Dr Alan McLucas
School of Information Technology and Electrical Engineering UNSW@ADFA
Australian Defence Force Academy
Northcott Drive
Campbell ACT 2600
AUSTRALIA
Tel +61 2 6268 8332 Fax +61 2 6268 8337
a.mclucas@adfa.edu.au

How to Deliver Multi-phase Software Development Projects: System Dynamics Simulation of Alternate Project Strategies

Abstract

This paper examines what differentiates multi-phase software development and integration projects from other complex projects. It argues that effectiveness in identifying defects in the early phases of a project results in early rectification (rework) and initially slows down the project. But, this is necessary to improve likelihood of successful delivery of subsequent phases. Lower levels of effectiveness in identifying defects early, creates the need for higher levels of subsequent rework, raising the possibility of rework of rework. The need for seemingly indeterminate amounts of rework is examined. The key drivers of successful reduction in amounts of rework are investigated through the use of system dynamics modeling and simulation. Most importantly for managers of projects involving software development is that scenario planning can be used to identify where management and engineering efforts are best directed. Through the use of system dynamics modeling and simulation, this can be done before committing to a particular software development project.

Correct at: 18 March 2008

Introduction

This paper addresses the systemic influences which underpin cost and schedule overruns in multi-phase software development projects, such as those undertaken for military applications. It examines how managers can make robust estimates of time and cost for software projects, and indeed for all developmental projects. It also examines how effective strategies for managing resources allocated to such projects, particularly for the management of quality might be developed.

What Makes Developmental Projects Different

Man has delivered complex projects for thousands of years. If the primary focus is on delivery of some highly desired capability, or impressive end products, then countless successes have been delivered. Around 4500 years ago the ancient Egyptians engaged in building the first of the mighty pyramids. Over 2000 years ago the Romans built cities with water reticulation and sewerage systems. Cities of the Roman Empire were connected by over 90,000 kilometres of paved roads. Some roads, bridges and aqua ducts built by the Romans are still in use.

Early last century, the Pacific and Atlantic oceans were joined by the building of the Panama Canal. In the 1960s the United States of America fulfilled the dream of President John F. Kennedy of taking man to the Moon and bringing him back alive. In the 1990s, England and France were joined by the Channel Tunnel.

These projects are held as exemplars of man’s achievements and when measured in terms of what has been ultimately delivered, doubtless they have been outstanding successes. However, if achievement of the original cost and schedule estimates were applied as key measures of success, then it is likely that we would be seriously challenged to defend any of these impressive projects against being classified as failures.

What these project share is that they were unique undertakings. At the time they were undertaken they involved new technology, which often had to be developed before the project could be finished. They involved innovation, experimentation and development of new ways of doing things.

Building the Panama Canal was not simply a matter of excavating a long canal wide enough to enable the passage of tankers and container ships. To build such a canal required engineers to find and survey alternate routes through jungles, swamps and over mountains. They had to find ways around rock that was too hard to excavate and too expensive to blast their way through. They had to find ways of raising ships many hundreds of metres over the mountains. Rates of progress in building the Canal were excruciatingly slow at times. The project stopped and restarted several times because sufficient funds and workers could not be found. During construction, thousands of workers became ill and some 20,000 died. In the face of a mounting death toll the project was stopped until mosquito-free barracks could be built to protect workers from mosquito-borne disease and medical facilities provided to treat the sick so they could return to productive work.

The Channel Tunnel had its own false starts and work stopped on several occasions. Just when it looked like it would be finally finished, extensive re-work was required. For example, means of extracting the heat, a consequence of friction between molecules of air forced through the small gap between the tunnel walls and trains travelling at high speed, had to be found.

Each of these projects involved substantial rectification, or rework. That is, work not done correctly or not producing the required performance or reliability had to be done again. What we have learned from centuries of complex projects is that the need for this rework is generally lessened as we gain experience about project-related processes or as we learn more about the technology involved.

Unfortunately, gaining experience is costly, requiring us to fail so that we might succeed in the future. Some types of contemporary projects are almost always delivered late, and prove to be much more expensive than the original estimates, provided at the time funding for the project was approved. Consistently amongst projects that overrun budget and schedule are those that involve software development. This remains the case despite a growing body of knowledge and experience about the detailed processes followed to deliver technical outcomes.

Modern Developmental Projects

Modern developmental projects can be likened to the mega construction and aerospace projects mentioned, insofar as new ways of working or developing new technology to overcome unexpected problems have to be found. However, all developmental projects are continually under threat of slipping behind schedule, with consequent increases in cost, for surprisingly simple reasons. Prime amongst these reasons is almost universal inability to reliably estimate the extent of rework needed. Even when we are able to describe the processes involved in rework cycles with some precision, our judgment fails us. Estimates we make of time to complete any project involving rework at best are unreliable, and at worst are not much better than informed guesses. Further, this situation is exacerbated by linear-causal thinking embodied in our training, our education and the prescriptive methods encapsulated in the body of knowledge about the management of projects.

