UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

Towards Evolutionary and Systematic Process Modelling using Components

by

Yvonne Margaret Howard

A thesis for the degree of Doctor of Philosophy

March 2004

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

Towards Evolutionary and Systematic Process Modelling using Components

by Yvonne Margaret Howard

There is growing recognition that successful software systems evolve. Similarly, the processes that produce software must also evolve in order to free evolutionary system growth. Systems Dynamics modelling and simulation has been used to support process improvement strategies; however ad-hoc modelling methods may cause failures of understanding that lead to failures of these strategies. If we are to build better, evolvable software development processes with predictable behaviour and outcome, we need to be able to use modelling and simulation in a more systematic way.

This thesis describes an evolutionary modelling method that uses quantitative simulation to ensure close correspondence between a Systems Dynamics model and the behaviour of a software development process. Secondly, through two experiments, we show how componentisation allows us to evolve process models in a dependable way, by breaking processes down into components that are well understood, with predictable behaviour. We suggest that we will be better placed to design evolvable, flexible processes that make good use of complex strategies like distribution, concurrency and feedback if we can develop re-usable process components, with well understood and predictable behaviour in the software development domain.

TABLE OF CONTENTS

LIST OF FIGURES

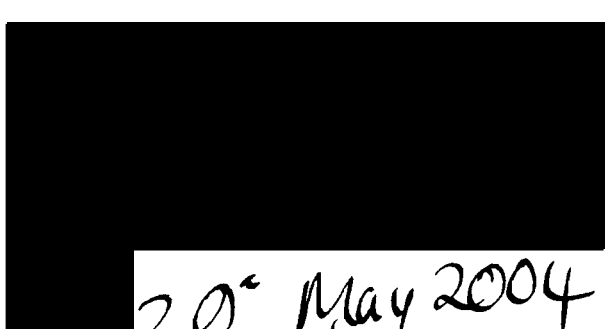
LIST OF TABLES

DECLARATION OF AUTHORSHIP

I, Yvonne Margaret Howard declare that the thesis entitled
Towards Evolutionary and Systematic Process Modelling using Components
and the work presented in it are my own. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as:

  Henderson, P. and Y. Howard (1998). "Simulating a Process Strategy for Large Scale Software Development using System Dynamics." Software Process Improvement and Practice(5): 121 - 131.

Signed: 

Date: 20th May 2004

Acknowledgements

My grateful thanks are due to the many people who have generously given their advice and encouragement to support me in this research. But in particular I would like to thank Peter Henderson, my supervisor, for his inspiration, constructive criticism and unfailing patience, and Andy Gravell for his guidance. On a personal note, I owe a debt of gratitude to my husband Tim Malone, and sons Joe and Jack for their generous and tolerant support. And to Frank and Margaret Howard, who set my expectations.

Finally, thanks are due to EPSRC grant (Engineering and Physical Sciences Research Council) for their financial support.

Yvonne Howard

Chapter 1

Introduction

Let's set a scenario familiar to many software practitioners; at the start of a project, despite resourcing it with the most able, motivated staff, it is hard to see progress. As you get nearer to your carefully planned, realistic and costed project deadline, your rate of progress towards that goal slows. We all recognise this behaviour, and anecdotally we may have many theories as to why we can observe it. Unfortunately, if we can't be sure of the causes of this behaviour, we can neither improve our prediction of project progress nor build processes that improve our performance.

So how can we improve our understanding of the behaviour of processes and have more certainty that any changes we make to our processes will have beneficial effects?

We may decide that, once we have a stable process, with all its flaws, we can treat it as a 'black box', basing predictions on past performance and adopting the view that, 'if it ain't broke, don't fix it'.

And yet processes cannot remain the same. In the same way that systems must evolve to meet new needs, process models must evolve to meet the needs of the developer and the system domain; thus systems development processes can be described as evolutionary systems. As proposed in one of Lehman's laws for software evolution [Lehman 1996], developed over thirty years study of evolution in software, user satisfaction declines unless steps are taken to evolve the product to meet new needs. We can view processes as similar evolutionary systems, and in

the case of process models, their users are developers who are building software products. The developer's satisfaction with the process model declines when the model no longer meets their need to respond to technological and cultural domain changes in order to meet market expectation.

There are many new process models that are claimed to provide an improved response to market demands, and answers to the problems of building large reliable and evolvable systems rapidly. For example, XP, Agile, Rational Unified Process, and Open Source. New processes use feedback mechanisms to evolve the product, and concurrency and distribution in order to achieve their project goals. Whether these new models will achieve their claimed results is the subject of much discussion but should also be the subject of evaluation.

Making changes to processes is expensive in time and money whilst potentially risking the operational stability of the organisation, so we need to reduce the risks and increase the benefits by improving our understanding and prediction of the effects of change.

Software developers are beginning to explore how componentisation allows them to evolve systems in a more dependable way, by breaking systems down into components that are well understood, with predictable behaviour. Similarly, process designers will be better placed to design flexible processes that make good use of complex strategies like distribution, concurrency and feedback, if we can develop re-usable process components, with well understood and predictable behaviour in the software development domain.

In the same way that systems developers have learned that using modelling and simulation tools to design systems products improves reliability and user satisfaction, process designers must learn that the processes that produce the systems deserve similar attention. The use of modelling and simulation is essential to achieving the goal of building effective, flexible, and evolvable processes.

Software processes have been shown to constitute feedback systems; [Belady and Lehman 1972; Lehman 1996] [Abdel Hamid and Madnick 1991]. Systems Dynamics [Forrester 1961] provides a methodology and tools that enable us to model and simulate dynamic processes and examine the feedback relationships that cause behaviour. Systems Dynamics has been used to successfully model individual processes, for example Adel-Hamid and Madnick's investigation of Brooks law (adding staff to a late project makes it later [Brooks 1995]) using Systems Dynamics showed that it is possible to add staff to a late project without making it 'later' and reduce the overrun provided that rules are followed.

Most Systems Dynamics modelling of software development processes has been ad hoc in nature, examining a particular instance of a process. In an ad hoc approach to modelling, we observe a dynamic behaviour, propose a plausible theory to explain it, and then attempt to replicate the behaviour using a modelling and simulation tool. If the behaviour can be replicated in the model with an understandable structure, then the plausible theory can be accepted as the cause of the real world phenomena. Unfortunately, from our subjective observations of the real world, we may add unnecessary process structure to our models that has no bearing on the real dynamics of the process and clouds our analysis of how to make process improvement decisions. This may lead us to make incorrect judgements about the real world we are examining.

Even within Abdel – Hamid and Madnick's successful model, later work has shown that the model may be over complicated and that some of the process structure, while depicting real world activity, has little or no bearing on the behaviour of the process [Houston, Mackulak and Collofello 2000].

Lehman and Ramil [Lehman and Ramil 1999] suggest that many process changes make no real or, at best, only marginal improvements to our ability to produce more reliable systems faster and more cheaply.

If we are to build better development processes with predictable behaviour and outcome, we need to be able to use modelling and simulation tools in a more systematic way.

This thesis describes how we can use Systems Dynamics to build process models from simple, well understood process components that can help us to explain, understand and predict software development process behaviour.

We describe two alternative process models that are both able to recreate the behaviour described at the beginning of the chapter; the asymptotic approach to completion of software. The process models have been created in Systems Dynamics using Vensim [Vensim 1988 - 1997] as a simulation and modelling tool, by connecting together simple process components. We show that we are able to explain the same behaviour with two very different and equally plausible theories. Were this behaviour to be modelled in an ad-hoc way, either theory may have been accepted as the cause of the real world behaviour, leading to process improvement decisions that would have marginal improvements on the process outcome.

We suggest that modelling using well understood components designed for the systems development domain would enable better process improvement and produce processes that are more evolvable.

The structure of this thesis is as follows.

Chapter 1, an introduction.

Chapter 2 provides a background for the study and covers evolution in software and processes; software quality and the software process.

Chapter 3 provides a background study of understanding process behaviour using modelling and simulation.

Chapter 4 presents a case study of ad-hoc process modelling based on the Cellular Manufacturing Process Model.

Following Chapter 4 are two chapters presenting work concerning how to move from ad hoc modelling to systematic modelling.

Chapter 5, 'Evolutionary Systems Dynamics Model Building'. In this chapter we use evolutionary model building to investigate behavioural congruence between models in different paradigms. We will use a simple process as a case study and examine the effects of resource allocation policies on the schedule and quality of the product. The simple process produces software that contains defects; it has policies that control defect removal activities that depend on the perceived quality of the software in production. The process is modelled firstly by Monte Carlo methods and secondly, in Systems Dynamics, using an evolutionary model building process.

Chapter 6, 'Simulation Experiments in Modelling Software Process using Components'. We will describe how Systems Dynamics can be applied to development processes in a systematic way. Most Systems Dynamics modelling is carried out in an ad hoc manner; a behaviour is observed and the modeller attempts to discover the feedback relationships that cause that behaviour and builds a model that reproduces that behaviour. Modelling in this way may cause inappropriate conclusions to be drawn about how best to improve the process. In this chapter we will show how abstract software development processes composed of simple, repeated components may be modelled and simulated to investigate their behaviour. Simple components with well understood behaviour that can be combined to form a process model will allow process modellers to have confidence about the causes of observed behaviour and propose process changes that will improve process outcomes.

Chapter 7, 'From Qualitative to Predictive Quantitative Models'. We discuss factors that affect our ability to build predictive quantitative systems dynamic models, using examples from the CMPM case study.

Chapter 8 provides a summary of the thesis and suggestions for future work.

Chapter 2

Background

There is significant interest within the software engineering community in improving the process of developing software in order to improve the software product itself, as well as to improve the predictability of development costs and schedule. This interest has generated research into the achievement of quality and predictability.

There have been many attempts to define software quality as we discuss later in this chapter but they coalesce around defining the quality of software products in terms of specific software attributes of interest to the user. These are external product attributes that represent the user's functional and non-functional requirements [Fenton and Pfleeger 1997]; they define the fitness for purpose of the software.

The International Standards Organisation's Single Universal Model (ISO 9126) has been developed to follow this view of quality using the definition: 'the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs'. It has six factors: functionality, reliability, efficiency, usability, maintainability, portability. In essence, perfect quality is the meeting of all requirements and the absence of defects.

Research into quality has focussed on two major areas, measurement of the product in terms of its attributes and measurement of the ability of an organisation and its processes to produce quality products. The second research direction has produced the Capability Maturity Model [Paulk, Curtis, Chrissis and Weber 1994] and

SPICE [Dorling 1993]. Explicit in the models is the recognition that an organisation must define, measure, control and finally improve its process in order to improve its capability. Progress through CMM and SPICE requires that organisations adopt an explicit development process model to make the delivery of systems more predictable.

Research into predictability in terms of effort, cost and schedule is led by research into project planning requirements. The COCOMO and COCOMO II estimation systems [Boehm, Clark, Horowitz, Westland, Madachy and Selby 1995; Boehm, Abts, Clark and Devnani-Chulan 1996] were created as a result of this research. COCOMO is based on a traditional waterfall lifecycle and a traditional development process estimating the cost of development from the number of person months a development will take based on an estimate of the size of the software in KSLOC (thousands of lines of source code).

Quality, cost and schedule are interdependent attributes (Figure 1); if you reduce the cost (effort) or time available (schedule), quality is affected; if you take steps to increase quality through quality assurance procedures or testing, cost and schedule are affected [Rus, Collofello and Lakey 1999]. Research into these dependencies within processes may prove to be valuable in providing the dynamic controls necessary for improvement.

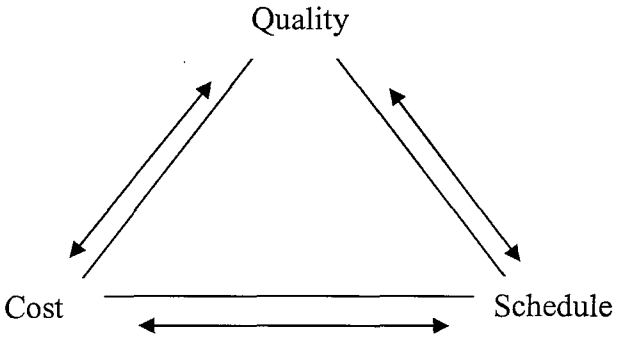


Figure 1. Cost, Quality and Schedule Interdependency

Producers of large-scale systems are aware that these systems, often involving both hardware and software delivered as one system, evolve over time but in today's fiercely competitive market place there is increasing commercial pressure to respond rapidly to new needs. In this environment, systems must evolve rapidly to achieve a fast time to market and secure competitive advantage.

Performance predictability in cost, schedule and delivered quality is critical to maintaining competitive advantage. Manufacturers use increasingly sophisticated techniques to reduce costs and improve delivery time. They make use of legacy code, have an asset base of reusable components and buy in specialist functionality. They reduce cycle time by looking for opportunities for development concurrency within the process. In order to make the development process more predictable, most organisations now undertake continual development process improvement using methods such as CMM or SPICE. Achievement of CMM or SPICE level 4, where measurement makes the development more predictable, requires the introduction of detailed process models and practical methods of process measurement [Pfleeger and Rombach 1994]. To get beyond level 4, a company must be able to use these measurements to control the process directly. Producers of large-scale software recognise that software evolves and that advanced process techniques must be used to maintain competitive responsiveness.

Process structure has been largely ignored as a determinant of process outcome when examining and modelling development processes [Ford and Sterman 1997], yet the research into CMM and SPICE shows that processes are an important determinant of product outcome, in terms of cost, schedule and quality. In the same way that products must evolve to remain successful and continue to satisfy market needs, the processes that produce software must evolve to support product evolution [Belady and Lehman 1985].

Software development processes are dynamic structures with complex interdependent feedback relationships. Understanding process behaviour is important to the success of any process control or improvement strategy. Modelling and simulation provide the means to develop, determine and validate the behaviour of new processes but the methodology chosen to model behaviour must be capable of capturing dynamic behaviour. Prediction systems that ignore complex, dynamic process structure cannot provide the predictability that manufacturers need to make competitive decisions.

2.1 Software Evolution

The evolutionary nature of software was first investigated and described in pioneering work by Belady and Lehman [Belady and Lehman 1972] more than thirty years ago, following a study of the IBM programming process. Continuing the examination of the nature and prerequisites for evolution in software to the present day, Lehman described eight laws for software evolution including a law recognising the role of feedback dynamics in the evolutionary process [Belady and Lehman 1985] [Lehman 1996].

Lehman defines and characterises two types of software: E-type systems that are used to solve problems in the real world and must be continually be fixed, adapted and extended, and S-type systems that are an executable model of a formal specification. An E-type system is judged by the continuing satisfaction of the

stakeholders in the real world whereas the criterion of acceptability of an S-type is that of mathematical correctness relative to the specification.

E-type systems evolve in an unbounded operational domain which itself evolves in response to the system. S-type systems have a bounded operational domain and so not evolve.

At this point it might be sensible to define what we mean by evolution in software and how it differs from maintenance particularly as the terms have been used interchangeably as if they were synonymous. There have been definitions of software maintenance since the 1970's and in the 1990's it was defined by two international standards; IEEE Standard 1219 defines it as:

> 'The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.'

The International Standards Organisation (ISO/IEC) definition is similar. The term evolution has been used to describe maintenance and adaptive development, but there are no standard definitions of software evolution as yet.

Lientz and Swanson in the late 1970's categorised maintenance activities into four classes [Lientz and Swanson 1980]:

- Adaptive – changes in the software environment
- Perfective – new user requirements
- Corrective – fixing errors
- Preventive – prevent problems in the future.

Bennett and Raglich [Bennett 2000] suggest that maintenance should be defined as general post-delivery activities whereas the goal of evolution is the adaptation of the system to the ever-changing user requirements and operating environment.

Thus evolution extends the idea of adaptive and perfective maintenance to be a continuous, feedback driven adaptation of software to meet stakeholders' expectation in the operational domain. We shall follow Bennett and Raglich's distinction.

E-type systems, therefore, after initial deployment with customers are continuously modified over a series of versions to meet new market needs.

In the Feast projects (Feedback Evolution and Software Technology), Lehman examines the nature of and the prerequisites for evolution in software [Lehman and Stenning 1996; Lehman and Stenning 1998]. The software studied in the project is large-scale software that has persisted over many years and meets the E-type criteria.

The Feast hypothesis is that software evolution is a feedback process and that unless the feedback dynamics are understood, major process improvement cannot be achieved [Lehman 1996; Lehman and Ramil 2000]. Lehman believes that external factors from the domain of the system limit evolution and that changes to process models have marginal effects. Process improvements that are made without understanding feedback dynamics will fail to deliver the expected benefits. The project proposes eight laws for evolution:

I	Continuing Change	E-type systems must be continually adapted else they become progressively less satisfactory.
II	Increasing Complexity	As an E type system is evolved its complexity increases unless work is done to maintain or reduce it.

III	Self Regulation	Global E-type system evolution processes are self-regulating.
IV	Conservation of Organisational Stability	Unless feedback mechanisms are appropriately adjusted, the average effective global activity rate in an evolving system tends to remain constant over the product lifetime.
V	Conservation of Familiarity	As an E-type system evolves, all associated with it must maintain mastery of its content and behaviour to achieve satisfactory usage and evolution. Excessive growth diminishes the mastery and leads to a transient reduction in growth rate or even shrinkage. Therefore the mean incremental growth remains constant or even declines.
VI	Continuing Growth	Functional content of E-type systems must be continually increased to maintain user satisfaction over its lifetime.
VII	Declining Quality	Quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII	Feedback System	E-type evolution processes are multi-level, multi-loop, and multi-agent feedback systems and must be treated as such to achieve major process improvement.

Lehman's findings are that evolution is constrained by feedback in the process. In the empirical studies carried out by Lehman, over the lifetime of the software, the rate of software growth declines. The inference is that the system loses its ability to be adapted to meet new needs; legacy, complexity and unwieldiness limit successful evolution.

Lehman proposes that there is a limit to the age and growth of software. However the early studies are of very old software (IBM OS/360) [Belady and Lehman 1985] and growth is represented by the physical size of the software. This equates growth in size with evolution and this may not be a good representation of evolution.

More recently developed software, e.g. Microsoft Windows 2000 and Netscape Navigator and Communicator have shown aggressive evolutionary growth in functionality and size. Navigator has evolved from a web browser into Communicator, a set of Internet communication tools, within 3 years and has grown in physical size from a code base of 100,000 units to 3,000,000 units. Microsoft Windows has evolved over 18 years from Windows 286 to Windows NT. The ability of Windows to continue to evolve may be based on the evolution of the process model used to develop the software [Cusumano and Yoffie 1998].

The example of Netscape shows how rapidly a software product needs to evolve in response to an expansion of the technological and cultural domain of the software and market expectation.

The characteristics of evolutionary change in software described by Lehman and the proposal that the process used to produce the software must support the requirements of software to evolve provide a basis for further investigation. Lehman stresses that an understanding of feedback is necessary to design processes that produce software that can evolve effectively; therefore any investigation should be supported by tools that allow feedback to be examined [Allen 1988].

The EPSRC funded SEBPC research programme indicates that evolution in business processes is necessary to maintain competitiveness and the ability of an organisation to change and adapt to new threats and opportunities [Henderson 2000]. Warboys also stresses the need to examine feedback [Warboys, Greenwood and Kawalek 2000].

In the case of software development business, the processes that need to evolve are the processes that produce software.

2.2 Quality

Fenton [Fenton 1996] [Fenton and Pfleeger 1997] asserts that we should define the quality of software products in terms of specific software attributes of interest to the user. These are external product attributes that represent the user's functional and non-functional requirements; they define the fitness for purpose of the software.

Returning to the ISO 9126 Single Universal Model, mentioned earlier in this chapter, it was developed to follow this view of quality using the definition: 'the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs'. It has six factors:

- Functionality
- Reliability
- Efficiency
- Usability
- Maintainability
- Portability

These external attributes are difficult to measure or compare, so the attributes are defined in terms of more concrete characteristics that are measurable. Within this model, each high level factor is composed of lower level criteria which define it, for example, reliability is defined as 'A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time'. A set of characteristics that define the attributes is suggested but not prescribed. The standard provides a framework for evaluation of quality.

The ISO model develops models proposed by Boehm [Boehm, Brown and Kaspar 1978] and McCall [McCall, Richards and Walters 1977], which are also tree structured decompositional models of the components of quality. Boehm used his model when developing the COCOMO cost estimation model; the McCall model is used to predict productivity [Fenton and Pfleeger 1997]. One of the difficulties in making quality comparisons is that the models used differ in their interpretation of the criteria that define the external attribute. For example, Boehm defines reliability as:

- Completeness
- Accuracy
- Consistency

McCall defines reliability as

- Accuracy
- Error tolerance
- Consistency
- Simplicity

The ISO model is an attempt to provide a universal standard but as the interpretation of the characteristics is not defined and is dependent on the evaluator, comparison of the quality of products is difficult.

Quality must be measurable for meaningful comparison to be made. External attributes are not directly measurable, so the quality models relate these external attributes through decomposition to internal attributes that are measurable, as shown in Figure 2.

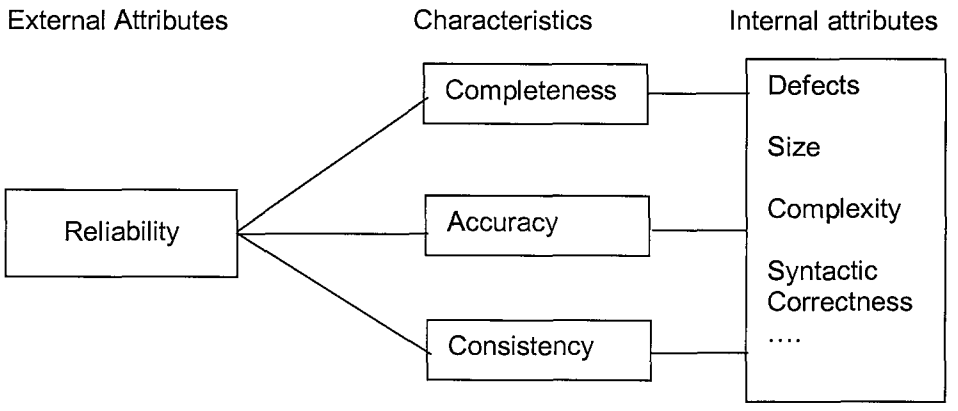


Figure 2. ISO decompositional model of the components for reliability

Whilst the user is the primary focus for consideration of quality, we should consider other stakeholders. Software producers, for example, expect to be able to reuse software components in order to reduce development cycle times. They need measures of the quality of the components that will be integrated into new and evolved software products; low quality assets may not be reusable or may cost more to integrate. Software quality is one of the measures that represents the value of their asset base. As well as reuse considerations, lower quality products will cost more to maintain and are less likely to evolve successfully. As described in Lehman's Laws for evolution, systems must be continually adapted else they become progressively less satisfactory. The observed quality of software will decline if the software doesn't evolve to meet new needs.

Producers' quality goals also include external attributes relating to process, cost, stability, timeliness and resource attributes of productivity. These attributes can similarly be represented by internal attributes. Fenton shows examples of the relationship between internal and external attributes for products, processes and resources, see Figure 3.

Entity		Attributes	
		External	Internal
product	Code	Reliability Usability Maintainability	Size, reuse, modularity, functionality, complexity ...
process	Detailed Design	Quality Cost Stability	Time, effort, number of faults found
resource	Personnel	Productivity	Experience, motivation, cost

Figure 3. Internal and external relationships for products processes and resources

The difficulty for producers is predicting the outcome of external attributes whilst the product is in the process of development. The product's internal attributes, for example, size, are often easier to measure than external attributes. The process must provide measurements of the internal product and process attributes in order to provide predictability and control. If it is difficult to predict the quality outcome of the product by measuring the product itself during development, an alternative would be to measure capability of the organisation and its processes to produce a quality product, based on the assertion that process predictability determines product predictability. Process measurement is discussed in section 7 of this chapter.

2.3 The Software Process

Humphrey [Humphrey 1990] defines the software process as,

> 'that set of actions required to efficiently transform a user's need into an effective software solution. A particular process is the set of tools, methods and practices we use to produce a software product'.

He identifies the characteristics of an effective software process as predictability; cost estimates and schedules must be met with reasonable consistency and products must meet functional and quality expectations, reflecting the interdependency shown in Figure 1

Development activities that are carried out in the software process include:

System conceptualisation	Software integration and testing
System requirements and benefits analysis	System integration and testing
System design	Installation at site
Specification of software requirements	Site testing and acceptance
Architectural design	Training and documentation
Detailed design	Implementation
Unit development	Maintenance

Process models are differentiated by the combination of tasks adopted, the development effort allocation to the tasks, the timing of activities and the feedback and control methods used to predict and control the process outcome.

Humphrey identifies three levels of software process models. The universal, U model provides a high level, abstract overview of the process model; the worldly, W model provides the working model adopted by an organisation and refines the U

model; the Atomic or A model is the instantiation of the W model for a specific project and is a refinement of the W model.

The universal models that are typically in use in software production are

- Waterfall [Royce 1970]
- Prototyping models - Spiral [Boehm and Bose 1994]
- Defect Prevention - V model [Dröschel and Wiemers 1999]

Process models developed in response to the increased demands placed by the market for fast time to market and fast growth of capability include:

- Incremental Development (synch and stabilise [Cusumano and Selby 1997], Objectory)
- Evolutionary (eXtreme Programming [Beck 1999], Open Source [Raymond 1999])
- Agile Methods (eXtreme Programming [Beck 1999])

2.4 Process Models

2.4.1 Waterfall Lifecycle

The Waterfall or Classic lifecycle (Figure 4) [Royce 1970]is the oldest attempt to define the software development process as a systematic, sequential engineering process. It is based on a systematic, sequential approach that begins with system requirements and progresses through analysis, design, coding, testing and maintenance.

