Agile Requirements

Articles

Posted by:
Posted on : 2007/8/31 7:34:16

In the last few years, the agile software development movement has created a paradigm shift in how we work to understand system requirements. Agile teams shape software systems using a collaborative process, with executable software at its heart and documents marginalised to a peripheral role. This creates a fundamental shift away from tools for managing requirements artefacts. Instead, we need tools that support collaboration and the gradual distillation of business rules into automated test suites.

Author: Rachel Davies, Agile Experience Ltd, UK, http://www.agilexp.com

Agile software development is a general descriptive term for an approach that is implemented by a number of different software development methods, such as XP, Scrum and DSDM. The common ground shared by agile methods is that they support the values and principles embodied in the Agile Manifesto [1]. Agile software development evolved in response to deficiencies with document-heavy waterfall development. It is useful to begin by outlining some of the limits of the traditional approach to requirements before moving on to the main subject of this article, Agile Requirements.

On traditional software development projects we talk about capturing the requirements -as if we are engaged in a hunt to cage some wild beasts that lurk in the jungle. What is actually meant by this phrase is a less exciting activity - producing documents, which specify a system in sufficient detail that software development can proceed by reference to these documents alone. If you cast your mind back and imagine all the requirements documents that you have ever read piled up, it is likely to be a hefty stack and the hours spent poring over those documents innumerable. Although, it is theoretically possible to capture my knowledge in a document from which the reader can extract the same knowledge without distortion, this may not be a practical or efficient way to communicate requirements for complex software systems.

In determining our process for developing software it is important to remember that the primary purpose of a development project is to deliver is a software system that generates value to a business. Models and documents are essentially by-products of the development; they do not generate value directly. If we are to optimize the flow of value from software development then we need to think seriously about eliminating process inventory and waste.

Limits of Documents

A typical programmer’s experience with traditional requirements is working from a document written in dry, formal language, that describes a desired software system. By studying this document, s/he gradually builds a mental model that guides the development of an executable software system. The requirements document acts as both means of communication and data storage (preserving

evidence of the request). When you put yourself in the place of this programmer there are several flaws with the use of documents as a communication medium that you are likely to encounter. Documents are selective and unidirectional; they may also be ambiguous, vague and conceal gaps.

Unidirectional

Documents are a one-way communication medium; information flows from author to reader. There is no opportunity for the reader to ask questions, offer ideas and insights. The author may try to anticipate questions but cannot realistically be expected to address everything and in an attempt to cover everything, a concise abstraction may be lost in a forest of pages.

A programmer working from a document is likely to find parts of the document difficult to understand. This may be due to poor choice of words by the author or lack of reader’s background knowledge. How does that programmer get to the bottom of the intended meaning? In a corporate context, s/he may not have met the author and the normal channel for questions will be correspondence via email, which takes time. If working under time-pressure a programmer may make their own conclusion about the intended meaning to avoid delays. Getting answers to questions takes time but guessing the answer can lead to developing the wrong software system.

From the author’s perspective, there is likely to be a delay between writing a document and receiving questions from readers. Any gaps spotted by readers will require the author to update the document. Documents take effort to maintain with significant ceremony to sign-off each new draft. This makes it possible for documents to fall out of step with current agreed working understanding of system requirements.

Selective

The author writes from their personal perspective on the system under development and naturally makes assumptions about the background knowledge of the readers. Requirements documents often describe what is required rather than why – outlining a desired solution rather than explaining the problem space. Little is said about the needs of the users and business context surrounding the system development. Following traditional development process it is assumed that the programmer does not need to know this information because requirements are chosen purely on business grounds. The job of the programmer is limited to the technical implementation of a system rather than contributing ideas on how to achieve business benefits. However, during the implementation many micro-level decisions will be made by the programmer, an awareness of the system context could help with these.

Cost

When we work in the abstract realm of pure analysis, we are detached from physical constraints such as implementation costs and limits imposed by technology choices. If requirements are considered from a purely business perspective then we may get so creative that we specify a system that exceeds our budget.

When planning a project it is useful to be able to identify the priority of requirements so that we can consider delivering the high value features early and can mark nice-to-have features as potential

contingency for schedule slips. However, traditional requirements specifications fix the scope of the entire development, all system features described are deemed necessary and their descriptions are intertwined throughout the documents. Expressing requirements in a meshed form makes it difficult to trim scope in response to development delays.

Freezing

The traditional approach is to document the requirements for the whole system and then freeze the requirements to provide a stable base for implementation to proceed. This assumes that we can come to a complete understanding of the system before developing real code. Perhaps this is possible in simple or well-known domains but this is a risky assumption to make for complex software development projects.

