

The Importance of Being Agile

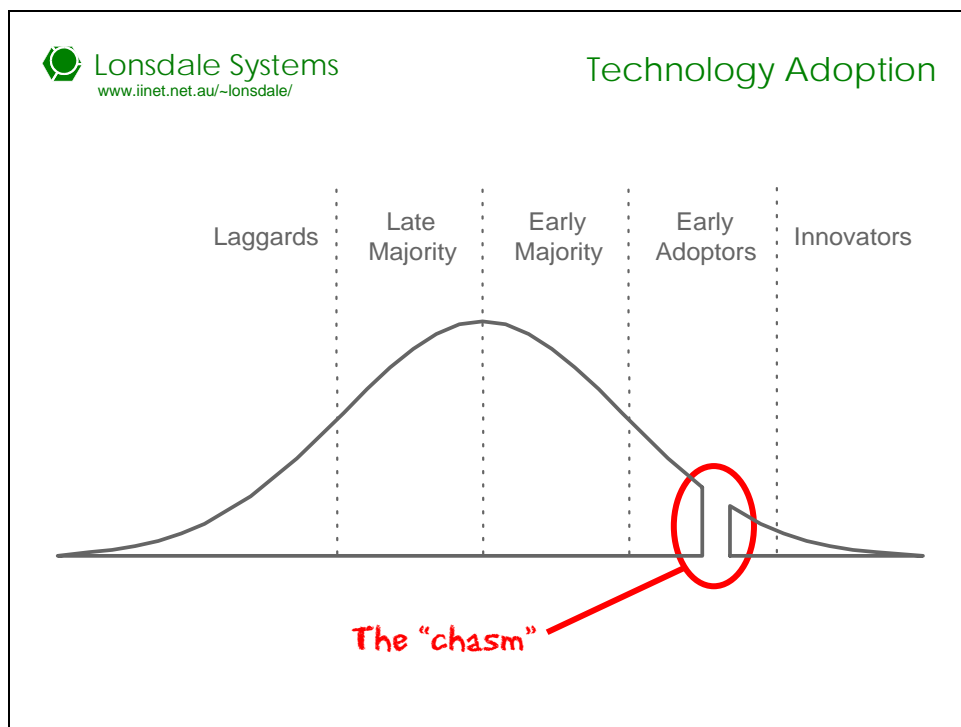
Phil Robinson

E-mail: lonsdale@iinet.net.au

Introduction

In some ways, “agile” development can be viewed as a “grass roots” reaction to the complexity of the “heavy-weight” UML-based methodologies. Sadly, as with many important developments in the IT sector, “agile” is being hyped beyond belief. This brief paper is intended to present the reader with an overview of the important aspects of agile development.

The Technology Adoption Cycle



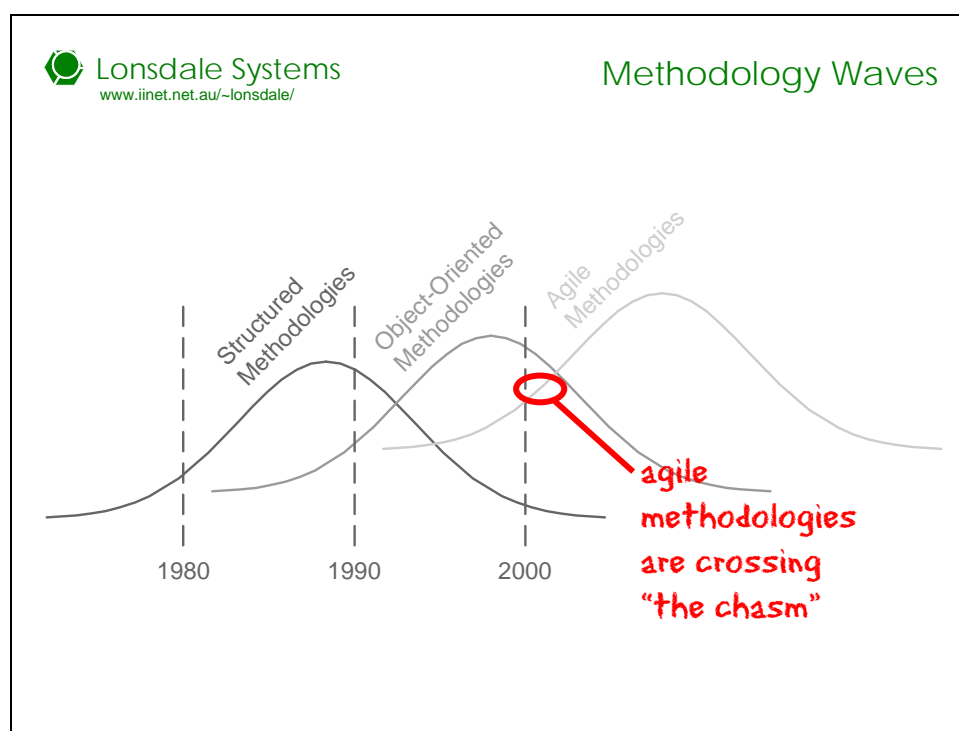
Adoption of new technologies follows a well-defined cycle which moves from initial interest by a small group of “innovators” to the tardy adoption by a group of “laggards”.

