The Journal of Contemporary Issues in Business and Government
2009 Volume 15, Number 1, pp 78–90

Managing IT: Does Agile Software Development Solve Buyer Needs?

John Brown
Department of Immigration, Commonwealth of Australia

Lloyd Conrade
Guy Callender
Curtin University of Technology

Abstract

A variety of innovations in software development practices, referred to by the collective term agile, have arisen in recent years partly as an attempt to deal with the considerable amount of complexity inherent in many software development projects. These agile approaches to software development are increasingly gaining acceptance with suppliers of software development services. Agile, in its various forms, appears to be a range of supplier responses to a set of problems traditionally incurred in contracted software development projects. Yet the extent to which the needs of the public sector (as one category of procurer) are faced is more questionable. Are the needs of a public sector procurer really satisfied with agile methods and practices as the framework for software development? Is agile really the solution to the original problem set?

Introduction

Core motivations incurred within the public sector often differ fundamentally from those experienced by the private sector (Sykes & Callender, 2009). For the latter, wealth-maximisation for shareholders, often through pursuit of maximisation of profits, is a strong driver for decision making. This pursuit and motivation can give rise to a wide range of risk-taking thresholds by both the owners and managers of private sector entities as they seek opportunities to render higher profits and higher returns on investment.

For the public sector, however, the profit motive is usually less significant. In its place, a greater responsibility to government and to the wider society is present, bringing with it diverse and competing goals both public and functional in nature. The need for a greater level of accountability within the public sector means that increased transparency in decision making is required, with standard and often rigorous procedures developed and instituted to accord with such transparency and accountability requirements. A general aversion to risk consequently arises in the public sector, which influences public sector procurement decision making. In recent years the rise of value for money as an increasingly dominant determinant of financial decisions reinforces a predilection for as much upfront certainty as possible concerning the various criteria which may influence a procurement decision. Knowledge (and, likewise, certainty) is power—the power to make effective procurement decisions.

Procurement of software development services is no exception in this quest for greater certainty, witnessed, for example, through the extent of general procurement guidelines issued by the Australian Commonwealth Government (CoA, 2008). The often large value of expenditure involved in software development procurement by public sector entities makes the overall *success* of software development projects much more vital. On the other hand, perceptions of public sector project *failure* (whether accurate or not), if played out publicly in the mass media, can have considerable undesired repercussions (Callender, Twomey, Chong & Huang, 2008).

If uncertainties associated with a project, particularly a large-scale project, are reduced at the beginning, then higher quality decisions can theoretically be made at the early procurement stage of the project, thus reducing the chance of ultimate project failure. The quest for upfront certainty is both a contributor to improved procurement decision making as well as mitigation against the potential for project failure later, with its resultant potential for political and public repercussions.

Recent moves within the software development industry to adopt what have become known as agile practices are an attempt to improve upon software development practices and to deal with certain types of failure recurrent in software development projects (Jamieson, Vinsen & Callender, 2006). Indeed, the incurrence of these failures has been sufficiently frequent that they have often seemed almost endemic to the provision of software development services.

Some Common Problems of Project Failure

Awareness of the relatively high frequency of specific software development failures stretches back several decades. Naur and Randell (1969) reported that a conference sponsored by the NATO Science Committee had recognised that software development projects often suffered a specific set of recurrent problems: they frequently and significantly exceeded their original budget; they frequently and significantly exceeded their originally envisaged period for development; and they frequently fell considerably short on initially envisaged functionality. Such was the perceived extent of these problems that the term *software crisis* was coined and has frequently been used since then to describe the common recurrence of these problems. However, some academics and practitioners, such as Glass (2005, 2006), question the extent of the problems and whether there has ever actually been such a crisis in software development, looking at the amount of well-functioning software on which we currently rely. Nevertheless, whether a project meets or exceeds its original budget, meets or exceeds the originally allocated development schedule, or fulfils or otherwise falls short on planned functionality have often been perceived as core criteria for determining whether a software project can be categorised as a success or a failure.

The most commonly referenced source of information for software project failure rates has, over the last one-and-a-half decades, been a series of Chaos Reports issued by the Standish Group. Most references are to the initial 1994 report which suggests that from a survey of 365 executive managers covering 8,380 applications (presumably meaning IT projects) only 16.2 percent of IT projects are *successful* while 31.1 percent of projects are cancelled at some point prior to completion. The remaining 52.7 percent are categorised as *challenged*, which is defined as: 'The project is completed and operational but over-budget, over the time estimate, and offers fewer features and functions than originally specified'. These are the same three aspects of project failure which were referred to by Naur and Randell in 1969 as bringing about a software crisis.

