**MBA (Technology Management) Programme**

T842 Final Report

The Software Project Manager’s Conflict

– to allow, or not allow, change

By Clarke Ching, T6615389

Due: April 02, 2004

**Executive Summary**

This research investigates if software development projects can be delivered on time and to budget in environments where requirements change frequently. Software development projects have a poor delivery record with most delivering late and over budget, many being cancelled, and only a few delivering software that meets the customer’s full requirements (Standish,1994).

A project's schedule and budget are determined and committed to in the early stages of the project when little is known about the product requirements. As the customer learns about the product they need to change the product. But change requires rework and this creates a conflict for the project manager: should they allow changes, to exploit their learning, or should they reject changes to protect the promised schedule and budget. The traditional *waterfall* software development approach tries to resolve this conflict by perfecting the requirements upfront and therefore preventing change. In contrast, *iterative and incremental* approaches try to resolve the conflict by frequently delivering small increments of top priority functionality and allowing the customer to reprioritise between iterations.

This research is a case study of one large software project within Standard Life Assurance Company. Data was collected from twenty-three staff interviews, three focus groups and two mid-project reviews and was analysed using Goldratt's (1994) "Thinking Processes" in order to understand the cause and effect within the project. The resulting cause-and-effect diagram (p. 24) clearly identifies the policies of the *waterfall* approach as the root-cause of the projects problems.

The main recommendation is to adopt an *incremental* approach. The benefits of this recommendation are outlined and then contrasted with the effects of the waterfall approach. Obstacles to successfully adopting this approach are identified and recommendations made to overcome them.**Contents**

1 Aims and objectives 5

2 Introduction and background 6

2.1 Introduction 6

2.2 The conceptual sub-problem 6

2.3 The real world sub-problem 8

3 The nature of the problem 9

3.1 The Waterfall approach 9

3.2 The Agile approaches 11

3.3 Two extremes 13

4 Research design and methodology 14

4.1 Research questions and initial hypotheses 14

4.2 Research design 14

4.3 Research framework 14

4.4 Research techniques 15

4.5 Data collection and analysis 16

5 Analysis and findings 17

5.1 How the data was analysed. 17

5.2 Completeness, reliability, and extensiveness of data 17

5.3 Outcomes of analysis 18

5.3.1 Theme 1: Preventing change by getting the specifications right up-front. 18

5.3.2 Theme 2: Change happens anyway 20

5.3.3 Theme 3: Management of frequent Change 20

5.3.4 Theme 4: Uncertain finish date 21

5.3.5 Theme 5 – “Big Bang” delivery reduces Return on Investment. 22

6 Conclusions, implications and recommendations 25

6.1 Conclusion and implications 25

6.2 Recommendations 26

6.2.1 Recommendation 1: Incremental delivery for the remainder of the current project. 26

6.2.2 Recommendation 2: Incremental delivery for future projects. 27

6.2.3 Recommendation 3: Manage the decision making bottleneck 28

References 29

**Acknowledgements**

This project completes a three-and-a-half year part-time, MBA in Technology Management with the Open University. During this time I have spent many weekends and evenings studying, so I thank my wife and family for their patience during my many absences.

My tutor, Dr Douglas Morrison, provided helpful and honest feedback as I struggled with my first research project. His support and suggestions have been indispensable. As has the assistance and participation of my sponsor Catriona Ronald and colleague Jackie Fairbairn.

The internet provides unprecedented access to industry experts. Gerald Weinberg, James Bullock and others from Weinberg’s Shape forum have been very generous with their time and feedback. Agile software development experts Mary Poppendieck and Ken Schwarber have patiently led me to understand their new paradigm.

Finally, my analysis was greatly improved with the help from my Theory of Constraints mentors Jim Bowles and Frank Patrick.

# Aims and objectives

This project’s overall aim is to investigate how software development projects can be delivered on time and to budget in environments where requirements change frequently. It is intended as a minor, but not insignificant, contribution to Technology Management since it addresses an area where Project Management in general, and software project management in particular, perform poorly. It should clarify the role of requirements management and how it affects the flow of work through software development projects.

My employer, Standard Life, will receive MBA level “informed consultancy” and recommendations in an area that is of particular concern. As a financial company most major projects within Standard life have a high software development component and yet these projects frequently suffer large budget and schedule overruns. This project focuses on the “analysis” task and complements a recently completed T842 project that focused on the “testing” task. My recommendations have already been partly implemented with success, and my employer recognises that the improvements are a direct result of the research.

Personally, this project provides the framework for planning, designing, and executing a structured research investigation and, finally, writing of this report. As well as consolidating previous MBA (Technology Management) courses, this research will provide evidence of my skills and abilities to my current and prospective employers. Moreover, it is an opportunity to learn about an area of Technology Management that fascinates me.

# Introduction and background

## Introduction

The commercial software industry is about half a century old. In 1975, Fred Brooks, in his classic text “The Mythical Man Month”, compared large software systems development to the dinosaurs, woolly mammoths, and sabre tooth tigers fighting the grip of the tar pit. He described an industry with excessive schedule pressure, long overtime, constant change and frequent overruns (McConnell, 2003).

In the intervening quarter-century, as software is integrated into more and more products and process, little has changed from the sad picture painted by Brooks.

The often quoted Standish (1994) “Chaos Report”, summarising a survey of over 8000 software projects, reports that:

* only 16.2% of software projects were completed on-time and on-budget,
* 31.1% of projects were cancelled before they ever get completed,
* 52.7% of projects cost 189% of their original estimates, and,
* more than a quarter were completed with only 25% to 49% of originally specified features and functions.

However, software projects aren’t alone as these recent examples show:

* The new parliament building was budgeted in 1997 to cost between £10M and £40M but is now predicted to cost about £420M when finished (Fraser, 2003).
* Edinburgh’s 2003 Hogmanay party was cancelled because of rain – with no contingency plans in place (Irvine, 2004)
* London’s millennium bridge was closed within days of it’s official opening when it “began to sway and twist in regular oscillations” (BBC News, 2004).

## The conceptual sub-problem

Projects face three competing requirements – budget, schedule, and product scope[[1]](#footnote-1) – commonly referred to as the “Iron Triangle” because changing one means at least one of the other two must also change (McConnell, 1997). For instance, to deliver a project earlier (schedule) then either the product’s scope must be cut – that is, build less - or the budget must increase – perhaps by hiring more staff or working overtime. The Standish (1994) results, above, show that software projects perform poorly with respect to all 3 factors.