It can be argued that software development approaches follow roughly the same sort of cycle.

Not all innovative technologies become mainstream. First, they have to cross the “chasm” in the adoption cycle. The “chasm” represents a “credibility” barrier for a new technology. The “chasm” is crossed when early adopters of the technology confirm its value to the larger group of potential users. Technology can also be “marketed” or simply “hyped” across the “chasm”.

There are many examples of technologies that fail to make it across the chasm. For example, Betamax video, quadraphonic sound and the Forth programming language never really achieved the big time!

Waves of methodologies



Since the 1980's there have been two major approaches to software development - structured and object-oriented. Agile development represents a third approach and there is every indication that it is in the process of crossing the "chasm".

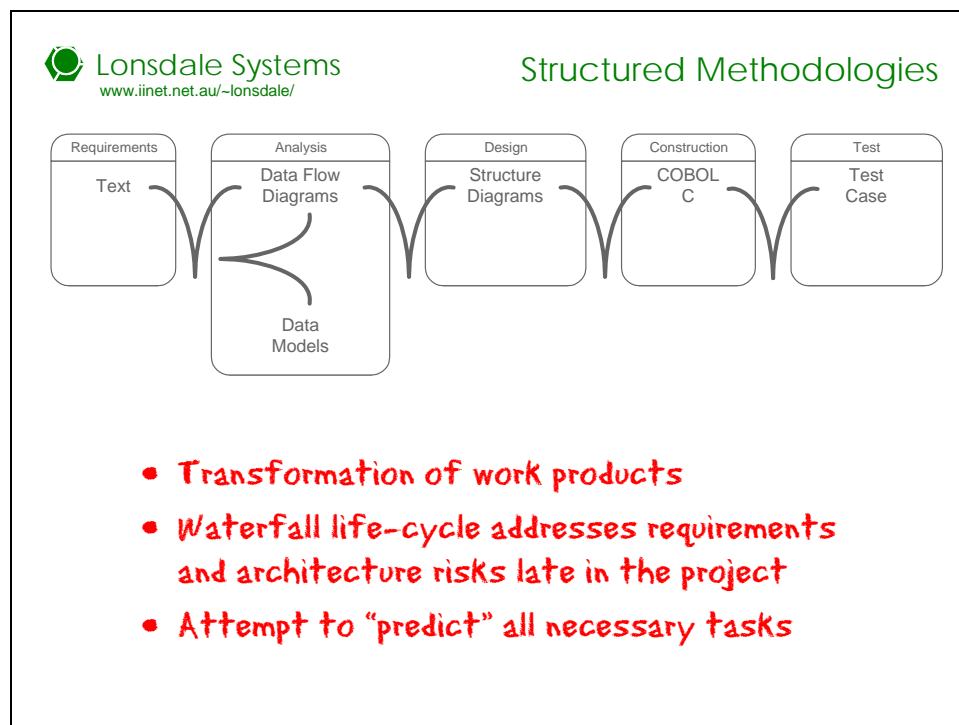
Before looking at agile development in detail we will briefly recap on the structured and object-oriented development approaches.

