people and developers talk at the beginning of the project (requirements) and at the end (acceptance testing) and pretty much ignore each other in the middle (though they may be some brief progress reports). The relationship is often somewhat adversarial as vaguely defined requirements are easily misinterpreted, and late requirements lead to arguments about whether they are inside/outside fixed-scope contracts and the additional time/resources needed to complete them.

Agile encourages a much richer partnership. There is an agreed vision that everyone is striving to reach together. Each side understands the other has a vital contribution to the project success. Agile is premised on the existence of a much higher trust level between business people and developers and it is extremely important that both parties act in good faith at all times.

The customer team and development team interact on a daily basis. If at all possible, a member of the customer team is co-located with the development team, so any customer-related questions developers have can be immediately answered. If the development team is going slightly off-course, the customer can immediately refocus them as appropriate. Developers are not wasting time seeking answers for key customer information – the onsite customer should be able to answer most queries as they arise and if not, should be able to access the relevant customer expert to do so.

3.5: Principle 5

“Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.”

Creative, experienced, educated, motivated people are the keys to success in any intellectual endeavour. Assuming the training and recruitment managers have taken care of the first three of these, how does one motivate software developers? For management to discover what motivates the development team, they should just ask them. Often the answers will include interesting projects, reasonable timescales and new technologies. They appreciate having the resources and organisational backing to complete projects and being trusted to participate in project decision making. Developers who are motivated will be more productive as individuals than those who are not. The fewer developers that are needed on software projects, the better the team as a whole performs (shortened communication paths, greater scope for tacit knowledge).

3.6: Principle 6

“The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.”

Face-to-face conversation has the highest knowledge transfer ratio of any communications mechanism. The body language of the speaker and the listener conveys useful information to the other person (it is often clear when the listener does not really understand what the speaker is saying – and hence the speaker can tailor the discussion). Any questions that arise can immediately be asked and an answer provided and then the conversation continues with this clarification. In contrast, imagine if the two people are in separate locations and knowledge is transferred via documents and questions can only be asked over email. When a question arises, there is significantly more effort needed to ask the question, it is likely that fewer questions will be asked; and it interrupts the current activity (reading the document) and the answer may be some time in coming (e.g. tomorrow); there is suboptimum activity in the meantime as the questioner either continues reading the document without knowing the answer or waits until the answer arrives (which will shed more light on what is being read).

The Agile Process encourages the development team and a customer to work together in the same room. This is feasible for teams of five to ten people. It greatly assists the generation and sharing of tacit knowledge. If the customer cannot travel to the development team, then it can make sense for the development team to travel to the customer.

People can be separated by time, by geography or by contract. The fewer separations there are within the development team and between the development team and the

customer team, the more successful the project will be. Though we recognise that the optimum communications mechanism is face-to-face conversation, we also acknowledge the need for other formats (e.g. documentation) where greater separation occurs. Knowledge stored in documents can be useful when sharing information over time or geography and is necessary for certain types of business contracts (fixed-scope projects, with no variation in scope permitted). For larger teams, there will also be greater separation and hence there is a need for more structured communications.

3.7: Principle 7

“Working software is the primary measure of progress.”

How does one know if a software project is progressing? Work may certainly be carried out but is its accumulation progressing towards a goal? Should we measure progress by how much money has been spent? Or the number of days the project has been running? Or the weight of technical documentation produced? Or the number of lines of production code written? Or the number of dialog boxes in the user interfaces or tables in the database? Or the amount of product sales of early iterations? Or what?

Construction crews building roads can easily measure progress – the length of new road built in any given period. Light bulb factory owners can easily measure the number of bulbs produced. Software teams and their customers have a much more interesting problem to measure progress.

Why do we wish to measure progress in software projects? Both the customer team and the development team have their own reasons. The customer team:

- will have a deadline in mind for deploying the software so it can generate business value;
- Will recognise that the software may be just part of a larger business initiative that involves expenditure of additional money not directly related to the software development (end-user training in preparation for deployment, marketing for new product offering, purchasing of servers) and the software availability plays an important role in optimum scheduling;
- Will wish to see concrete results from its early investment – so often will pay for software development in instalments and wish to see regular progress as a result

The Development Team:

- Probably has another project lined up after this one and needs to allocate wisely its time commitments;

- Has a budget for the current project and would like to keep costs within that;
- Will wish to manage cash flow sensibly, which often means receiving regular payments from the Customer Team based on software value delivered

We need a common measurement of progress that both the development team and the customer team can agree on and that suits their different needs. Increments of working software is the ideal basis for measuring progress. This is end-to-end functionality that the customer can actually use and both the customer team and development team can agree is present or not.