Our inability to make reliable estimates of the extent of rework and implications this has for schedule and cost is an insidious problem. This is derives from:

a. human cognitive limitations when faced with dynamic feedback (Dörner, 1980; Diehl and Sterman, 1995; Kleinmuntz, 1985; 1993; Kline, 1995; Richardson, 1991; Sterman, 1989a; 1989b; 1989c; 1994); and
b. time dependent behaviour being difficult to predict, with dynamic feedback response to remedial strategies being almost always counter-intuitive (Meadows, 1980; Forrester, 1968; 1971; 1975; 1987; Nuthman, 1994)

It might be argued that unlike building of the first pyramid, first time construction of an inter-ocean canal, putting man on the Moon for the first time and bringing him back alive or developing novel tunnel-boring methods for building the world's longest under-sea tunnel, developing software for military applications would be relatively straightforward. This might appear to be so when our accumulated knowledge about projects is considered. Undertaking military software development projects surely is a matter of applying well known methodologies for capturing requirements, design the software, writing code, integrating modules and acceptance testing, for example.

Systems engineering methodologies (Blanchard and Fabrycky, 1997; Stevens, *et al*., 1998) which provide the foundation for the engineering of software were spawned by NASA's Apollo space program. These methodologies have become highly developed in the last 50 years, and continue to evolve. The software engineering body of knowledge is now comprehensively defined. Many engineers and project managers are trained in software engineering and software project management.

Billions of lines of commercial software code have been written. Some software applications we use every day such as Microsoft™ Word are exceedingly complex, contain millions of lines of code. There is a large pool of experience in writing software. Indeed, some of the world's largest corporations have built their success on their declared ability to develop software.

Unfortunately, this commercial experience does not readily translate through to success in the development of software needed for military applications. This is particularly so if we measure success in terms of achieving planned cost and schedule. Whilst the software used in military often contains orders of magnitude fewer lines of code than commercial applications this does not improve success rates in delivering military software projects. By comparison with Microsoft™ Word which contains round 3,000 thousand lines of code (3,000 KLoC); the C code which controls the operation of a contemporary combat net radio contains some 30 KLoC. In one particular instance, software built to control the operation of a modern military combat net radio was subjected to field trials and its functionality was demonstrated. However, substantial rework was required because documentation needed to assure that quality management and engineering activities had been comprehensively followed was incomplete. The customer expressed doubts about the possible performance and maintainability of the software. The original equipment manufacturer (OEM) who developed the software offered to rectify this situation, estimating that the rework would take three to six months and costs incurred would not be passed onto the customer. The rework took around three years and is understood to have cost a further $15 million.

One key attribute that differentiates software for military applications from commercial software and increases the amount of rework is the need for military software to be ultra-reliable. Fielding beta-version software for operational use in anticipation of a final release, then entering into an extended program of reliability growth, is not an option for military systems. Further, military software is frequently embedded in hardware and this brings unique demands for creating working interfaces between the software and hardware.

Each of these issues places particular demands on software engineering and project management skills needed by programmers and quality management. But, before any project can be approved, viable estimates must be made of the time and cost associated with expected successful delivery of the required functional performance. Further, if success is to be repeatable in subsequent projects there must be confidence in the likely achievement of schedule and budget, as well as functional performance.

Perhaps the greatest threat to schedule and budget is not how to deliver technical performance, *per se,* but how to make reliable estimates of the rework needed. The project's sponsor needs to be convinced of the veracity of these estimates made before the project is approved. Further, the basis for claiming the veracity of these estimates needs to be established for currently planned and future projects to be successful.

Rework and Quality Management – A Simple Metaphor

A bathtub is filled with a very large number of marbles. For our purposes, these marbles are all green in colour. Each marble represents an amount of work to be completed, such as the writing of a number of lines of software code. Marbles are scooped out of this first bathtub one bucketful at a time. Each bucketful represents, for example, the writing of code planned to be completed in a single day.

Unfortunately, a small fraction of the marbles is damaged as they are scooped up from this first bathtub. This damage is akin to defects (errors or bugs) inadvertently being added to the software as it is written. Fewer defects are likely to result when highly skilled programmers are employed in writing the code, but there will always be some defects. The second bathtub is progressively filled with our metaphorical green marbles, some of which have been damaged by the very method used to move them from the first to the second bathtub. All marbles, including the damaged ones, are placed in this second bathtub.

Damaged marbles are not easily identified except by close and careful inspection by skilled inspectors with special equipment. Marbles from the second bathtub are carefully collected and inspected. Unfortunately, even the most rigorous inspection process will not find all damaged marbles. Each damaged marble found is painted red at this point. All marbles, both green and red are placed in a third bathtub by the inspectors.