The 1st Wave – Structured Methodologies

Structured methodologies view software development as a process of transformation. Requirements are transformed into designs which are, in turn, transformed into program code and working systems. The major flaw with this approach is that no one ever fully explained how the transformation was performed. The result was a series of discontinuities in the development life-cycle. In practice when people moved from analysis to design, they effectively started all over again!

Structured methodologies frequently employ a large number of different techniques. Each technique is often based on a different underlying paradigm, such as Data Flow Diagramming or Entity-Relationship Modelling. The different paradigms often leads to further discontinuities within individual life-cycle stages.

Also, most structured methodologies are based on a “waterfall” life cycle. This assumes that the work products of a stage must be fully defined and approved before the subsequent stage can commence. This approach means that the two biggest risk factors in software development – requirements and software architecture – do not get addressed until the end of the development cycle.



Typically, structured methodologies attempt to predict and sequence all of the tasks required to develop software. They do this so that less experienced developers may benefit from the wisdom of more experienced developers.

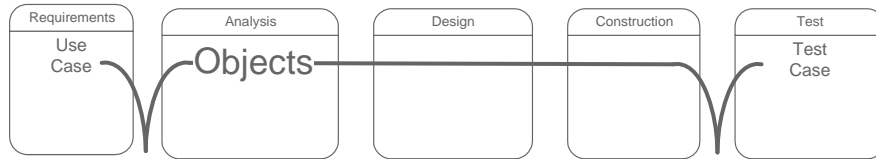
SSADM (www.ogc.gov.uk/index.asp?docid=1342) and the German Government’s V-Model (www.informatik.uni-bremen.de/uniform/gdpa/) are typical examples of structured methodologies.

The 2nd Wave - Object Oriented Methodologies

Object-oriented methodologies are all based on an “object” paradigm that can be used throughout the majority of the development life-cycle. The object paradigm encourages the development of software to be viewed as a process of evolution rather than one of transformation. This helps to remove the discontinuities between the different stages of the life-cycle.

The consistent paradigm also encourages an iterative approach to development. Iterative approaches allow the requirements and architecture risks to be addressed much earlier in a development project.

By far and away the most widely known object-oriented methodology is the Rational Unified Process (RUP) (www.rational.com/products/rup/).

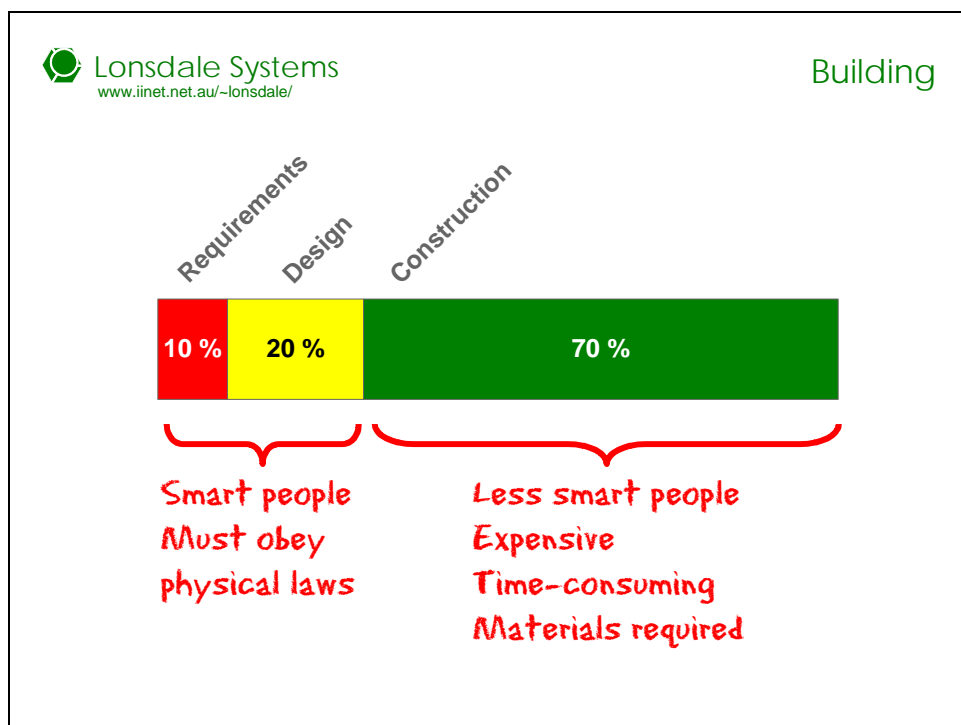


- Evolution of work products
- Iterative life-cycle addresses requirements and architecture risks early in the project
- Still attempts to "predict" all necessary tasks