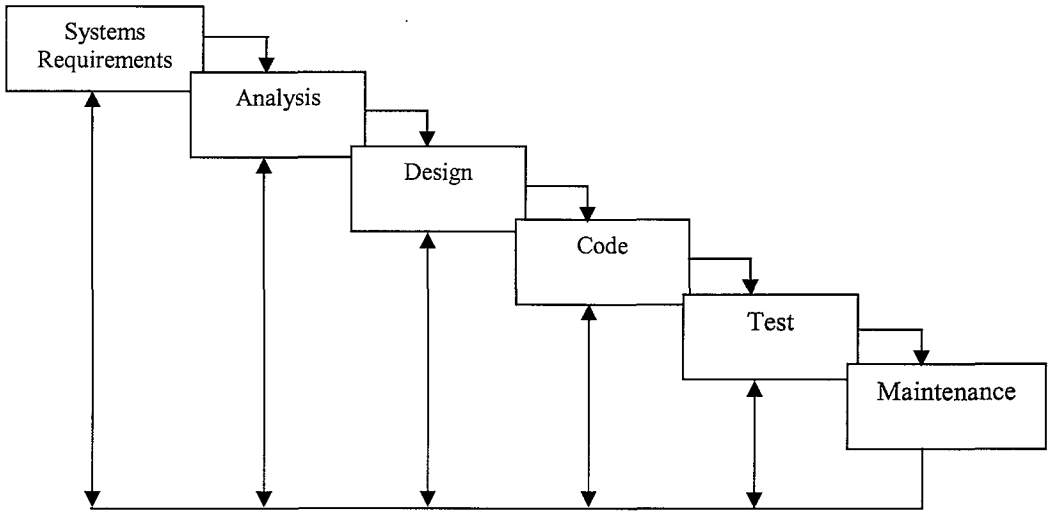


Figure 4. The Waterfall Model

The paradigm can be criticised because real projects rarely follow a sequential flow and iteration always occurs. Iteration is desirable in that it follows the change in mental model produced by understanding and learning as the project progresses. Customers evolve their understanding of their requirements as the project progresses yet under this lifecycle they are required to fix their requirements early in the lifecycle. Major failures of understanding can remain undetected because the customer sees a working version late in the cycle. The model assumes that all projects start from a 'clean sheet' whereas modern software producers use their resources of knowledge and predictable, stable software assets to shorten time to market.

Parnas in 'How and why to fake it'[Parnas and Clements 1985], proposes that there are benefits in retrofitting the model to the project as a method to document an idealised version. This historical, idealised version of the project smoothes out the iteration that actually takes place. Parnas uses the process model as a schema for documentation to aid maintenance and support legacy development. This acknowledges that the Waterfall Model does not effectively support the actual

process followed by developers or the planning, estimation and control of the process.

2.4.2 V model

The V model of development is a sequential model of development, similar to the waterfall model, that incorporates verification and validation of the product throughout the sequence of tasks, see Figure 5. The model follows Humphrey's [Humphrey 1990] definition in that it models the frameworks of tools, methods and standards that support the process as well as the sequence of activities to be followed. Validation and verification is ensured though inspection, review and testing at each stage in the development cycle. Testing is a planned, not ad hoc activity.

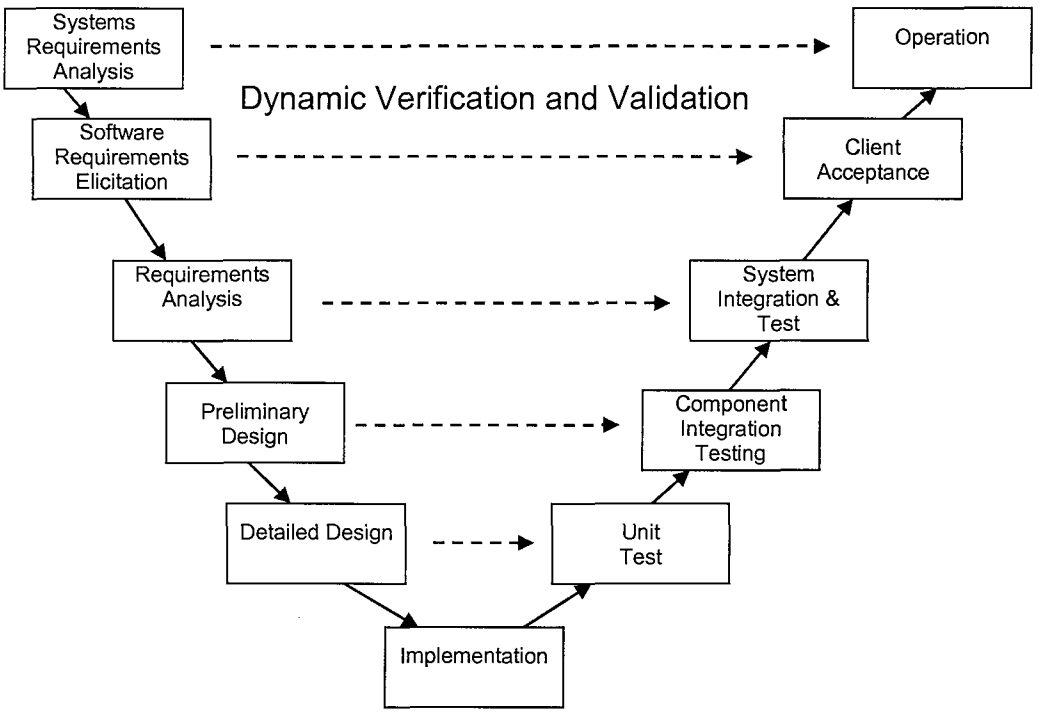


Figure 5. V model of development

The model addresses the need to ensure quality of outcome by incorporating defect removal and prevention activities in the process model; for example unit tests are an outcome of the detailed design.

2.4.3 Rapid Application Development - Spiral Lifecycle model

Rapid Application Development uses iteration of the development cycle to provide a progressively more complete version of a product. Prototyping is used as a method of eliciting requirements and feedback about each successive version and for exploring the problem domain. Prototyping as a technique was developed in response to the difficulties encountered in sequential development where requirements are fixed early and the changes that are a natural response to learning about the problem domain are difficult to feedback into the requirements. Whilst users may know many of the objectives that they wish to address with a system, they may not know all the details of the problem domain or the capabilities of the system domain. Process models that use prototyping allow for these conditions and offer a development approach that yields results without fixing requirements too early. In Rapid Application Development models, the developer builds a simplified version of the proposed system and presents it to the customer for consideration as part of the development process. The customer in turn provides feedback to the developer, who refines the system requirements to incorporate the additional information. Prototype code may be thrown away or refactored and entirely new programs developed once requirements are identified.

The Spiral model developed by Boehm [Boehm 1988] Figure 6, shows the iteration inherent in development . The spiral model formalises earlier rapid application development models characterised by prototyping and includes risk assessment of the project outcome.

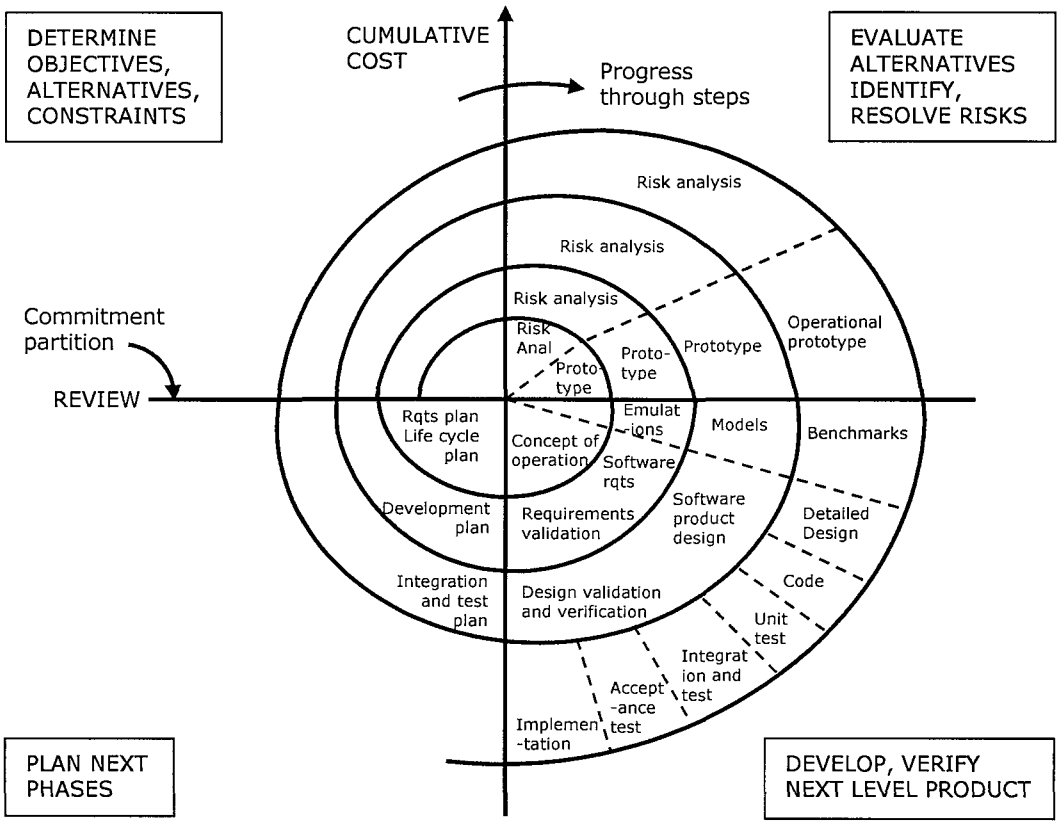


Figure 6. Spiral Model of Development

A spiral flow through four quadrants describes the process model. The quadrants are planning, risk analysis, engineering and customer evaluation. At each circuit around the spiral, increasingly more complete versions of software are built that can be evaluated by the customer; risk analysis determines whether the project should continue or whether the process should be modified or halted. For example, if an increase in cost or project completion time is identified during one phase of risk assessment, the customer or the developer may decide to limit the product features because the increased cost or lengthened timeframe may make continuation of the project with the current feature set impractical or infeasible.

Customer evaluation of each successive version of the product is used to modify the next more detailed version.

Each flow around the Spiral is made up of the following steps:

- determine objectives, alternatives, and constraints.
- assess risk, evaluate alternatives, identify and resolve risks.
- develop and verify.
- plan next phase.

At this point in the spiral the commitment to proceed is reviewed.

The model is designed to show dynamic development that is modified by the development process itself.

The Win-Win Spiral Model [Boehm and Bose 1994] described by the diagram in Figure 7, is an evolution of the original Spiral model. It incorporates a framework (Theory W) for the product stakeholders (e.g. developers, users) to negotiate mutually satisfactory sets of objectives and alternatives and constraints for each successive version.

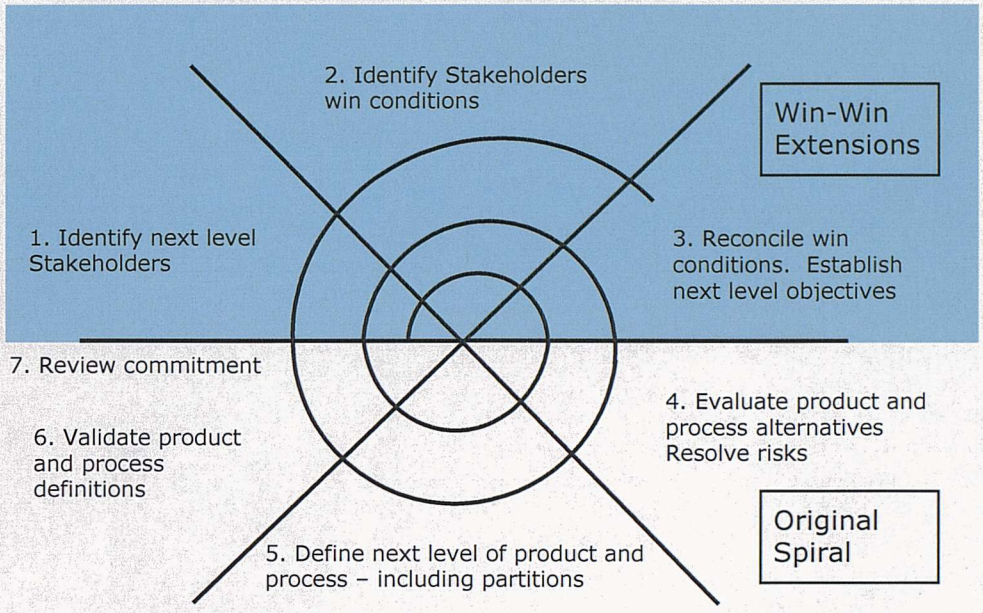


Figure 7. Spiral Model with Win-Win Extensions

2.4.4 Incremental Builds- Microsoft 'Synch and Stabilise'

Incremental models of development [McConnell 1996] have been developed in response to the need to reduce the time to market by using high levels of concurrency in the process. Development and testing are done in parallel and the specification evolves from a high-level vision statement of prioritised goals through incremental releases of successively more complete versions of the product.

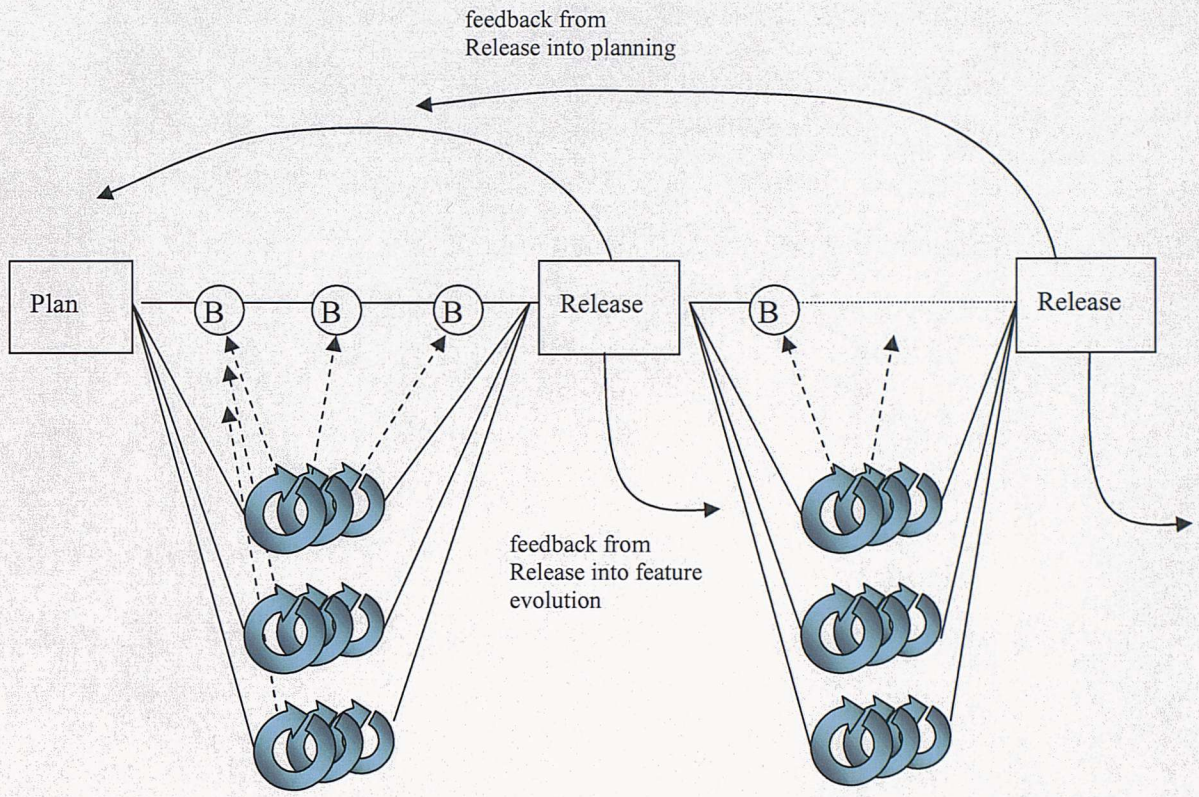


Figure 8. An Instance of a Synch and Stabilise Process

Incremental models based on 'Synch and Stabilise' [Cusumano and Selby 1997] use distribution to create concurrency, Figure 8. The product is broken down into feature sets that can be tackled by a team. The project is broken down into sequential sub projects (milestones) representing completion points for major portions of the product, each with a prioritised set of features. Feature teams complete a complete cycle of development (design, code, fix, integrate) for each milestone. Each milestone completes an increment of the product, Figure 9. Developers work in teams aligned with the components or requirements they are making, each team has the skills needed to complete the development cycle. The teams synchronise together by submitting the components they are building into a daily build of the product. The product is stabilised at each release milestone.

The team structure breaks the communication barriers that can occur between functionally aligned groups. In comparison, Waterfall process models, with sequential phasing create functional organisation structures of developers aligned with phases. A team of analysts would work on a specification, and pass the completed and signed off specification to a team of designers in the design phase, who pass off the signed off design to a team of implementers.

Developers improve their competitive advantage by reducing their time to market by planning multiple release cycles with fixed release dates. The release content is not fixed too far ahead so that the release date can be achieved irrespective of whether feature completion plans have been achieved.

Customer feedback from each successive release is used to evolve the specification for later phases.

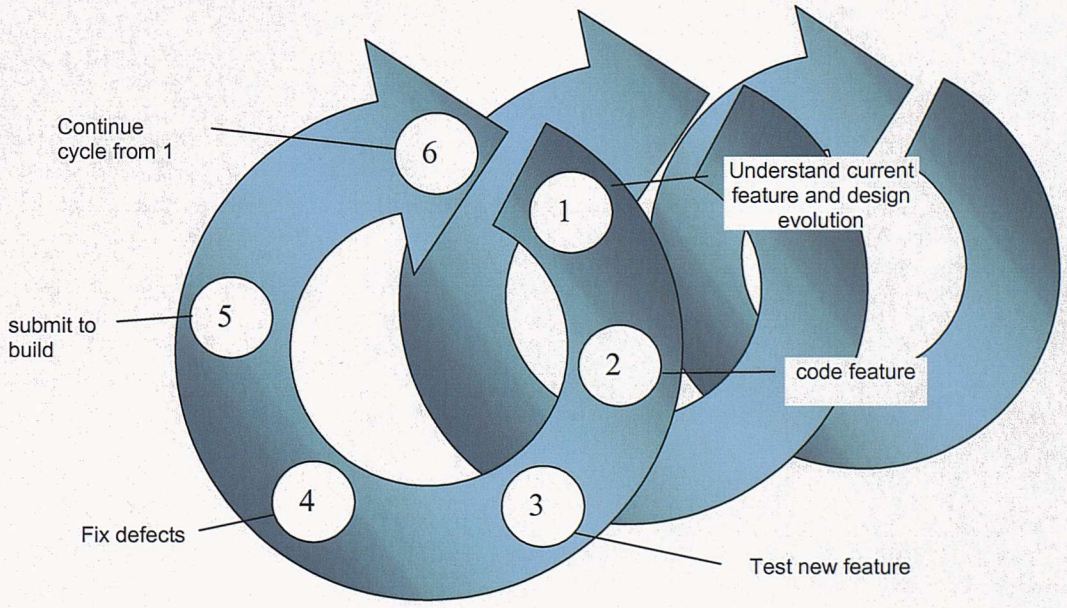


Figure 9. An Individual Cycle in Microsoft Synch and Stabilise

2.4.5 Agile Methods - eXtreme Programming

Agile Methods [Cockburn 2002] [McConnell 1996] are low procedural overhead methods that accept that software development is difficult to control. Agile methodologies emphasise values and principles in software engineering, rather than procedures and documentation. Projects minimise risk by ensuring that engineers focus on small units of work and work in close collaboration with customers. The method uses short, iterative project cycle; at the end of each cycle project priorities are re-evaluated. Agile Methods include eXtreme Programming, and DSDM (Dynamic Systems Development Method).

The most well known agile method is eXtreme Programming (XP) [Beck 1999] uses similar techniques to Synch and Stabilise and has been developed for use with small teams building software quickly in an environment of changing requirements. The model emphasises high levels of communication and feedback from the customer and the development process. The process follows these techniques.

- negotiate requirements with on-site customer domain expert, simple design using storyboards, CRC (Class Responsibility Collaboration) cards
- no design/architecture review, design is refined through the code-fix-release cycle
- design is broken down into coding work packages/components that can be completed in a day
- development using well defined and strictly adhered to tools and coding standards
- pair programming
- iterated code, test and fix cycle (writing tests before code)
- daily builds

- incremental release of frequent updates in functionality in a short cycle.

The model shows similarities with rapid application development models that use iteration and prototyping to evolve successively more complete versions of software.

2.4.6 Open Source

The Open Source process model developed and described by Torvalds [Raymond 1999] was most famously used for the development of the Linux operating system. The model has been used to produce collaborative software using the web as a community infrastructure:

- in the academic world,
- networked computer game world e.g Half-Life: Counter-Strike
- and for middleware infrastructure e.g Apache TomCat web server, Mozilla web browser

The last two communities produce successful, mature Open Source products that are able to create or replace commercially viable products [Scacchi 2002].

Scacchi reports the one Open Source community portal for collaborative development, SourceForge [OSDN 2002-2004] had more than 30,000 open source projects listed in 2001, with more than 10% indicating the availability of mature, released and actively supported software.

The source code is published and freely available. Interested users find and fix bugs and offer improvements. The owner selects fittest component from those offered, integrates it into the product and includes it in the published system. The system is a co-operative venture between the publisher and the user. The market decides how the product will evolve and evolutionary success of the product is dependent on the willingness of the market to participate in the development process. The process model is distributed and concurrent [Moon and Sproull

2000]. However, in this process model, the publisher is not concerned with the development activities that produce a component.

Open Source is probably the most innovative process model to emerge in the last ten years that challenges how complex systems can be constructed, evolved and deployed. Scacchi identifies Open source development as 'a complex web of socio-technical processes, development situations, and dynamically emerging development contexts' [Scacchi 2004].

Feedback is used implicitly and explicitly as the dynamic that evolves an Open Source product. The Open Source developer solicits and uses feedback from the developer and user community. Software evolves when the product is successful in the marketplace and interest and commitment has been stimulated from the stakeholder community.

The Open Source process model is being used successfully in developing and evolving infrastructure and network game software both in academic and commercial environments.

2.5 Comparing Process models

Classical lifecycle models such as the 'waterfall' and V-Model are sequential and static, they are non-evolutionary and assume that the product is created solely within the organisational boundary. In traditional software development environments, they aided process understanding and improvements but cannot support the more complex evolutionary processes described here or predict their behaviour.

More recently introduced lifecycle models such as Spiral [Boehm 2000] have been developed to take account of iterative dynamic development. However, Perry, Staudenmayer and Votta [Perry, Staudenmayer and Votta 1994] conducted empirical studies using diaries and direct observation in two organisations to examine the software process and in particular how developers spend their time.

They found that neither the waterfall nor the spiral models reflect what really happens. Their findings were that many iterative and evolutionary processes were performed concurrently.

Powell and Mander [Powell, Mander and Brown 1999] have described iterative development where concurrent pipelining techniques have been used to reduce cycle times. The model shows a traditional sequence of development phases from requirements through design, implementation and testing. Each Phase is iterated producing successively more complete versions of the product to be passed from each phase to the next, thereby enabling the phases to overlap.

In these cases the process model follows the traditional development sequence of phases. There remains the problem of predicting performance when work is distributed and may not all be within the organisational boundary.

Incremental 'synch and stabilise' models have been successfully used at Microsoft to allow fast evolution of large systems, and fast time to market. The process is a sequence of sub-projects each of which is a complete development cycle producing an increment of functionality of the product. Because of the iterative development of increments of functionality, there is always a system that is ready to ship. Customer and developer feedback into requirements ensures that the product can respond quickly to changing user needs and changes in the system domain needs. The high level of concurrency within sequential sub projects allows a faster time to market for large systems. The process model addresses the problems of the lack of responsiveness in the waterfall model by breaking barriers between functional groups, and providing feedback mechanisms from users and developers to evolve the product rapidly. Whilst the code-fix-build cycle is clearly defined in the model, there is no clear definition of how design activities can be as responsive as requirements. The model incorporates code and fix rather than defect prevention, which may allow design defects to proliferate. Criticisms have been made that,

unless these aspects of the process model are better defined, the process model may not scale to development of systems where high reliability is required.

Open source is a distributed process model. The system publisher selects and integrates components provided by interested users outside the organisational boundary. At component level any lifecycle model may be used. The benefits of the process model are a fast time to fix, market driven, and fitness for purpose, evolution of the whole system guaranteed through competition for survival of evolutions of components. However the publisher needs to generate sufficient market interest or users may not supply new and evolved components in which case the product will not evolve.

Competition for the most successful fix evolves components quickly but the cost of development effort is hidden because most development is unpaid. The resources of many programmers are used to find and develop fixes for any one successful fix. The cost of thrown away work, (unsuccessful evolutions) is not measured or measurable.

The emphasis in the model is on a code and fix cycle of evolution; requirements and design are not formalised. The product design evolves through competition for successful components. Failures of design are cast aside through competition and the cost of design failures is not measured. Thus upstream design defects are ignored in favour of down stream code fixing which takes more development effort.

The process model does not define how the product architecture is developed, in Open Source developments described so far (Linux), the architecture was well defined and stable before the source was opened.

The successful application of the model is likely to be determined by the nature of the software to be developed for example, operating system software and infrastructure. If this model is to be used in vertical markets, then developers will

need to use a different model for their revenue stream, possibly based on selling product related services rather than the products themselves.

2.6 Evolution in process models.

In the same way that systems must evolve to meet new needs, process models must evolve to meet the needs of the developer and the domain system. As suggested by Lehman's work on software evolution, user satisfaction declines unless steps are taken to evolve the product to meet new needs. In the case of process models, the users are developers who are building software products. Their satisfaction with the process model declines when the model no longer meets the needs of the developer to respond to changes in the technological and cultural domain to meet market expectation.

Process improvement models like Humphrey's [Humphrey 1990] use feedback from the current model to evolve new models. Feedback from evaluation of world W-models and changes in the system domain leads to evolutionary changes in universal U-models. Evolution in process models can be observed by examining how process models have developed over time.

The Waterfall Lifecycle was the first attempt to define a process model; it sequences and defines development activities.

The V model is an evolution of the Waterfall Lifecycle that includes validation and verification, but is still sequential. This evolution was in response to developer and user needs to improve the quality of software products.

The Spiral model is an evolution of rapid application development, incorporating a formal framework of risk analysis. In turn, the Win-Win Spiral model is an evolution of the Spiral model that includes formalised negotiation of requirements. This evolution to include prototyping and requirements feedback was in response to dissatisfaction with the Waterfall Model early requirements fixing and late

realisation leading to undetected failures of understanding between developers and users.