Marbles are progressively taken from the third bathtub and the red ones are repaired. This is the rectification work carried out by programmers. When repaired, the red marbles are painted green. Now all marbles are green, appearing exactly as they did when they were placed in the second bathtub for the first time except that only a small number are defective, and those defects have not yet been discovered.

All the marbles so far inspected, those rectified and a small number of marbles having undetected damaged are placed back into the second bathtub. Of the green marbles now placed back in the second bathtub, the portion likely to be green again, having been identified defective and therefore rectified, is hopefully nearly 100%, but always effectively less than 100%, of those that were defective before they entered the inspection process. How close we are likely to get to finding 100% of the defects, of that portion of the total actually inspected, depends on the skill, effort and tools applied during inspection. The effectiveness of the inspection process is highly non-linear, meaning that results obtained are not achieved simply in proportion to the effort expended. Further, there is a minimum threshold in terms of effort which must be applied to achieve any significant results, and above a certain higher level further incremental increases in inspection effort will return marginally diminishing results.

When they are sufficiently satisfied with the quality of the rectification work, our team of inspectors select marbles from the second bathtub and pass these on to the next stage of the process or phase of the project. The aim here is to empty the second bathtub so that work can proceed through the next phase of the project. It is important to note here that marbles from the first bathtub and the third bathtub appear the same. They are randomly mixed. One consequence of this is that some defects will never be found. In the general case, some of our marbles are passed on to the next phase without being inspected.

A further complication here, not considered before now, is that our green marbles are related to each other. Some marbles will be related to a large number of marbles. Some marbles will be related to a smaller number of marbles. All marbles are related in some way, through the functional structure of the software being built. Our metaphor does not extend to taking this functional structure into account, but we can accept this limitation for now. In practice we would use knowledge of this structure to direct the focus of work, inspection and rework effort on critical lines, modules or functional blocks of code.

Each of the phases is similar in that each involves rework. Rework can also occur as a consequence of defects discovered in a later, or downstream, phase of the project. This inter-phase rework demands input into one or other of the earlier, or upstream, phases of the project. So, whilst rework is an element of each phase, there is always the possibility that the need for some inter-phase rework, or rework of the rework, will arise.

Meeting the Challenge of Managing Developmental Projects

Unfortunately, project management estimating is based too-often on a simple linear relationship between work to be done and rate of applying effort. Of course, the rate of applying effort is not uniform, and experienced project managers claim to know this. Indeed, this is well documented in the body of knowledge. Rates of progress are initially slow before ramping up to higher rates then slowing down as the project of phase nears completion. Regardless of the variations in rates of working as the project of phase progresses, the fundamental totality of work / work rate relationship is embedded in the way calculations of the estimate at completion (EAC) for the project, that is, what the whole project will cost to complete, are made.

In conventional project management terms, rework is almost universally accommodated by providing management reserves or contingency, where the provision of these is made on the basis of the experience of the estimator. Infrequently, or rarely, is the extent of rework quantified with any rigorous understanding of rework mechanisms, with the consequence that projects such as those involving software development result in significant cost and schedule overruns. It does not require much imagination, and reflection on the rework metaphor, to realize that our usual methods of estimating the extent of rework will fail us except where rework is minimal such as that for routine, technically naïve projects. To make reliable estimates of the impact rework will have on software development projects, we need different techniques.

Dynamic Model of a Single Phase Project Involving Rework

Single phase development projects involving process feedback, that is, having a single rectification or rework cycle have been described in the system dynamics modelling literature (Abdel-Hamid, 1984; Abdel-Hamid and Madnick, 1990; Ford and Sterman, 1998 and Taylor and Ford, 2006). In each case these studies have set out to explain why even experienced project managers consistently and seriously underestimate how long such projects are likely to take. System dynamics modeling techniques (Coyle, 1996; McLucas and Linard, 1999; Wolstenholme, 1990; Wolstenholme and Coyle, 1983; Sterman, 2000; McLucas, 2005;) are most suitable for this analysis because they focus on feedback causality. The application of these techniques, firstly to a single phase developmental project, and secondly to a multi-phase software development project is described.

In system dynamics modelling, stock-and-flow diagrams depict the relationships between state and rate-controlling variables. The basic rework cycle for a single phase software development project, depicted as a stock-and-flow diagram, is shown at Figure 1. The rework cycle involves a physical feedback loop where primary rates of flow through the phase *Rate of Writing Code*, *Rate Code Approved* and *Rate Code Released* and the rework cycle rates *Rate of Inspecting Code* and *Rate of Rectifying Code Found to be Defective,* are constrained by application of limited resources and the need to share resources within this single phase.