In its 2006 Chaos Report, the Standish Group found a trend towards greater success in IT projects since their 1994 report and that successful projects by 2006 constituted approximately 35 percent of projects, cancelled projects totalled 19 percent, and challenged projects had fallen to 46 percent of the total number of IT projects. The reported figure of challenged projects was still a very significant percentage of the total, even if approximately correct as an indication of over budget, over time, and functionally short projects in the software development industry. Any such extent of failure would probably be thoroughly unacceptable in other industries.

However, and despite the frequency of references to the Standish reports, deficiencies in the conclusions reached in these reports, as well as doubts about the research methodologies employed, have been highlighted by Jørgensen and Moløkken-Østvold (2006) and Glass (2005). Indeed, it is not at all clear if Standish classifies a project as challenged if only one or two of the three aspects of failure of challenged projects apply to that project. The Standish definition of challenged seems to require all three aspects of failure to co-exist in the one project. Clearly there are deficiencies in the Standish conclusions which need to be interpreted and applied with caution. Further, although one or more of these aspects of failure may have been incurred within a particular project, the final implemented product may actually meet with considerable success due to sizable gains arising from functionality, productivity or financial improvements (Callender, Twomey, Chong & Huang, 2008). Is such a project a failure or success, or does each aspect of failure or success, however defined, need to be considered on its own merits? Perhaps there is a need for more rigorous, comprehensive and up-to-date research to be conducted in this area.

A partial-corollary of the problem of projects falling short on functionality in order to meet the time or cost constraints is the alternative problem of *scope creep*. Scope creep is the repeated pressure to extend the scope of a project to include ever-increasing levels of newly-desired functionality. If the time and cost constraints are allowed to loosen, scope creep often replaces the functionality-deficiency problem. As the scope continues to widen, the project takes even longer and costs even more, and thus scope creep fuels these two other problems. Consequently, the existence of scope creep is often viewed as a warning sign that a project may ultimately fail (Jamieson, Vinsen & Callender, 2006; Thorp, 2008).

With a lack of currently available information providing a more comprehensive and precise assessment of how much projects do not actually meet their time, budget and functionality targets, or incur scope creep, it is commonly agreed that these types of failure in software projects remain a sufficient problem to be an ongoing concern, even if their extent may not currently be considered sufficient to constitute a software crisis as originally described by Naur and Randell (1969).

The Public Sector and Project Failure

Any occurrence of these types of failures of software development projects can be significant when the project is in-house within the private sector. Such failures are particularly critical when a software development project is the subject of an outsourcing agreement as the contractual relationship brings legal responsibilities as well as legal and financial risks associated with agreed outcomes not being met.

The vital aspect of the project outcomes may assume even greater significance when the buyer of the software development services is from the public sector. This is because a public sector procurer usually has heightened accountability for outcomes, is often constrained by considerably fixed financial budgeting, and often has rigorous and complex procedures to follow in an atmosphere which pressures for transparency in relation to, for example, procurement of goods and services. Software development success (in terms of not incurring excessive scope creep, meeting the agreed budget, meeting the allocated period for development, and meeting the functionality agreed upfront) is therefore often a critical requirement for a public sector procurer of software development services.

Bellamy and Taylor (1998) suggest that public sector IT development proceeds through three stages. *Automation*, as the first stage, represents the replacement of manual (mainly paper-based) work with much the same work being conducted by machines. *Informatization*, as the second stage, focuses on making information more accessible through integration of data sources, improved use of organisational and external data, and flexibility of communication of, and access to, data. *Transformation*, as the third stage, involves business process re-engineering, including across organisational boundaries, to reorientate delivery of services around the composite needs of specific types of customers. Progress of IT development through each of these stages accords with a continual increase in inherent complexity and associated uncertainty. This is particularly true in the third stage of transformation which may often require the satisfaction of seemingly conflicting or competing needs from multiple public sector entities each with their own political and economic agenda.

In recent years there has been an increasing acceptance and adoption by the software development community of what have become known as agile practices and methods of software development. These practices and methods have been specifically aimed at dealing with the problems associated with projects containing considerable uncertainty and complexity.