Incremental models are a response to the need to be able to react to market expectations by developing large systems rapidly. To achieve rapid growth in software products, developers need concurrency in development activities, and feedback relationships to allow requirements evolution. Incremental models use build and release iteration for requirements evolution and concurrent development cycles to reduce the time to complete each release. This shows evolution from both Spiral models and Waterfall models. Synch and Stabilise iterates concurrent waterfall cycles in a sequence of subprojects. eXtreme iterates concurrent rapid application models.

Open Source has evolved to provide a model for component-wise distributed development across organisational and geographical boundaries. The product reacts rapidly to market needs and expectations of fast defect correction. There appears to be evolutionary interdependence between Open Source systems and their user-developer communities so that they co-evolve.

2.7 Process Predictability and Control

A process is said to be under statistical control if its future performance is predictable within established statistical limits. When it is under statistical control, it should be repeatable with similar results. Deming [Deming 1982] states that measurement is the basis for statistical process control. He applied these techniques to manufacturing industries. Humphrey [Humphrey 1990] asserts that these techniques are applicable to the management of software development processes and they have been incorporated into the Capability Maturity Model [Paulk, Curtis et al. 1994]. Following the Representational Theory for Measurement [Fenton and Pfleeger 1997], numbers must properly represent the process being controlled and must be sufficiently well defined and verified to

provide a reliable basis for action. Measures of the product and process are required for predictability of product quality, cost and schedule.

2.7.1 Measuring Software Products

According to the Representational Theory of Measurement [Fenton and Pfleeger 1997], data obtained as measurements must properly represent attributes of the observed entities; measurement must be consistent and manipulation of data should preserve the relationship observed between entities. The theory provides rules for consistency in measurement and provides a basis for interpreting data.

The definition of measurement using the representational theory provided by Fenton is:

'Measurement is a mapping from the empirical world to the formal relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute.'

The theory can be applied to product and process quality measurement in order to find concrete internal attributes that truly represent the external attributes. Internal attributes are more measurable during the process so their measurements are easier to use, to assess and to control progress towards target outcomes.

Fenton defines internal attributes 'as those that can be measured purely in terms of the product, process or resource on its own, separate from its behaviour'.

Commonly measured internal attributes are size, modularity, redundancy, reuse, syntactic correctness, structuredness and defect density.

Size is one of the most important measurements used for predicting cost and schedule, for example in the COCOMO and COCOMO II estimation models. If we examine size, Fenton shows that it is not one-dimensional and that it has at least three dimensions, length (physical size), functionality and complexity (of the underlying problem solved by the software).

One aspect, length, is the most commonly measured in Lines of Code (LOC) or KLOC (thousand lines of code) yet there are many definitions of a LOC. They can be blank, comments or data declarations; the counted size of the product depends on the counting method used and the definition of a LOC. Jones [Jones 1986] reported differences of five times in the counted size of a product depending on the definition used.

When we consider comparisons of length in LOCs between two products, one composed of components and one developed from scratch, can a comparison of the effort required to produce the products be made? Comparisons can be spurious, because methods of counting differ. Measurements of size seem uncomplicated but there are difficulties in achieving comparable, consistent and meaningful results.

The internal product attribute typically used as a measure and predictor of quality during the process is defect density. If defect density is to be used as a predictor, we must be certain that the representational theory of measurement holds, and that it represents the reliability of the product (mean time between failure). However, Adams at IBM found that the density of defects in a product did not necessarily indicate its rate of failure [Adams 1984]. Neither are comparisons of defect densities reliable if the measurement of size is not consistent.

Narrow measures of internal attributes are often used because they seem easier to count and collect; there is a tendency to measure only the readily measurable attributes rather than the full representation of the external attribute.

2.7.2 Measuring Process

As well as measuring the quality of their products, producers need to be able to measure how successful they are at producing software efficiently. Producers need to be able to predict:

- How much effort it will take to produce a software product,

- How long the process will take to complete,
- The productivity of the process.

As discussed in section 3, these are process and resource attributes. Process, resource and product attributes are similarly measured in terms of their internal attributes. There are similar difficulties in achieving precise and comparable measurements, for example, manpower costs may include only those working directly on the product, or they may include administrative staff.

Effort required to complete a process is determined by the amount of work that is required to be completed. The time that the process will take to complete is determined also by the amount of work to be completed. One definition of the work to be completed is the size of the product. Size in terms of LOC is a concrete product measure but it is only available when the product is complete. In order to predict effort and time, size must first be predicted. This shows the interdependence of product and process measures.

2.7.3 Cost and Schedule Estimation COCOMO and COCOMOII

In recognition that not all development follows traditional process paradigms, COCOMO II [Boehm, Abts et al. 1996] has been developed to take account of prototyping life cycles (spiral), reuse and COTs style development (integrations of Commercial Off-The-Shelf components through exposed interfaces). Whilst COTS style development is recognised within COCOMO II, only components integrated as part of a tool-bed or an infrastructure are modelled. It does not address COTS integration development where the delivered system is an integration of components.

COCOMO II is designed to be applied at three stages in the target lifecycles, at project start up, early design and post architecture, with the intention of providing increasingly more accurate costs as more information becomes available about the delivered system. The model is intended to be applicable to a wide range of

organisations and products and incorporates a set of parameters that must be set up to tailor the model to a specific organisation and process. Twenty four cost driver parameters are required to set up the model for a two stage estimation process, covering product factors (required reliability, complexity, reusability, documentation), platform factors (execution time, storage constraints, volatility), personnel factors (experience and continuity), project factors (tools, schedule and inter site communication). This requires a sophisticated knowledge of the software to be developed, the organisation, resources and process. For example, to calibrate PCON, personnel continuity, (high is desirable) the producer first must assess their project's annual personnel turnover, APT using data from previous projects and other evidence, and use it to derive their PCON rating as shown in Figure 10. A low value APT is desirable and converts to a high value PCON.

PCON	Very Low	Low	Nominal	High	Very High	Extra High
APT	48%/year	24%/year	12%/year	6%/year	3%/year	

Figure 10. Calibration parameters for Personnel Continuity

The definition of the parameters and calibration of the model requires considerable effort and it is very difficult to calibrate the model for an organisation. However the cost drivers show the areas that affect productivity.

Project costs are estimated from person months required to produce software of size S. The fundamental relationship modelled is:

$$PM = A \times (Size)^{B} \times \prod_{i=1}^{x} (EM_i)$$

Where:

PM is person months adjusted by cost drivers,

A is a constant used to capture the multiplicative effects on effort with projects of increasing size,

S is size in KSLOC, adjusted by a breakage factor, of the delivered system

B is a scale factor used to account for economies or dis-economies of scale

EM is a set of cost drivers.

The breakage factor represents the amount of code developed but not delivered in the system because of discarded or changed requirements and due to rework through error. It therefore represents the extra work required to achieve the desired quality in the delivered system.

COCOMO II is interesting because it identifies that new process models require new techniques for estimating project costs and schedules; it identifies that the size of the delivered source code does not represent all of the work done to make the product (the breakage factor); and that the behaviour of the project process affects the estimate (cost drivers). The research also indicates that more effort needs to be expended when the requirements for reliability are high.

2.7.4 COCOTs

The research into COCOTs [Boehm 1997] by USC began in recognition that COCOMO II did not address delivered systems composed of components. The researchers found that although many people within the software industry are talking about such systems as yet there is little empirical or theoretical research in this area.

Their early research is based on lexical maps of surveys of risk factors identified by software professionals in the Risk Repository of the Software Engineering Institute at Carnegie-Mellon. COCOTs researchers have used this information to suggest cost drivers for a cost estimation model analogous to COCOMO II.

The cost model calculates the effort in person months required to make the planned product. The researchers recognised that the size of the delivered product in thousands of source lines of code used in COCOMO and COCOMO II was an inappropriate measure of the amount of development work done by developers

because it also included those lines of code developed by the component suppliers. Henderson describes the work of integrating developers as 'glueing' [Chatters, Henderson and Rostron 1998] components together. The cost or person-months is related to the amount or 'size' of the glue code. In COCOMO II an alternative method of calculating KSLOC is by counting function points and then converting them to KSLOC using a conversion table. The function point method of sizing the glue code, without conversion to KSLOC, has been used in COCOTs.

Following the model of COCOMO II, in the COCOTs model version 1, person months are calculated from the size of the code adjusted by linear and non-linear scaling constants, rework and a set of cost drivers (effort multipliers).

$$ESIZE = UFP \times (1.0 + BRAK/100)$$

$$PM = A \times (ESIZE)^{B} \times \prod_{i=1}^{13} (EM_i)$$

Where:

ESIZE is effective size of developed glue code

UFP unadjusted function points

BRAK rework percentage

A linear scaling constant

B non-linear scaling constant

EM 13 effort multipliers or cost drivers

PM is the estimated effort in person-months for the COTs integration task

The 13 cost drivers include COTs specific factors identified through the Carnegie-Mellon risk assessment; supplier maturity and performance, component reliability, integrator experience. Interestingly many of the project, personnel and product factors considered important in COCOMO II do not appear in the COCOTs model, for example, personnel continuity, process maturity, team cohesion, yet these

factors exist in an organisation whether using component integration methods or more traditional development processes. The researchers assert that the 13 drivers are the most significant factors affecting cost prediction for COTs style development. Organisations may find that some COTs integration projects will include some self built components; in this case they will need one cost estimation model, COCOMO II, for the self built component and another for the COTs part of the project, the difficulty will then be of holding sufficient information to set up and calibrate the COCOMO II parameters and the COCOTs parameters.

Estimation models provide a snapshot prediction of the costs of developing a product however they do not capture how the project performs against the prediction until the project is complete. The method of estimating the size of the product means that it is not easy to assess progress during the project. Achievement of KSLOC against target is not visible during all phases of the project (requirements, design, testing). Without this information the costs cannot be controlled.

Estimation systems do not provide control of the dynamic system even if they are applied at more than one point in the development (as is suggested for COCOMO II). Cost and schedule metrics can indicate exceptions to the planned costs and schedule but estimation tools do not provide the insight necessary for the dynamic control of the process

2.8 Process Improvement

Humphrey [Humphrey 1990] identifies process improvement as a key management task to reduce costs, make more effective use of resources and improve the organisation's ability to prosper. One goal of process improvement therefore is to improve predictability in schedule, cost and quality of the products of the process. Another goal is improvement of the quality of the products of the process.

Humphrey defines a continuous cycle of process improvement with six repeated actions, Figure 11. At each iteration of the cycle, the starting point for improvement is understanding the current status of the process. The completion of each iteration is starting the next iteration. The cycle provides evolutionary

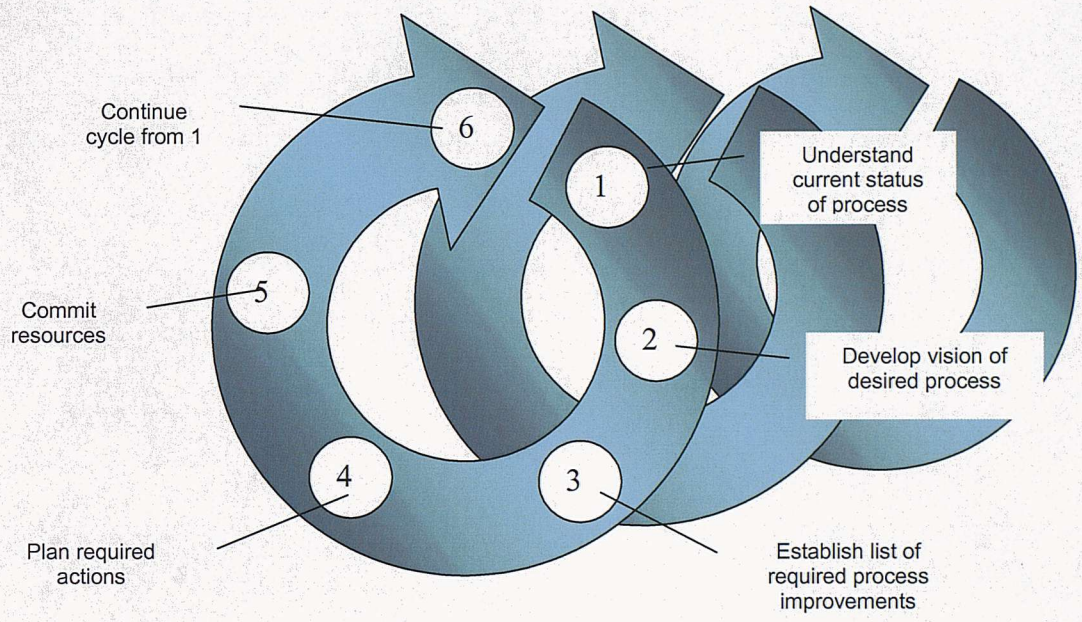


Figure 11. Humphrey's Process Improvement Cycle

progress for the process.

The Software Engineering Institute's Capability Maturity Model, CMM, [Paulk, Curtis et al. 1994] and SPICE [Dorling 1993] assess the ability of an organisation and its processes to produce quality products, Figure 12. Explicit in the models is the recognition that an organisation must define, measure, control and finally improve its process to improve its capability. CMM and SPICE provide a framework for organisations progressively to improve quality through the adoption of quantitatively controlled process models. The models follow the improvement cycle proposed by Humphrey; each progression from one level to the next is an iteration of the cycle where the improvement priorities are made obvious.

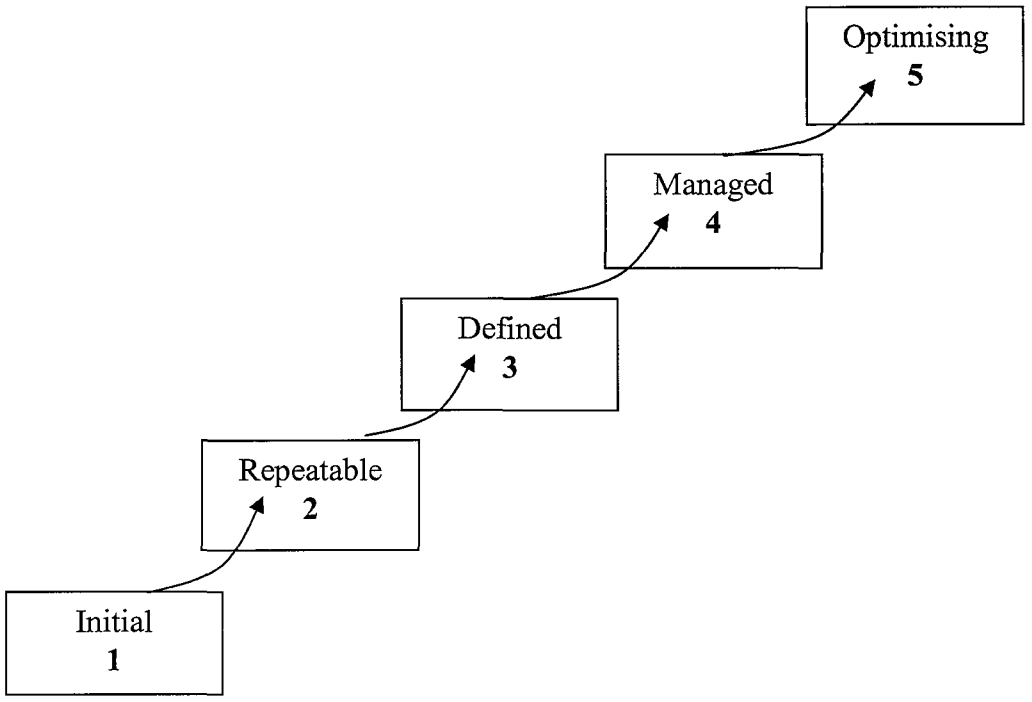


Figure 12. Capability Maturity Model

CMM define five levels of maturity of an organisation, where level 1 is the lowest and level 5 the highest. SPICE has 6 levels, the top 4 are comparable to CMM but Spice further differentiates the lowest level into 0 and 1. The levels were chosen to represent the actual phases of evolutionary improvement of software organisations and the difference between each level represents a measure of improvement reasonable to achieve from a prior level [Paulk, Curtis et al. 1994].

The following description refers to the CMM model.

Level 1. At level 1, an organisation has no defined process and is chaotic.

Level 2. At Level 2, an organisation has a stable, repeatable process, although will not have defined the processes or have any insight into the behaviour of the process. The process is a 'black box'.

Level 3. At level 3 the organisation has insight into the process and has defined it.

Level 4. At level 4 the organisation is able to manage the defined process by using quantitative control information. The organisation aims to operate the process within quantitative performance limits by taking measurements of process performance, analysing them and adjusting the process to maintain performance within limits.

Level 5. Organisations have detailed validated models of their processes. They can make major changes to their processes and be able to predict the effect on performance of those changes with a high degree of confidence.

The cost estimation models such as COCOMO and COCOTs support planning key process areas but do not provide the dynamic information to match performance against plans that is necessary for control of the dynamic process fully to achieve level 4 and level 5 in the maturity model.

2.9 Summary

In this chapter we have provided a background study for this thesis in which we examine evolutionary behaviour of both software and the processes that produce it.

We described some of the technological and commercial dynamics involved in producing successful products in the marketplace. Performance predictability in cost, schedule and delivered quality is critical to maintaining competitive advantage.

We presented an evaluation of software quality and showed that quality is not a static measure or characteristic but depends on the stakeholders' perception. In fact, quality declines unless active steps are taken to evolve the software to meet stakeholders' evolving expectations.

We described software process models, from the earliest sequential model, the Waterfall, proposed by Royce [Royce 1970] through to process models that employ distribution, concurrency and iteration. Each new process model was

proposed to improve product quality, reduce time to market and costs, either all or some of these. Alongside process model developments have been innovations in methods and tools to measure or predict the achievement of project goals. We show that software processes are key to producing and evolving high quality software products. Thus software process is an important determinant of product outcome, in terms of cost, schedule and quality.

Grady Booch's interpretation of the central COCOMO equation shows that the performance of a software project (the effort required to complete a product) depends on the complexity of the software being produced, the process, the team and tools used, where size is a determinant of complexity [Booch 2004] .

$$\text{Project Performance} = (\text{complexity})^{(\text{process})} * (\text{Team}) * (\text{Tools})$$

In this interpretation, it becomes clear that an effective process damps down complexity, a poor process exponentially increases the effects of complexity.

The overview of software processes suggests that processes evolve too, and are also multi-level, multi-loop feedback systems. In the same way that products must evolve to remain successful and continue to satisfy market needs, the processes that produce software must evolve to support product evolution. Processes that recognise and support feedback are more likely to produce evolvable software

This chapter underpins the need for better understanding of process behaviour In Chapter 3 we show that this understanding can be achieved through modelling and simulation and the same techniques can be used to support process improvement, providing predictability. Unless we understand the evolutionary nature of software products and processes, and use process improvement techniques that are supported by the modeling and simulation of dynamic behaviour, then product

quality improvements, and cost and schedule predictability through process improvements, are unlikely to be achieved.

Chapter 3

Understanding Process Behaviour using Modelling and Simulation

Simon proposed the 'principle of bounded rationality' in which he described the difficulty that we have in making accurate mental models of processes and therefore making accurate predictions of outcomes.

> 'The capacity of the human mind for formulating and solving complex problems is very small compared with the size of the problem whose solution is required for objectively rational behavior in the real world or even for a reasonable approximation to such objective rationality' [Simon 1996]

Human judgement is bounded by limitations of the mental models we create and limited further by our failures in interpreting the models through lack of attention, information processing capability, biased perspectives and unchallenged accepted 'truths'. As process models become more complex, mental analysis is unable to cope with the complex interactions that take place that determine the outcome [Sterman 1989]. The Waterfall model is one of the simplest process models adopted by software development organisations yet predictions of cost and duration based on it are notoriously inaccurate because interactions due to iteration are not captured in the mental model.

Formal computer models can help to overcome the limitations of mental models because they are able to inter-relate many factors simultaneously, make assumptions explicit and open to reasoning, reveal causes of behaviour and, most importantly, they can be simulated to allow experiments to investigate outcomes and behaviour without risk to the real environment [Sterman 1989].

However, this doesn't mean that formal modelling can solve all of the problems of understanding process, for the following reasons:

- The model may not represent the real world
- Correlation is mistaken for causal behaviour
- Models can become so complex that they become a copy of the real world and therefore understanding and reasoning about the model become as difficult as understanding the real world
- Models are only as good as the expert knowledge captured within them. The data should include 'soft' data (team experience, management resourcing policies etc.)
- Models can be difficult for non-modelling practitioners to understand
- The modelling technique may not model dynamic behaviour and feedback in the system

These weaknesses can be overcome by choosing modelling methodology and tools carefully to support the purposes of modelling and if the modelling methodology is applied using process modelling guidelines.

The modelling techniques chosen should:

- explicitly model feedback relationships
- represent both hard and soft data

- have a graphical representation that helps non-practitioners to understand the models
- be simulatable to aid model validation and understanding.

Process modelling guidelines suggest how potential weaknesses in models can be overcome.

Modellers can avoid over-complex models by carefully defining the objective of making the model and then defining a perspective from which to abstract from the real world to examine behaviour patterns rather than single events. By using these principles, the modeller can abstract from detail that is not the focus of the model.

The problems of lack of representation and incorrect structure can be overcome if modellers validate their models against real world behaviour.

Validation tests include checking that the model is able to replicate past behaviour in the real world, testing the model's assumptions, the correspondence of the model structure to the system and the robustness of the models behaviour. The modeller needs to use all sources of information including interviews, direct observation and historical and experimental data to capture the structure of a system to ensure that soft data is not overlooked.

Simulation provides an important tool for handling the complexity of dynamic feedback systems [Abdel Hamid and Madnick 1991].

> "The behaviour of systems of interconnected feedback loops often confounds common intuition and analysis, even though the dynamic implications of isolated loops may be reasonably obvious. The feedback structures of real problems are often so complex that the behaviour they generate over time can usually be traced only by simulation ." [Richardson and Pugh 1981]

Abdel Hamid and Madnick assert that not only does simulation make possible more complex models and models of more complex systems but also provides a means of experimentation.

Simulation models make possible controlled experiments that solve the problem that "isolation of the effect and the evaluation of the impact of any given practice within a large complex and dynamic project environment can be exceedingly difficult" [Glass 1982]

Zelkowitz and Wallace describe simulation as one of the types of controlled method experimentation that is useful in the software engineering domain.

> "We can evaluate a technology by executing the product using a model of the real environment. We hypothesise , or predict, how the real environment will react to the new technology. ... a simulation is often easier, faster and less expensive to run than the full product in the real environment." [Zelkowitz and Wallace 1997].

Simulation therefore, provides a means of revealing and understanding the behaviour of complex dynamic models and also of making controlled experiments on the model to examine the effects of different behavioural policies.

3.1 Using Modelling and Simulation to Support Process Improvement

Effective use of modelling and simulation can be seen if we examine how these techniques can be used to support the process improvement cycle described by Humphrey [Humphrey 1990].

Adopting or changing a process model is a significant undertaking for any organisation and is a risk to the success of the organisation.

Weiss [Weiss and Basili 1985] wrote that,

> '...in software engineering it is remarkably easy to propose hypotheses and remarkably difficult to test them. Accordingly, it is useful to seek methods for testing software engineering hypotheses.'

The feasibility of adopting a process model needs to be underpinned by evidence that the new process will achieve the aims of the organisation for improvement. Evidence of the suitability of a process can be gained from case studies, but for a novel process model this may not be available and for an established model, it may be difficult to find case studies in organisations with an analogous profile. In both cases there may be insufficient evidence to predict the performance of a process model within the organisation and support a case for its implementation.

Risk can be reduced if the new process is modelled and then simulated to provide predictions of the process behaviour. Decisions about whether to adopt a new process can be made with greater confidence about the outcome. Creating a model of the process and then simulating its behaviour over time allows us to understand and predict process behaviour 'off-line'. The models can take into account the specific profile of the organisation. This enables us to validate the expected benefits of the new process before implementation in the organisation.

Simulation and modelling support the six stage process improvement cycle suggested by Humphrey and improve the predictability of improvement outcome by adding two new stages and improving the effectiveness of the others, Figure 13.

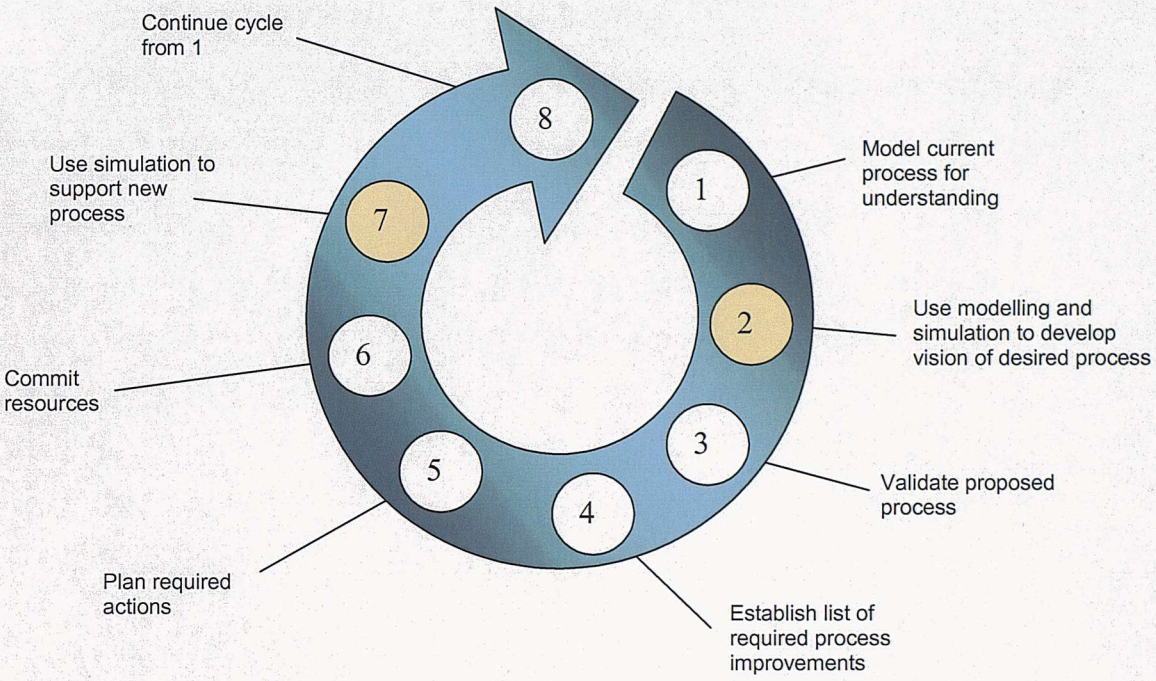


Figure 13. Humphrey's process improvement cycle extended with modelling and simulation

- Understand existing process – Model existing process, simulate to ensure that model represents real world (and understanding is complete)
- Develop vision of desired process - Model new process
- New stage, validate new process - simulate new process to evaluate effects
- Prioritise improvement actions - simulation shows effects of improvement actions
- Plan process changes - use simulation to examine effects of plans
- Commit resources - use simulation to work out staffing requirements
- New stage, support new process - use models and simulations to train users in the new process; use tools derived from the models to control the process.