The problem arises because the schedule and budget are usually estimated and committed at the start of the project, when the product is poorly understood. But, as the project proceeds, the customers learn more about the product and ask for changes to the product scope. These “change requests” put the project’s manager in a difficult conflict, as shown in Figure 1, below . In order to have a successful project the project manager must both, deliver on time and on budget and deliver the functionality required by the customer. In order to deliver the required functionality they must allow the customer to change their requirements - otherwise the project doesn’t benefit from the customers learning and the product is inferior. But, if they do allow the change then the promised schedule and budget are jeopardised.



Figure 1 The project manager’s conflict

I call this the *project manager’s conflict*: they can’t both allow change and not allow change. They can’t have their cake and eat it too.

There are five common responses to this conflict. The first is to try to prevent change by getting things right upfront. The second is to establish “change control” procedures that only allow the most important changes to be introduced. The third is to include contingency within their project plans and budgets, to absorb change when it happens. Fourth, product triage, or de-scoping, is used to pare back the product’s scope when it threatens to exhaust the contingency. Finally, the schedule is allowed to slip.

This research investigates the project management of software projects and considers if this conflict can be better managed in environments where the product scope is unclear, or changes, frequently.

|  |  |
| --- | --- |
| **Area** | Software Project Management |
| **Field** | Requirements management |
| **Aspect** | Managing changing requirements. |

Table 1: Area - Field – Aspect analysis

Many studies cite requirements management issues as a primary cause of software project failure:

* The Standish (1995) “Chaos” report, above, identified three requirements management failures - *lack of user input, incomplete requirements, and changing requirements* - as the top three causes of project failure.
* Demarco and Lister (2003) identify *requirements inflation* (commonly known as “scope creep”) and *specification breakdown* as two of the top five software project risks.
* Johnson (2003) showed that, on average, 45% of built requirements are never used, and 19% are only used rarely, which contrasts with the Standish (1994) conclusion that a quarter of the projects finish with only 25% to 49% of the features and functions originally specified.

The frustration of frequent change is not limited to software development. The contractor on the Scottish parliament project, mentioned above, blamed scope management for the project’s tenfold budget increase:

“Between October 2002 and May 2003 the construction team faced 1825 architect's instructions, which led to 4600 variation instructions to trade contractors. Although there was an attempt to simplify the project, with a freeze on design changes imposed in April 2003, 545 new instructions were issued in May.” (Fraser, 2003).

## The real world sub-problem

This research investigates a large software development project within Standard Life Assurance Company (SLAC). My research sponsor - the software project's information technology programme manager - is concerned about the number of late, over-budget, under-scope projects within the company. An informal survey of my colleagues suggests that none our divisions major projects during the last 5 years have been on time, on budget and fully functional.

This case study's project's schedule, budget and product scope are all under pressure. The schedule is critical because each month's delay in going to market is estimated to cost £1,500,000 lost profit. This figure is dependant on being first-to-market, so prompt delivery is vital. Product scope is critical because, in order to exploit its first-to-market advantage, the product must be highly functional - in fact, the project sponsor says that launching an inadequate product would be devastating for the company as a whole. Finally, the budget is unusually constrained because of strategic financial considerations.

She believes the analysis task - a major part of requirements management - takes too long and that it is often incomplete or ambiguous.

# The nature of the problem

This problem – how to manage changing requirements – is at the core of one of the major debates in software project management circles: the debate between traditional and agile methodologies (Glass, 2003). The main distinction between the two methodologies is their approach to change. The traditional methodologies, epitomized by the waterfall approach, endeavour to prevent change by getting everything correct and signed off upfront. The agile methodologies, such as Extreme-Programming and Scrum claim it is impossible to prevent change. So they deliver software in short, frequent iterations, adapting to changes in requirements and priorities between iterations.

## The Waterfall approach

Most software projects use some variation of the “Waterfall” software development lifecycle. In their simplest form waterfall projects, as shown in Figure 2, consist of sequential phases of analysis, design, build, and test, with all functionality being delivered at the end. Each phase is done by a different role or speciality – that is, analysts do analysis, designers do design, programmers write programs and testers test the programs.



Figure 2 The Waterfall lifecycle

The waterfall lifecycle tries to *prevent* requirements change by spending more time upfront to ensure the requirements specifications are correct and signed-off by the customers.

The rationale for using the waterfall approach is based on Boehm’s (1981) landmark research that the cost of making requirements changes is dramatically more expensive to make as the project progresses. This is known as the “cost of change curve” (see Figure 3). During the 1970’s, Boehm studied several large software companies and discovered that, for example, it was about 100 times more expensive to change a requirement once the software has been released to its users than if the change had been discovered during the requirements phase. Recent studies have confirmed the 100:1 ratio (Boehm and Turner, 2003).



Figure 3 Traditional Cost of Change curve, (Ambler, 2003)

As software development has matured many techniques have been developed to improve requirement’s definition. Weinberg and Gause (1989) focus on requirements ambiguity and suggest many techniques to reduce it. McConnell (1996) suggests 27 best practices, several of which – evolutionary prototyping, inspections, joint application development (JAD), principled negotiation, requirements scrubbing, theory-w management, user interface prototyping, throwaway prototyping – are intended to involve the customer more and bring feedback into the process much earlier than traditional techniques. Boehm (1988) blames conflict and compromise for many requirements change and has elegantly incorporated win-win negotiation techniques into his “Spiral” development approach and software. Gilb (1988) faults ambiguous success criteria and suggests a precise and unambiguous language for defining requirements. Many of these techniques are used in Standard Life.

Despite these techniques, Boehm and Papaccio(1988) show that a typical software project will experience a 25% change in requirements. Reinerston (1997), discussing product development in general, says that requirements change happens because we learn more about the product as the design progresses (see Figure 4). To deny change is to ignore project learning.



Figure 4 Information arrival in product development projects, adapted from Reinerston(1997)

When changes happen, waterfall projects invoke “Change Control” procedures. First, the customer identifies that a requirement needs changing and they formally request the change. The impact of the change is assessed and a change committee, or project manager, approves or rejects the change (McConnell, 1997). Some project managers will include “contingency” in their plan in order to absorb changes and prevent the finish date from slipping (Goldratt, 1997). Others require customers to swap requirements from scope to compensate for the extra work for the change that has been approved. There is considerable administrative overhead in managing change requests. Requirements triage, or de-scoping, is used to slash the product scope in order to meet a schedule (Davis, 2003).