Software Development Suffers From Too Many Analogies

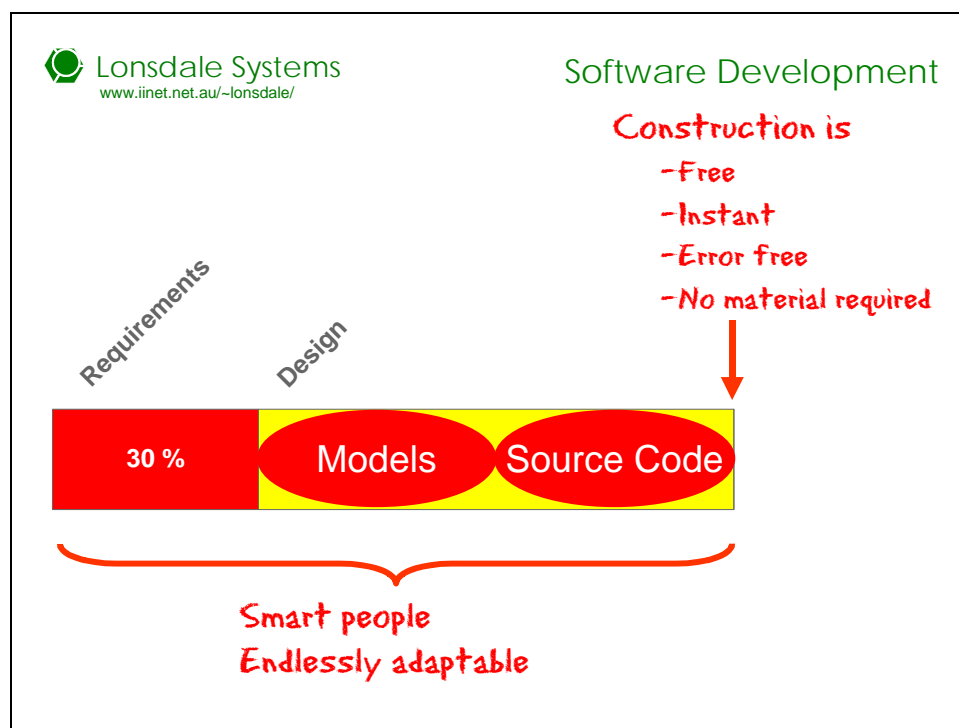
Because it is a relatively new discipline, it is very tempting to make analogous comparisons between software development and other disciplines. Building is a popular analogy. However, while this can be useful in a limited way, taking the analogy too literally can be dangerous. Lets explore why.

Characteristics of the Building Life-Cycle



Building construction has three stages – requirements, design and construction. Construction occupies 70% or more, of the time it takes to create a building. Requirements and design are done by smart people so that less-smart people can do the construction. Designs have to take account of natural laws such as gravity. Also building construction must address the logistic problems of the timely provision of raw materials at the building site.

Characteristics of the Software Development Life-Cycle



In contrast, software development does not have to obey any natural laws and does not require any raw materials. Unlike bricks and mortar, software is endlessly adaptable. There is very little opportunity to use “less-smart” (and less expensive) people.

This paper supports an emerging point of view that software construction is in fact **free**, **instant** and **error-free**. This is because the actual construction of software takes place when source code is compiled into executable code. Everything else prior to this point (including writing the source code) should be regarded as design. Source code should be regarded as nothing more than a very detailed design specification.


Even the designers of buildings, would agree that it is much harder to predict all of the tasks to create a design in advance. Many tasks only become obvious as the design emerges. If nearly all software development activities are related to design, then it is also very difficult to predict all of the necessary tasks in advance.

The 3rd Wave - Agile Methodologies

Agile methodologies take the uniqueness of software development as a starting point and attempt to define development approaches that take account of the uniqueness.