- Continue cycle from 1

Modelling and simulation can produce tailored tools for the organisation to manage the live process to predict and improve performance.

The use of modelling and simulation in the improvement cycle can be seen by examining where it can be applied in the progression through the Capability Maturity Model. In the Capability Maturity Model, an explicit development process model is fundamental for achieving higher levels of maturity and is a requirement for level 3. It can be argued that capability improvements through process definition can be enhanced if simulation techniques are used [Christie 1998], because the effects of complex interactions are revealed.

Simulation can be useful in improving capability in these respects:

- Using modelling tools for process definition, the process can be validated before introduction (*level 3*).

- Through the use of process simulation in training, the model can be embedded throughout the organisation (institutionalised, in CMM terms) *(level 3*).

- Simulation in project management and control, using metrics from the process, allows prediction of possible outcomes for a range of control decisions, making explicit complex feedback relationships that are difficult to reason about (*level 4*). This enables managers to balance targets in quality, cost and schedule.

- Dynamically changing the process, possibly through technology insertion, can be evaluated and the effects on quality cost and schedule targets predicted before implementation (*level 5*).

In these respects, simulation supports process improvement in the Capability Maturity Model.

3.2 Process Simulation and Modelling Methods

When simulation and modelling has been applied to software processes, the two techniques that have been most commonly used are Discrete Event (or State – based) and Systems Dynamics.

Both combine graphical representations of the process with a simulation capability based on a mathematical expression of the relationships between the model entities.

Knowledge based systems have been developed [Scacchi 1999] but the models produced have not been satisfactory in terms of encouraging a shared understanding of the behaviour of the modelled system or promoting process improvement or redesign because model representations and animation are not sufficiently well designed to aid shared understanding. Shared understanding, particularly where there are complex relationships, is aided by tools with a graphical representation and interface.

Wolfgang Kreutzer [Kreutzer 1986] describes that in discrete – event simulation, models are viewed as structured collections of objects bound into webs of relations and transformations. Time is advanced as an event occurs; event driven models assume that nothing relevant happens between successive state transitions. Discrete event modelling is useful for tracking individual entities within a process and finding deadlock and livelock. The effects of process decisions on individual entities and interacting entities can be examined.

Systems Dynamics is the application of feedback control systems principles and techniques to managerial, organisational and socio-economic problems. Interdependent flows of objects and conditions are modelled and simulated continuously.

Kellner [Kellner, Madachy and Raffo 1999] suggests that the decision to use one technique in preference to another should be based on the following criteria:

- continuous, for strategic analyses
- discrete for scheduling

Kellner points out that process modelling situations are not so clear-cut and that in fact discrete processes may have sub processes that are continuous e.g. human-resourcing and continuous models have to compromise to describe interruptions, queues and delays.

There are tools available which allow both techniques. Their ability to do this is questionable if you consider the way time is calculated and advanced in the two techniques. Discrete event modelling advances time with each event. Systems Dynamics solves partial difference equations that underlie the model at time intervals (dt); time is therefore advanced at dt interval.

Systems Dynamics is a suitable methodology to examine the strategic behaviour of new software development processes and investigate how the structure of the process affects its behaviour and the outcomes. Discrete event modelling is suited to examining how an individual entity will interact with others and behave within the process. Within this thesis, processes will be examined at a strategic level, therefore the thesis will concentrate on modelling and simulation using Systems Dynamics.

3.3 Systems Dynamics, Systems Thinking

'The purpose of modelling is insight not numbers'

states Kreutzer in System Simulation Programming Styles and Languages [Kreutzer 1986].

Systems Dynamics was first developed by Jay Forrester in 1961 [Forrester 1961] and further developed by, amongst others, Barry Richmond [Richmond 1990], Peter Senge [Senge 1990], Geoff Coyle [Coyle 1996] and John Sterman [Sterman 1989].

Systems Dynamics is described as 'the art and science of making reliable inferences about behaviour by developing an increasingly deep understanding of the underlying structure'[Richmond 1990]. Systems Dynamics is a methodology for capturing, modelling and simulating processes. Relationships within a system or process cause the dynamics it exhibits (feedback) but intuitive judgements are unreliable about how systems change over time. Simulating the process over time shows the effects of complex relationships.

Systems Dynamics abstracts from individual entities and discontinuities. Instead of examining the behaviour of each single entity, Systems Dynamics considers how accumulations of entities behave. These are described as stocks or levels, for example within software development, completed code can be considered to be a stock. Stocks can also be intangible, for example the experience of a team of developers can be described as a stock.

Stocks are increased or depleted by activities, described as flows. In the example given, Figure 14, coding is the flow that increases the stock of completed code.

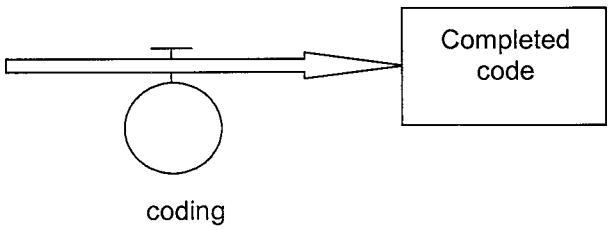


Figure 14. Coding increases the stock of completed code

The stock of experience is accumulated by a learning flow. If there is a stock of something, then it must have been generated by an activity or flow; conversely if there is an activity then, associated with it, something will be either increased or depleted.

Stocks can also be resources in the process. These can either be consumable resources that are depleted by a flow, or producing resources that generate flows but are not consumed in the process.

The difference can be shown if we look at two examples, the first illustrates a consumable resource, Figure 15; a stock of defects is depleted or consumed by a flow of detecting and fixing the defects. Physical stocks obey conservation laws.

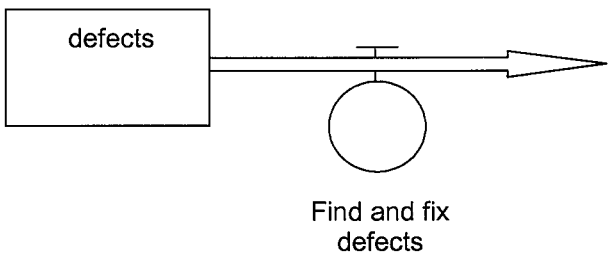


Figure 15. Finding and fixing defects reduces the stock of defects

The experience process illustrates a producer stock, Figure 16; the stock of completed code generates a flow of learning about the software being produced. This increases the stock of experience, but the stock of completed code is not reduced by the learning flow.

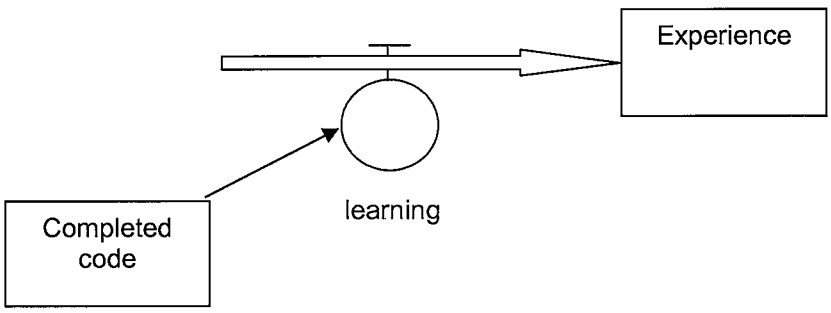


Figure 16. Increasing stock of experience as code increases

Systems Dynamics explicitly models feedback relationships within processes. These are circular loops of cause and effect that run from stocks to flows and back to the stock. Stocks give rise to flows of activities and flows change stocks or conditions. The effect of feedback is to generate goal seeking behaviour with respect to some target so that when deviations from targets occur, feedback relationships inspire, and then trigger corrective actions to bring the condition back into line. When there is more than one feedback loop in an interdependent process, the goal-seeking behaviour of each loop may conflict, as activities designed to bring one condition to meet its target simultaneously knock another condition out of line. These effects are very difficult to predict through mental analysis. Sterman reports that in controlled experiments people are shown to misperceive the effects of feedback structure in even small, simple processes. The strength of Systems Dynamics modelling is that the effects of goal seeking and goal conflict in complex webs of relationships can be exposed and accurately determined.

A simple feedback relationship can be illustrated in the experience example, Figure 17. As the stock of experience increases, there is reinforcing feedback that affects the productivity of developer and increases the rate at which code is completed.

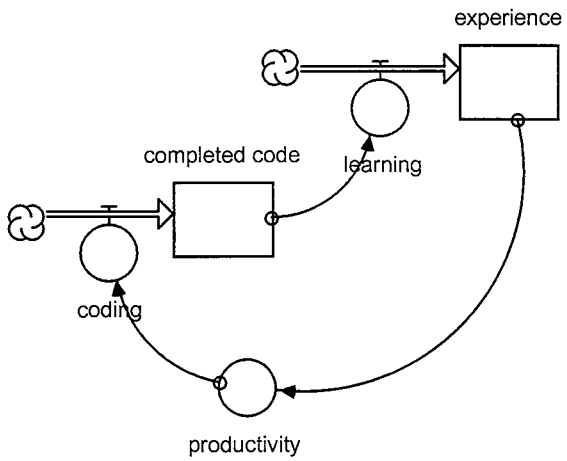


Figure 17. Systems Dynamics model showing feedback relationship between experience and productivity

We can extend this simple model to show how feedback relationships interact to create conflicting goal seeking behaviour as shown in Figure 18. The developers with increased experience are more marketable and may be inclined to leave to improve their salary. This would deplete the workforce within the organisation and reduce the rate at which code can be completed.

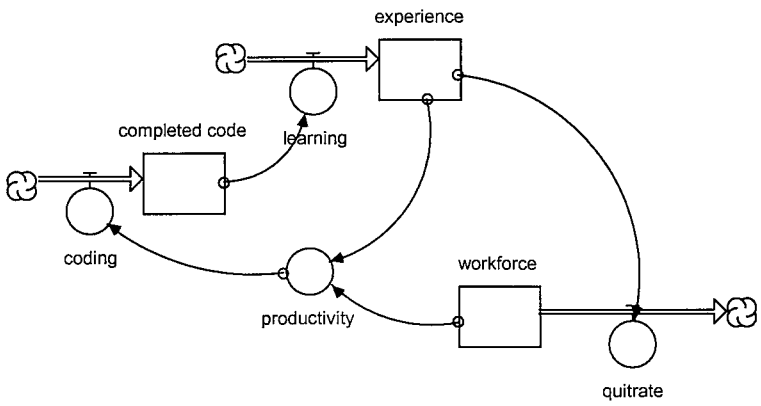


Figure 18. Systems Dynamic model showing the effects of goal conflicts in feedback relationships

We can show how Systems Dynamics models are simulated over time using mathematical algorithms. A set of discrete calculations is used to approximate to the idealised curve. The software divides the time axis into equally spaced intervals each with a width of delta time (dt). Calculations are performed at discrete intervals of delta time.

Equations behind the model diagrams define the calculations. These are Finite Difference Equations that are initialised for a simulation and then iterated for each delta time.

The two calculation methods most commonly used in Systems Dynamics are:

- Euler's method which is a simple linear extrapolation which is fast and suitable for most models but it is less suitable for examining a process on the edge of instability because it will always overshoot a turning point on a curve.

- Runge Kutta uses higher order differentiation which makes it more suitable for examining the trajectory of a changing process on the edge of instability. The number of calculations increase and therefore the time to perform simulations is greater.

Systems Dynamics has been used successfully to model systems and processes in many domains:

- Socio-economic systems including population growth dynamics, effects of economic policies [Sterman 1985]
- Business strategy and policy assessment [Forrester 1980]. The U.S. Department of Energy has produced detailed forecasts and policy analyses of domestic and international system using Systems Dynamics
- Biological systems – insulin process in the human body [Stella 1990 - 1998], population growth cycles.
- Development Processes for example, in software, construction, shipbuilding, electronics [Ford and Sterman 1997].

Rodrigues and Bower [Rodrigues and Bowers 1996; Ford and Sterman 1997] have identified three domains in development processes that have been addressed using Systems Dynamics; monitoring and control, rework and human resources [Abdel Hamid and Madnick 1991; Rodrigues and Bowers 1996], [Madachy 1996]. Sterman identifies that a fourth domain, the structure of the process itself is a key determinant of the behaviour of the process and its predictability.

3.4 Modelling Software Development using Systems Dynamics

In 'Software Project Dynamics – an integrated approach' Abdel-Hamid and Madnick [Abdel Hamid and Madnick 1991] use Systems Dynamic models to understand the process of managing software development, and the problems of

controlling cost, quality and schedule. This work is one of the most comprehensive studies of software projects using Systems Dynamics and has been the basis of work by other researchers in the field. Their aim was to gain insight into how the management of software development maybe improved to reduce cost overruns, late deliveries and user dissatisfaction. They stress that without an improved understanding of the process, real improvements are unlikely to be made. They particularly wanted to explore how typical critical management problems can be understood and how the outcome of alternative decisions can be predicted, for example:

> 'If a project is behind schedule, what are the implications of increasing the workforce, or changing the completion date? How may Brooks Law [Brooks 1995] (adding people to a late project makes it later) be explained?'

What are the reasons for and implications of the differences among potential productivity, actual productivity, and perceived productivity?

The focus of their work is management of software development projects, therefore rather than examining components of development processes; they abstract management activities, human resource management, controlling and planning and show how these are interrelated with software production. These activities form the subsystems of an integrated Systems Dynamics model. Abdel-Hamid and Madnick chose this integrated approach to

> 'prompt and facilitate the search for the multiple and potentially diffused set of factors interacting to cause software development problems'.

They give an example that the schedule overshoot problem can arise not only because of schedule underestimation but also because of management's hiring and firing policies.

The figure below, Figure 19, shows the structure of the model and the flows that connect the subsystems.

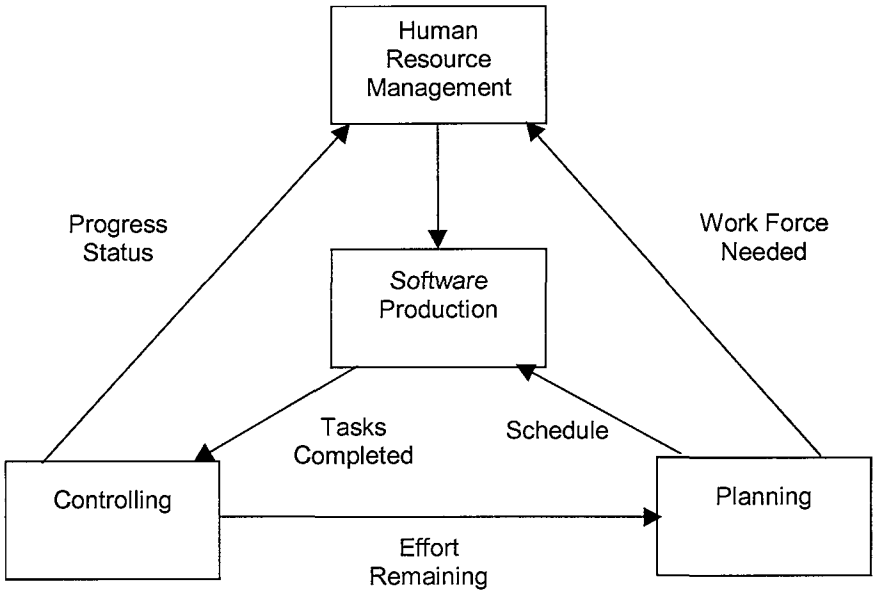


Figure 19. Overview of Abdel-Hamid and Madnick's model showing subsystems

Abdel-Hamid and Madnick's work sees the software development process as continuous. Individual objects are not identified. They represent the process as a sequence of stages through which requirements flow (coding, testing, quality assurance and rework) transformed at each stage by a process following the waterfall lifecycle model. In order to simplify their model, they set a model boundary that commences with design and coding and finishes with testing. They have made assumptions that requirements are fully defined and stable and that all defects are found and removed by the end of the development process. Their reasons were that they wanted to exclude from the model behaviour that was not caused by the developers and managers within the project. Their investigation therefore focuses on cost and schedule overruns that are a result of the policies, decisions and actions of the project team in spite of stable requirements. It should be noted that the cost estimation model that they have used is COCOMO, which also excludes requirements definition. These assumptions restrict the quality assurance methodology to defect detection through review and testing.

Their sources of empirical information were DEC, SofTech, and MITRE, all producing medium size (64KLOC) systems and extensive systems dynamics models were developed and tested against case studies at NASA to confirm the model's behaviour.

Having confirmed that the model was able to replicate behaviour observed in case studies, Abdel-Hamid and Madnick carried out extensive simulation experiments to explore the effects on cost and schedule of different planning, QA and resourcing policies and the decisions and actions based on them.

When carrying out simulation experiments on typical quality assurance policies, where manpower is allocated to QA as a percentage of development man-days, they discovered that there are diminishing returns in QA effort. QA expenditures that were either significantly lower or significantly higher than the optimal range increased the total cost of the project. Too low expenditure increased the time spent on testing and too high expenditure consumed cost without detecting more errors. Significantly, they realised the importance of emphasising quality during the software development project.

Their simulation experiments on manpower policies in relation to explaining Brooks Law, show that adding more people to a late project always makes it more costly but does not always cause it to be completed later. They were able to show that the interaction between adding additional manpower, increasing training and communication requirements caused a reduction in average productivity and an increase in man day requirements and therefore cost. Whether the project will be further delayed depends on whether the drop in productivity is so severe that the additional person's net contribution is negative.

They show that project estimates and resulting schedule affect the outcome project itself.

'A different schedule creates a different project'

Abdel-Hamid and Madnick touch on, without making explicit, recognition that the tools and process used are structural in determining outcomes, as Sterman found.

Abdel-Hamid and Madnick's work shows one of the problems with Systems Dynamics; models can rapidly become very large and complicated. The model may be criticised for its level of detail in that it has very many converters (variables). However, they deliberately chose this level of detail because they were concerned that simpler models might leave out vital aspects of the real world environment that would misguide project managers making decisions based on predictions from the model.

Wernick and Lehman [Wernick and Lehman 1998], [Lehman and Stenning] in their work on the evolutionary nature of software, believe that Systems Dynamics models must be abstracted as far as possible in order to retain understanding and insight. Systems Dynamics is a useful tool for revealing the dependencies that cause behaviour, a model that is at too high a level may not reveal sufficient information to show the basis for complex behaviour or enable understanding of strategic issues whilst models that are too complex can too closely attempt to model the real world rather than model the behaviour of the real world.

This shows the importance of choosing the correct focus and level of abstraction for the purpose of the model. Kreutzer [Kreutzer 1986] uses the example of Occam's Razor, a good model is the simplest one that can still be justified.

Madachy [Madachy 1996] has used Abdel-Hamid and Madnick's [Abdel Hamid and Madnick 1991] models as a basis to develop models that investigate the behaviour of inspection based processes. Lehman [Lehman and Stenning] used Systems Dynamics to produce the models that underpin his work on the laws for software evolution discussed in chapter 2. Mander and Powell [Powell, Mander et al. 1999] have used Systems Dynamics models to investigate the behaviour of a highly iterative, concurrent, pipelined process model. Sterman [Ford and Sterman 1997] has shown that Systems Dynamics can be used to investigate and explain the

behaviour of development projects in many domains, and Allen [Allen 1988] has shown how Systems Dynamics can be used to investigate evolving systems.

The work done by Abdel-Hamid and Madnick, Madachy, and Lehman in the domain of software development shows that simulated Systems Dynamics models can replicate the behaviour of software development projects within the chosen focus. Each has chosen a different focus for their work, Abdel-Hamid chose to investigate the effect of project management policies, Madachy investigated the effects inspection based process improvement and Lehman chose to examine product evolution. Furthermore, they show that simulation experiments can be performed on the models that will explain and predict how the real world will behave.

Chapter 4

The Cellular Manufacturing Process Model

The Cellular Manufacturing Process Model, CMPM, [Chatters, Henderson et al. 1998] has been proposed by Peter Henderson to support the evolutionary nature of modern development, concurrency and distribution and to take account of reused and bought in components. Henderson based the model on Watts Humphrey's network models of software development [Humphrey 1990] and on the value chain model for competitive advantage developed by Michael Porter [Porter 1985]. In proposing a new process model we must be sure that we can predict its behaviour, support process design and planning and show the benefits in improved control and management.

In this chapter, we will describe CMPM and how it may be applied in large-scale development projects. We examine the issues that affect the ability of cells to achieve their output targets, showing how simulation with Systems Dynamics is giving us insight into the behaviour of CMPM [Henderson and Howard 1998].

CMPM, the Cellular Manufacturing Process Model is an advanced process strategy based on components, which uses concurrency and distribution to reduce cycle times. In CMPM, networks of semi-autonomous producing cells co-operate to produce a complex large-scale system. The model views development as a manufacturing activity where large scale systems are built from components, which may be a mixture of self built components, re-used components from the producers own asset base and from bought in components. Viewing large-scale

software development as a manufacturing activity may be considered to be contentious when software development traditionally has been seen as essentially a creative or an engineering activity. This model seeks to show how large scale software producers use the techniques of manufacturing assembly of available components with known behaviour and properties in order to satisfy the market cycle.

The model is hierarchical in that any component may itself be a product of other components. Components are stable sub-assemblies of an evolving product; each sub-assembly can evolve independently. Our conjecture is that a process model based on integrating systems from components, which are separately evolving sub-assemblies, will enable evolutionary growth beyond what is possible in the monolithic systems, described by Lehman [Belady and Lehman 1985]

Software producers need predictability in cost, quality and schedule when competitive advantage demands a short time to market. Predicting the cost, quality and schedule outcome of CMPM depends upon the behaviour within the cell (intra cell) and the co-operative behaviour between the cells (inter cell) in a dynamic environment.

In a cellular manufacturing environment each cell produces a series of versions of a component each of which meets the requirements imposed on it by its customers or by some internal or external decision process. A cell will have a role both as supplier to other cells in the process network and also as customer for components from other cells.

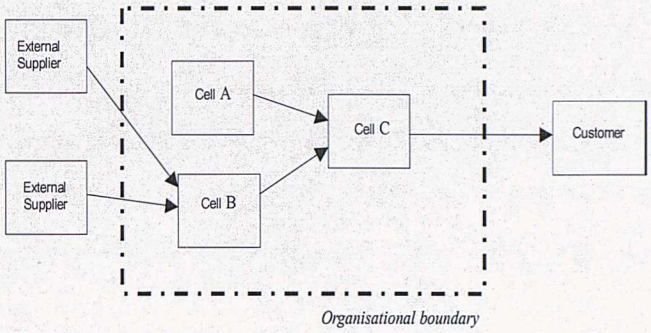


Figure 20. Supplier and Customer Relationships in CMPM

Figure 20 shows three supplier-customer relationships, an external supplier provides components for Cell B, Cell A is an internal supplier to Cell C and Cell C is also supplier to the Customer. There is another possible relationship where the organisation collaborates with another supplier to supply the customer.

Each cell has responsibility for producing their product (which may itself be a component) from components supplied by other cells within the organisation (internal suppliers), external suppliers or made within the cell, (Figure 20). The work of the cell is to design their product by selecting and sourcing components, build the product by gluing the components together using component interfaces, and to remove defects to ensure that it meets output quality targets.

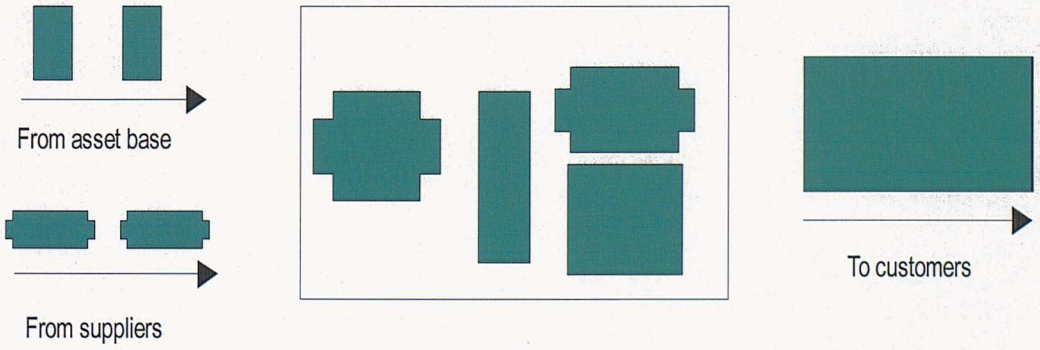


Figure 21. A cell integrating components in CMPM

The cell has goals that are dependent on the goals of its suppliers and customers. In order to meet an obligation to a customer it will have to meet imposed targets for quality, cost and schedule but will be dependent on the targets achieved by its own suppliers. Predicting the achievement of these targets for each cell is important for predicting the performance of 'downstream' cells and of the whole network.