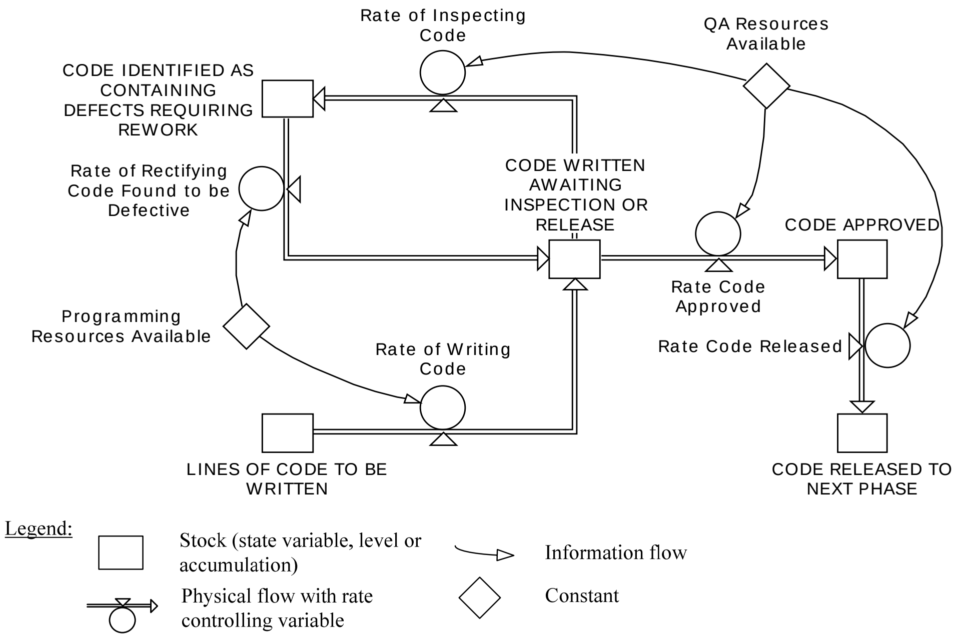
Rate of Inspecting Code
QA Resources Available
CODE IDENTIFIED AS CONTAINING DEFECTS REQUIRING REWORK
Rate of Rectifying Code Found to be Defective
CODE WRITTEN AWAITING INSPECTION OR RELEASE
CODE APPROVED
Rate Code Approved
Programming Resources Available
Rate of Writing Code
Rate Code Released
LINES OF CODE TO BE WRITTEN
CODE RELEASED TO NEXT PHASE
Legend:
Stock (state variable, level or accumulation)
Information flow
Physical flow with rate controlling variable
Constant

Figure 1. Stock-and-Flow Diagram Development Phase Involving Rework

As code is written, defects are inadvertently added. The rate of adding defects is linked to the rate of writing code, and can be estimated according to the maturity of the software engineering practices and the skill of the programmers. Depending on the intensity and effectiveness of effort applied to inspection, many of these defects will be discovered. Another sector of the simulation model (not shown in Figure 1) contains linked variables. These are used in calculating, for example, the likely incidence of defects being present in written code, probability defects will be found through routine quality assurance (QA) processes, allocation of resources to programming and rectification work, and allocation of QA resources.

In the following examples the focus is on a single phase. Later, focus will shift to a project which has several such phases. Whilst resources available to do the work are limited they are initially dedicated to the current single phase. As the number of phases in the project increase so the need to share the project's total resources across each of the phases increases. As will be shown later, allocation of resources across the phases constrains the rate of working and priorities need to be set for the dynamic re-allocation as the work to be done shifts from earlier to later phases.

Figure 2 depicts the rates of work achieved and residual defects when there is no inspection and no rework. Note that the y-axis indicates relative scales for the different variables and that this figure and the two that follow, Figures 3 and 4, should be used for comparison.