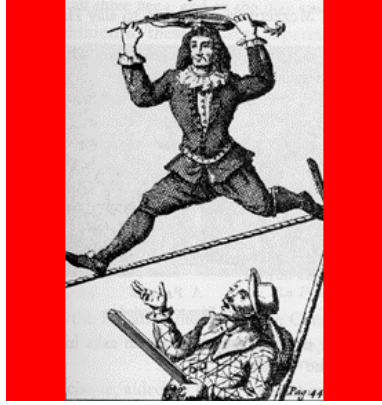
In order to clarify the principles of agile development, a group of developers got together and drafted the “Agile Manifesto”. The manifesto provocatively states that developers should value:

- individuals over processes;
- working software over documentation;
- customer collaboration over contract negotiation; and
- responding to change over following a plan.

 Lonsdale Systems
www.iinet.net.au/~lonsdale/

Agile Methodologies

- “Agile”
 - Characterised by quickness, lightness, and ease of movement; nimble
 - Mentally quick or alert: an agile mind
- Agile methodologies attempt to be “adaptive” rather than “predictive”



There are a number of competing agile methodologies. Some of the better known are listed below. In this paper we shall concentrate on the features of extreme Programming (XP) as this approach adopts a number of unconventional strategies to software development.

- eXtreme Programming (XP)
- Crystal
- Scrum
- Feature Driven Development (FDD)
- Dynamic System Development Method (DSDM)

eXtreme Programming (XP)

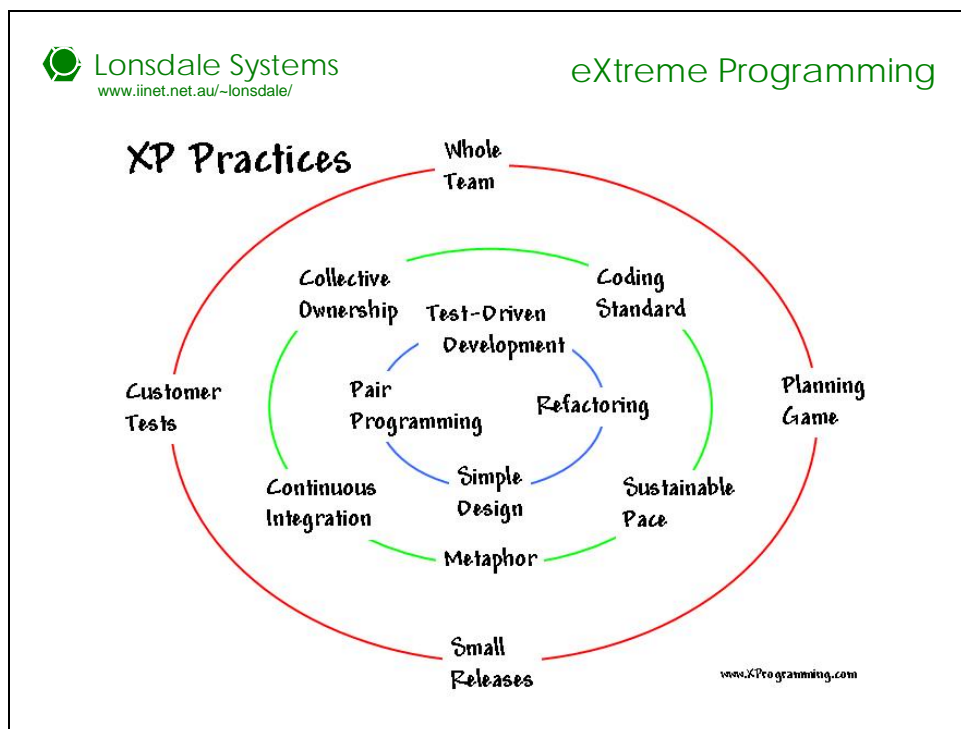


Diagram courtesy of www.Xprogramming.com
(This site also has many more details about XP)

Pair Programming

Peer reviews are a proven technique for improving quality. Pair programming is simply a logical extension of peer reviews. In fact although this is one of the most controversial aspects of XP, there is a body of empirical evidence to support the practice.

Testing

Testing is fundamental to software quality. Designing test cases before coding and employing automated test tools is simply a strengthening of an existing practice.

Refactoring

Even simple designs suffer the effects of entropy over time. Continuous design improvement (refactoring) is a way of preventing this from happening.