This raises a fundamental question of whether the human brain is capable of building a virtual system in our imaginations that we can explore and document. In a traditional waterfall process, requirements documents are the culmination of an analysis phase, during which the problem space has been explored via abstract models. To make this task manageable, we break the system down into pieces. However, we still need to keep track of all the pieces so that we may compose a coherent system from them. The human mind has limited capacity for complex problems, it is to be expected that when working in the realm of abstract ideas, we may miss scenarios, stakeholders’ needs and functions that need to be supported by the system. It is not clear that we can we fully determine the system requirements without starting to develop some software to explore possible interaction patterns. In his book "Serious Play", Michael Schrage [2] suggests that we can only really determine what we want by interacting with a prototype.

Let’s contrast the traditional approach to requirements in software development with how we operate outside the software domain. For example, if we are investing in home improvements such as a new kitchen or bathroom – do we just place an order based on drawings and make no comment until the installation is complete? Most people would be keen to see samples of new fittings and retain the right to change their mind about some aspects as installation proceeds. We would naturally expect that, as the fittings are installed in situ, we are likely to spot some ergonomic issues that were not apparent from the original two-dimensional plans.

The final flaw with freezing requirements early is that we are developing the system for use in a changing world. Time does not stand still during software development, the world around us changes – new business opportunities may arise during the development. When we freeze requirements early we deny the chance to adapt the system to a changing context.

Agile lifecycle

The core of agile software development is building system features incrementally by working in a collaboratively. Agile methods are all built around a key assumption – we continue to learn about a system and it’s operating environment during the project. When we recognize that our understanding develops during the project then we realize that freezing requirements during early project phases will either cause rework or the delivery of a system that does not meet the customer needs. We need an approach to software development that can embrace change during the project. Agile methods iteratively evolve both requirements and the system under development by planning

the development in short cycles (weeks rather than months). This is not a new idea [3].

When we develop the system incrementally then this allows the customer/user to try the software developed during the project rather than waiting until the end. Their feedback can be used to refine the system behaviour.

When we move to an iterative lifecycle, we lose the traditional waterfall phases: analysis, design, code and test. These activities now run in parallel. This has an impact on how we organise a project team. Rather than assembling specialist teams dedicated to waterfall phases with handover artefacts, it makes more sense to form a cross-functional project team; a team of individuals whose task is to shape and deliver a software system together. Business people are needed throughout the project to explain how the system under development will generate business value and support the user community through different scenarios. Technical people are needed throughout the project to translate those requirements into executable code and verify the system meets conditions of satisfaction.

Agile communication

When a team work on analysis and software development in parallel, this creates an opportunity to work with requirements in a different way. Specifically, we can consider other communication channels - we can open up dialog. Agile teams primary means of communicating requirements is conversation. The medium of conversation creates the possibility for information flow between all parties. The problems imposed by communicating via documents drop away.

When agile teams plan development, this done by the whole team in a workshop. If you listen in, you will hear conversation flowing around the group. The whole team are partners in this conversation. They discuss scenarios and development options to explore the requirements before development is planned. The beauty of exploring requirements through team conversation is that we develop a shared understanding and have the opportunity to offer our own ideas. Having a two-way conversation between business and IT and flushing out misinterpretation early may prevent developing code that has to be thrown away.

Stories

The difference is made clear in Extreme Programming (XP) [4]. The use of Stories is a primary practice of XP. An XP story describes a candidate system feature in language that is meaningful to both customer and developer. Stories have three essential parts: Card, Conversation and Confirmation [5]. Stories must be small enough to implement in a single development cycle (XP teams use a one-week cycle).. To support collaborative working, stories are written on index cards. Index cards provide a low ceremony way to document development plans. Each story card represents a conversation that has taken place. Automated tests are defined for each story, which are used to confirm that the system supports that story.

Other agile methods do not explicitly use stories but also emphasis that requirements descriptions need to be short and loose, the bare-minimum needed to plan development. Typically, requirements are listed as one line description maintained in a backlog in priority order. Features may be swapped in and out of this backlog and it is assumed details about each requirements will be determined as

part of each development cycle.

Requirements specification documents are usually expressed in impersonal and cold language giving us "whats" without business motivation. This is quite different from the way we tell stories. Using story form we start by describing the context and give background to the players in the story. A story takes the players through a chain of events that communicates a desired outcome. A story breathes life into a limp requirement. Through the story, the listener grows to understand the pain experienced by the users and the burning need for the software – this can have a profound effect on motivation within the software development team. By discussing explicit situations and our reactions to them (via the system under construction) we can achieve a shared understanding. Understanding the big picture can help programmers to develop software solutions that are a better fit.

Cards

A document is both a communication medium and a storage mechanism. Although a conversation provides us with a rich communication medium, the words may be lost in the ether with no record except in the memories of the participants. During team planning workshops, XP teams use index cards to make notes about stories being discussed. There is no formal form for a story card; it can be written in any way that is useful (although a short name for the story as a heading helps). These cards are used after the weekly planning session to create a visible plan on the wall. Each day the team holds a meeting around the plan to review the work remaining for the week. It’s possible that a few days may elapse before I start work implementing a story but because the team sit together I can replay the conversation with them before I start work.