In the waterfall methodology, quality is judged according to Crosby’s (1978) notion that quality is “conformance to requirements” and that quality is measured by the number of defects (compared to the requirements). Weinberg (1982), Miller (2003) and Reinerston(1997), amongst others argue that this approach to manufacturing is valid in manufacturing but is not valid in product development or design activities such as software development. Gillies (1997) says that Juran’s idea that quality is “fitness for purpose” is “particularly influential when we come to consider the use of quality management ideas in software development” (p. 104).

## The Agile approaches

Since the late nineteen-nineties an alternative to the Waterfall approach has emerged. These approaches, collectively known as “agile” (such as Scrum, Extreme Programming and Feature Driven Design) - argue that preventing change is impossible so, instead, they work in ways that they claim “embrace change”.

From a work flow point of view they deliver software in small, frequent –e.g. weekly – iterations. Each iteration includes the customers highest priority (taking into account both risk and cash flow) requirements. At the end of each iteration the software is fully built and tested and could – but doesn’t have to be – released to the customer. The customer may reprioritise their requirements to benefit from any learning. See Figure 5.



Figure 5 Incremental and Iterative development, Larmen (2003)

Some agile practices are technical. *Refactoring* allows programmers to make significant design changes to their programs safely (Fowler, 1999). Test Driven Development “mistake proofs” such changes by efficiently building comprehensive test suites (Beck, 2002). Some agile approaches – such as Scrum (Schwaber and Beedle, 2002) and Critical Chain (Goldratt, 1997) - are change friendly alternatives to traditional Critical Path project management. Larman (2003) recommends using the critical path “Rolling Wave” planning technique to report agile projects in a traditional manner.

Gelbwaks (2003) describes how Critical Chain project management was combined with agile techniques to rapidly commercialise the Segway Human Transporter – not only a new product, but a new technology category: “barely 18 months passed from the time the design was completed to when the first units began rolling off the factory line” (p. 28). In a personal email he described the process:

“Rather than having the project network be comprised of specific tasks as is the usual case, our network consisted of stories - those larger entities that are the hallmark of agile development efforts. The network was derived in the typical way (backwards decomposition), but only to the level of detail necessary to define the stories. Then, as each story was undertaken (using the agile approach), the rest of the picture would be a little clearer and subsequent stories would begin to jell and progress could be made.” (Gelbwaks, 2004)

Agilest Kent Beck argues that by using these practices the cost of change curve looks more like Figure 6, below. If this is true it invalidates the main reason for using the waterfall approach – that we have to prevent change because change is expensive.



Figure 6 Kent Beck’s cost of change curve (Ambler, 2003).

However, Boehm and Turner (2003) point out Beck provides only anecdotal evidence for the lower cost of change and that three recent large-scale studies corroborate Boehm’s original curve in Figure 3.

However, my assessment is that since neither Boehm’s original studies nor the recent studies distinguished between traditional and agile projects and, since agile projects are only a small proportion of all projects, then it seems likely that the recent studies were based almost exclusively on waterfall style projects. The two approaches are so fundamentally different I don’t believe that studies of waterfall projects can tell us anything about the cost-of-change for agile projects. I suggested this interpretation to an agile group and it was well received and, judging from feedback, a plausible and novel interpretation.

However, Boehm and Turner (2003) point out two little-known features of the original 1981 study:

* + The cost-of-change curve for *small projects* is relatively shallow – a ratio of 5:1 compared to 100:1. This distinction couple help explain why agile projects, which are generally smaller, have been successful. While some agile approaches acknowledge that they don’t scale easily (extreme programming, for instance) others (such as Scrum and FDD) claim they can scale by working in small teams that are specifically designed to isolate changes from one another - in technical terms, the teams must be designed so their work is highly cohesive but loosely coupled.
	+ When the changes are broken down the Pareto principle applies: 20% of changes caused 80% of the work. This point is little known amongst either of the agile or traditional communities and means that 80% of the changes have much shallower curves and the remaining 20% have very steep cost of change curves. Boehm and Papaccio (1988) say “the major implication of this distribution is that software verification and validation activities should focus on identifying and eliminating the specific high-risk problems to be encountered by a software project, rather than spreading their available early problem elimination effort uniformly across trivial and severe problems.”

These two points suggest that, provided proactive risk mitigation is done and the developers work in small, well designed teams, then the cost of change curve for agile projects could be much shallower than in traditional waterfall projects.

## Two extremes

This section has presented two extremes on a continuum of software development approaches where the distinguishing characteristic is the size of each “phase”: the agile approaches have many small phases; the waterfall approaches have just one “big bang” phase. There are many variations in between these two extremes. Table 2 summarises the two extremes.

Table 2 Comparing Waterfall and Agile Software Development.

|  |  |  |
| --- | --- | --- |
|  | Waterfall | Agile |
| Attitude | * Prevent Change
* Heavyweight processes
* Command and Control
 | * Embrace Change
* Lightweight processes
* Self Management within iterations.
 |
| Technical Tactics | * Prototyping
* Joint analysis sessions
* Inspection
 | * Customer on Site
* Test driven development, Refactoring, Pair programming
 |
| Management Tactics | * Sign-off requirements before starting design.
* Serial phases before big bang release.
* Change control manages and discourages change.
 | * Product owner prioritises features.
* Small iterations of most valuable features.
* Frequent inspection, evaluation and adaptation
* “Fail early” by tackling risky features first.
* Early release(s) where possible.
 |
| Paradigm | * Software is like manufacturing
* Software quality “conformance to specifications”.
 | * Software is like Designing
* Quality is “fit for purpose”.
 |

# Research design and methodology

## Research questions and initial hypotheses

My project sponsor initially suggested researching how to speed up the analysis phase of a project (see Figure 2). She thought it took too long and prolonged the projects. Following my literature review I broaden the focus in two respects. First, the goal should be to speed up projects, not just the analysis phase- we could easily speed up analysis by doing less of it, but the project would take longer. Second, I should focus on requirements management – the prevention and management of requirements change - across the project life, rather than just the analysis phase.

The research question I settled on was:

**Can requirements management be done differently so that projects can be completed more quickly?**

## Research design

Despite an extensive literature review, I felt that the problem was ill-defined and messy. I decided that for the research to be useful “informed consultancy” I needed to understand the underlying cause-and-effect within projects. So, although I had developed several hypotheses[[2]](#footnote-2) I decided to adopt an inductive, rather than deductive, approach and let the theory emerge from my research. I felt my original hypotheses were two narrow and would not generate any new insight or knowledge since their solutions were both well documented. I rejected a *review* approach, in favour of an *empirical* approach, because I didn’t understand the real world problem well enough to decide which part of the theory to review.