Traditional Software Development: Waterfall and its Variations

In describing his own experience with large software developments, Royce (1970) illustrated a basic flow of seven successive steps involved in developing software where each step is completed prior to the commencement of the next (see Figure 1). The basic form of his model has since become widely known as the waterfall software development lifecycle. Contrary to many references to the waterfall lifecycle since, Royce did not propose that software developers follow his basic process model. Rather, he considered that its '*implementation … is risky and invites failure*' (Royce, 1970: 329). In addition, he viewed iterations of the steps within the entire process, together with elements of re-design, as being essential to successful project completion, and he also recommended prototyping of the final software product envisaged. However, Royce's suggested embellishments were ignored in the mainstream and the strict sequential form of the waterfall process has been treated since as the standard conceptual base framework for software development, albeit with some naming and interpretive alterations of the steps involved (see Figure 2).

The waterfall software development lifecycle provides a considerably fixed structure for the sequence of the types of activities that need to be performed in software development. A traditional belief is that there should be an emphasis on sound documentation being developed as comprehensively as possible throughout every stage of the development process and 'that no phase is complete until the documentation for that phase has been completed' (Schach, 2007: 51). Comprehensive documentation theoretically provides all parties associated with the project with the information they need to make decisions and to progress the project further, and theoretically provides confidence that the project is on track. For example, by thoroughly specifying the project requirements by the end of the requirements and analysis stage and then incorporating a budget for the entire project, financial decision-makers have sufficient information to decide whether to proceed with the project and can then budget and allocate funds accordingly. This information is crucial in the public sector, for example, where there are strong capital budgeting constraints and budgets are most often set at least a year in advance.

Figure 1: Royce's Model That Became the Basis for the Waterfall Software Development Lifecycle.

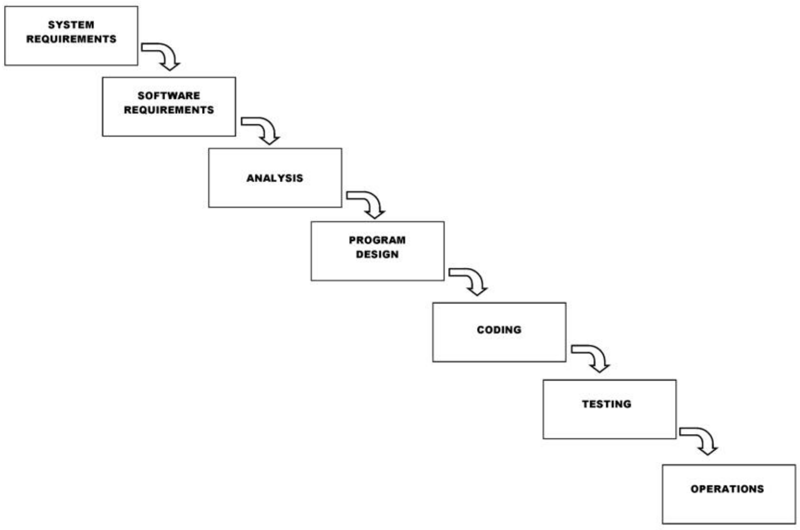
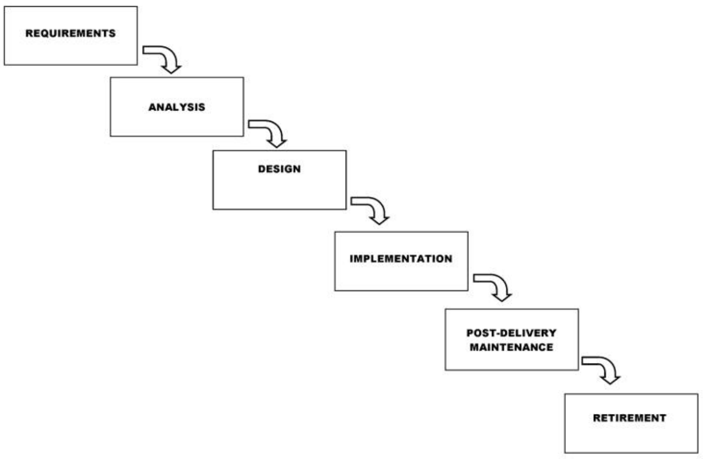
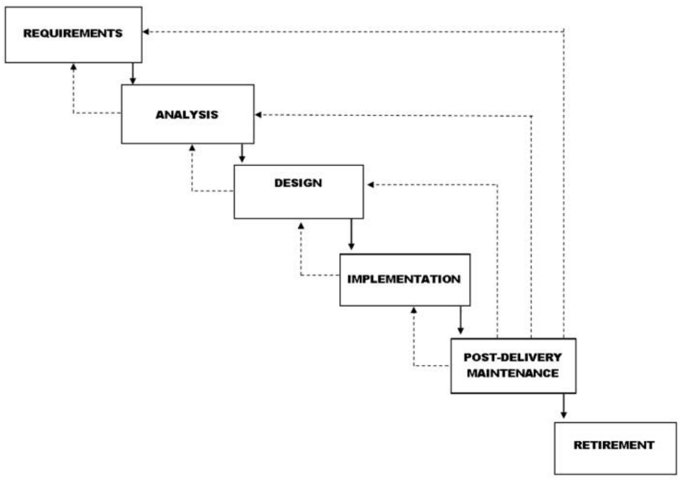