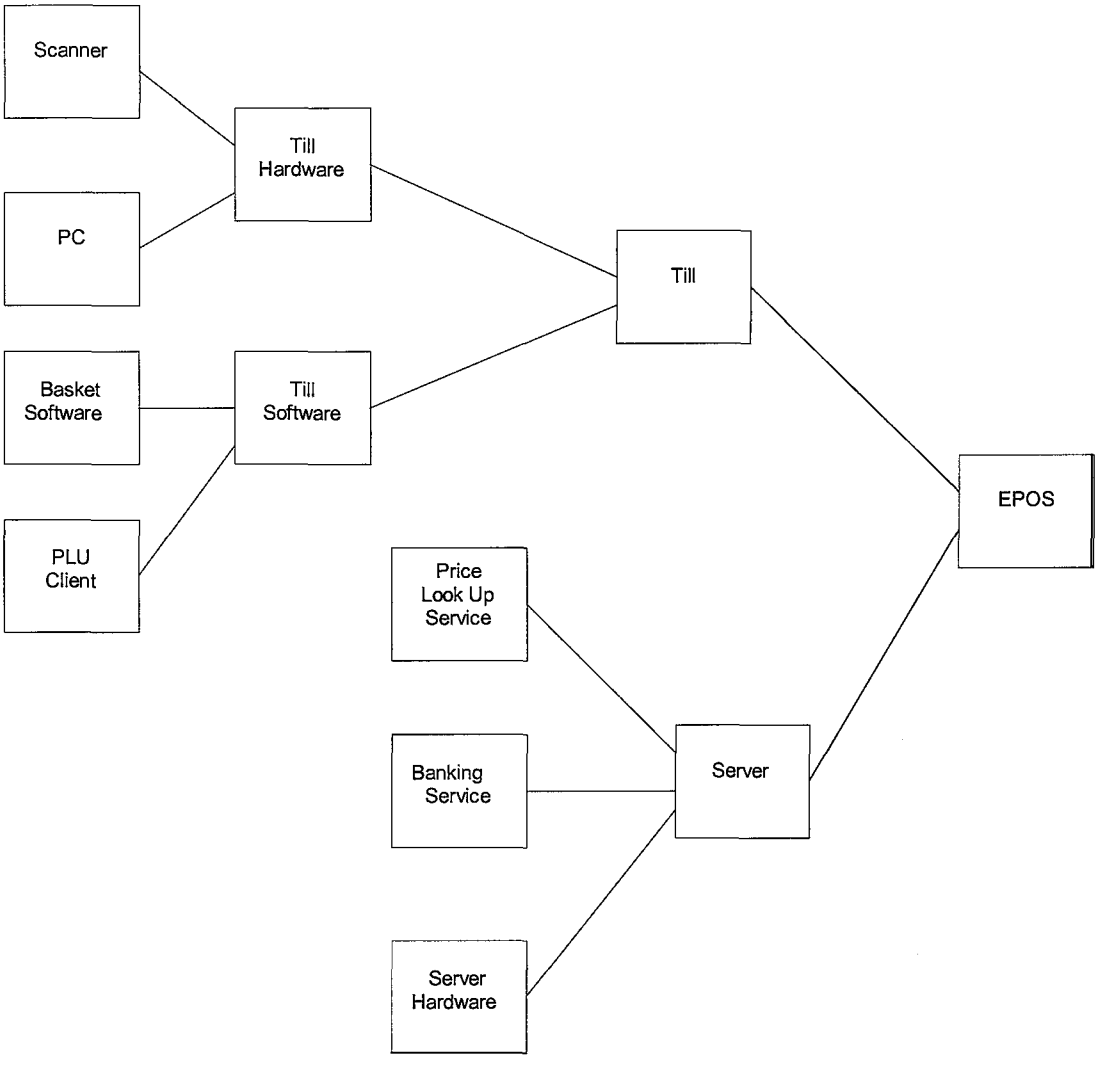


Figure 22. Electronic Point of Sale System component structure

Here we consider a simplified Electronic Point of Sale System, comprising a network of tills using a set of services, as shown in Figure 22. When customers present their purchases for payment, each item in their basket is scanned electronically and is transferred to a virtual basket held in the till. The till uses remote services to lookup the price and description of each item. The till calculates the value of the 'basket' and uses the banking service to control the financial transaction with the customer. Figure 22 shows the simplified component structure of such a system. In this example there are hardware and software components to be integrated into the delivered system and the system includes client and server components.

Consider the Till client component, which is made up of a hardware component comprising a scanner and PC based till, and a software component comprising a number of components including a 'basket' component that stores the items and calculates the value of each shopping transaction. The Server component comprises hardware components and software components that include banking and price look up (PLU). The complete EPOS system comprises the Till hardware and software and the Server hardware and software. We can evolve the system to include a loyalty card scheme, creating a new loyalty card component for the till and a component for the loyalty card service for the server.

4.1 EPOS CMPM – mapping hierarchy to process

The Cellular Manufacturing Process Model for this system maps directly onto the product breakdown structure to give a network of manufacturing cells, Figure 22. This work breakdown structure is likely to survive many generations of product. The stability of the structure is caused by semi-autonomous cells independently evolving their own product.

Each cell is responsible for one level of integration. Each cell receives components from suppliers, builds some components locally, glues the components into an integrated product (which is tested to output standards) and shipped to a customer

or to the next level of system integration. If we again consider the Till component, in this case the architects have chosen to source the scanner hardware from an external supplier. The Till Hardware cell integrates their own components with the externally supplied scanner. Concurrently the software cells are manufacturing their component, gluing together components they have built themselves and components from internal and external suppliers. The Till Integration cell (labelled Till in Figure 22) can start to work integrating the till components as soon as they are available but in order to collapse time-scales, the cells can choose to start integration with an earlier version of any component from the repository. Thus if the software components are available before the new version of the scanner is delivered, the cell may use an older version recognising that when the new version is available some of the integration work may need to be redone, trading effort for schedule completion.

Should the PLU Client cell produce a component with an output quality lower than its target, the Till Software cell will be faced with a choice of working around the low quality component and consequently increasing the effort they must spend on gluing the component in, or reverting to an earlier version. In both cases the output quality of the Till component delivered to the Till Integration Cell will fall short of its target.

The challenge facing planners is to allocate effort over the process network. Too much effort allocated to a cell creates cost overruns, too little creates either schedule delays or quality problems for downstream cells. With an early prediction of upstream problems, planners can mitigate the effects and avoid an increasing 'bow-wave' of delays or quality shortfall.

4.2 Intra Cell Behaviour

The behaviour within a cell can be as formalised or as ad hoc as the product demands. The network is not dependent on detailed knowledge of how each cell performs its integration task, only how it meets its external obligations. However,

in general, the behaviour within a cell will be a pumping action over time in that versions of products (components to the customer) will flow out, meeting an ever-evolving customer requirement. Figure 23 shows one elementary possibility for the behaviour of the cell. Component flow is shown by solid arrows, requirement flow and other control flows are shown by the broken arrows. Rectangles show repositories and ellipses show activities.

The behaviour is as follows. From customers (or elsewhere) the cell receives external requirements for evolutions of the product for which this cell is responsible. The cell is a team of architects, designers and engineers who are knowledgeable about the components and previous builds the cell maintains in its asset base. They develop build plans and requirements for new or changed components. Some of these requirements will be internal requirements, which determine new or revised components to be built by the team working in the cell. Other requirements will be passed back to a supplier. Either way, the new and revised components are assumed eventually to arrive in the repository. At some stage, all of the necessary components will be available to undertake integration and test activities leading to the delivery to the customer of a version of the product.

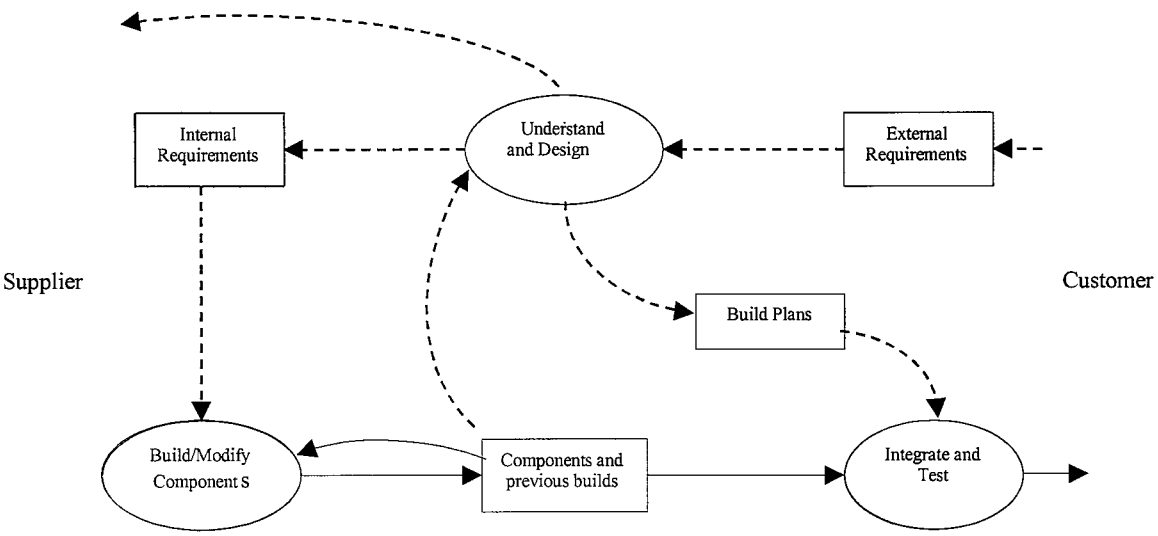


Figure 23. The pumping behaviour of a cell

4.2.1 Relationship between Work and Quality

If we examine a simple cell, corresponding to the internal behaviour of the pumping model fig 4, Work (effort), W is applied to complete the component of size S and to remove defects (detection and rework). Q is the input quality of a component and P is the output quality of the component, informally estimated on a scale where 1=perfect and 0=useless.

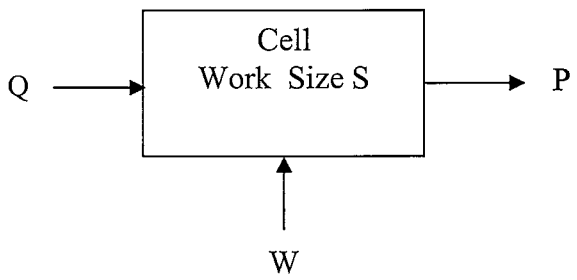


Figure 24. Work allocation in a CMPM cell

If Q is low or the target quality is very high we must put in more effort. We implicitly understand that perfection is unattainable and that as we approach it the return on investment in effort diminishes [Chatters, Henderson et al. 1998].

Early theoretical predictions [Chatters, Henderson and Rostron 1998] of the behaviour of quality and work using simple EXCEL models based on the COCOMO relationships between Work and Size and naïve reliability models, have produced a family of curves all asymptotic approaching P=1 as shown in Figure 25.

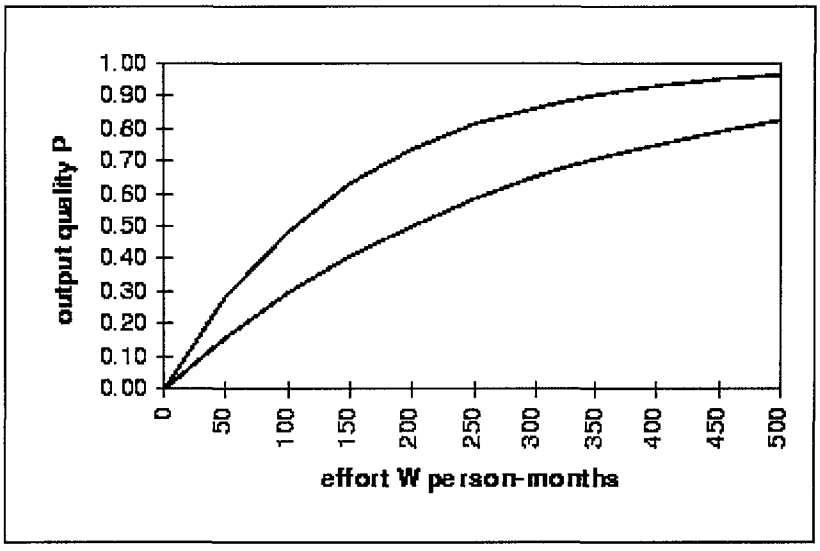


Figure 25. Predicted growth of quality

As we expend effort, our quality eventually reaches about 95% in the upper line, but the last 10% takes something like 35% of the effort. The steepness of the curve is greater if the quality of the components being integrated is higher. The upper line shows the process integrating from components with an average input quality of 95%. The lower line shows the same process where the input quality averages

50%. With the same effort expended as in the first case, we only reach 85% output quality.

The internal effort allocation W1, W2,...,WN depends on management and policy decisions, for example whether there is an inspection-based process [Madachy 1996]. We expect that a range of policy decisions would produce behaviour within the family of asymptotic graphs.

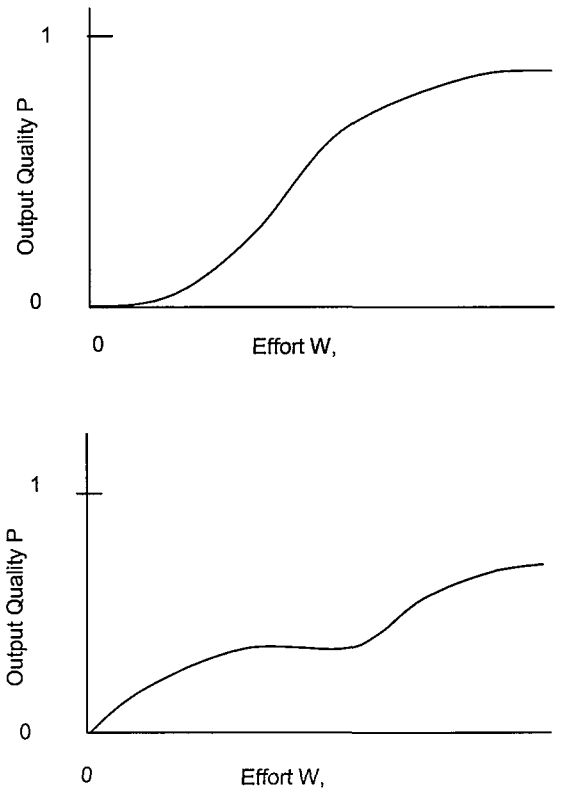


Figure 26. Graph shapes from industrial partner data

In fact, data from our industrial partners is beginning to show different behaviours, Figure 26, shows two graph shapes that have emerged. The data was collected over the period of the CMPM investigation and includes both a historical data and data captured after CMPM metrics had been established [ICL 1999].

Our Systems Dynamics models reproduce the first shape in Figure 26, which shows an asymptotic relationship between output quality P and effort W. There is a range of asymptotic curves where P approaches 1, depending on a range of values of effort, W1, W2, ..., WN and Q. If Q is high, the curve is steeper and the output quality P is higher. We believe that the difference from the naïve EXCEL models is due to the dynamic feedback of error detection and rework within the process.

The second shape is discussed in Chapter 7.

This may indicate that the quality profiles of components are much more sensitive to the policy and process control decisions about the allocation of effort. Correctly allocating effort between process phases may be crucial to achieving quality targets on which the network of cells depends.

4.3 Modelling and Simulating CMPM

The use of modelling and simulation to examine the feasibility of implementing process improvements and the difficulty of finding sufficient empirical evidence has been discussed in Chapters 2 and 3. The most extensive change that an organisation can make is changing its adopted process model. Modelling and simulation forms part of the process model development, risk assessment and also the definitions and tools which will support the management process.

We are investigating CMPM at a strategic level rather than examining what happens in a specific project. As discussed in Chapter 3, Systems Dynamics, which is a continuous technique, is the most appropriate dynamic modelling and simulation paradigm for investigating dynamic behaviour at strategic level [Kellner, Madachy et al. 1999], [Kreutzer 1986]; we have therefore chosen this method for our investigation of CMPM. In this chapter we investigate the causal loop structures that represent the dynamic relationships within the process. We create a model of the process showing these feedback relationships and then

simulate the behaviour over time. In Chapter 7, we describe the process of moving from simulatable, qualitative models to quantitative models where data from our industrial partners is used to check whether the model exhibits similar behaviour to the live system.

At a strategic level CMPM may be viewed as a continuous process where requirements for new versions of a product are received, products are made and delivered to internal or external customers, therefore suited to a Systems Dynamics approach. We used Stella [Stella 1990 - 1998] as a graphical environment for producing systems dynamics models and running simulations of the models. By simulating the cell behaviour at the intra-cell level we can explore the effects of varying inputs to the cell on the achievement of its output targets and gain understanding of the relationship between the targets.

Figure 27 shows a representation of a single CMPM cell derived from the pumping model abstraction. In this example we have focussed on two targets, quality of a component (reliability, fitness for purpose) both at an input level and the target level to be achieved by the cell and the effort that must be expended by the cell to achieve the target quality. The difference between the input and output quality is a measure of the value added to the component by the cell. The effort required is a measure of the cost of adding the value [Porter 1985].

The Systems Dynamics model abstracts the Pumping model into two activities, doing work to integrate components, *Complete Tasks*, and doing work to remove defects. Work to integrate components covers the pumping model activities 'understand and design', 'build/modify components' and 'integrate'. Work to remove defects covers the 'test' activities identified in the pumping model. The amount of work that the cell must do derives from the external requirements imposed on the cell, generating work to build or modify internal components and work to integrate internal and externally produced components. In the Systems Dynamics model this is identified as *Planned Work.*

In developing these models I have used tried and tested systems dynamic structures [Systems; isee systems 1998], [Kreutzer 1986] as modelling components, for example, co-incident flows (completed work and defects) and producer flows (completed tasks produced by effort).

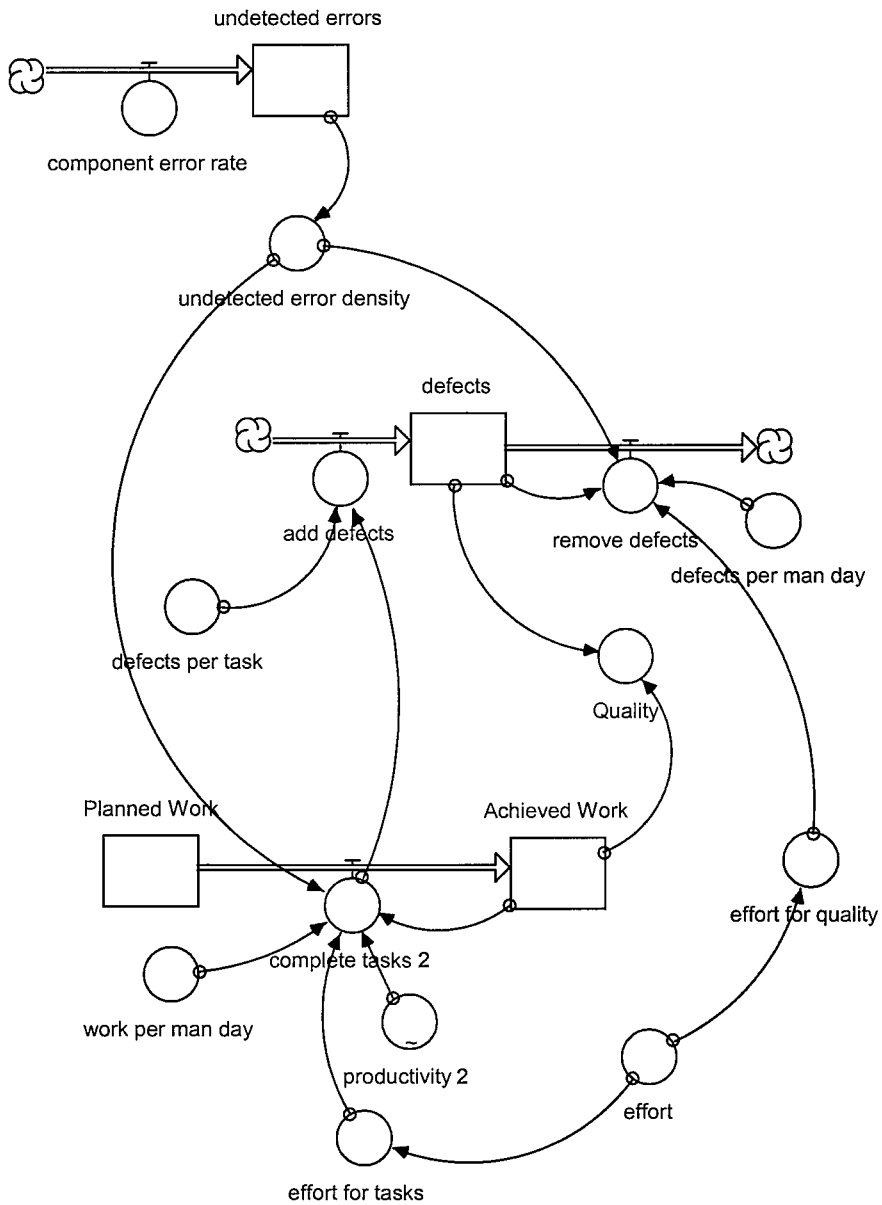


Figure 27. Systems Dynamics Model of a single CMPM cell

Planned Work, Achieved Work, Defects are defined as stocks which accumulate and are drained by work processes. *Undetected Errors* represents errors in delivered components that the cell will integrate.

Complete tasks is the flow which accumulates *Achieved Work.* The resource for task completion is *effort* in man-days. As work is done to complete the component, defects are co-incidentally generated (a co-flow, *add defects*). The stock of defects is drained by the work process *remove defects*. *Effort* in man-days is divided between the work processes that complete work and those that detect and remove defects. The percentage applied to each flow can be varied to examine the effect of different process policy decisions.

Quality is a measure of absence of defects and completeness, and because quality is a coincident flow with work, an increase in quality can only be achieved by work to complete the component and remove its defects. This is shown by the connection between *Quality, Achieved Work* and level of *Defects*. The cell can only remove defects in the components it builds itself; incoming components from internal or external suppliers that are of low quality cannot be improved by the cell, only by the supplier, but the amount of work that the cell has to do increases because of the difficulty of integrating low quality components. The model shows a feedback relationship between the level of defects and the rate of defect removal. Littlewood [Littlewood 1991] observes that errors with a high frequency are fixed first, and that as the error density decreases, those remaining are infrequent and more difficult to find.

In this model, for the purposes of focussing on the work processes and quality, *Effort* (manpower) has been defined as a constant, representing fixed team size and availability for the duration of the project. Other models, in particular Abdel Hamid and Madnick [Abdel Hamid and Madnick 1991] and Madachy [Madachy 1996], explicitly show that manpower is a stock that increases through a recruitment process and decreases by people leaving. This was because they were

investigating the effects on costs and schedule of varying manpower through resourcing policies. In this model, manpower resourcing behaviours are not being examined and have been excluded.

Effort for each delta time interval is the same and represents the cost of each time unit; expenditure of time is equivalent to expenditure of cost.

The effort available for work is converted by *Productivity*. In the best case, all of the effort is productive, but productivity is reduced by human resource factors, for example: level of experience, team size, motivation [Boehm, Abts et al. 1996]. In their empirical research Perry, Staudenmayer and Votta [Perry, Staudenmayer et al. 1994] found that developers spent only 40% of their time working directly on development, and for the remaining time they were either waiting or doing other work.

The work process is converted by a factor representing the process quality. In the best case the process will have a high quality. The quality of process can be assessed using CMM [Paulk, Curtis et al. 1994] or SPICE [Dorling 1993]. Assessing overall process maturity and Key Process Area maturity in the Capability Maturity Model is similarly used in COCOMO II [Boehm, Abts et al. 1996] as one of the Scaling Drivers (PMAT) for project estimation.

Simulation shows the interaction between interdependent goals of cost (effort) and quality as we vary the effort and its division between work done to complete the component and work done to remove defects, planned work, productivity and process quality.

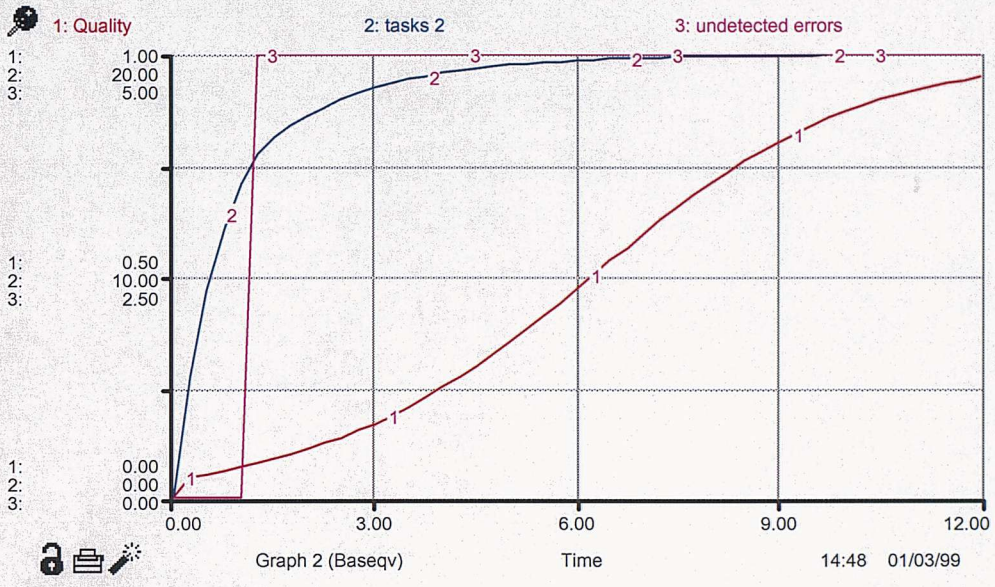


Figure 28. Graph from simulation of Stella model

The graph of quality over time (effort), Figure 28, shows an asymptotic approach to its maximum value.

As work is done to complete the component, defects accumulate and work is done to remove them. Since the number of defects in proportion to the size of the completed work reduces it takes more work to remove them. The amount of effort to detect an error increases as error density decreases. Errors therefore become more expensive to detect and fix.

If overall effort (manpower is insufficient) or the proportion of effort allocated to removing defects is reduced, the product may be completed but the quality will be lower at the end of the simulation time.

We can simulate the effects of receiving a faulty component by injecting extra defects that have not been generated by the cell's work process. The achievement of quality by the end of the simulation time is reduced. Our collaborators feel that

this graph shape, an asymptotic approach to completion of size and achievement of quality Figure 28, reflects their experience.

4.3.1 Systems Dynamics Model Evolution

The model shown in Figure 27 is an evolution of earlier Systems Dynamics models, descriptions of some of which follow.

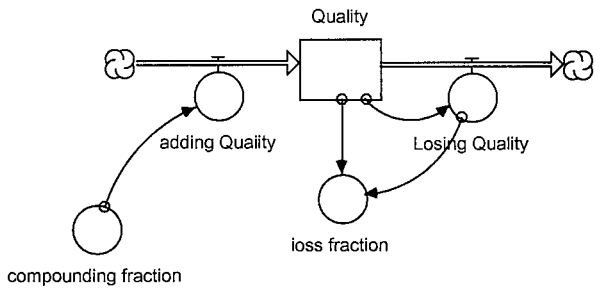


Figure 29. Early Quality Model

In the early models, quality was modelled as a stock that could be lost through adding defects or gained by doing work to add quality (Figure 29). A later model (Figure 30) showed quality as the inverse to a stock of defects; quality was defined as an 'absence of defects'.