3.8: Principle 8

“Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.”

Agile is based around short iterations and frequent releases. It is important that all activities are sustainable. There may be some variation in commitment over time, but if there is excessive work effort on one iteration it will tend to have a negative effect on the next iteration.

The term “death march” was coined to define the key characteristic of waterfall projects from a developer's viewpoint – that of being forced to work all possible hours, including late evenings and weekends. This was caused by waterfall being an inefficient way of organising software development projects; that integration of components occurred late in a project and only then were mismatching component interfaces detected and customers only saw the implementations of their requirements near the end of the project and this often caused many late change requests. Excessive workloads are unsustainable – leading to de-motivated developers, poor health, bug-infested code and ultimately when the developers get sense, they leave the project. It is noted that some integrated circuit design companies has a rule that their designers never work late or on weekends. Their work must be of extreme high quality and studies show most bugs are inserted outside core working hours by tired workers.

Agile strives to even off the pressure over the entire project. It is more efficient and more productive. It avoids late breakage during integration by continuously integrating (a few times a day). It encourages constant communication with the customer so there are no late surprises. It establishes a sustainable rhythm to the project, based around the subscription model, with regular delivers to customers.

3.9: Principle 9

“Continuous attention to technical excellence and good design enhances agility.”

The fact that a team is using the agile process to create software does not alleviate the need for technical excellence and quality design. Developers need mastery of the technical foundation (programming languages, frameworks, operating systems) for their projects. Developers need to understand modern design paradigms (service oriented architectures, object-oriented design, design patterns, etc.) .

By ensuring technical excellence and good design in the working software produced, the development team enhances flexibility and ultimately makes it much easier to respond to changing customer needs, which is the essence of agility.

3.10: Principle 10

“Simplicity--the art of maximizing the amount of work not done--is essential.”

The best software engineer is the one who writes least amount of code. Since half of all software functionality is never used, surely the easiest way to improve productivity is not to bother to write half of it? “But which half” is the usual reply.

Heavyweight methodologies have a tendency to produce massive sets of requirements. Customers get the impression they have a single shot at defining requirements upfront (and if they forget something, there will be an exorbitant additional cost), so there is a naturally tendency to request everything one can think of. Unfortunately in this scenario, at the time customers have the least clarity about what is needed - at the beginning of a project – they must somehow correctly identify these requirements. They really do not know in detail what they want - they only have a fuzzy picture.

With agile, moving from the fuzzy picture to the crisp picture happens over multiple iterations. The customer defines a little of what is needed at the beginning, the working code is delivered, and then the customer can define some more. At each iteration, the customer defines the most valuable requirements and these are usually the ones which it knows most about at that time. Over the iterations, the customer learns more and more about what is really needed. Hence with agile, customers are in the most knowledgeable position just as they define the requirements.

3.11: Principle 11

“The best architectures, requirements, and designs emerge from self-organizing teams.”

Micro-management of software development teams has not been a success on many software projects. Developers intensively dislike being treated similar to workers on an assembly production line. Frederick Taylor (of Scientific Management fame) and his stopwatch have no place in a modern software development environment.

The most creative software developers will be those given the freedom to organise their own development. Teams will come up with appropriate organisational structures and practices to deliver quality work products.

When using the Agile Process, the development team is empowered to seek out optimum organisational solutions for the tasks it needs to carry out. This is not some abstract concept, but rather the customer team has faith in the development team to make wise decisions with regard to how the software is developed.

3.12: Principle 12

“At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.”

Since the development team is self-organising, it needs to ensure it is progressing in the best possible direction. The Agile Process encourages tuning and customisation. Over the course of a project developers see what is working or not working, and can reflect on how to improve their development process. The end of a development iteration (e.g. monthly) seems to be a good time to carry out such a reflection.

4: Practices Of the Agile Process

4.1: Overview

The Agile Process is not a perspective process, in the sense that it does not mandate that developers first do A, then must do B and then must do C. Rather, the Agile Process describes a number of practices that each in their own way have provided useful on projects in the past. Each development team is empowered to pick and choose from among these, to modify or to invent its own additional practices, to optimally deliver value to the customer.

4.2: Information Radiators

A term coined by Alistair Cockburn, an information radiator is anything that radiates information. Flipcharts, whiteboards and walls covered with posters are all samples of information radiators. By presenting useful information on public display around the development team, developers can see them from where they sit and as they walk past them on a daily basis – and hence are far more likely to absorb their contents.