Figure 2: A Common Version of the 'Standard' Waterfall Software Development Lifecycle.



It is widely believed that the strict sequential flow of the waterfall lifecycle is never actually followed in practice. Rather, as was recognised by Royce (1970), software projects always undergo some degree of iteration of the phases involved. This is the result of a variety of factors. For example, the complexity of many large software development projects means that, no matter how detailed the development of requirements in the beginning, some level of detail will inevitably be omitted in that phase and only noticed at a later stage, perhaps not until after the project has been implemented. Refinement of the quality of the product may also be recognised and require further work or re-work. The near inevitability of human error also contributes to the need to revise and improve upon the work of an earlier phase. Consequently, a range of variations to waterfall have been proposed over the years (see Figure 3, for example) which attempt to portray the iterative nature of software development projects within a waterfall lifecycle context. With each iteration or additional level of work resulting from unforeseen complexity, the project incurs the risks of scope creep and running over budget and over time. The common method of dealing with this is to build additional time and cost allowances into the project from the beginning. But what constitutes a sufficient allowance in an environment of indeterminate levels of uncertainty and complexity? Clearly, with project failure still so prevalent, this level of management of uncertainty and complexity seems to frequently be proven inadequate.

Figure 3: A Variant of the Waterfall Software Development Lifecycle showing Iterations to Development.



The Rise of Agile

In recent years various alternatives collectively referred to as agile processes have become more popular as mechanisms used by software developers to attempt to deal with project uncertainty and complexity. In early 2001, 17 software development practitioners gathered, over a series of days, for the purpose of identifying commonly-held ideas which could collectively form the core of an alternative to standard software development practices. The outcome of their gathering included what became known as *The Manifesto for Agile Software Development* (2001), which states that:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> - ***Individuals and interaction**s over processes and tools*
> - ***Working software** over comprehensive documentation*
> - ***Customer collaboration** over contract negotiation*
> - ***Responding to change** over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more.*

The participants backed their Agile Manifesto with a set of twelve associated principles which, in combination with the manifesto, have spurred an agile software development movement. It should be noted that the term *agile software development*, often simply referred to as agile, is not a reference to one strict agile methodology. In fact the original proponents of the Agile Manifesto themselves had between them a variety of approaches to software development. They continue to pursue these approaches under various terminologies such as Scrum, Extreme Programming, Crystal Development, Lean Development, Feature-Driven Development, Dynamic Systems Development Method, Adaptive Software Development, and (Agile/Rational) Unified Process. Each approach has its own methodologies and/or processes but they all endeavour to comply both with the values of the Agile Manifesto and the twelve principles.

Regardless of the specific approach adopted, there are various typical characteristics of projects developed through agile processes, as explored at length by Highsmith (2002) and Sliger and Broderick (2008). For example, there is an emphasis on first delivering to the customer those aspects of functionality that are considered most important. A requirement of strong customer involvement throughout each iteration is seen as being amenable to the generation of immediate feedback regarding the appropriateness or otherwise of a product delivered. Practices such as pair-programming (two programmers working together with one checking the work of the other) are frequently promoted for rapid, focussed development turnaround.

There is also a strong emphasis on frequent delivery to the procurer of progressive iterations of *workable* software, with the development team often seeking to deliver some additional functionality as frequently as every month, every fortnight, or even every week. At this rate of iterative and incremental development, there is automatically far less emphasis on developing comprehensive documentation—a fact agile proponents readily admit. This rate of iterative delivery also fundamentally alters the software development lifecycle so that, for each iteration delivered, the developers by necessity work on every phase inherent in the waterfall software development lifecycle but with specific emphasis on the short-term work relating to that specific iteration.