Agile software development is not a series of mini-waterfalls. The notes on index cards are not the counterparts of requirements documents; this is a different process model. There are several tools available for managing electronic story cards – that seem to miss the point that the story card description is not a requirements document in the traditional sense and treat these as the prime artefact rather than the story tests. Index cards provide a non-intrusive way of documenting a conversation without disrupting the flow. The cards are not a simple input from which a conversation is generated; they are also an output of the conversation. A story card is not a stand-alone document describing the full requirement it is really a brief note that a token that to remind us of key details from a conversation. Our primary source of information during the development is conversation, made possible by co-locating the team.

The return to handwriting on cards appears to be a retrograde step. Surely, people in the software industry should be using the latest software tools for planning? To understand why index cards are preferred for planning with stories, you need to understand that paper has different "affordances" – interactions that it supports – and limitations. For example, spatial layout – it is easy to shuffle tasks written on index cards around to create different layouts. When planning it helps to review prioritization from different perspectives – by risk, by value, etc. Another affordance is visibility - it is easier for a group of people to read index cards spread out on a table than gather around a computer screen. Index cards can be annotated by grabbing a pen. Grabbing the keyboard in a meeting usually means changing seats. I have worked with teams who have used data projectors in meetings to make the computer screen visible to all but what happens is the group end up waiting for the operator to keep up with them. The way we interact with today’s planning software slows down the meeting dynamics. If we are to use software tools then we need to review

how they support group interaction and to remember, rrecording the conversations is a secondary activity, we don’t want it to disrupt the primary activity understanding the desires around the system.

Index cards have some limitations too. If your meeting group is large then some may not be able to read the text on the card. I encourage teams to "write big" using marker pens and move to using sticky-notes on a wall rather than a table if members of the team have problems.

It’s worth remembering that the drive for a "paperless" office [6] was motivated by the drive to reduce storage costs of large quantities of data - to remove paper records in filing cabinets. Index cards don’t hold a lot of data and can be easily bundled with a rubber band. The notes on the index card are a temporary holder of the requirement. The cards are only used for the purpose of planning; the cards are not intended to become a lasting record. Index cards can be thrown away after they have been translated into acceptance test scripts and committed to version control.

Confirmation

The first task in implementing any story is to create a set of automated system-level tests that clarify the scope of the story. These tests are used to confirm the story has been implemented as expected. These tests remove ambiguity by creating executable tests capturing examples of how business rules should fire.

In 2002, Ward Cunningham published the Fit framework [7] this provides a way for business users to write story tests using desktop tools such as MS Word and Excel. When using a test framework like Fit, tests are typically worked on by a customer-developer pair, fleshing out scenarios to be implemented as part of the story.

The great thing about these tests is that non-technical people can read them and so become living documents that specify the system behaviour and also check that the system continues to support the full suite of stories developed to date. In agile software development it is these customer level story tests that form the requirements repository.

Summary

The software industry has been stuck in an illusion that dry language is less ambiguous and that facts can be isolated from the context and the needs of the business driving the development of a software system. Requirements can be documented in an ivory tower of analysis before engaging with technical labourers who will be building the monumental system. Agile teams have learned that limited abstract ideas ought not to be frozen at the start of software development but need to be refined and explored within an iterative development process.

In agile software development, the sharing of stories binds the team together, as the storyteller offers up their ideas they are woven into the system. People show their interest by asking questions and sharing insights. Suddenly, everyone starts to feel like an equal partner in a project that they are engaged in together. Traditional barriers between departments fall away and people feel energised to make the possibilities discussed a reality.

References

1. Agile Manifesto - http://www.agilemanifesto.org
2. "Serious Play: How the World's Best Companies Simulate to Innovate" by Michael Schrage Harvard Business School Press (1999) ISBN: 0875848141
3. "Agile and Iterative Development" by Craig Larman - Addison Wesley (2003) ISBN: 0131111558
4. "Extreme Programming Explained" 2nd edition by Kent Beck, Cynthia Andres - Addison Wesley (2004) ISBN: 0321278658
5. "Essential XP: Card, Conversation, Confirmation" by Ron Jeffries [http://www.xprogramming.com/xpmag/expCardConversationConfirmation.htm
6. "The Myth of the Paperless Office" by A.Sellen, R.Harper - The MIT Press (2003) ISBN: 026269283X
7. "Fit for Developing Software" by Rick Mugdrige, Ward Cunningham - Prentice Hall (2004) ISBN: 0321269349 Originally published in the Fall 2005 issue of Methods & Tools