It is up to the team to decide what should be on the information radiators and over the iterations of a project this may change. Sample topics include burndown charts showing how progress maps against the planned end of the current iteration, open defect count, CRC diagrams or architectural alternatives. One useful display could highlight which user case each developer is currently working on.

Information can be created by hand (often a result of a few developers communicating near a whiteboard) or electronically (often the result of collection of information from different sources over time). With the availability of moderately priced large format (e.g. A3) laserprinters and pen plotters, it is now possible to print out interesting information.

4.3: Roles of People in the Agile Process

The agile process has far fewer roles for people compared to other processes. Many of the roles are combined and overlap is permitted.

On the customer team, one could have:

- Sample User (on-site with development team)
- Business Expert (deep understanding of customer's business – unlikely to be continuously available to work on development team)

- Executive Sponsor / Product Manager (ultimately responsible for deciding on scoping/priority issues; paying the development team)
- Operations Manager/System Administrator – ensure the orderly running of the software (server software)
- Professional Services - Tech support, consultancy, training, custom development

On the development team, one could have:

- Project Manager
- Architect
- Developer
- User Interaction Designer
- Security Specialist
- Tech writer
- CTO / IT Manager

4.4: Shared Code Ownership

“Code ownership” is the term developers use to describe which developer is allowed to change which source files. Some teams have individual ownership, whereby a single developer is responsible for each component and only that developer edits code in that component. Other teams use collective ownership, whereby the entire team have collective responsibility for all the components and any developer can edit any piece of code.

Having individual ownership means but that a single person can get to know a block of code and the technologies it uses extremely well, that there is coherence between all the ongoing changes to that code and that there is clear demarcation of control. However, it also means that if that one person leave the project permanently, someone else has to get up to speed on the departing person's code, if the person is away temporarily (holidays, sickness, visiting a customer), then no changes can be made to that code, which slows the team down. Some teams relax the arrangements in this situation, whereby if someone is offsite that that person's code can be changed. The individual ownership approach also sometimes leads to what we might call “skill silos”, where developers on a project know one aspect of it only, and do not have a good overall picture.

Having collective ownership means that all developers get to somewhat know the entire codebase and the technologies it uses. Anytime a change is needed it may be

implemented immediately by whoever is present and available on that day. If a particular developer is under scheduling pressure, then a different developer can come to their support and perform some of their allocated tasks. When a developer permanently leaves a team there is no vacuum of knowledge, as other team members already know the entire codebase. The first problem with collective ownership is scaling. Once you go beyond a certain teamsize, it is unlikely that a developer will be competent to edit the entire code base. There is too much code, too many technologies and too many things can go wrong. At what teamsize this becomes a problem depends on the capabilities of the team members and the diversity of technologies on the project. Often the same code needs to be edited for multiple use cases, and having this work performed by a single developer increases the likelihood of an optimum solution for all of them. A developer who early edited a piece of code, and now needs to call it from somewhere, has a mental model of how it works, but this may not be a validate model as in the meantime someone else may have changed that code.

The Agile Process advocates shared code ownership, whereby a group of developers is responsible for each piece of code. There should be more than one developer in the group, so that is one if offsite/leaves someone else can easily take over; but there should be a small number in the group (perhaps 3-5) so that we do not have too many people editing the same code and causing the problems with full collective ownership. Shared code ownership is a model that will scale well, and strives to maxmise the advantages and minimise the disadvantages of both individual code ownership and collective code ownership.

4.5: Frequent Integration

When constructing a tunnel from both ends, you really have to wait until you are almost finished to determine if the integration will work.. With software, it is quite different. It is quite feasible to integrate the components as they become available and solve any small problems immediately, before they become large problems later in the project.

The classic problem with the Waterfall model is that of “late breakage”. Different groups of developers work away on their own little pieces; and the integration does not occur until late in the project (the integration phase). When mismatches are detected, it is quite expensive – both in terms of costs and time, to fix. To counter these problems, the Agile Process recommends frequent integration, whereby code is integrated as frequently as is feasible. At a minimum teams should strive for daily builds, and if possible even a few times a day. There are plenty of problems during software integration and this approach will raise their visibility much earlier and much closer to when the code was written. As it is fresh in the developers' minds, they will fix it much

quicker.