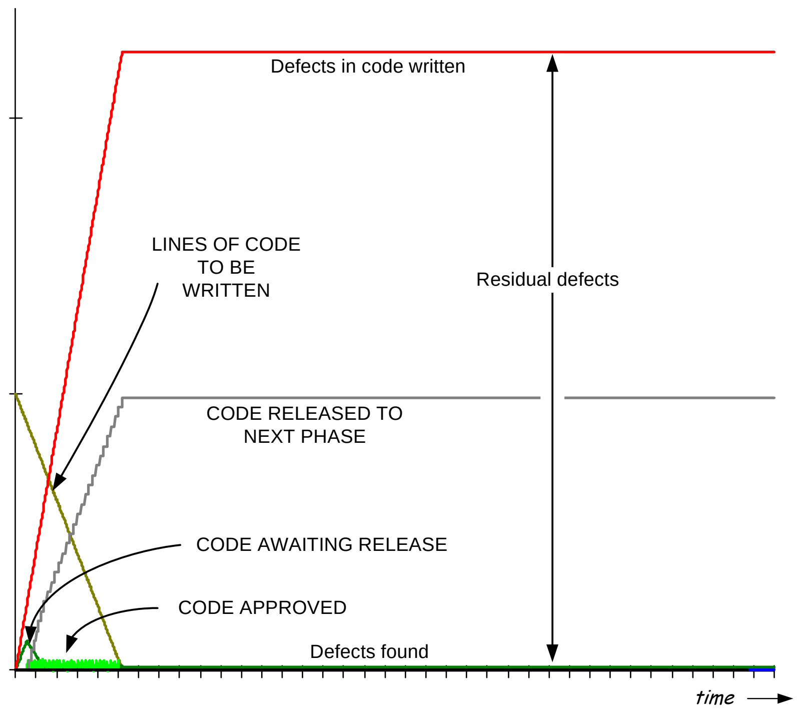

Figure 2. No Inspection No Rework

In this case code is released almost as quickly as it is written and time taken is determined simply from information about the amount of code to be written and the average expected time taken to write each line of code.

When high rates of effort are allocated to inspection and consequent rectification, the residual defects are greatly reduced, as shown in Figure 3.

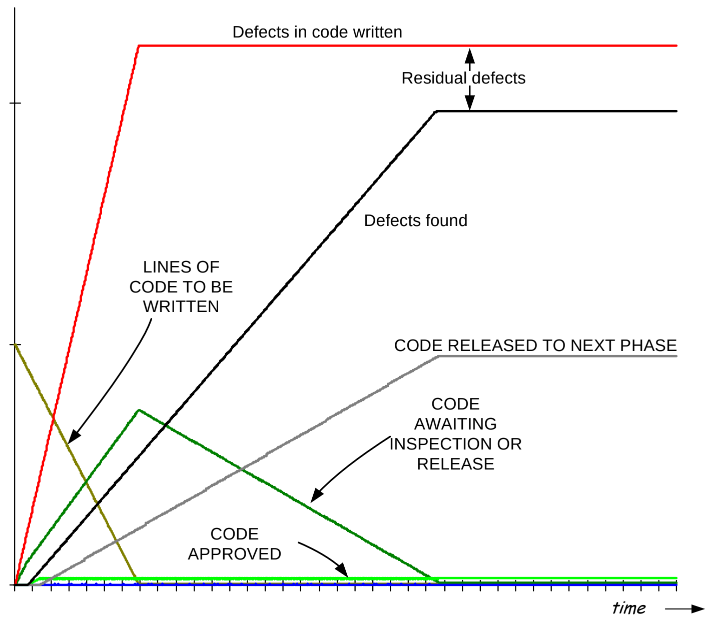

Figure 3. High Rates of Effort Allocated to Inspection and Consequent Rework

When moderate rates of effort are allocated to inspection and consequent rectification, the residual defects are greatly reduced, as shown in Figure 4. Note that whilst there is an obvious reduction in residual defects, the rate of writing code is slowed because effort must be applied to rectification work. Code awaiting inspection or release accumulates before being reduced because programming effort is divided between the writing of code and rectifying defects. Underlying the behaviour over time depicted in Figure 3 is an assumption that the efforts of programmers, shared between writing code and rectifying defects, are not dynamically re-allocated with priority of effort being given over to rectification work once initial writing of code has been completed.

However, if priority of effort is shifted to rectification once initial writing has been completed, in an equivalent time and with the same effort applied to inspection additional defects are found and rectified. This is shown in Figure 4.

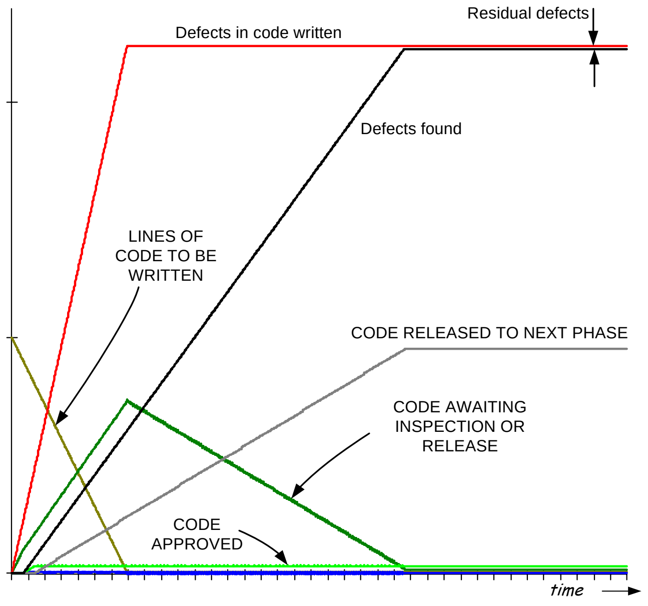

Figure 4. Dynamic Re-allocation of Effort to Rework and Consequent improvement in Detecting Defects

Figures 3 and 4 represent two of range of possible options for re-allocating available resources. Investigating such options becomes more important when there is more than a single development phase, in which case dynamic allocation of resources across the phases becomes increasingly important. Re-allocation and other such options can be investigated through a series of ‘what if’ scenario-based simulation runs.