This produced behaviour where the quality of a component was at its maximum when no work had been completed, Figure 31.

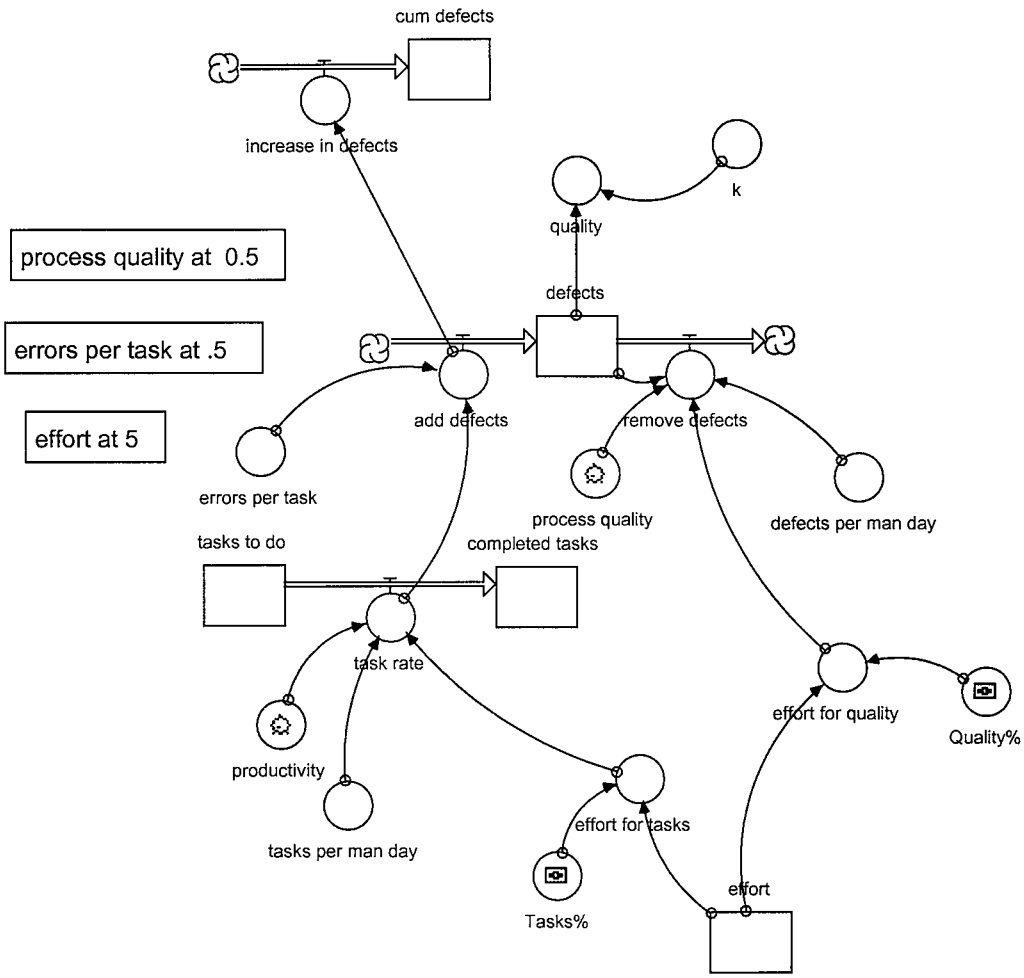


Figure 30. Stella Model of Development Process, Quality is a simple measure of defects

Our understanding of quality was improved by recognising that a more reasonable definition of quality has to include the completeness of the product and is therefore a measure of absence of defects and also of satisfaction of requirements.

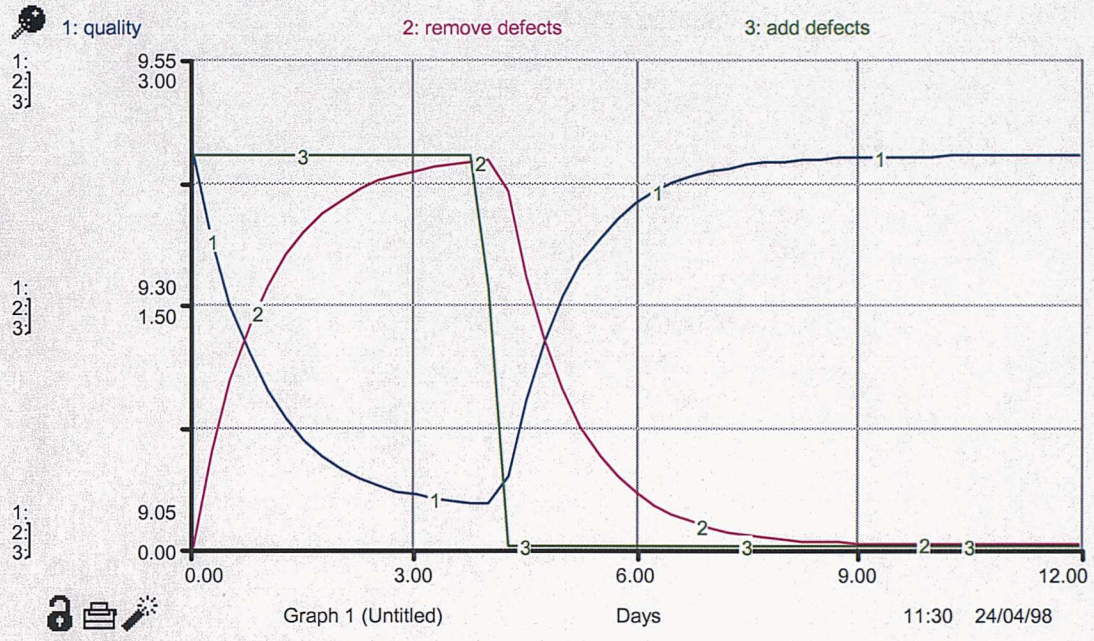


Figure 31. Graph produced from simplistic Stella model, Quality is greatest when no work has been done

The models were developed further to include the effects of defects in components being integrated.

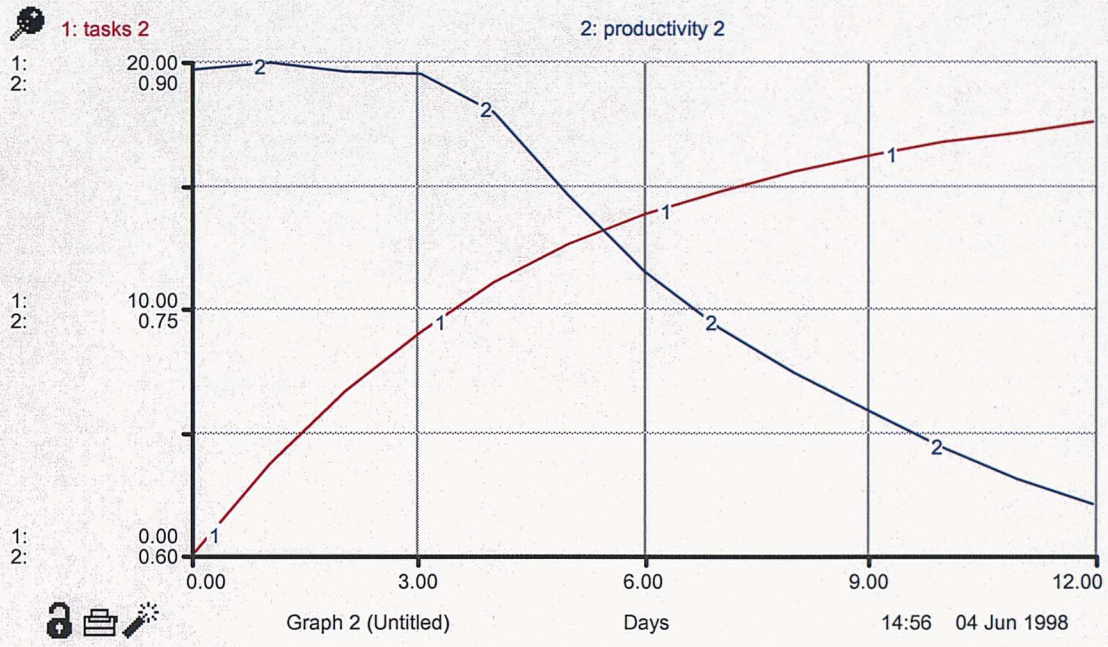


Figure 32. Asymptotic growth in size, tasks 2

The growth in size of the product was modelled to investigate how the rate of completion reduces as size increases, Figure 32. The following models (Figure 33) show three structures that produce an asymptotic growth in size.

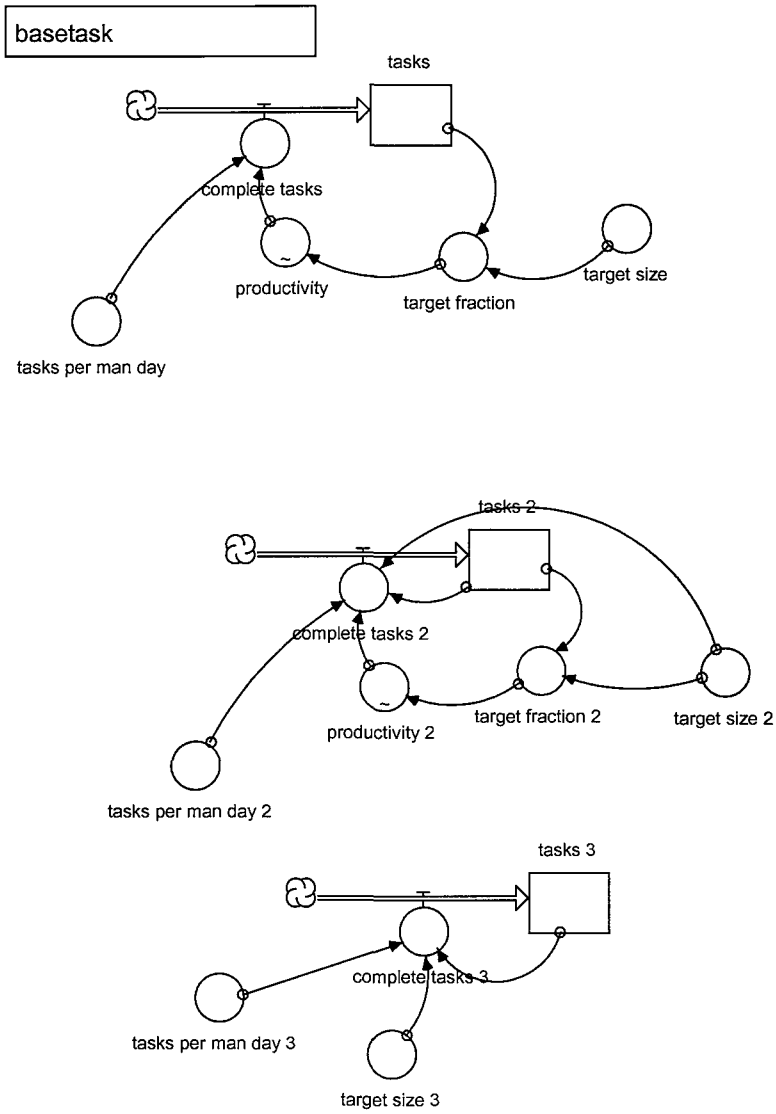


Figure 33. Three task completion models showing structures where the completion of tasks takes longer as the size of the stock increases

Experience in developing the models showed the dangers of increasing their complexity to a level where understandability is reduced, so that the purpose of the model, which is to increase understanding is not achieved.

4.4 Modelling CMPM Networks of Cells

A CMPM process instance is a network of independent single cells that co-operate. We have shown the behaviour of a single cell but the dynamics of the network are created by the individual cell in dynamic relationship with the other cells. In modelling a process network the dynamic couplings will be investigated.

4.4.1 Repeating structures

Wernick and Lehman [Lehman and Stenning 1996], [Wernick and Lehman 1998], in their work on the evolutionary nature of software, believe that systems dynamics models must be abstracted as far as possible in order to retain understanding and insight. While Systems Dynamics is a useful tool for revealing the dependencies that cause behaviour, a model that is at too high a level may not reveal sufficient information to show the basis for complex behaviour or enable understanding of strategic issues. On the other hand, models that are too complex can too closely attempt to model the real world rather than model the behaviour of the world. Kreutzer [Kreutzer 1986] follows the principle of Occam's Razor which is that a good model is the simplest one that can still be justified.

The abstraction at the heart of the CMPM models is that there are repeating process structures; each process model is a network of co-operating cells, each one repeating a generic model structure.

We can abstract from the pumping model described (Figure 23) and show each cell as a 'black box', with the pumping model as its internal behaviour. effort, W, is applied to a component of Planned Size S to complete the component and remove defects. Q is the input quality of a component and P the output quality of the component, informally estimated on a scale where 1=perfect and 0=useless.

A process model is defined by creating the cells and connecting them into the network. Using generic, repeating structures we can generate a model for each instance of a process. The cells are dynamically coupled by inter cell relationships.

There are two views of CMPM where repeating structures can be observed. The first view, which has been described, is a network of co-operating cells, each of which is repeating a pumping model process to produce sub assemblies of the delivered system.

The second view shows the repeating structures within the evolution of the product. Each delivered instance of the system is an evolution.

Each evolution is a build (product design); each build has one or more integrations of sub assemblies and each integration is made of one or more components.

We investigated the dynamic coupling between cells, the inter cell relationships. The connecting relationships that affect the behaviour of the CMPM process network are as follows; the amount of work that the delivery of a component supplies to the customer cell, (traditionally this has been related to size), the quality of the component supplied, the required output quality of the component to be built, and the schedule completion. These relationships between cells may be described as a set of metrics. [Chatters, Henderson et al. 1998]:

$$W = f(Q,P,S)$$

Where:

W = effort; S = size

Q = input quality of supplied components

P = delivered quality of system

The rapid rate of evolution of products means that planning and control information needs to be recorded much more frequently. In this environment it is important that this information is captured easily and speedily. In the same way interpretation needs to be quick and simple to enable control decisions to be made in time to achieve the target outcome. In modelling network coupling we must be sure that the relationships are expressed in measures that are feasible to collect.

We discuss below our investigations into these relationships.

4.4.2 Size

Systems Dynamics simulation by Abdel-Hamid and Madnick [Abdel Hamid and Madnick 1991; Abdel Hamid, Kishore and Daniel 1993] has helped us understand the behaviour of traditional sequential paradigms and the complex, concurrent behaviour described by Mander and Powell [Powell, Mander et al. 1999] but existing simulation models assume that all of the work to complete a development product is done within the organisational boundary. This allows the abstraction that a unit of code (size) can be directly related to a unit of work. We can say that, for example, in the process modelled by Abdel-Hamid and Madnick (Figure 34) the work to produce a kloc (thousand lines of code) comprises the sum of the work at each phase in the process that the organisation has adopted.

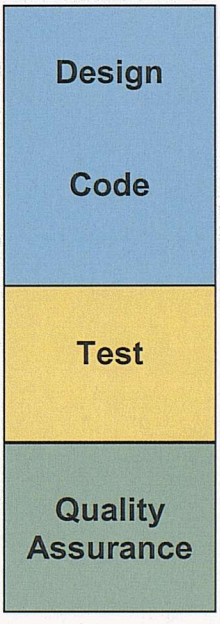


Figure 34. The work to produce KLOC

The process phases are dynamically coupled, as the work outputs from one phase are the work inputs to the next phase. However, new size metrics are needed when

the work output units of one cell do not form the work input units to the downstream cell and the final size of the product is not directly related to the work required to produce it. The input work units are related to the amount of work required to 'glue' components together via component interfaces and the required work to produce a cell's self-produced components. The exported work units are the size of the produced component's interface. The size, S, of the required work of a cell is defined by:

S = component Glue + self built components.

The exported work size SE, delivered to the customer cell is defined by:

SE = Interface size of the component

The amount of 'glue' needed should be related to the sum of the interface sizes of the components. We expect inter cell dynamic coupling to be related to interface size.

However, Garlan, [Garlan, Allen and Ockerbloom 1995] in proposing a theory of architectural mismatch, presents a case study of the problems associated with assembling a system out of existing large scale parts. The study showed that considerable effort was required to solve problems of interfacing the parts that were not related to the size of the interface, but were to do with mismatched architectures. Solving the problems required 'work–around glue code' or additional components in order to compensate for the mismatched interfaces as described in 4.2 . Whilst the Garlan study attempted to compose large-scale existing systems that were not designed or intended to be components, it does indicate that the architecture and closeness of fit of components may be an important determinant of the effort and glue required to compose them.

The degree of fit may be regarded as an aspect of the incoming component quality Q, in terms of usability, efficiency, reliability and functionality of the component to be composed. In this view, the perceived quality of a component (and therefore

suitability for selection as a component) varies according to the components with which it will be composed.

The size of total glue code required is related to the size of the interface and the closeness of the fit. A component that fits perfectly requires no additional glue code whereas a component with a poor fit requires additional glue code. The size of the additional glue code is a measure of the degree of fit of the component. It represents the extra work required to achieve the desired quality in the delivered system. In COCOMO II, Boehm uses a breakage factor to indicate code created but discarded as a result of error or changed requirements (see Chapter 2.7.3). The additional size required to create glue code to work-around poorly fitting components is a similar factor, but in this case, the code is delivered not discarded. So we must modify our earlier definition of S to be:

S = Interface Glue + work-around glue + self built components.

In his work on prediction models and tools, in particular COCOTs [Boehm 1997], Boehm provides an analysis of the effects of COTs style development on prediction and provide an alternative size framework based on interface points as a method of estimating the amount of work that an interface generates.

4.4.3 Quality

The quality abstraction used by Abdel-Hamid and Madnick [Abdel Hamid and Madnick 1991] and Madachy [Madachy 1996] assumes that all defects are removed and all requirements achieved by the end of the simulation. The dynamic effect on the process is the rework required. This is a reasonable abstraction in a traditional process environment where the simulation covers one complete lifecycle where at the end of the process the software product moves outside of scope of the simulation and output quality has no dynamic affect on the process. In CMPM each cell has a complete lifecycle of its own and the output quality of a component produced by one cell provides the input quality of components for downstream customer cells. The cells are dynamically coupled by quality.

Informally we estimate the quality (reliability, fitness for purpose, closeness of fit) of the components we must build from and the product we must build. Thus quality is an estimation of completion of requirements both functional and non functional (NFR). A component with completed requirements and without defects would have quality =1, a useless product would have quality = 0. Quality predictions are vital process control information for managers who must trade off their own schedule, cost and quality targets. If the predicted quality of a component from an upstream cell is poor, the downstream manager may decide to use an earlier evolution of the component, trading requirement fulfilment against schedule fulfilment.

4.4.4 Schedule

The network of cells also have to meet schedule targets, the schedule achievement by an individual cell affects the downstream cells ability to meet their targets. If a cell is predicted to fail to meet its target, the customer cell may have the flexibility to rearrange their integration tasks to minimise the effect of the delay, or it may use an earlier version of the component or in the worst case be delayed in the achievement of its own schedule target. However the delay imposes schedule pressure which acts on the downstream cells.

Boehm's work in COCOTs [Boehm 1997], is based on an analysis of the experience of component integrators, contained in the SEI Software Engineering Risk Repository; the study confirms that delays in the supply of components affect the ability of the integrator to meet their performance targets. Supplier performance is used as a cost driver in the COCOTs tool.

Ford and Sterman [Ford and Sterman 1997] also show the importance of modelling customer and supplier relationships, although in their model, the suppliers and customers are development phases. They show the dynamic effect on work availability and therefore performance of external concurrence relationships.

4.4.5 Predictability

Organisations indicate that predictability about project performance is critical to the success of the organisation. Decisions to invest in a product are based on risk, costs, potential revenue and whether the time to market matches a predicted window of opportunity for the product (when the product is likely to achieve its sales potential). If predictions under-estimate schedule then an organisation will commit expenditure to a project that will fail to meet its sales potential. If the predictions significantly over-estimate schedule then organisations will fail to invest in potentially profitable products. To date, achieving predictability has been difficult, since traditional static project planning and estimating systems do not model the dynamic nature of software processes.

Chapter 5

Evolutionary Systems Dynamics Model Building

There are two motivations for this set of investigations and simulation experiments.

- Showing how systems dynamics models can be built in an evolutionary manner, with successive models making an evolutionary step towards a closer correspondence to a real world model; this may be described as evolutionary model building.
- Demonstrating that observed behaviour can be reproduced in Systems Dynamics, using an evolutionary model building process and validated by quantitative data.

Whilst qualitative Systems Dynamics models can be useful as means of understanding a problem domain, it is very easy to construct a model that appears to convincingly explain a particular behaviour but there may be other equally plausible models that could explain the behaviour.

A model's correspondence with real world behaviour is more likely to be demonstrated if qualitative data from the real world, when used in the model, reproduces the real world behaviour. When taking a real world process and modelling it in Systems Dynamics, it can be very difficult to move beyond qualitative models to quantitative models because unfortunately, in many complex systems it is very difficult to find accurate data from the problem domain that can be used to validate the model. This happens for many reasons; historic data at the

level of detail that the model needs may not be available; recording data specifically for the purpose of modelling may not be economically viable.

In this simulation experiment I have used a simple process with precisely defined behaviour. The process is defined by activities governed by probabilistic choices and can be simulated to produce quantitative data using Monte Carlo methods [Metropolis and Ulam 1949], [Fishman 1996].

The Monte Carlo method models probabilistic choices of activities; Systems Dynamics models stocks and the flows that create and deplete them. Both paradigms model continuously over time but Monte Carlo methods calculate a choice over a probability distribution at each tick of time, whereas Systems Dynamics calculates, using partial difference integration, the effects of flows on stocks at each tick of time. Systems Dynamics explicitly models causal and feedback relationships and abstracts from single entities to a population.

The investigation shows an evolutionary method of Systems Dynamics modelling; feedback at each evolution from comparing the model to the real world behaviour is the dynamic that generates a closer correspondence in the next evolution of the model. In this investigation the Monte Carlo model provides an alternative model of the simple process which gives qualitative and quantitative visibility into the process to support the evolutionary process.

I will show how Systems Dynamics models can be produced in an evolutionary manner to reproduce the behaviour of our simple process using qualitative and quantitative results from the Monte Carlo model simulation to bring the correspondence of the two models closer together at each evolution of the Systems Dynamic model.

In the earlier evolutions, when the Systems Dynamic model and simple process behaviour is furthest apart, the comparisons are typically qualitative; in later evolutions we can use quantitative comparisons.

5.1 The Simple Process

Let us examine a simple software development process. A piece of software must be built to a specified size (this may be a number of components or function points or any other measure of the size of software). How long it will take to complete and the quality of the resulting software will depend on three things; the work applied to the task, the efficiency of the process in producing good work, and the quality assurance practices used to detect and fix faults.

The quality assurance practice can be represented as a policy of tolerating only a certain proportion of bad code throughout development before applying effort to fix the bad code.

The efficiency of the process can be represented by the proportions of good and bad code produced throughout the process and its ability to fix bugs.

This simple process may be described by Monte Carlo methods to produce a model in which effort is allocated according to the perceived defects in the already completed work

The process has a probabilistic choice of four activities:

- Doing good work
- Doing bad work
- Fixing bad work
- Or finishing

At every tick of the clock a randomised choice between the activities is made; this corresponds to a unit of work being carried out.

The choice between activities is weighted towards fixing bugs if the bugs are in excess of a fixed proportion of code or if the code has achieved its target size.

When making new work, weighting towards making a bug increases with the proportion of code completed. In the example provided, the weighting increases from 30%, when the proportion of completed code is nil, to 95% when the code approaches target size. This likelihood of making a bug may be used to represent the efficiency of the production process, both in making good code and also in finding and fixing bugs.

Although the process is simple, these structural dynamics cause interesting behaviour in terms of the quality outcome of the code and the time it takes to complete a target amount of code.

5.2 The Simple Process Modelled as a Decision Tree

How the choice is made between activities may be modelled as a decision tree (Figure 35). For every iteration of the process (a 'tick' of the process clock), the choice is between correcting bad work (*correct*) and doing new work *(move on)*. If the choice to correct bad work is made, either the path to make a fix *(make fix)* is taken or the path for an unsuccessful fix *(don't fix)* is followed. If the choice is to do new work, then either new work finishes *(finished)* when the work has

completed its target size, or a new piece of code may be added *(add bit)*. If a new piece of code is added, then either the path to make good code may be followed *(make good work)* or the path to make bad work *(make bad work)* may be taken.

The leaves of the tree show the effect on cumulative work w, cumulative bad work *b*, and cumulative code *n*.

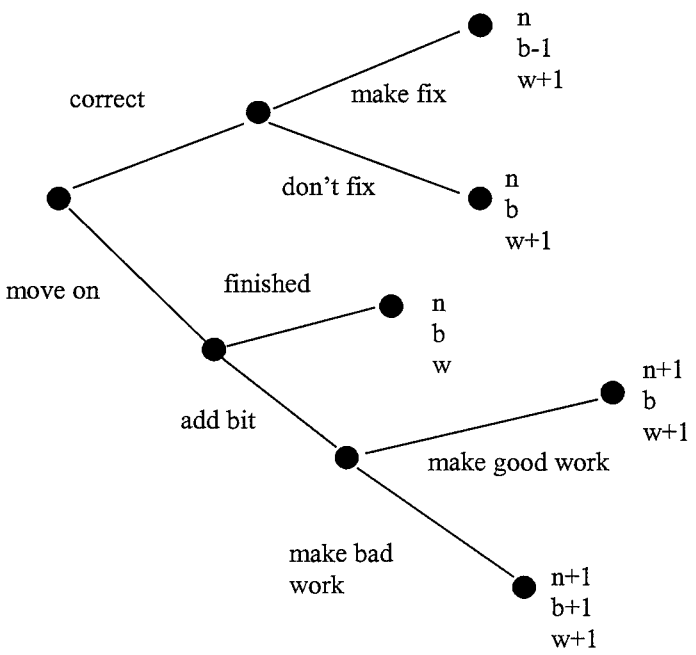


Figure 35. The simple process modelled as a decision tree

5.3 Monte Carlo Model (Mathcad)

The process described by the decision tree can also be described by Monte Carlo methods to produce a statistical simulation. The process then becomes a set of probabilistic choices, (corresponding to the nodes of the decision tree) which determine the paths followed. A simulation is the graph of paths followed as each choice is made randomly, according to the probability distribution.