The Waterfall model attempts to avoid late breakage by striving to create the perfect design before coding starts. There is a tendency to create copious amounts of documentation that rigorously defines how the code should behave. The Agile Process places less emphasis on documentation and more on experimentation. It encourages developers to write exploratory pieces of code and if these solve problems integrate them with the rest of the codebase and fix any problems as they arise.

When carrying out frequent integration, the question sometimes arises as how to handle the situation when a piece of software takes a few days to develop, and is not usable before that. First we should say that most classes/components can be developed in multiple stages and with a bit of imaginative work most will be usable to some degree on a daily basis. For those that remain, we suggest the use of interfaces, stubbing or temporary subprojects. If one piece of code must make calls into a class, and a different version of that class is being developed, then the simple use of interfaces allows both classes to be part of the main build – the calling code uses the interface, and the implementations of the interface are interchangeable. Stubbing allows the function signatures of a class to be defined as part of empty methods and over time their implementations may be added. Temporary subprojects allow major blocks of code to be developed independently and later integrated into the main project. If this takes a few days, that is fine – it takes longer the risks grow that undetected integration problems exist.

4.6: Value-Driven Development (VDD)

What should be the key driver for a software development project? The literature discusses “use case driven development”, “feature driven development” and “test-driven development”. The driving force behind all these approaches is delivering customer value, so we use the term value-driven development. The only people who can decide what constitutes customer value are the customers themselves. From a customer's perspective, the terms “use case”, “feature” and “test” are all the same. Developers used to the waterfall model find this difficult to understand, as from their experience requirements happen at the beginning of the project and testing at the end. Agile developers understand that requirements are a description of how the finished system should behave and tests are also a description (actually, a far more precise description) of how the finished system should behave.

The Waterfall model can be considered a plan-driven approach, where each step is carried out strictly in conformance with a plan. These may or may not indirectly deliver customer value.

Any knowledge activity, such as software development, involves tangible and intangible results. With software, tangible results include working software and developer documentation. It is a common human trait that what is measured will be optimised, so it is important to choose wisely. When we base our measurement of progress on delivering value to the customers, that that is what will be optimised, which makes sense.

4.7: High Level Vision

Each project should have a high level vision that defines the goals of the project. The team can decide how this should be presented, perhaps as an information radiator, or as a short document (e.g. 2-pages) or as part of the requirements document.

The vision should articulate what the projects is trying to achieve, who it is for and any assumptions or dependencies that exist.

Some independent software vendors (ISVs) create the marketing data sheet for a product at the beginning of the project. The motive is that it is useful for the development team to see how the product will be presented to customers.

4.8: Gather requirements as they become available

Customer requirements do not arise neatly at the beginning of a project or even at the beginning of development iterations. Requirements are discovered throughout the project and it is imperative that the development process cope works with this (agile does) rather than against (the plan-driven approach does not cope well with late requirements).

Whether requirements should be stored in an (archiveable) document or on index cards that will later be destroyed once the software is complete is a decision for the customer. If the project is once-off and will not be followed by updates, or the same team stays on the project then having temporary requirements storage any suffice. For most projects with a team that changes over time, with multiple iterations where later ones often modify existing requirements, a requirements document becomes more interesting.

Visibility of the requirements to be implemented during the next development horizon should be high, after that it may be somewhat obscure. However, if requirements are available early on, they should be collected. It does help developers to have a broader understanding of what is coming up in future iterations to optimally implement the current iteration. Imagine a user interface form that needs to display a list of items. If

the quantity is 3, then radio buttons would seem appropriate; if it is 20 then a listbox could be used; and if it is 2000 then a Google-like approach of display the first 10-20 items with options to display more or to refine the search would be appropriate. If that quality requirement is available early, it should be captured so that the UI developer can implement it correctly first time.

4.9: Encourage Flexible Scoping

The customer team is concerned about what it is getting for its investment. The development team is concerned about what it is committing to deliver. The agile process is premised on a high trust level between the customer team and the development team. Fixed scope projects are based on the twin premises that it is easy to define precisely what the customer wants, and the customer knows what it wants. When building a road between two cities, With software, it is not quite so easy. Customers often really do not know what they want until they see an early implementation of the working software. Also, it is quite difficult to precisely define requirements. An online retailer might have a requirement for a product display page for its website – is this something a student can put together in a day or the likes of the Amazon product page? With software, there can be some variability in the meaning of requirements. A third issues with scope and software is that requirements tend to be far more dynamic. (if a government needs a road between two cities this year, it probably needs it next year also). Dynamic requirements and fixed scope projects tend to be a problem.