I chose to conduct an *exemplar case study* of my current project, rather than a survey or experiment. I rejected an experimental approach as inappropriate and impractical. A case study, according to Yin(1981a, 1981b) “investigates a real life phenomenon within its real-life context when the boundaries between phenomenon and context are not clearly evident and which multiple sources of evidence are used”. I chose to do an *exemplar*, rather than *cross-sectional* or *longitudinal*, for practical reasons – I had easy access to only one project.

One concern about doing a case study was the risk of the project failing mid way through, or deviating significantly from the plan. I mitigated this risk by starting my data collection as soon as possible.

## Research framework

Since my approach was focusing on the underlying cause-and-effect I needed a technique to capture and model the data. I decided to use Goldratt’s (1994) Theory of Constraint’s “Thinking Process” approach as my framework for collecting, scrutinising and understanding the underlying cause and effect within the case study. I chose Goldratt’s approach, rather than other alternatives such as fish-bone diagrams, multiple cause diagrams, or system dynamics diagrams because I have prior experience and success using Goldratt’s process.

The process has the following steps:

1. Identify the system under consideration – in this case, the project.
2. Collect a list of frustrations, issues or problems of the system from the various stakeholders
3. Map these symptoms into a hierarchy of causes and effects, while constantly scrutinising the logic, filling any knowledge gaps (e.g. by clarification with interviewees) until the hierarchy coherently explains the underlying cause-and-effect within the system.
4. Identify common root cause(s) of the undesirable features symptoms.

The value of using this framework is that it identifies the few root causes from amongst the many symptoms, and therefore identifies where to focus improvements.

The resulting cause and effect tree is shown in Figure 8 on page 24.

## Research techniques

I chose to use face-to-face research, such as interviews and focus groups, as well as opportunistic observation, to gather my data. This approach was necessary because I wanted to gain a broad understanding of the system under scrutiny, which necessitated open-ended, semi-structured questioning techniques. Since the answers could be ambiguous I needed face to face questioning where feedback was immediate. This choice of technique also played to my particular working style and strengths.

I did attempt to write a questionnaire to find out the kind of information needed but when I tested it with colleagues they said the questions were too open-ended to answer, quickly, in writing.

A concern with my approach is that the interviews and analysis were subject to the bias of my interpretation. I was able to mitigate this risk by triangulating my results with the findings of two independently conducted project reviews, as well as reviewing my analysis in two focus groups. I also used Goldratt’s categories of legitimate reservation, to scrutinise and verify the internal validity of the cause-and-effect tree’s “if … then” relationships (Scheinkopf, 1999).

Twenty-three semi-structured interviews - I interviews 23 colleagues, each taking 15-30 minutes. I took notes during and after each interview. I interviewed a cross section of staff – customers, analysts, managers, developers and testers - at different of seniority from the project.

My intention was to understand the underlying cause-and-effect at play within the case. Therefore my interviews started by asking the interviewees to describe their high level frustrations, then drilled down into why these were frustrating (i.e. the consequences) and asked the interviewees to speculate on their causes. I used the 5-Whys technique, Causes and Consequences mapping and 5W&H techniques[[3]](#footnote-3).

Two Project reviews -

I had access to the results of two mid-project reviews which, although from different teams on the case study project, reached similar conclusions to each other and to my own analysis which was useful for triangulation purposes.

The first was a mid-project review by the information systems team, which was facilitated by our project manger. I was a participant. We produced an affinity diagram - staff were asked to (a) list the project’s good and bad points on post-it notes; (b) stick them on the walls, then (c), group them - in silence - until the group was in consensus – followed by a group discussion. The results were thoroughly documented.

The second was a mid-project review done by the business analysis team. I was given full access to the written findings.

Three focus-groups - I conducted three focus groups using senior technical staff to review my partial findings and provide direction. I intentionally invited staff that I thought would offer honest and forthright feedback.

The first was opportunistic, happening mid-way through the research, when I was asked to give a presentation to our divisions “Accelerated Development Team” about *agile-software-development* techniques (which I had come to understand through my literature review). I conducted a two-hour focus group with 4 senior staff – an analyst, two project mangers, and a designer – where we discussed what we could learn from the agile approaches in relation to the problems identified so far in the research and what obstacles prevented their use.

The second focus group happened near the end of the research analysis, when I reviewed my cause-and-effect tree (see Figure 8 on page 24) with senior colleagues, leading them through a thorough scrutiny of the tree’s logic. Although this was, strictly speaking, part of the research analysis it also provided new data.

The third focus group reviewed my recommendations with senior colleagues.

Company records – In order to investigate the extent that the analysis task causes defects in testing, and therefore rework, I sampled and classified the test-defects database from a recent, large, project. This was time consuming and fruitless.

## Data collection and analysis

Although the data collection and analysis are distinct steps and discussed in separate sections of this report, the process was, in reality, very iterative. I started building a very rough cause-and-effect model following my first interview and continued revising and adding to it throughout the process. The one-page view helped me to collate my findings and identify gaps and inconsistencies in my data, which helped guide future interviews or led me to revisit previous interviews.

# Analysis and findings

## How the data was analysed.

When the bulk of the data had been collected I reviewed the notes from my interviews, the first focus group, the 2 project-reviews and the rough-draft cause-and-effect tree, writing each piece of data onto a yellow post-itTM note. I arranged the notes into an affinity diagram categorising them into high-level themes. I then used mind-mapping software to categorise the themes and sub themes in a hierarchy, which I laboriously turned into a causes-and-effect Tree, scrutinising and refining as I went.

When I had reached my limits with the cause-and-effect tree, I presented it to several groups for further scrutiny:

* I conducted a focus group with senior colleagues.
* I presented my findings to a local agile special interest group.
* I joined IT-guru Jerry Weinberg’s internet based "Shape forum" where I was asked to write an article based on my ongoing research. Several iterations of the article were reviewed and discussed by the forum. The article will be published later this year.
* I have found writing this report has clarified my ideas as I have struggled to turn them into words.
* I continued to read relevant literature - in particular Larman (2003), Poppendieck (2003) and all of Barry Boehm’s works provide invaluable insights.

This process took place over several weeks, during which I continued to collect data as needed.

## Completeness, reliability, and extensiveness of data

I interviewed twenty-three staff working on the project. I interviewed at least one, but often more than one, person from each role in the project (see Table 3). If I discovered any inconsistencies or gaps during the analysis I returned to the interviewee to clarify.