Metaphor

All project teams have their own vocabulary to describe systems. A system metaphor is simply a way of formalising this practice.

Sustainable Pace

It is obvious that tired and exhausted developers do not give their best. Maintaining a sustainable pace is a sensible way to ensure that people give their best to a project.

Whole team

Communication is an important aspect of all projects. Locating all developers together and allocating a full-time customer representative improves communication.

Planning Game

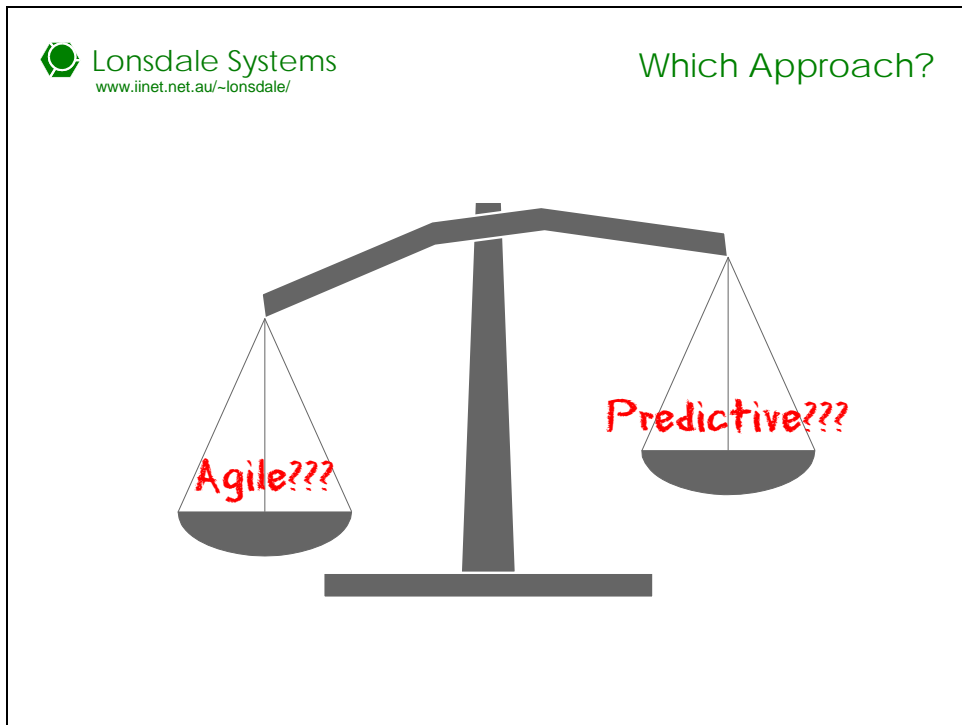
The tradeoffs encountered during a project are neatly summarised by the "Project Equation" which states:

$$\text{product scope} + \text{product quality} = \text{project time} + \text{project cost}$$

Both sides of the equation must balance. Any increase in product scope must be accompanied by an increase in project time and/or cost. Decreasing the cost of a project or the time available, while holding the product scope fixed will inevitably lead to reduced product quality.

In the planning game, developers estimate project time and cost while users prioritise product scope and quality. The project Time and cost then remain fixed for a single iteration based on the planned product scope.

Which Approach is right For You?



Is the agile approach right for you? The answer to that question is “it depends...” Remember that the most widely used approach to software development is still “Code and Fix”. In many cases, any methodology, however agile or lightweight would lead to some improvement in product quality and staff productivity.

	Agile	Predictive
Developers	Superior skills	Adequate skills
Customers	Committed	Available
Requirements	Emerging, changing	Known early, stable
Architecture	Current requirements only	Current and future requirements
Refactoring	Inexpensive	Expensive
Size	Smaller teams	Larger teams
Objective	Rapid value	High certainty

If you currently follow one of the more traditional methodologies be careful about ditching too much of it, too fast. It is important to avoid misconceptions such as - “Responding to change over following a plan”
“Great! Now I have a reason to avoid planning and just code whatever comes up next...”

The table above provides a comparison of agile and predictive methodologies with some attributes that may help you make a choice.

At the end of the day “agile” may disappear with a “puff” into its own cloud of hype. This has happened in the past with 4GLs, CASE tools, RAD... Remember, there are no “silver bullets” that make software development simple!