Sometimes (e.g. government contracts) fixed-price/fixed-scope projects are mandatory and for these a best effort need to be made to gather detailed requirements.

Note that chapter 7 of [Poppendiecks] has an interesting discussion of types of software contracts. Multistage contracts seems useful in many situations. There is incentive for the development team to perform well so that it wins the next stage of the contract, and the customer knows it can get rid of the development team if that is appropriate.

4.10: Daily Stand-up Meetings

Agile encourages a much higher visibility of progress and problems within the team. Once a day (usually first thing in the morning) the development team should briefly meet to identify progress, outline plans for the day and identify anything that is slowing progress. Such a meeting should not last longer than ten minutes and this usually can be encouraged by the simple expedient of everyone standing up for the duration (also means it can be done on the common area where the developers work and does not require a dedicated meeting room).

Such a meeting avoid the problem of developers “going dark” when working on a problem and not reporting on progress for a long duration. Within any development team, there is a wide range of experiences and learning. For difficult problems, it is sensible to harness that. If a problem is taking too long to solve, there is a need to apply “swarm intelligence” of the team to overcome it.

4.11: Occasional Pair Programming

Pair programming is when two developers work together at the same computer to code or debug. Members of the development team have different characteristics - different skill sets, different perspectives and different ideas. By pairing, they bring a great range of talents to bear on a problem, carry out continuous code review and encourage knowledge exchange. The fact that two people are working on the same problem does not mean productivity is halved. In many cases, they should get the job done quicker and better, thus avoiding later rework. It is questionable whether all developer activities should be paired. The Development Team is empowered to decide for itself how much pair programming is needed.

4.12: Sustainable Pace

To be written!

4.13: Risk Management

4.14: Avoid Functional Decomposition

4.15: Responsibilities and Behaviours are Key to Good Architecture

4.16: Initial Architecture Skeleton

4.17: Common Workspace For Development Team

4.18: Iterations & Release Planning

4.19: Team of Teams

4.20: Clearly Define Contract Models

4.21: Understand Stability vs. Flexibility Tradeoffs

4.22: Travel Light

4.23: Templates and Coding Standards

4.24: Avoiding Multiple Independent Representations of the same Information

4.25: Customer Team

4.26: Test-first

4.27: Waste Elimination

4.28: Learning Through Exploration and experimentation

4.29: Set based development

4.30: Decide at the last responsible moment

“Wisdom blooms late in a project”.

5: Work Products

5.1: Overview

The primary delivery of software developers is working software. In performing this task efficiently - for the current iteration and future iterations, additional work products need to be delivered. We divide what is needed into two broader categories – department-wide and project-specific.

5.2: Engineering Group Specifications

These are specifications that are applicable to all projects within a software development department in an organisation.

5.2.1: Corporate Software Development Strategy Specification

This document defines the general strategy for a software development organisation. This includes what products it will be developing, lists their general capabilities and high-level features, identifies what user needs the organisations should fulfil and outlines the competitive offerings. It also includes a version and technology plan. Consider it a starting point for the projects the organisation will be working on over the next 18-24 months. .

It should be jointly written by the various stakeholders – CTO, product managers, project managers and marketing personnel (if targeting external customers) or appropriate operations management (if targeting internal users).

It should be reviewed by those involved in coordinating different product offerings and decide on build/buy analysis; by those gathering requirements for projects so they are clear what requirements need to be collected; by technical architects so that at this early stage what is identified makes sense to build; and by sample end-users so that it identifies their real needs.

5.2.2: Coding Standards and Code Review Specification

All code should be written to a consistent coding standard. As developers move between projects and take over code written by others, common coding techniques eases the joint development. Developers should be encouraged to regularly carry out code reviews and a checklist of what to look out for can assist in this endeavour.

5.2.3: Software Development Process Specification

A software company should be able to explain to potential customers and new employees how its development teams approach software projects.

For teams that have decided to use the agile process, this document you are currently reading aims to satisfy this role.

5.3: Project Specifications

Project specifications are the documents that may be created for a specific project. The development team is empowered to decide for itself what specification are needed.

5.3.1: Requirements Specification

This document defines the requirements that the software project must satisfy. It makes strong use of use cases for functional/behavioural aspects of project. It should be written by the Requirements lead and requirements team. It should be reviewed by all members of the project team, product managers and marketing personnel, along with sample end-users and their managers.