Table 3 – Interviewee Roles and numbers

|  |  |  |  |
| --- | --- | --- | --- |
| Business / IT | Seniority (1 = senior– 4 junior) | Role | Number interviewed |
| Business | 1 | Project Sponsor | 1 |
| Business | 2 | Business Analyst (Senior) | 1 |
| Business | 2 | Marketing manager | 1 |
| Business | 2 | Actuary  | 1 |
| Business | 3 | Business Analyst | 3 |
| Business | 4 | Tester | 1 |
| IT | 1 | IT programme manager | 1 |
| IT | 2 | IT project manger | 2 |
| IT | 2 | Test Manager | 1 |
| IT | 2 | Designer | 2 |
| IT | 3 | Developer / Programmer | 4 |
| IT | 3 | Systems Analyst | 4 |
| IT | 3 | Database administrator | 1 |
| Total |  |  | 23 |

Trochim (1991: 1) points out that a research piece’s validity can only be established or evaluated after the research has been undertaken. I designed my research so that the data collection and analysis was done in many iterations and had as much scrutiny and feedback as possible during the execution. The cause-and-effect tree changed and grew with each iteration, based on the constant scrutiny and feedback.

Now that the research is complete, I feel the design was valid because:

1. I was able to interview over 50% of the people involved in the project and, although my sampling was often opportunistic, I achieved fair representation from all stakeholder groups.
2. The second focus group thoroughly scrutinised my data and logic against their own experiences and suggested several improvements.
3. The project reviews reached similar findings about the symptoms (they didn’t consider the root causes) which was useful triangulation.
4. I published my partial findings to several internet discussion groups and received positive feedback and confirmation from them, providing external validation.
5. My findings are consistent with state-of-the-art external literature, particularly Larmen (2003) and Boehm and Turner (2003), providing external validation.

One concern with data validity is that I only interviewed permanent staff, even though about half of the developers are short-term contractors. They were not included in the formal project reviews either. I didn’t deliberately exclude them, but rather I found them evasive about answering my questions. One contractor commented that they “get paid the same each hour, not matter how the project goes”; others implied that providing feedback could jeopardise their contracts; and others said it wasn’t in their interest to make projects run faster because then their contracts would be shorter, and therefore their income would be lower.

## Outcomes of analysis

The data collection and analysis covers part of the development project – up to the early stages of testing.

The outcomes of the analysis fall into five major themes:

* Theme 1: Preventing Change by getting the specifications right up-front
* Theme 2: Change happens any way.
* Theme 3: Consequences of change.
* Theme 4: Uncertain finish date.
* Theme 5: “Big Bang” delivery delays cash flow.

The full cause-and-effect tree is shown in Figure 8 on page 24. It should be read from the bottom up, following the direction of the arrows, with the bottom entity causing the top entity.

### Theme 1: Preventing change by getting the specifications right up-front.

The first phase in a waterfall project is to analyse the customer’s requirements in order to produce specifications that the designers can use to design the software. The project’s sponsor was concerned that the analysis phase seemed to take a long time and, despite this, many subsequent changes seemed to be caused by gaps in the analysis.

However, the systems analysts I interviewed said the opposite: that they needed longer to do complete the analysis properly and they were sometimes rushed to complete it too quickly and so made mistakes. These mistakes caused change and rework later.

They suggested several reasons why the technical aspects of analysis appeared to take too long:

* First, using the waterfall approach, *everything* within scope must be analysed and *documented* before beginning the design. That is: the amount to be documented is large and the process is time-consuming.
Customers are asked to “sign off” documents forming a “contract” between the business and software teams (Gillies, 1997). This is intended to prevent change later. Analysts noted it is extremely difficult to write documents that are unambiguous and easy to understand for non-trivial systems.
Analysts, developers and testers said they preferred less documentation and more discussion. Developers said it was faster to develop software if they understood the product rather than just using a specification and suggested the product would be better if they helped with the analysis. The single tester I interviewed described how it was frustrating to base his test scripts on frequently changing written specifications and suggested both analysis and testing would benefit if analysts and testers worked more closely.
* Second, since the customers only had one chance to ask for their requirements they tended to include everything they could think of, no matter how minor. Johnson’s(2002) finding that, on average, 45% of features are never used and 19% of features are only used rarely, back up this point. Poppendieck(2003) describes this unneeded functionality as the biggest waste in software development since it makes not just the analysis phase but every following phase take longer.
* Third, they said good analysis looks simple, but the simplicity belies the effort required to achieve it. For instance, one analyst described how it took over six months to analyse and decide the products commission structure but less than six hours and three A4 pages to specify what to build. Most of the effort was taken up with understanding the problem in detail, exploring and evaluating many possible solutions, and resolving conflict between stakeholders.
* Fourth, many decision-makers were not available when needed in the early stages – particularly the actuary team as discussed in the next section.

They also described how, several times during the project, developers joined the project before work was available for them. In some cases the project managers diverted the analysts from critical-path tasks to lesser priority tasks in order to prepare work for the developers. Although this diversion created work for the developers it made the analysis phase take longer. Other times the developers were asked to start work using incomplete specifications, the project managers acknowledging that there would, inevitably, be rework required later once the analysis was complete. Developers said that this was one of the major causes of change and rework (see 5.3.2).

Finally, Business analysts, in interviews and their mid-project review, said each of the five systems analysts had different working styles and used different techniques and it was difficult and time consuming trying to adapt to the different styles. Several senior IT staff echoed the comments saying this made the analysis task take longer and increased the potential for mistakes.

### Theme 2: Change happens anyway

The top frustration from ***every interviewee*** and both project reviews was the high rate of change within the project and the administrative overhead and rework that this caused. Boehm and Papaccio(1988) shows that a typical medium to large software project experienced a 25% change in requirements.

Systems analysts said they were frustrated because they were often blamed for not getting the requirements right to start with. They pointed out that although some changes could have been avoided by more careful preparation in the early stages of the project, many of the changes came about because they – the analysts, customers, and developers - learned more about the product, the market and the technical constraints as the project progressed (see Figure 4, above). Many of the requirements they made at the beginning of the project were “best guesses”. They pointed out that because it takes a long time between making the initial “best guess” and receiving the software there is a large window during which the original requirements may change - for instance, it was over one year between completing the first specification and when the customers first tested it. Many changes are discovered during the testing phase with one study attributing 41% of defects to requirements (Sheldon, F. et al. 1992) .