Clearly, agile software development is oriented towards attempting to manage development projects in an adaptable manner (by both the development team and the customer) with less comprehensive planning (especially at the beginning of the project) and a shorter-term focus throughout. Proponents of agile view their processes as an attempt to deal with relentless speed and change through an ability to act quickly to deliver fast results. Highsmith (2002) suggests that 'problems characterized by change, speed, and turbulence are best solved by agility', with XP, Crystal and Scrum being 'particularly relevant to extreme or complex projects – those that have an accelerated time schedule combined with significant risk and uncertainty that generate constant change during the project' (p. xxii).

Agile software development processes are a clear recognition by the proponents of agile of the existence of inherent complexity and uncertainty in software development and a response by developers seeking improvements in the way they conduct their development work. Agile attempts to deal with the complexity and uncertainty of projects by ensuring a shorter-term focus and more frequent delivery of smaller units of product in order of the most-needed functionality. Thereby they perceive that they provide their customer—the purchaser of their services—with the elements of a product most needed by that customer. In fact their customer's needs might critically be the totality of product, not some prioritised portion of the totality. Agile processes are therefore a developer's perspective in seeking to resolve the perceived problems inherent in software development.

The Cost of Agile

Proponents of agile software development processes, however, have had little to offer a public sector procurer in terms of the reliability of the pricing of agile projects. Most proponents publicly remain totally silent on this issue. Sliger and Broderick (2008) see a difference in how the costing of agile projects must occur compared with how costing of traditional (waterfall and related) projects are usually conducted. For traditional projects, Sliger and Broderick (2008) believe, the entire project work is broken down into specific tasks, cost estimates of the likely resources required for each task are ascertained, and the total cost of the project is the total of all the task cost estimates. This approach, referred to as bottom up estimating, is possible because the project has been analysed in considerable detail upfront.

However, with agile projects no such detail of project breakdown is performed upfront. Such a level of detail of the project is worked out, if at all, progressively during the various iterations of the project's development and a total cost of the project may not be known until the final iterations of the entire project. The focus of the development team at any time is only on the shortest time horizon—the work immediately at hand (Sliger & Broderick, 2008: 42):

> *A team should only plan one release at a time: this release. Likewise, a team should only task out one iteration plan at a time: this iteration. Estimating top-down and relative to the horizon lowers the upfront planning costs as well as the cost of change later in the project.*

Consequently, there is no firm basis at the beginning of undertaking a project for an agile developer to reach any reliable measure of the total project cost. Sliger and Broderick seek to diffuse the responsibility of the software developer (as supplier) to accurately place a cost on their services by stating that 'it's very difficult, if not impossible, to predict costs for a complex situation' (Sliger & Broderick, 2008: 113). Yet estimating costs and providing the public sector purchaser with a price upfront is exactly what is required of a supplier of software development services seeking to bid for the project, no matter how difficult or seemingly impossible it appears to be to do so. And the same responsibility exists regardless of whether or not the supplier chooses agile development processes. If the chosen process of the supplier does not allow the supplier to meet the essential requirement of the public sector procurer (that is, an accurate estimate of price for a given scope and declared project duration), then that is a limitation of the development process chosen.

Sliger and Broderick further attempt to diffuse this fundamental issue by declaring that: 'Agile projects have the ability to always be on time, and within budget, if the scope is flexible' (Sliger & Broderick, 2008: 117). The problem with this is that scope is rarely flexible on the downside. With an outsourced IT project, this is akin to the supplier suggesting to the buyer: 'If we don't get all the work completed that we committed to finish within the time period we committed to, we won't charge you more than we committed to charge you. But if you really want the project completed in its originally foreseen entirety, it may take longer and cost more than we previously identified'. With this lack of full commitment from the agile supplier in relation to the supplier's own tender, the fundamental problem of outsourced software development projects is not addressed, let alone resolved, by engaging in agile processes.

If the price of commitment to project duration and project cost by an agile developer, as supplier, is the project scope (and scope could also quite validly in this context incorporate product functionality), then it is quite reasonable from the public sector procurer's perspective for the credibility of an agile supplier's tender to be called into question from the outset. The procurer is seeking a commitment from the supplier in relation to cost, scope and duration and the agile supplier readily admits that the procurer can, at most, have a commitment to only two of these three project fundamentals.

If this is the case, the tender and agile development processes appear to be fundamentally in conflict from the procurer's perspective regarding requirements of and constraints upon the software development project. The price of agile software development may well then be more than just the total financial cost to the buyer. There may also be ultimate pressure on the procurer to forego one of the procurer's fundamentals of the project—to either forgo the limit on time, or forgo the limit to cost, or (even more likely) forgo the scope and functionality of the product produced.