5.3.2: Architecture Specification

This document defines the high-level architecture for the system. Identifies sub-systems and how they interact. Includes discussion of process/thread architecture, messaging constructs; file system layout and installation system startup. It should be written by the technical architect. It should be reviewed by all members of the project team.

Design notes and models are also recommended where appropriate.

5.3.3: System Test Specification

This document defines the testing arrangements for the system as a whole.

It should be reviewed by all members of the technical team. If it is to also represent acceptance test, it should be reviewed by customer representatives who will be performing the acceptance test.

5.3.4: Security Threat Model Specification

A document to explain what threats the application faces and how they can be defended against.

5.3.5: User Interaction Design Specification

Need to be able to describe the task model, interaction model and view/navigation model of the user's interface to the system.

5.4: Other Communications Techniques

A variety of team communication techniques can be used to store and represent information. Project-specific information should be stored in one accessible place and this information should be achieved along with the rest of the project artefacts (e.g.

Source code). They include:

- wiki / blog
- mailing list
- Instant messaging or irc
- Video
- Digital Camera pictures of “information radiators”

5.5: Project Lifecycle Documents

- Technical Presentation – often need to be able to explain the technical concepts of projects to interested parties
- Project Plan / Status – need to specify what will be completed over the next iteration and more high level plans for future iterations and releases
- Risk & Issue List – Need to keep track of risk and other issues and determine that they are alleviated

5.6: Additional Documents

Additional documentation could be provided for some of the following:

- Product Marketing Plan (marketing collateral)
- Product Introduction Plan
- PR Plan
- Professional Services Plan
- Training Plan
- Helpdesk
- Community Plan

These documents are usually prepared by people outside the development team, but they may need input from developers from time to time.

6: Agile Process Review Checklist

6.1: Overview

The Agile Process defines values and principles that guide development teams in how they carry out projects. It is not prescriptive in the sense that it does not mandate a precise set of steps that developers must follow in every situation. Instead, it offers a series of base practices that are compatible with the agile values and principles. From these developers are encouraged to pick and choose the practices that benefit their current project and to create their own practices as they see fit. Each team's situation is different and hence tuning and customisation of the process they use is expected.

Since the agile process is such a flexible approach, the team is encouraged to regularly review how it develops software and to continuously improve how it functions. It is recommended that a short review be carried out at the end of each iteration (e.g. monthly) and a more substantial review once or twice a year.

6.2: Review Questions

The following is a selection of questions that may be considered during a review. Each team will of course have additional questions specific to their project.

6.2.1: The primary review question

- How can customer satisfaction be further improved?
(When using the agile process, our highest priority is to satisfy the customer, so our review should start with this)

6.2.2: Development Team

- What is an honest assessment of how the development team is performing?
(consider productivity, quality, motivation)
- The development team must deliver projects within constraints based around quality, cost and time. For each, is it being optimally achieved or are there shortcomings?
- Is project scope being correctly managed over the iterations?
(Agile encourages dynamic scope on projects and customers using learning from early iterations to decide on scope for later iterations)
- Of the existing practices in use, which should we keep, customise or dispose of?
- What new practices should we incorporate into our development?

6.2.3: Development Environment

- Has the development team got appropriate development resources (machines, software tools, books, etc.)?
(For example, if developing multithreaded software, need to be able to test it on both single-processor and multiprocessor boxes).
- Is the office space for the development team organised appropriately for agile concepts? (Common work area, occasional pair programming, limits to interruptions from outside the team)
- Are appropriate team communications facilitated? (“information radiators”)

6.2.4: Work Products

- Is the customer satisfied with the permanent work products (in addition to code) that the development team has produced?
- Are temporary work products discarded before they get out of sync with running code?
- Is the development team delivering work products that satisfy the key goal of setting up for the next iteration?
(The types of work products - e.g. documentation - needed depends on a variety of factors - such as whether all/most of the development team will continue on the next iteration – and hence tacit knowledge may suffice).
- Are contract interfaces (for code that traverses team boundaries) clearly documented and not subject to multiple (incompatible) interpretations?

6.2.5: Customer Team

- The development team has a dependency on the customer team to perform well in accordance with agile concepts. What is an honest assessment of how the customer team is performing?
- Has the on-site representative of the customer team delivered a rapid and accurate representation of customer requirements and concerns by the time a feature is being implemented?
- Where necessary, has the on-site representative of the customer team sought out expert customer knowledge for more advanced requirements?
- If misunderstandings have arisen between the development team and customer team over any topic despite the genuine best efforts of participants, why has this occurred and how can it be avoided in future?