Two causes of change deserve special mention. First, as discussed in section 5.3.1, sometimes developers were available before analysis was available and started work based on incomplete specifications, which then changed as they were completed. The project sponsor said she felt the advantage of starting the work outweighed the cost of rework. Figure 3 suggests that this trade-off would only pay if a small amount of change resulted. I am not able to quantify whether this is the case.

Second, the actuaries team were a significant bottleneck to the analysis effort. This team was only available for a small amount of time during the first year of the project and yet they were one of the project’s key decision makers. The analysts had to make their “best guess” at many requirements, in lieu of the actuaries input. When the actuaries did became full-time on the project they needed to correct many of the “best guesses” (which caused much change and rework) and they were a severe bottleneck to the whole project. At one stage I checked our “issues” database and all “requirements” related issues were being held up by the actuary team. Apparently, this team is well know for being the bottleneck on many large projects within the company.

The bottleneck situation was exacerbated because, in order to keep so many people happy, the actuaries multitasked – working on many tasks at once but switching between them before they were complete – giving the impression of progress on each task but making each task take many times longer to complete (Goldratt, 1997). This meant that not only were staff waiting on the actuary team, but they were waiting much longer than necessary.

### Theme 3: Management of frequent Change

When staff realise that a change is needed they request a “change control” which is then evaluated (effort versus importance) and either approved or rejected. Staff were frustrated when change controls were rejected because it meant that the product was sub-optimal and because many rejected changes were considered more important than many requirements still in scope. The approved changes had often been watered down to get them approved. Business staff were all disappointed with the product that would finally be delivered.

Project managers said they recognised the importance of being flexible to changes and described how they included contingency tasks within their plans to absorb the approved changes. Sometimes though they initiated prioritisation exercises where lower priority tasks were dropped, or moved to later phases, in order to meet the schedule. In these ways the scope is flexed to try to meet an uncertain finish date, the subject of our fourth theme.

### Theme 4: Uncertain finish date

When this report is published (April 2004) the project will still have over six months to delivery. The project has recently been rescheduled and the management team has committed to a new delivery date. Many staff privately expressed doubts about the viability of the new date.

Part of this uncertainty comes from the natural variation within complex systems –plans are, after all, predictions based on many task estimates which themselves are subject to natural variation (Goldratt, 1997). Although the project managers commit to a fixed finish date that date is still uncertain. Figure 7, adapted from Demarco and Lister(2003), shows the cumulative probability of completing a mock project within for each month in the following year.

**Cumulative Probability of Completing**

**Mock Project by month**

0

20

40

60

80

100

1

2

3

4

5

6

7

8

9

10

11

12

**%**

Assuming a 30% chance

of cancellation

Figure 7

Another cause for the uncertainty is the risk of “late failure” (Larman, 2003) inherent within waterfall style projects. Nearing the later stages of the projects many of the paths on the project plan start to converge and the software looks, according to the plan, like it is nearly finished. This is commonly known as the “90% complete syndrome” (Abdel-Hamid, 1988) since many tasks remain at 90% complete for a large part of the project. However, the percentage complete is a poor indicator of actual progress because it only shows elapsed time based on the original estimate. For instance, it doesn’t reflect the following two “late failure risks”, both inherent in the waterfall approach and both the result of delayed feedback:

* The first was highlighted during the mid-project IT review. Staff were concerned about integrating the software. Integration takes many distinct units of software, that have been built by a single developer, and assembles them[[4]](#footnote-4). It is regarded as one of the highest technical risks in a project (Larman, 2003). Since waterfall projects don’t integrate their components until late in the project the staff were concerned because they had spent many months building their software but had no proof that the system would work as a whole.
* The second concern came from business staff. For some it had been over two years since they started work but they had not yet seen any working software. Despite putting the effort in upfront they were nervous that the software delivered would not be what they expected. Many changes are requested very late in the project when customers realise that the software doesn’t do what they expected (Larman, 2003).

Despite many staff privately discussing their concerns about meeting the published finish date the management team maintain publicly that, although it will be a struggle, the published finish date will be met and all requirements within scope will be delivered. For political reasons, staff only voiced these opinions privately. For the same reasons, I did not seek clarification from the management whether their public position was their true belief. While I cannot attribute the cause for this, one likely reason is management’s belief that *keeping the pressure on* motivates staff and improves productivity.

Another cause may be cultural. Demarco and Lister (2003, p. 42) suggest many corporate cultures have an unspoken rule that “It’s okay to be wrong, but not okay to be uncertain” or “… you can ask for *forgiveness* for being late (afterward), but you can’t ask for *permission* (beforehand).” They say that this attitude can thwart the best-intended efforts at risk management. My opinion is that the public confidence in the schedule prevents the management team from taking overt steps to mitigate the risks. In the Section 6, based on Demarco and Lister’s (2003, p.128) recommendation that the “best bang-per-buck risk mitigation strategy we know is incremental delivery”, I recommend the project adopts an incremental delivery approach as the safest and most efficient way of meeting the schedule. Yet, I suspect that the first obstacle to implementing this recommendation will be the taboo of admitting that there is a fair probability the project will not meet its deadline.

### Theme 5 – “Big Bang” delivery reduces Return on Investment.

Software Development is a value creation exercise (Denne and Cleland-Huang, 2003). This projects is being conducted in order to make money and yet the “big bang” delivery of the waterfall approach delays cash flow until the very end of the project.

The product’s sponsor estimates the new product should deliver about £1,500,000 profit each month when it reaches the market. The project will delivered in two main phases. The first delivers the majority of the functionality on one date and then, a few months later, the second delivers functionality de-scoped from phase one. By following the waterfall methodology the product is delivered in one, what is commonly known as, “big bang” and no income is received from the product until the “big bang”.

Several staff – both technical and business - suggested that the product could have been delivered to market, much earlier, with a small subset of the functionality finally being built. It could have catered for the needs of the majority of customers and situations and started earning income much sooner, reducing the payback period significantly. Early delivery would also increase the company’s chances of being “first to market” which the sponsor considers vital.



Figure 8 – Cause and Effect Tree

# Conclusions, implications and recommendations

## Conclusion and implications

*My prime conclusion* is that the problems described within sections 2 and 3 – schedule overruns, budget overruns, frequent project cancellations, bloated and yet deficient, software - are present in the real world case study and *are directly caused by using the waterfall methodology*. It is clear from Figure 8 on page 24, which graphically encapsulating the above analysis, that the root cause of the “real world” problems is that the project is using the waterfall methodology. Many skilled colleagues from across the software development industry have reviewed my conclusions and all say it is a fair and accurate summary of waterfall projects in general; it is also consistent with the writings of many industry authorities such as Larman(2003), Boehm and Turner(2003), DeMarco and Lister (2003) and Denne and Cleland-Huang (2003).