Dynamic Model of Software Development Projects

A dynamic model of a generic software development project described in subsequent sections of this paper is assumed to have four interconnected phases:

1. Code writing.
2. Integrating coded modules.
3. Integrating modules into functional blocks.
4. Acceptance testing.

Each phase containing its own rework cycle. For the analysis described in this paper, it is assumed that rework outside that defined within each of the four phases is not required. This means that, for example, during the acceptance testing phase code which is found not to function as required is reworked during that phase rather than being sent back to an earlier phase for the rework to be done. This is an alternate representation of the Waterfall Development Model rather than a simplification of that model.

As the project proceeds, resources are progressively re-allocated depending on a balance of competing priorities for:

1. initially completing work, such as writing code or integrating coded modules;
2. managing the quality of work done in the first instance;
3. applying inspection effort to finding defects; and
4. applying programming effort to rectifying defects found.

The dynamic re-allocation of resources is aimed at achieving most effective employment of available. An unintended consequence, albeit a necessary consequence, is that some activities proceed more slowly than they might otherwise. For example, coding takes longer than would be possible if all resources were allocated to that task first. However, the actual allocation is based on considerations of the need to allocate resources to concurrent activities.

The options for resource allocation are investigated through a series of scenario-based simulations which seek to investigate time and cost implications. Through repeated simulations under a variety of scenarios, strategies are identified for:

1. making resource allocations which are effective in reducing the time taken to complete the whole project, rather than a particular phase;
2. making resource allocations which are effective in reducing the cost taken to complete the whole project, rather than a particular phase;
3. identifying where resource allocations are critical, and where supplementation may be needed;
4. rate-determining steps in processes, particularly where a number of activities are concurrent or otherwise interdependent;
5. allocating QA effort to reduce defects to target levels across the whole project; and
6. making trade-offs of cost and time for delivering the project.

These simulations reveal that traditional methods of estimating the work needed to complete the project are inappropriate.

Results of Simulations

Figure 5 (adapted after Elwell, 2007) demonstrates the simulated results of the project being completed over time. Rates of completion of each phase are shown. Note that the y-axis does not depict comparative scales for the extent of the work conducted in each phase. Rather, the maximum amount of work conducted in each phase has been normalized to 100%. For example, the range of values for *Phase 1 To Be Completed* is 32,000 to 0 lines of code, whilst *Phase 4 To Be Completed* is 1 to 0 where 0 represents completion of acceptance testing of the whole operating system. Each represents 100% of the work done.

For the intermediate phases, represented by *Phase 2 To Be Completed* and *Phase 3 To Be Completed* the maximum value in Figure 5 represents the amount of work that accumulates before the Phase is allowed to commence. For example, assembling of modules in Phase 2 does not commence until a specified number of modules, say 10, are available to be assembled. A similar control applies to Phase 3.