The equations below describe a Monte Carlo model of the process in Mathcad [Mathcad 1999].

The model has the following functions:

- corr
- bug
- addbit
- fixbug

Corr is the choice to correct a bug, or to do new work (*move on* in the decision tree) provided that the work has not reached its target size, N. (Table 1, page 103 provides a key to the symbols used in the following equations)

$$\text{corr}(pc,b,n,t,k) := \left(b_t \geq 1\right) \cdot \left[\left(n_t \geq N\right) + \left(k \cdot pc_t \leq \frac{b_t}{n_t}\right)\right]$$

corr is true (choose to correct a bug) if there is at least one bug, the code is not at the target size and the bugs in the completed code are greater than the maximum percentage of bugs tolerated *k*, weighted by a random choice function *(pc)*.
When *corr* is false, the '*move on*' path is followed.

Bug is the choice that work (either new work or work to correct a bug) is good work.

$$\text{bug}(pb,b,n,t,aa,bb) := pb_t < aa + bb \cdot \frac{n_t}{N}$$

bug is true (a work unit is incorrect) when the random choice *(pb),* falls between the minimum rate of faulty work set for the process *(aa)* and a maximum that

increases as the completed code increases (bb*n/N). When *bug* is false, a unit of good work is added if new work is being done, or a bug is removed if the path to *correct* has been followed.

Addbit is the choice that adds a piece of new work, which may be either good work or bad work, determined by the choice, *Bug*.

$$\text{addbit}(pb,b,n,w,t,aa,bb) := \text{if}\left[\text{bug}(pb,b,n,t,aa,bb), \begin{pmatrix} n_t + 1 \\ b_t + 1 \\ w_t + W(n_t) \end{pmatrix}, \begin{pmatrix} n_t + 1 \\ b_t \\ w_t + W(n_t) \end{pmatrix}\right]$$

Fixbug is work to remove a bug which may be effective (*make fix* in the decision tree) or ineffective which does not remove the bug (*don't fix* in the decision tree), determined by the choice, *Bug*.

$$\text{fixbug}(pb,b,n,w,t,aa,bb) := \text{if}\left[\text{bug}(pb,b,n,t,aa,bb), \begin{pmatrix} n_t \\ b_t \\ w_t + W(n_t) \end{pmatrix}, \begin{pmatrix} n_t \\ b_t - 1 \\ w_t + W(n_t) \end{pmatrix}\right]$$

At each tick of time of the simulation, the functions are evaluated;

$$\begin{pmatrix} n_{t+1} \\ b_{t+1} \\ w_{t+1} \end{pmatrix} := \text{if}\left[\text{corr}(pc,b,n,t,k), \text{fixbug}(pb,b,n,w,t,aa,bb), \text{if}\left[n_t \geq N, \begin{pmatrix} n_t \\ b_t \\ w_t \end{pmatrix}, \text{addbit}(pb,b,n,w,t,aa,bb)\right]\right]$$

Key:

n is cumulative code.

N is final size of code (target)

w is a unit of work; in this case a unit of work is added for every tick of Time, t

t is a tick of time

T is time

Two random functions determine the probability distribution of choices (where *runif* is the Mathcad random generator):

pb is randomly generated probability of making a bug

$$pb := runif(T, 0, 1)$$

pc is randomly generated probability of choosing to correct

$$pc := runif(T, 0, 1)$$

k is the maximum percentage of bugs tolerated in the code

Two constants weight the balance of producing good and faulty code:

aa minimum rate of bug production

bb variable rate of bug production

Q is quality of the incoming components

$$Q_t := \frac{(n_t - b_t)}{n_t}$$

P is quality of the outgoing components

$$P_t := \frac{(n_t - b_t)}{n_t}$$

Table 1. Key to Monte Carlo model equations

At each tick of the simulation,

if *corr* evaluates to true,

the simulation follows the *fixbug* path.

if *corr* evaluates to false,

we evaluate whether the product has reached its target size;

if it has, no new work is added;

if it is less than the target size,

the simulation follows the *addbit* path.

At each tick of the clock, whether new good or bad work is added or a bug is either fixed or not fixed, a unit of work is added until the process finishes. The process finishes when the target size for the code is complete and all of the bugs have been removed. During a typical simulation of the process, the time allowed for the simulation may expire before the process is complete.

By varying the initial settings for the simulation; we can produce a range of results for the time taken to complete the code, the rate of growth in size and the quality of the code.

5.3.1 Simulation 1, varying the quality of the incoming component

We can simulate a typical process scenario in which we receive a component from which we are to build our product. The component is 200 code units in size and our finished product will be 400 code units. The component is not perfect and has a proportion of bad code which will affect the quality of our product. Within our process we will add new code until we have reached the target size and attempt to fix the bugs. We will set the threshold for tolerance of bugs in the code at 25%.

The first simulation was set up with the following values:

Initial size, n0	200
Target size, N	400
Initial bugs, b0	25
Threshold for bug tolerance, k	25%
Minimum bug rate, aa	0.3
Variable bug rate, bb	0.95

Table 2. Initial settings for Simulation 1

When simulating the Monte Carlo model in MathCAD, the initial value of bugs was set at 25 and the initial value of code was set at 200. Thus at the commencement of the simulation, the number of bugs in the code was below the maximum tolerated.

Figures 36 to 39 show graphs from simulations of the model which trace the growth of size and defects as work is done. For each simulation, probabilistic choices at each tick of time determine the outcome of the unit of work. Thus repeating a simulation will not repeat the results, but produce a distribution of values of n, and b. However, the general shape of the graphs is similar; these graphs are representative of the graphs that were produced.

Figure 36 shows a graph of the growth in size, n, as work, w increases. Completion of the required 200 code units took 860 work units which takes 860 time units to complete, (because in this instance, one work unit takes one time unit *(tick, t)*. As the units of completed code approach the target size, the rate of faulty work compared with good work increases, slowing the rate of completion and showing an asymptotic approach to the target size. As new bugs increase, the bug

tolerance threshold is exceeded and at every tick of the clock the random choice to fix bugs is more likely.

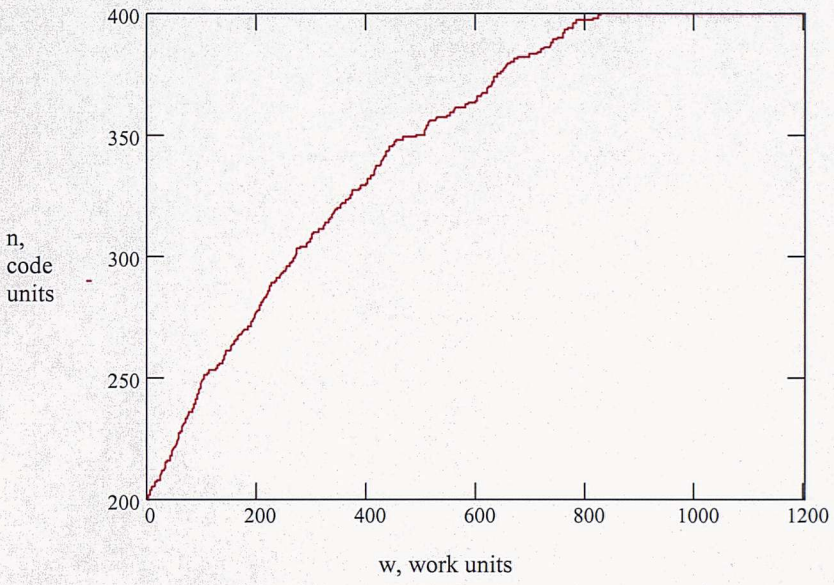


Figure 36. Graph from Monte Carlo simulation showing growth in size n, as work, w, is done

Figure 37 is a graph from the same simulation showing the growth in bugs, *b*, as work *w* is done. The level of bugs increases to a maximum of 85 when the code reaches its target size, subsequently new work stops and no new bugs are added. The simulation stops before the all of the bugs are removed, taking 340 work units to remove 37 bugs; this is a consequence of the effectiveness of the bug removal policy in the process.

The growth of bugs against work appears to show an S shaped population growth phenomenon.

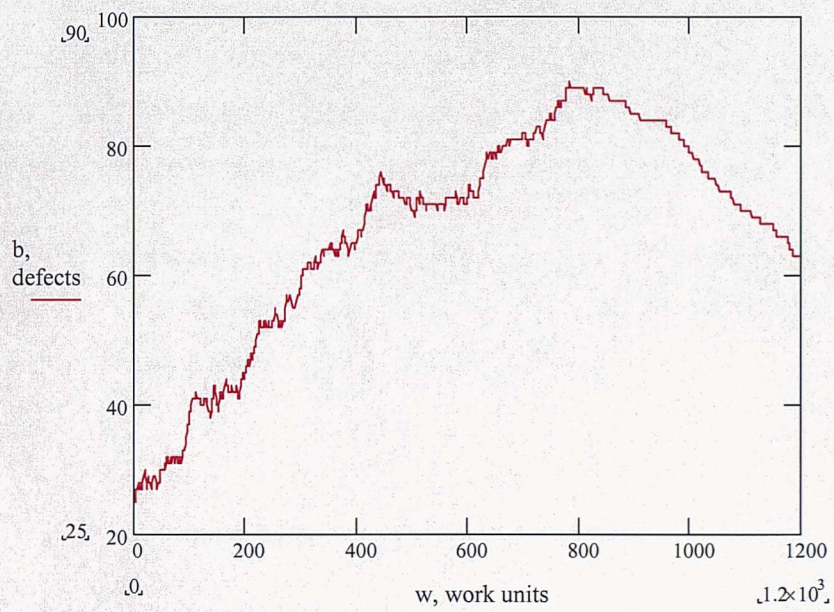


Figure 37. Graph from Monte Carlo simulation showing growth of defects, b against work, w

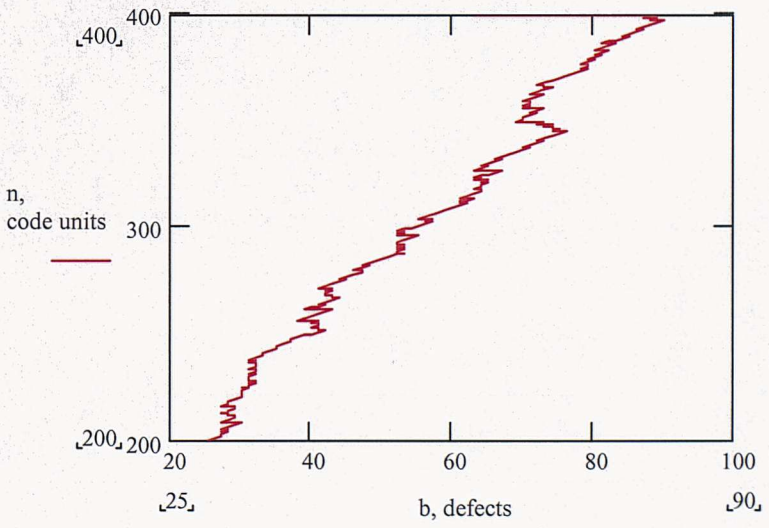


Figure 38. Monte Carlo simulation, graph showing growth of defects, b with increasing size, n

We can see the effects on the amount of work to achieve target size and the outgoing quality of the product of a different quality of the incoming component; in this case, there are 60 defects in the code from which we must build. The target size of 400 units and the defect removal policy remain the same.

The initial level of defects is higher than threshold level of bugs tolerated in the code, as a consequence work is done to reduce the level of bugs to below the tolerance threshold before any new work is done.

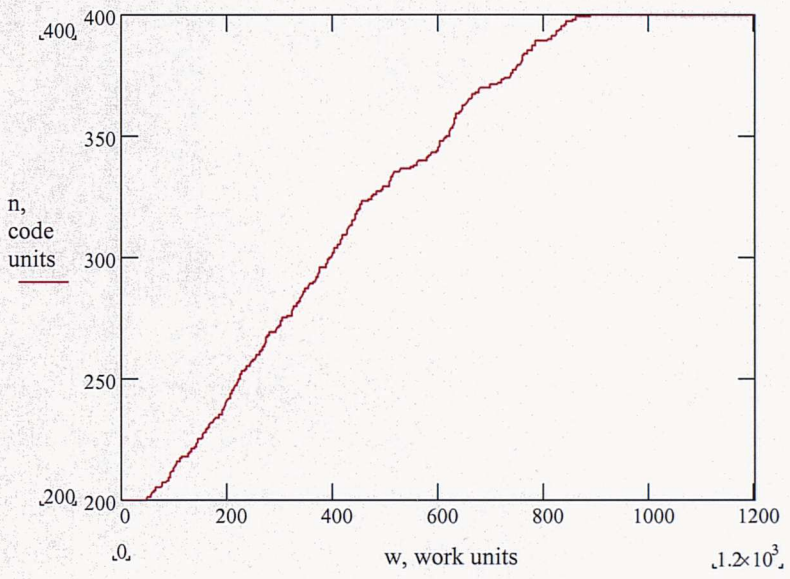


Figure 39. Monte Carlo simulation, initial defects = 60, graph of increasing size, n as work, w increases

Figure 39 shows the effect of high initial defects on code production; no new work is done so that the code does not begin to increase in size until t = 40 and target size is not reached until t = 850.

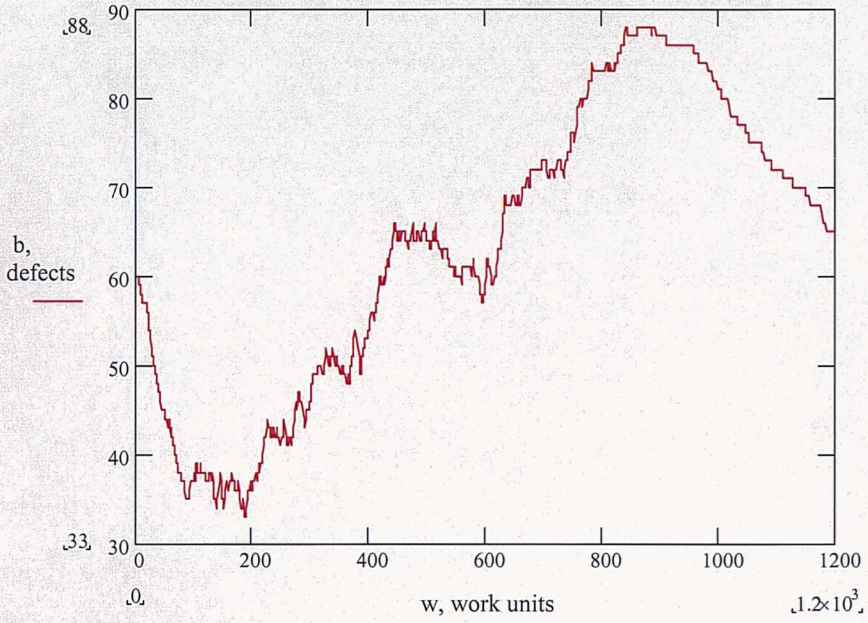


Figure 40. Monte Carlo simulation, initial defects = 60, graph of defects, b as work, w is done

Figure 40 shows the effect on the number of defects in the code; initially work is done to reduce the defects in the code until the threshold level has been reached. After that, new code can be made and the balance of work between making new work and fixing defects has the same probability distribution as when the incoming component had only 25 defects. The component takes 65 work units longer to complete and there are 2 more defects in the component at the end of the simulation time than in the previous example. Results for other values of incoming defects are summarised below.

b0	Maximum defects	Residual defects	Work to complete N, 400	Final Quality, P
0	89	59	780	0.853
25	88	63	825	0.843
60	88	65	890	0.843
100	100	68	940	0.83

Table 3. Results from Monte Carlo simulations where the number of initial defects were varied

Overall, in our simple process, the effect of defects in incoming components on the final product and the time it takes to complete is very small. For example, the work to complete the product increases by 20% when defects are increased to 50% of the incoming product. The residual defects increase by 10%. This is because the effects are only in the work done to reduce the defects to below the threshold of tolerance. Thereafter the process behaviour and the balance between new work and defect removal is the same, whatever the quality of the incoming component.

5.3.2 Simulation 2, varying defect removal policies

The second simulation I shall describe shows the effect of different policies for defect removal. As before, we are building from a component of 200 code units in size, and our finished product will be 400 code units. The component has 60 defects, and during the process we will attempt to produce a perfect product. Within our process we will vary the threshold of defects in the code above which we will choose to fix defects rather than produce new work.

The results from other simulations are summarised below; showing how varying the threshold, k affects the simulation outcomes of maximum and residual bugs, work to complete the product and the final quality.

k	Maximum defects	Residual defects	Work to complete N, 400	Final Quality, P
0.10	38	28	1110	0.93
0.25	88	65	890	0.843
0.50	150	115	690	0.713

Table 4. Results from simulating The Monte Carlo model with varying defect fixing policies, k

If we tolerate only 10 % defects in the code in production, we will achieve a low number of residual defects, but the time to complete the component will be nearly double the time to complete the product with a tolerance of 50% defective code. At the higher threshold of tolerance, the time to completion is short, but the residual defects are four times higher than if the 10% tolerance policy is followed.

In our simple process, varying the defect removal policy has a much greater effect on the time taken to complete the process and the quality of the finished product than the quality of the incoming components.

5.4 Systems Dynamics Representation of the Simple Process (Vensim)

We can recreate the Monte Carlo simulation developed in Mathcad, in Systems Dynamics using Vensim [Vensim 1988 -1997], by direct analogy. Here we describe an evolutionary modelling process producing a series of models, each iteration building on the last, as we understand more about the process until a close replication of the Monte Carlo model is achieved.

For each evolutionary cycle, we start with the real world behaviour and attempt to recreate it with the simplest possible Systems Dynamics structure. We then compare the Systems Dynamics model with the real world process and determine

the points of correspondence between the model and the real world behaviour. Where we have good correspondence, we can take the structure into the next evolutionary cycle. Where the correspondence is incorrect or absent, we add or remodel the structure in the next evolutionary cycle. In this way we are looking for successively better synchronisations of the model and real world behaviour.

This is a feedback driven process, the modelling activity increases understanding of the problem domain, which feeds back into increasing correspondence between the model behaviour and the real world behaviour.

To return to our simple process, we have a good understanding of the behaviour, both externally because we can simulate it and collect quantitative data, and internally because we can express the process precisely mathematically.

We can begin to represent the simple process in Systems Dynamics by making an initial abstract model of the stocks in the process and the flows that increase or deplete them, and using the evolutionary process described, evolve the model to include structures that affect the dynamic behaviour.

We produced eleven models in all; I will describe in detail six models that show the major evolutionary steps made towards the final representation of the simple process in Systems Dynamics. The six successive models show the evolutionary path to the successful final model. The other models were alternative paths that could have been followed but that were discarded. At each evolutionary step quantitative and qualitative analysis of structure and behaviour identified the model with the closest correspondence, which became the next successful evolution.

The Systems Dynamics paradigm abstracts from the individual entities to a population, therefore I have not used a randomised choice between the activities.

5.4.1 First Evolution - Monte 3

The first significant model is a simple abstraction of stocks and flows.

The obvious stocks in the system are code, expressed in code units, and bad code, expressed in bad code units. The activities that increase and deplete the stocks are making code and fixing bugs. For the first evolutionary cycle, I will concentrate only on the activities that increase the stocks.

There are two stocks, all code *(code)* and faulty code *(code that's bad)*. The code stock is increased by two flows, *make good code* and *make buggy code.*

Code that's bad is increased by the flow, *make buggy code.* The activity *make buggy code* increases both the stock of code and the stock of bugs.

The first approximation of the flows splits making new code between good and bad work in the ratio 65:35. There is no flow that removes bugs.

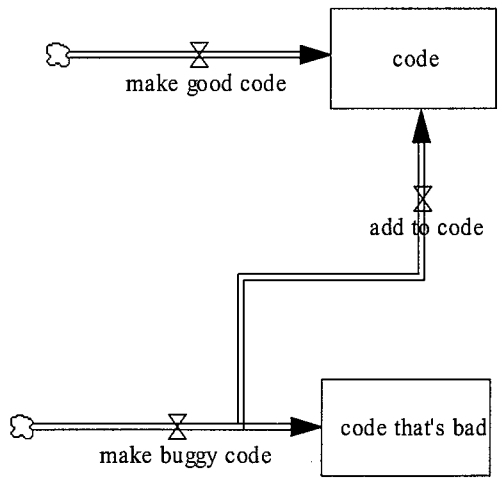


Figure 41. Systems Dynamics model of simple process

The model can be tested for correspondence with the simple process defined in Monte Carlo methods.

The systems dynamic model abstraction of flows is comparable to the Monte Carlo function *addbit* and the part of the decision tree coloured red in Figure 42.

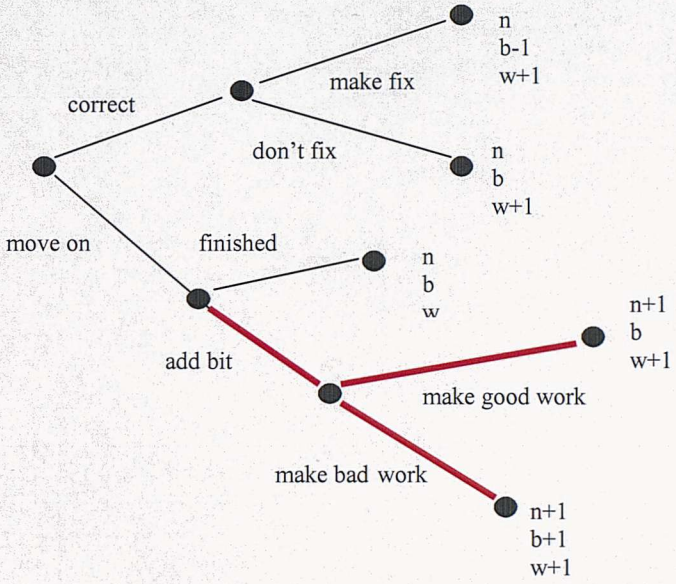


Figure 42. Decision tree of simple process showing coverage of first Systems Dynamic model

$$\text{addbit}(pb,b,n,w,t,aa,bb) := \text{if}\left[\text{bug}(pb,b,n,t,aa,bb),\begin{pmatrix} n_t+1 \\ b_t+1 \\ w_t+W(n_t) \end{pmatrix},\begin{pmatrix} n_t+1 \\ b_t \\ w_t+W(n_t) \end{pmatrix}\right]$$

addbit makes both defective and good code with every tick of time, *t*. The size of the code *n*, is made up of defective code and good code. The function accumulates the number of bugs, b within the code, n. The probability distribution between making good code and defective code is governed by two constants, the minimum rate of bad code production, *aa* and maximum rate, *bb*; these were defined for the simulation as $aa = 0.3$ and $bb = 0.95 - aa$.

The systems dynamics model approximates the probability distribution to a ratio of good and bad work. We can say that the Systems Dynamic model is in good

correspondence with the Monte Carlo model in the structure of stocks and the flows (or activities) that create them. We have a corresponding structure that determines the proportion of defective code in the total code. The structure is an abstraction of the Monte Carlo probability distribution, but it is good enough for the first evolution of the Systems Dynamics model

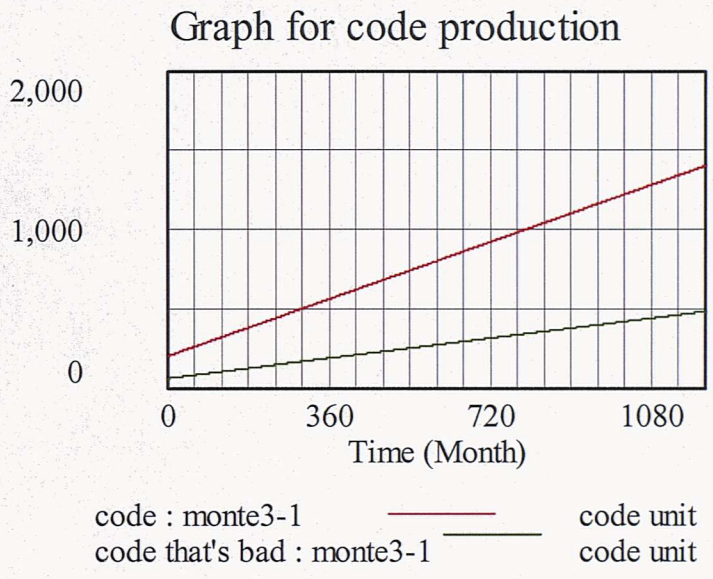


Figure 43. Simulation of first Systems Dynamics model, graph of code and defects as time increases

If we examine a graph of code production from simulating the Systems Dynamics model, Figure 43, we can see an increase in both total code and defects within the code. The code does not show an asymptotic approach to a target size that the full simple process simulated in Monte Carlo methods shows. However, the underlying structural correspondence is good, in that we accumulate the same stocks of good and bad code and they are increased by equivalent activities.

We can compare this model with an earlier discarded version that fails comparison with the simple process defined by Monte Carlo methods, Figure 44. In this model, although we have stocks of code and defects, the flows that create them do

not correspond closely with the *addbit* function. The model is too complicated as it has flows that increase stocks and flows that decrease stock whereas *addbit* only increases stocks. Instead of the stock of code being increased by two flows, code is increased by *add code* and *bugs* is increased by *add bugs*. The two flows are shown as coincident flows (one flows as a result of the other). The mathematical representation does not correspond either; in *addbit* the code is increased by both adding a flow of good code and adding the flow of defects, in the failed Systems Dynamic model, the stock of code is increased by increasing the flow *add bugs*.

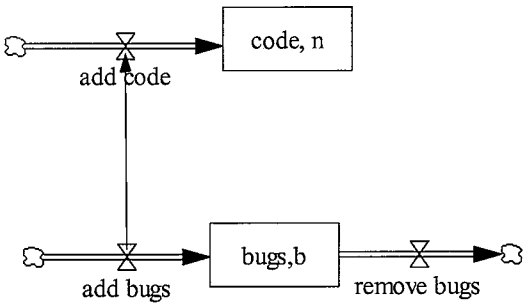


Figure 44, Alternative Systems Dynamics model of the simple process

5.4.2 Evolution 2, Monte 4

The second evolution of the Systems Dynamics model (Monte 4), Figure 45, has two new features, making an evolutionary step closer to the simple process.