Figure 9, is a summarised version of Figure 8 and it shows the link between the waterfall approach and the problems described.



Figure 9 Cause and effect Tree – implications of using the waterfall approach

Table 4 summarises the effects of using the waterfall approach against 4 factors: product, learning, schedule and finance.

Table 4 Summarising the effects of the Waterfall approach

|  |  |
| --- | --- |
| **Product** | Since the project’s customers only have one chance to state their requirements they include every requirement they can think of upfront, no matter how minor, resulting in bloated software, bloated plans, and bloated budgets and reduced return-on-investment. |
| **Learning** | Staff working on the project learn as it progresses but much of this learning is “lost” as changes are rejected, or watered down, during the “change control” process.  |
| **Schedule** | The schedule is bloated and uncertain until near the end. It is at high risk of late failure risk because (a) the components are not integrated technically or across work packages until late in the project, and (b) the customer doesn’t see the final product until late in the project and it is often different to what they expected. |
| **Finance** | The opportunity to reap value from early, partial releases is lost because cash flow is delayed until every requirement is delivered in one “big bang”.The first to market advantage is risked because the bloated schedule and “big bang” release delay going to market.  |

## Recommendations

Without a clear understanding of the root cause of the system’s problems and with so many symptoms being related to requirements and analysis tasks it is understandable that most improvement efforts focus on doing the analysis task “better” by, for instance:

* Spending more time, upfront, getting the requirements right.
* Thoroughly reviewing and signing-off specifications.
* Preventing designers or developers from working until the requirements are signed-off.
* Tightening “change control” procedures to discourage change requests.

However, these efforts only attack symptoms. They assume we can fully prevent change upfront which we cannot – too much change is the result of *learning by doing* the project. The unwanted effects of the waterfall methodology - bloated, yet incomplete, software; bloated, yet unreliable, schedules; late failure risk, and late cash flow – would remain.

**My main recommendation is, therefore, to attack the root cause – the waterfall methodology – and to manage projects using an incremental delivery approach, instead.**

In particular, I recommend using the “Scrum” project management framework recommended by Schwarber (2002), coupled with the risk management emphasis of DeMarco and Lister (2003).

### Recommendation 1: Incremental delivery for the remainder of the current project.

As it stands, with several months to go, the current project:

* is trying to deliver a fixed set of requirements.
* is committed to a delivery date.
* has many outstanding high-risk tasks, such as end to end integration.

The incremental approach recognises that (a) some requirements are vital and some are trivial and that (b) the probability of meeting the committed delivery date is less than 100%. The customer prioritises every requirement and the team delivers the most important first. Nearing the delivery date, if not every requirement is built, the customers decide if the software is adequate and can either slip the schedule or go live with the most important software built. High risk tasks, such as end-to-end integration, are done much earlier when there is time to recover. Actual task progress is much clearer – a function is either done or not, rather than long-term 90% complete – and decision making is improved.

During my third focus group I discussed this recommendation with the team’s senior technical staff and all agreed that this is, technically, the way we should proceed with the project. However, I believe the main obstacle to accepting this recommendation is the cultural difficulty of admitting the project may not meet the committed date with the committed functionality, as discussed in section 5.3.4. I will raise these issues when I present the findings of this report to my sponsor.

In short, by adopting the incremental approach the current project will dramatically improve its ability to deliver the most useful software possible on the delivery date.

### Recommendation 2: Incremental delivery for future projects.

My recommendation is that future projects also adopt the incremental approach using the Scrum framework. For new projects, the rewards of managing projects incrementally - from the start - can be very high. Projects can release the most valuable software sooner, significantly shortening the payback period and increasing ROI (Denne and Cleland-Huang, 2003). They can mitigate common project risks much earlier, increasing the odds of success (DeMarco and Lister, 2003). Table 5 summarises the benefits of adopting this incremental approach against the four factors: product, learning, schedule and finance.

Table 5 Summarising the effects of the incremental approach

|  |  |
| --- | --- |
| Product | The product being worked always contains the top priority items. When it is released it will always contain the most important requirements. No work is done on low priority items, therefore minimising the schedule and budget, and maximising return on investment.Since the customer sees the product sooner they can provide feedback sooner. |
| Learning | Learning and feedback takes place early and the customers may reprioritise as they need at anytime. Unlike waterfall projects where changes usually cause rework, the cost of changing is minimised, because:(a) The top priority requirements are analysed in depth before being started, are better understood, and are less likely to change, and, (b) the lower priority requirements can be changed at no cost since they haven’t been started yet. |
| Schedule | The schedule is more certain because, before the release date is reached, the customer can decide whether the product is good enough to release or if they should reschedule.Late failure risk is mitigated earlier:* Since each iteration must be potentially shippable, it forces integration to be done much earlier, when it is easier to fix if problems arise, or, in the worst cases, it is cheaper to cancel the project.
 |
| Finance | The iterations may, if the customer decides, be released early and the project will pay for itself earlier.The possibility of being first to market is improved.Risks are mitigated much earlier, so the project’s expected return is higher. |

However, while the benefits are potentially very high, the incremental approach is risky if not adopted properly. I am aware of at least two prior Standard Life projects that tried to adopt the new approach but failed. Without going into the details of those projects, I believe they failed because they adopted the idea of delivering in increments but chose their increments badly. Therefore, I recommend that if Standard Life does adopt my recommendations they do so, first, using a small project where they can learn with lower risks, before moving to progressively larger projects, and, second, using advice from external specialists.

Two suggestions compliment this recommendation:

* 1. Within the waterfall approach, documentation is used both as a contract and as a communication medium. It often substitutes for face-to-face communication. With much less work-in-process and much closer deadlines it will quickly become apparent how much time is spent preparing written documentation. I suggest Standard Life adopt the process-light, people-heavy documentation techniques from the agile approaches. This will both quicken analysis tasks and improve product knowledge across all roles.
	2. The incremental approach obsoletes many of the traditional project management tools. However, for large projects, with many non-software tasks a tool is still needed to manage the large-scale task dependencies. Larman (2003) recommends the “rolling wave” technique, although I suggest that Standard Life investigate Goldratt’s (1997) Critical Chain project management technique combined with the incremental approach for the software tasks, as described by Gelbwaks (2003 and 2004).