Conclusion

Several core problems continue to arise in software development projects. These are the problems of functionality being eliminated from the end product as originally envisaged, software projects taking longer than forecast, original budgets being exceeded, and/or the continual issue of scope creep. These are of particular concern for the public sector with its strong accountability requirements which necessitate considerable certainty associated with procurement decision making. These problems, from the procurer's perspective, are actually issues of uncertainty: where what was considered to be certain or true at the beginning of the project becomes less and less certain, or less and less true, as the project progresses.

To this extent, agile software development contributes no advancement to the solution of the core problems faced by the public sector procurer. Rather, it has the potential to contribute further uncertainty upfront in terms of the scope and the functionality to be incorporated in the final product, the time to develop the product, and the total cost of the product to the procurer. The extent to which agile software development contributes to this uncertainty constitutes a backward step for the public sector procurer and extends the problems incurred.

Agile software development is the software development supplier's response to the problems the developer faces in the course of his/her own contributions to software development projects but it is not a solution to the repetitive occurrence of the specific software development problems faced by the public sector procurer. The public sector incurs specific constraints and needs simply because of the very nature of the public sector. These constraints and needs do not change to accommodate an agile software development supplier. Proponents of agile software development need to recognise this and then deal with the problems that arise from the public sector procurer's point of view rather than to potentially add to those problems.

If proponents of agile software development fail to deal with these issues, then perhaps it is appropriate to view agile processes and practices as falling short of the needs of the public sector procurer. Developers applying agile software processes are seeking improvements to how software is developed from their own perspective. That is commendable, yet their improvements are not solutions to the core problems faced by the public sector procurer. They have yet to deal with the core, long-standing problems of their public sector customer from their public sector customer's perspective.

Acknowledgements

The authors wish to thank reviewers and delegates at the 2009 meeting of the Global Business and Technology Association in Prague, Czech Republic, for their comments and advice on this paper.

References

Bellamy, C. and Taylor, J.A. (1998) *Governing in the Information Age*. Open University Press, Buckingham.

Callender, G., Twomey, A., Chong, S. and Huang, X-Z. (2008) Influencing government contracts: What the media said about success. Proceedings of 17th IPSERA Conference, Perth, March, pp 343-8.

Commonwealth of Australia (CoA) (2009) Commonwealth procurement guidelines. Department of Finance & Deregulation, Canberra. Retrieved: 20 March 2009 from <http://www.finance.gov.au/publications/fmg-series/procurement-guidelines/index.html>

Glass, R.L. (2005) IT Failure Rates – 70% or 10-15%? *IEEE Software*, 22 (3) pp 110-12.

Glass, R.L. (2006) The Standish Report: Does it really describe a software crisis? *Communications of the ACM*, 49 (8) pp 15-16.

Highsmith, J. (2002) *Agile Software Development Eco-Systems*. Addison-Wesley, Boston, MA.

Jamieson, D., Vinsen, K. and Callender, G. (2006) Agile procurement and dynamic value for money to facilitate agile software projects. Proceedings of 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06), Cavtat/Dubrovnik, August/September, pp 248-55.

Jørgensen, M. and Moløkken-Østvold, K. (2006) How large are software cost overruns? A review of the 1994 CHAOS report, information and software technology. *Amsterdam*, 48 (4) pp 297-301.

The Manifesto for Agile Software Development (2001) Retrieved: 22 March 2009 from <http://www.agilemanifesto.org>

Naur, P. and Randell, B. (1969) Software engineering. Report of a conference sponsored by the NATO Science Committee, Garmisch, October.

Principles behind the Agile Manifesto (2001). Retrieved: 22 March 2009 from <http://www.agilemanifesto.org/principles.html>

Royce, W.W. (1987) Managing the development of large software systems. Proceedings of 9th International Conference on Software Engineering, Monterey, CA, March/April, pp 328-38.

Schach, S.R. (2007) *Object-Oriented and Classical Software Engineering (7th Edition)*. McGraw Hill Higher Education, Boston, MA.

Sliger, M. and Broderick, S. (2008) *The Software Project Manager's Bridge to Agility*. Addison-Wesley Professional, Reading, MA.

The Standish Group International (1994) The Chaos Report, 1994. The Standish Group International.

Sykes, M. and Callender, G. (2009) Deconstructing the public-private mix in procurement: A preliminary study. Presentation to 18th IPSERA Conference, Oestrich-Winkel, April.

Thorp, J. (2008) IT project cancellations: Pay now or pay later. *Information Systems Control Journal*, 1 (Jan) pp 18-19.