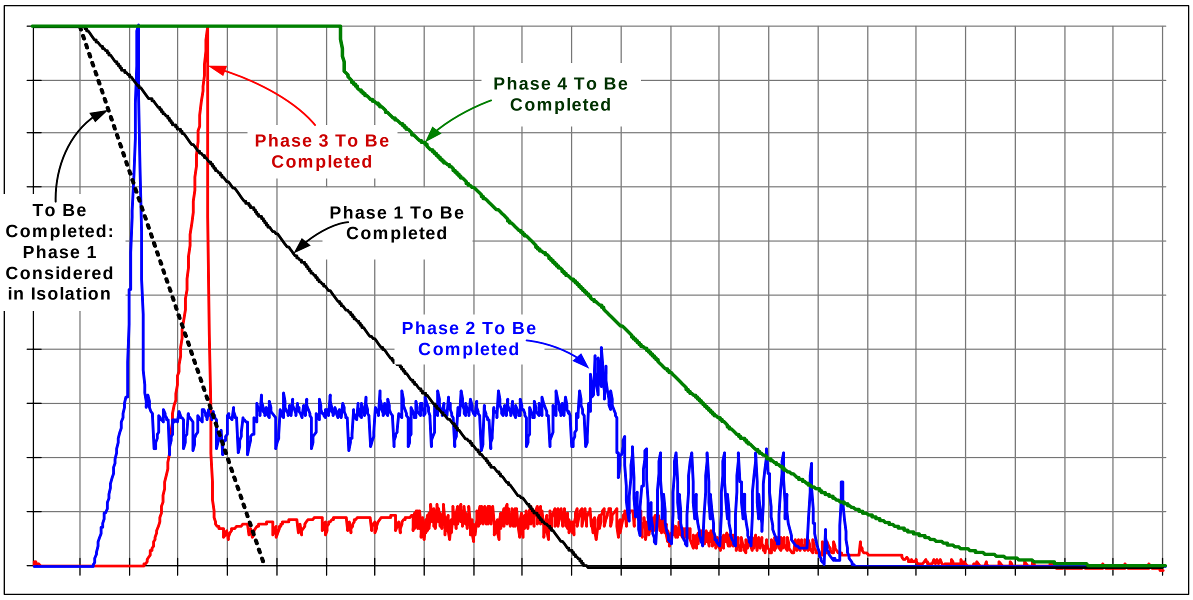

Figure 5. Simulated Completion of Project Phases

The dotted line *To Be Completed: Phase 1 Considered in Isolation* depicts the estimated work needed to complete Phase 1. Whilst this estimate is based on the model depicted in Figure 1, insofar as takes into account the need for inspection and rework to keep the defects at or below the target level, it does not take into account the need to share available resources with other phases. When concurrent working, and rework, with limited resources is taken into account, the rate of completing is significantly reduced. See *Phase 1 To Be Completed*.

The need to share resources across the other phases, whilst allocating resources for work, rework, inspection and QA activities within phases impacts on the rates of working that can be achieved in each of the phases. The rates of working and progress to completion of the project are non-linear.

It is of particular interest that when Phase 4 is 90% complete, 25% of the total time to complete the project is needed to complete the last 10%. In this case applying additional resources in Phase 4 will not improve the situation. The most effective way to reduce the time to complete the project is to employ additional resources from the outset.

The rates of finding and rectifying defects are non-linear and different across each of the phases. This is depicted in Figure 6, where the later three phases are considered. The negative rates of finding and rectifying defects arise as a consequence of the need for some work to build up before work in the phase can commence. Again, the rates of finding and rectifying defects slow after initial success and become very low as each of the phases, and the projects, near completion. Note that although rates of finding and rectifying defects are shown for these phases as they near completion, concurrent work continues albeit at diminishing rates until the end of the whole project.

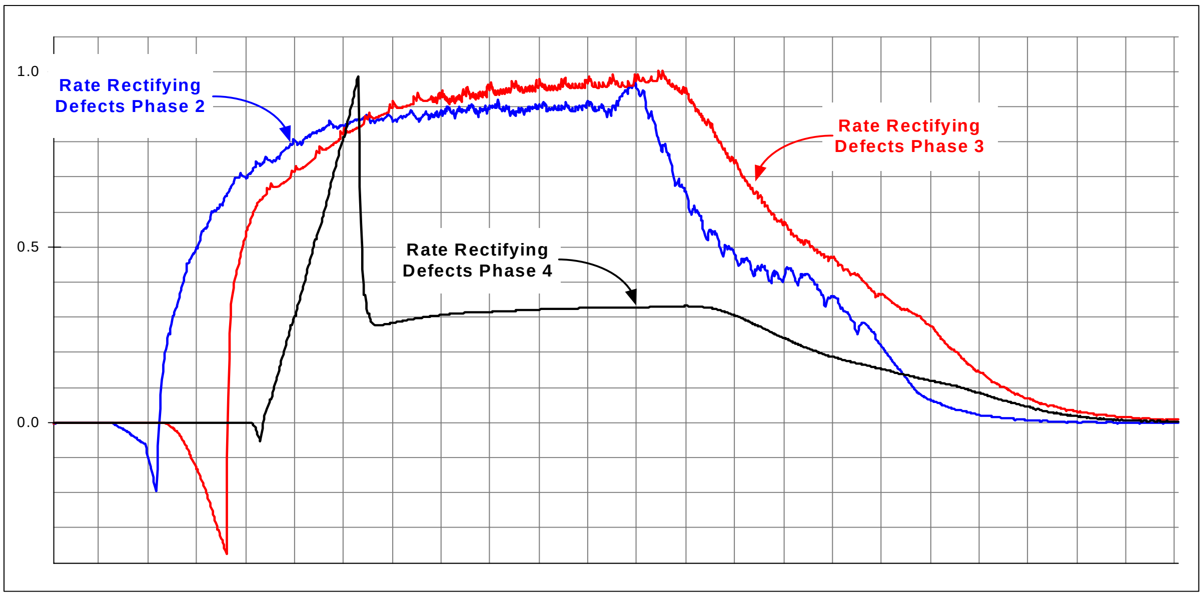

Figure 6. Rates of Finding and Rectifying Detecting Defects

Discussion of Simulated Results

A surprising number of non-linear dynamics in the observed behaviour over time appear as a consequence of:

a. the non-linear relationship between intensity of QA effort, and the impact this has on success in finding defects;
b. the non-linear feedback dynamics of the rework cycles within the phases; and
c. the dynamic sharing of resources both within phases and between phases of the project.

Greatest gains in terms of reducing time to complete accrue from being able to most effectively allocate resources from the very beginning of the project. This requires that those resources be made available when need which, in turn, can only be achieved if project managers know of this need well beforehand because the availability of resources is frequently characterized by long lead times.