### Recommendation 3: Manage the decision making bottleneck

My final recommendation is vital for future projects, no matter if the previous recommendations are accepted.

The actuarial team was a bottleneck for this, and many prior, projects. Their late arrival delayed this project considerably because of slowed decision making and increased rework. When the team did arrive they were swamped and their output severely reduced by unproductive multi-tasking. Senior management should decided how to manage this bottleneck, either increasing their availability or deciding how to use them most efficiently when they are available.

# References

Abdel-Hamid, T (1988). Understanding the "90% Syndrome" in Software Project

Management: A Simulation-Based Case Study”. Journal of Systems and Software, 8 pp. 319-330.

Ambler, S (2003), Examining the Cost of Change , Available from: < www.agilemodeling.com/essays/costOfChange.htm > [accessed 19 March, 2004]

Beck, K (2002) Test Driven Development: By Example, Addison Wesley, Boston

BBC News (2004) Millennium Bridge – In depth BBC News, Available from: < http://news.bbc.co.uk/hi/english/static/in\_depth/uk/2000/millennium\_bridge/happened.stm > [accessed 19 March, 2004]

Boehm, B (1981) Software Engineering Economics, Prentice Hall, New Jersey

Boehm, B (1988) A Spiral Model of Software Development and Enhancement, IEEE Computer, May, pp. 61-72.

Boehm, B and Papaccio, P (1988) Understanding and Controlling Software Costs, IEEE Transactions on software engineering, volume 14, number 10, pp.1462-1477

Boehm, B and Turner, R (2003) Balancing Agility and Discipline: A Guide for the Perplexed, Addison-Wesley, Boston

Crosby, P (1978) Quality Is Free: The Art of Making Quality Certain, McGraw-Hill, London

Davis, A (2003) Requirements Triage, IEEE Computer, March, pp. 42-49

Denne, M and Cleland-Huang, J (2003) Software by Numbers: Low-Risk, High-Return Development, Prentice Hall, New Jersey

DeMarco, T and Lister, T (2003) Waltzing with Bears – Managing risk on Software Projects, Dorset House, New York

Fowler, M (1999) Refactoring: Improving the Design of Existing Code, Addison Wesley, Boston

Fraser, D (2003) Holyrood contractor: 19 changes a day demanded. Sunday Herald, 22 June, Available from: < <http://www.sundayherald.com/print34822> > [accessed 19 March, 2004]

Gelbwaks, M (2003) Segway and an Agile Critical Chain, Cutter IT Journal, Vol. 16, No. 3, Available from: < [http://www.cutter.com](http://www.cutter.com/freestuff/itj0112c.html) > [accessed 19 March, 2004]

Gelbwaks, M (2004) Personal communication

Gillies, A (1997) Software Quality – Theory and Management, Thomson Computer Press, London

Glass, R (2003) Agile versus traditional: make love, not war!, Cutter IT Journal, Vol. 14, No. 12, Available from: < <http://www.cutter.com/freestuff/itj0112c.html> > [accessed 19 March, 2004]

Glib, T (1988) Principles of Software Engineering, The Bath press, Avon

Goldratt, E (1994) It’s Not Luck, North River Press Publishing Corporation, Massachusetts

Goldratt, E (1997) Critical Chain, North River Press Publishing Corporation, Massachusetts

Irvine, P (2004) Hogmanay: the inside story. The Scotsman, 17 January, Available from: < http://news.scotsman.com/archive.cfm?id=57632004 > [accessed 19 March, 2004]

Johnson, J (2003) ROI – It’s Your Job, The Standish Group International, Available from: < http://xp2003.org/xp2002/talksinfo/johnson.pdf > [accessed 19 March, 2004]

Larman, C (2003) Agile and Iterative Development—A Manager's Guide, Addison-Wesley, Boston

McConnell, S (1996) Rapid Development - Taming wild software schedules, Microsoft Press, Washington

McConnell, S (1997) Software Project Survival Guide, Microsoft Press, Washington

McConnell, S (2003) Professional Software Development: Shorter Schedules, Better Projects, Superior Products, Enhanced Careers, 2nd ed, Addison-Wesley, Boston MA

Miller, R (2003) Managing Software for Growth – Without Fear, Control and the Manufacturing Mindset, Addison-Wesley, Boston MA

Poppendieck, T and Poppendieck, M(2003) Lean Software Development: An Agile Toolkit for Software Development Managers, Addison-Wesley, Boston MA

Reinertson, D (1997) Managing the Design Factory – A product developers toolkit, The Free Press, New York

Scheinkopf, L (1999) Thinking for a Change, St Lucie Press, Boca Raton

Schwaber, K and Beedle, M (2002) Agile Software Development with Scrum, Prentice Hall, New Jersey

Sheldon, F, Kavi, K, Tausworth, J, Yu, J, Brettschneider, A, and Everett, W (1992) Reliability Measurement from Theory to Practice IEEE Software, July

Standish (1994) The CHAOS Report, The Standish Group International, Available from: < http://www.standishgroup.com/sample\_research/chaos\_1994\_4.php > [accessed 19 March, 2004]

Trochim, W (1999) Introduction to Validity, <<http://trochim.human.cornell.edu/kb/introval.htm>> [accessed 19 March, 2004]

Weinberg, G (1982) Rethinking Systems Analysis and Design, Addison-Wesley, Boston

Weinberg, G and Gause, D (1989) Exploring Requirements: Quality Before Design, Dorset House, New York

Yin, R.K. (1981a) “The case study as a serious research strategy.” Knowledge: Creation, Diffusion, Utilization. 3 (September), pp.97-114

Yin, R.K. (1981b) “The case study crisis: some answers.” Administrative Science Quarterly, 26 (March), pp.58-65.

1. *Product scope, scope, functionality and requirements* are used synonymously throughout this report. [↑](#footnote-ref-1)
2. For example:

H1 - the analysis phase could be done faster (my sponsors initial suggestion)

H2 - the analysis phase is done poorly and causes many subsequent, more expensive, changes (from literature review & discussion with colleagues) [↑](#footnote-ref-2)
3. These techniques were taught in the Open University MBA course “B822 - Creativity, innovation and change”. [↑](#footnote-ref-3)
4. In technical integration, a PC Windows program written in the Java programming language may be integrated with a mainframe transaction handler, such as CICS or IMS, which runs a COBOL mainframe program, which calls other COBOL programs and retrieves data from a DB2 database. Work package integration tests work flowing end-to-end through the system, for the first time. [↑](#footnote-ref-4)