There is very little, if anything, to be gained by adding resources later in the project. The effectiveness of various options for allocating resources cannot be determined by use of intuition and judgment: they can only be identified through testing of alternate scenarios in a series of simulation runs.

This simulation modeling approach can be extended to take into account the sensitivity of project delivery cost and time to complete to other factors such as programmer skill levels, time to assimilate new programming techniques, inter-phase rework, alternate inspection strategies, enhancements in project management maturity and use of CASE tools.

References:

Abdel-Hamid, T.K., 1984, 'The Dynamics of Software Development Project Management: An Integrative System Dynamics Perspective'. Doctoral Thesis, Massachusetts Institute of Technology, Cambridge, MA.

Abdel-Hamid, T.K. and Madnick, S.E., 1990, 'Software Project Dynamics: An Integrated Approach'. Prentice-Hall, Edgewood Cliffs, NJ.

Blanchard, S.B and Fabrycky, W.J., 1997, 'Systems Engineering and Analysis, Prentice-Hall.

Coyle, R.G. 1996, 'System Dynamics Modelling: A Practical Approach', Chapman and Hall, London.

Diehl, E. and Sterman, J. 1995, 'Effects of feedback complexity on dynamic decision making', In: *Organisational Behaviour and Human Decision Processes*, 62, 2.

Dörner, D. 1980, 'On the difficulties people have in dealing with complexity', in: *Simulation and Games*. 11: 87-106.

Elwell, R. 2007, 'Simultech Case Study B: Report on Software Development Pilot Project', submitted for *ZACM8325 System Dynamics of Project Organisation*, University of New South Wales, Australian Defence Force Academy, Canberra.

Ford D. N. and Sterman J. D., 1998, 'Modeling dynamic development processes, in: *System Dynamics Review*, vol. 14, no. 1: 3-36.

Forrester, J.W. 1975, 'The impact of feedback control concepts on the management sciences', in: *Collected Papers of Jay W. Forrester*, Productivity Press: 45-60.

Forrester, J.W. 1987, 'Lessons from system dynamics modeling', in: *System Dynamics Review*, vol. 3, no. 2, (Summer) 1987: 136-149.

Kleinmuntz, D. N. 1985, 'Cognitive heuristics and feedback in dynamics decision environment', in: *Management Science*, vol. 31, no. 6: 680-702.

Kleinmuntz, D. N. 1993, 'Information processing and misperceptions of the implications of feedback on dynamic decision making', in: *System Dynamics Review*, vol. 9, no. 3 (Fall 1993): 223-237.

Linard, K. and McLucas A.C. 1999, 'Addressing complexity and systemic behaviour in engineering management: A tutorial for real-life problems', in: *UICEE Proceedings of the 2nd Asia-Pacific Forum on Engineering & Technology Education*, July 1999.

McLucas, A.C. 2005, 'System Dynamics Applications: A Modular Approach to Modelling Complex World Behaviour', Argos Press, Canberra, ACT.

Nuthman, C. 1994, 'Using human judgement in system dynamics models of social systems', in: System Dynamics Review, vol. 10, no. 1 (Spring 1994): 1-27.

Richardson. G.P. 1991, 'Feedback thought in social science and systems theory', University of Pennsylvania Press, Philadelphia.

Sterman, J.D. 1989a, 'Misconceptions of feedback in dynamic decision making', in *Organisational and Human Decision Processes*, no. 43: 301-335.

Sterman, J.D. 1989b, 'Modeling managerial behavior: Misperceptions of feedback in a dynamic decision making Experiment', in: *Management Science*, vol. 35, no. 3: 321-339.

Sterman, J.D. 1989c, 'Misperceptions of feedback in dynamic decision making', in: Milling, P.M. and Zahn E.O.K. (eds), *International System Dynamics Conference: Computer-Based Management of Complex Systems*. International System Dynamics Society, Stuttgart: 21-31.

Sterman, J. D. 1994, 'Learning in and about complex systems', in: *System Dynamics Review*, vol. 10, no. 2-3, (Summer-Fall): 291-330.

Sterman, J. D. 2000, 'Business dynamics: Systems thinking and modeling for a complex world', Irwin McGraw-Hill.

Stevens, R., Brook, P., Jackson, K., and Arnold, S. 1998, ‘Systems engineering: coping with complexity’, Prentice Hall, London.

Taylor, T. and Ford, D.N, 2006, ‘Tipping Point Failure and Robustness in Single Development Projects, in: *System Dynamics Review*, vol. 22, no. 1, (Spring): 51-71.

Wolstenholme, E. F. 1990, ‘System enquiry: A system dynamics approach’, John Wiley and Sons, Chichester. UK.

Wolstenholme, E.F. and Coyle, R.G. 1983, ‘Development of system dynamics as a rigorous procedure for system description’, in: *Journal of the Operational Research Society*, vol. 34: 569-581.