The first feature stops code production when a target amount of code has been completed. The code production flows (*make good code* and *make buggy code*) stop when *code done*? is set to true, which is the case when the code already completed is the same as the target size (*final code size, N*). This structure describes goal-seeking behaviour in the process.

The model has a new flow, *remove bugs*, which decreases the stock of bugs.

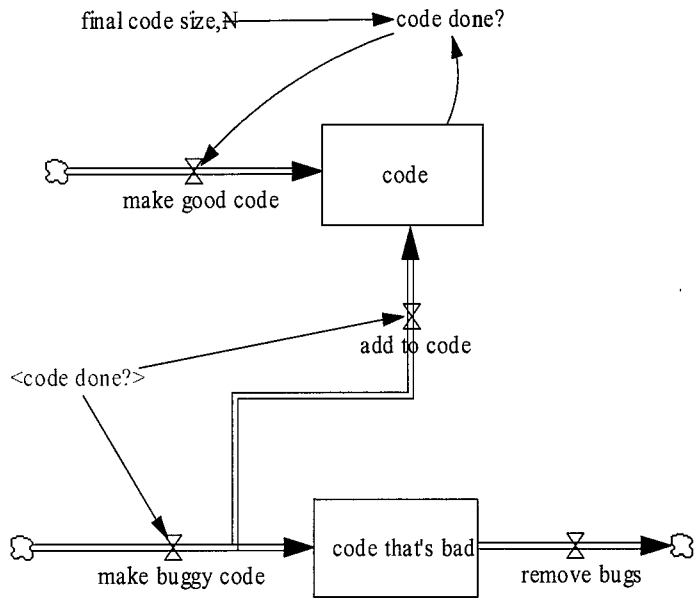


Figure 45. Second evolution of Systems Dynamics model of the simple process

We can examine the Systems Dynamics model for correspondence with the simple process.

The structure that stops production when the final size is complete is similar to the decision tree choice between *finished?* and *addbit*. In the Monte Carlo method equation, repeated below, it is modelled by evaluating $n_t \geq N$ (code at simulation time, *n* less than or equal to the target size, *N*) before evaluating *addbit* in for every tick of the simulation. This stops the production of all code, both good and defective.

$$\begin{pmatrix} n_{t+1} \\ b_{t+1} \\ w_{t+1} \end{pmatrix} := \text{if}\left[\text{corr}(pc,b,n,t,k), \text{fixbug}(pb,b,n,w,t,aa,bb), \text{if}\left[n_t \geq N, \begin{pmatrix} n_t \\ b_t \\ w_t \end{pmatrix}, \text{addbit}(pb,b,n,w,t,aa,bb) \right] \right]$$

We can conclude that the Systems Dynamic structure limiting code growth has a good correspondence with the Monte Carlo equation.

The flow *remove bugs* in the Systems Dynamic model represents *fixbug* in the Monte Carlo Model and *correct* in the decision tree. However, the representation is approximate; the simple process has a probabilistic choice that any attempt to correct a defect will fail, increasing a unit of work, w, but the number of defects, b remains the same. The Systems Dynamic model approximates defect removal rate over the defect population. Therefore whilst the structure corresponds to the simple process defined by Monte Carlo methods, the probabilistic behaviour is approximated.

The paths of the decision tree are now all represented in the Systems Dynamics model, Figure 46, however except for the decision to stop code production, the choices at the nodes of the decision tree that determine which paths will be followed have been approximated.

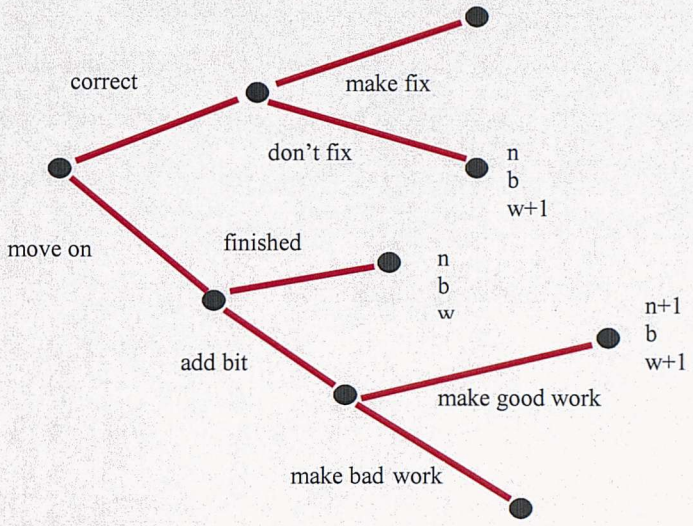


Figure 46. Decision tree coverage by Systems Dynamics model evolution 2

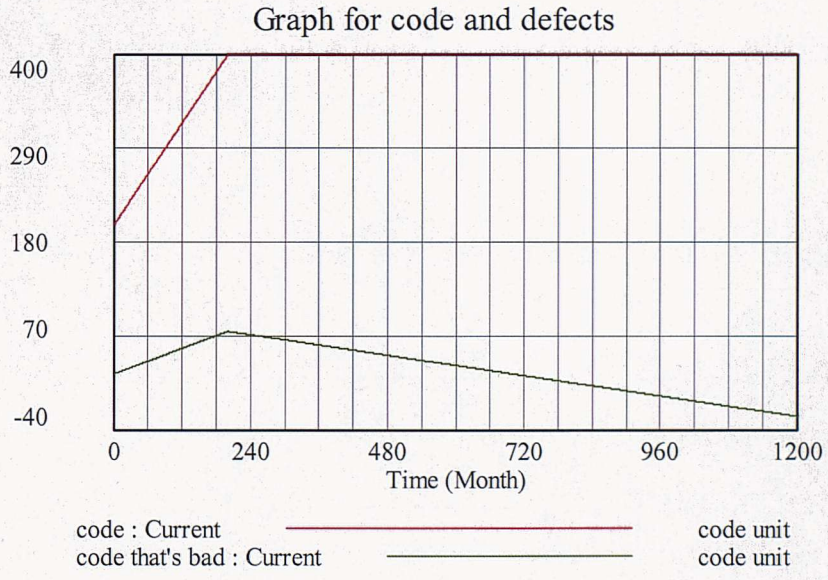


Figure 47. Simulation of 2nd evolution Systems dynamics model, graph of code and defects as time increases

The simulation results from the Systems Dynamic model, Figure 47, show that the code and new defects in the code stop increasing when the code reaches the target size, 400 units, as we expect from the correspondence with the Monte Carlo model. After that point, defect removal continues, again following the behaviour of the Monte Carlo model; except that the Systems Dynamic model has no structure that will stop defect removal after defects have reached zero. The increase in code does not show an asymptotic approach to the maximum produced by simulation of the Monte Carlo model.

5.4.3 Evolution 3 – Monte 6

The third evolution focuses on dynamic behaviour that affects the choice between making new code and defect removal activities.

Monte 6, Figure 48, adds another goal seeking structure to follow the policy that up to 25% bugs in the code in progress will be tolerated. This is the variable *'willingness to tolerate bugs'* and associated links. The function that defines the structure in the systems dynamic model follows:

willingness to tolerate bugs=

IF THEN ELSE("code done?">=1,0,(IF THEN ELSE(XIDZ(code that's bad, code, 0)>=0.25, \0,1)))

The function compares the percentage of bugs in the code completed so far with a constant *tolerance percentage*, in this case 25% (equivalent to setting the constant k = 0.25 in the Monte Carlo model). *willingness to tolerate bugs* evaluates to false when the percentage of bugs in the code exceeds the tolerance percentage or when the code has reached its target size.

The new structure affects the flow *'remove bugs'* as shown in the following code fragment:

remove bugs=

IF THEN ELSE(willingness to tolerate bugs=0, 1,0)

~ *bad units/Month*

Remove bugs flow is set to zero while *willingness to tolerate bugs* is true. Remove bugs flow is non-zero while *willingness to tolerate bugs* is false. Thus the variable *willingness to tolerate bugs* acts as a switch that turns defect removal activities on and off. The *make good code* flow is reduced by the *remove bugs* flow, so the bug tolerance policy affects the flow of new work, distributing work over the two activities.

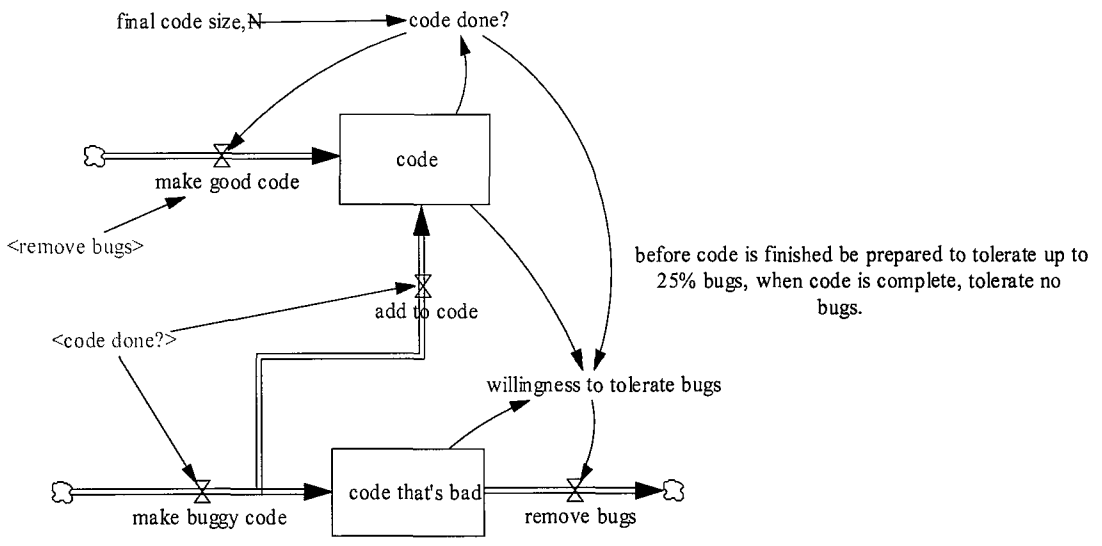


Figure 48, Systems Dynamics model evolution 3

The new structure models the choice *corr* in the Monte Carlo Model,

$$corr(pc,b,n,t,k) := (b_t \geq 1) \cdot \left[(n_t \geq N) + \left(k \cdot pc_t \leq \frac{b_t}{n_t} \right) \right]$$

and in particular, $\left(k \cdot pc_t \leq \frac{b_t}{n_t} \right)$ which randomly provides the probability distribution that *corr* will evaluate to true (choose to correct a bug) if the ratio of bugs in the code is greater than k at each tick of the simulation. The correspondence between the Monte Carlo model and the Systems Dynamics structure is good, but limited by the Systems Dynamics abstraction of a modelling a population flow, rather than discrete entities.

We can simulate the evolved Systems Dynamics model to examine the effects of the new structure on the behaviour of code and defect growth and on defect removal in comparison with both the previous model and the behaviour of the Monte Carlo model.

The following graph, Figure 49, shows a simulation where there was an initial stock of defects higher than the tolerated percentage. The effect on behaviour of the structure is that code and defect growth is slower, and no new code is produced until defects are below the tolerated percentage. This is in better correspondence to the simple process modelled in Monte Carlo methods than the previous evolution of the systems dynamics model. However the code and defects increase in a straight line rather than showing an asymptotic approach. The Systems Dynamics model has to evolve further to correspond to the simple process behaviour.

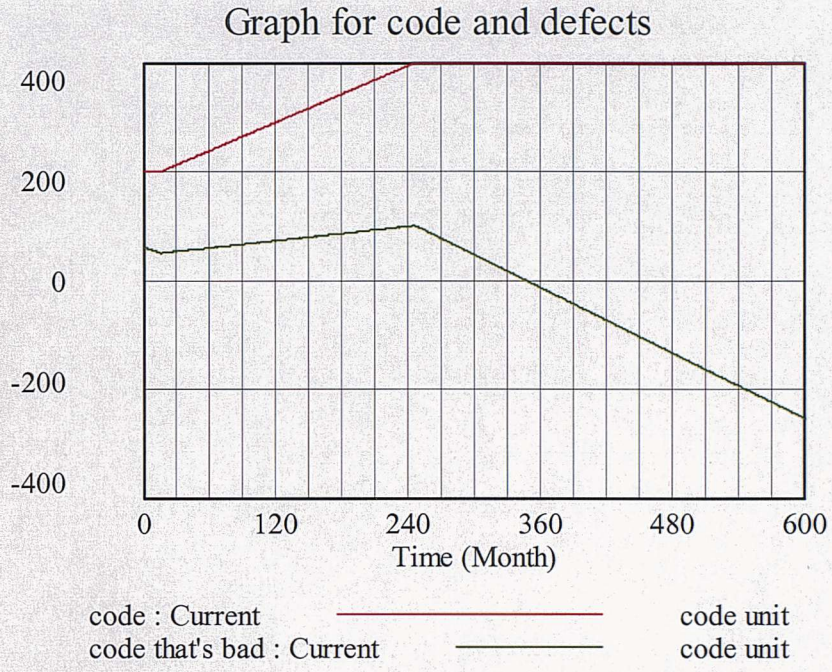


Figure 49, simulation of 3rd model evolution, graph of code and defects as time increases

5.4.4 Evolution 4, Monte 8

Monte 1 – 6 models use an approximation to allocate the work between making new work (good and bad) and fixing bugs. Monte 8, Figure 50, refines the approximation towards a better representation of the split between activities in the Monte Carlo model.

In the definition of the simple process, and the Monte Carlo model, the rate of production of defective code increases as the proportion of code completed increases.

The new structure in the Systems Dynamics model, proportion of code complete models the ratio of code completed to the target code size, N.

proportion of code complete=

code/"final code size,N"

~ *dmnl*

The new variable is used to refine the definition of the flow *make buggy code*, which in turn refines the definition of the flow make *good code*.

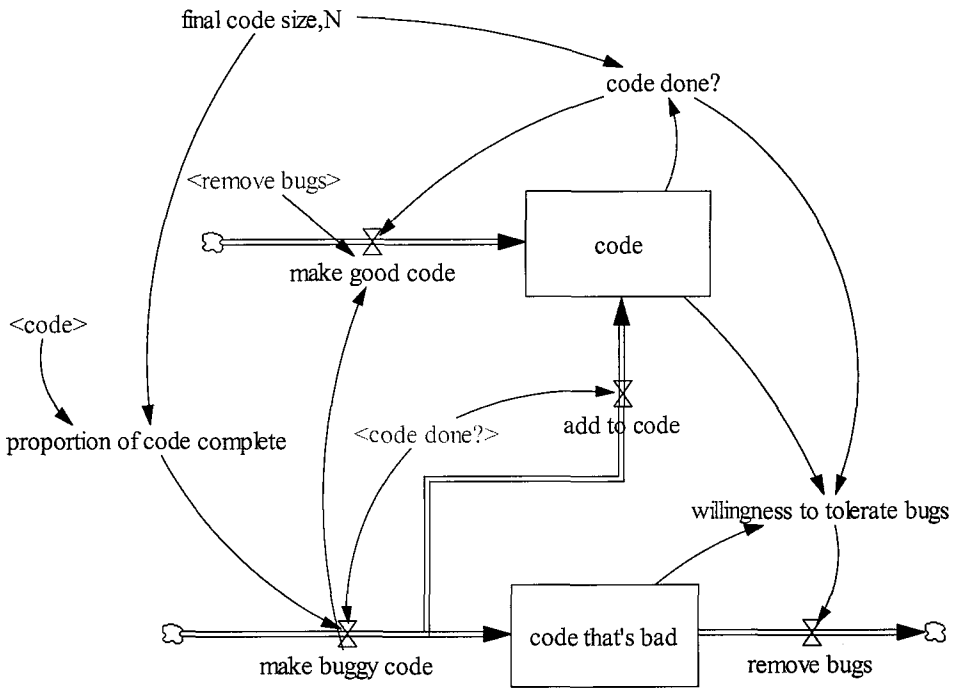


Figure 50. Systems Dynamics model evolution 4

The following equations from the Systems Dynamics model define the dynamic split between code production activities.

make buggy code=

*IF THEN ELSE("code done?">=1, 0, (0.3+ (0.65*proportion of code complete)))*

~ *bad units/Month* ~

make good code=

IF THEN ELSE("code done?"=1, 0, (1-make buggy code-remove bugs))

~ *code unit/Month*

0.3 is the value of the constant *aa* in the Monte Carlo model which represents the minimum level of bad code production. 0.65 is value of the Monte Carlo constant *bb* which represents the variable rate of bad code production above the minimum. As the proportion of code completed increases to a maximum of 1, the rate of bad code production increases to 0.95.

The behaviour of good code production is defined by defect production and removal behaviour, in correspondence with the simple process and Monte Carlo model where it is defined by the choice not to correct a bug (in *corr*) and not to produce a defect (in *addbit*).

5.4.4.1 Exploring the behaviour of Systems Dynamics Model in Comparison with Monte Carlo Model

At this stage of evolution of the Systems Dynamic model, where we have evolved the model of the structure of the simple process, and have good structural correspondence through qualitative behavioural comparison, we can begin to use quantitative data to check correspondence further.

The Vensim model was simulated with the same parameters as the Monte Carlo Model:

Initial Bugs	60
Initial code size	200 units
Target code size	400 code units
aa	0.3
bb	0.65
Bug tolerance level at	25%

Table 5. Initial parameters for the Vensim model simulation

The Vensim model shows similar behaviour to the Monte Carlo simulation. The graph of bad code over work (or time), Figure 52, shows that initially work is done to reduce bugs in the code to below the bug tolerance threshold, then new code is added (including new bugs) and bugs are removed until the code reaches its final size. When the code is complete, work is done to remove bugs.

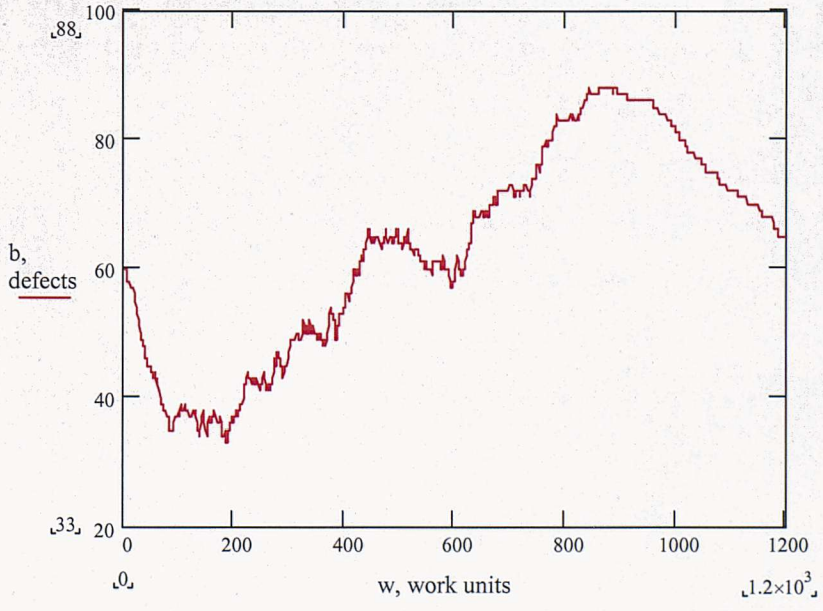


Figure 51, Monte Carlo simulation graph of defects b as work w is done

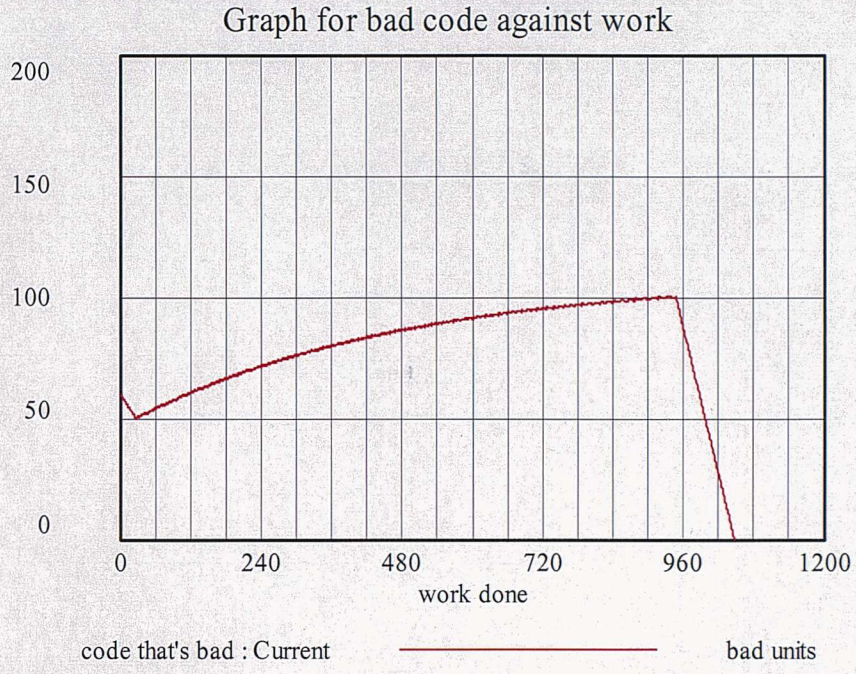


Figure 52, Systems Dynamics model simulation, graph of code and defects as time increases

The graph of code against work (time) generated by the Vensim simulation, Figure 52, shows a similar behaviour to the Monte Carlo simulation graph, Figure 51. At first, no code is added because work is done to reduce the level of bugs to below the tolerance threshold. Then new code is added until the code reaches the target size. Both graphs show an asymptotic approach to the maximum. This asymptotic approach is closer to the simple process behaviour than the previous systems dynamics model evolution. The evolutionary change has been the closer representation of the tendency to make a bug when making new code or not fix a bug in *fixbug*. The dynamic structure is a feedback relationship where the tendency increases with increasing code size. As no other behavioural changes were made, we can assume that it is the feedback relationship that has produced the asymptotic effect.

The target size is reached in a similar time for both simulations (approximately 960 work/time units), Figure 53 and Figure 54.

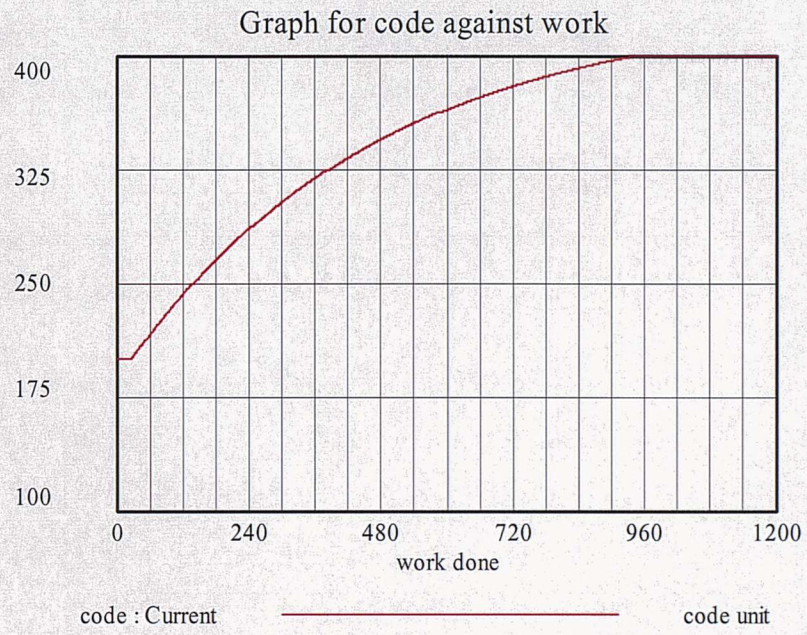


Figure 53, Systems Dynamics model simulation graph of code against work done

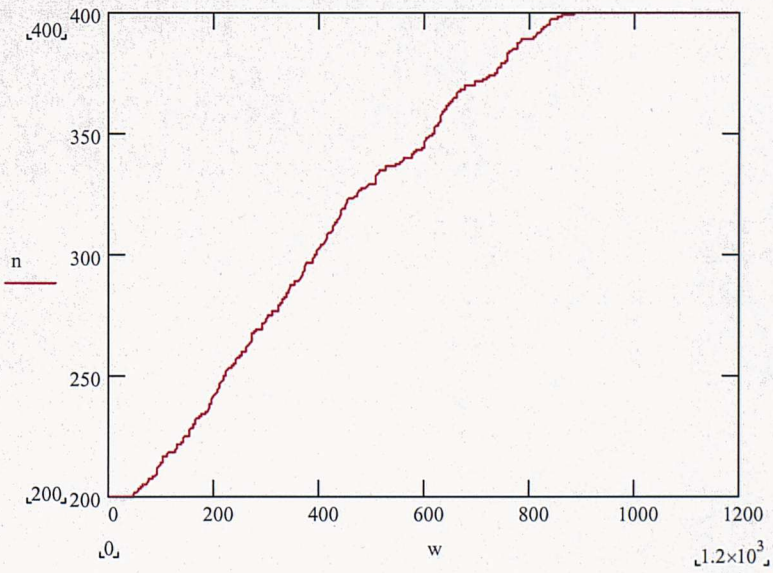


Figure 54. Monte Carlo simulation, graph of code, n against work, w

5.4.4.2 Varying the policy on Bug toleration

The Monte Carlo model simulations showed that the simple process is sensitive to the level of bugs tolerated in the code before defect removal activities were undertaken; if we have good correspondence between the models, we would expect that the systems dynamic model would show a similar sensitivity.

Therefore, we simulated the Systems Dynamics model with the bug tolerance level set at different levels for consecutive simulation runs to explore the growth of code and defects for bug tolerance levels of 10%, 25%, 50%, and 90%.

Using Vensim's graphical capability, Figure 55, it is easy to see that this process exhibits some interesting behaviour if you are willing to tolerate a high level of bugs, say 50%, or even 90%. In this case, the new code is completed faster but with a higher level of bugs. When the target size has been reached, the only activity possible at each tick of time is to fix all the bugs, thus the overall time to achieve the target size is reduced and also the time to complete code and fix all bugs. At 25% toleration of bugs, total time to reach target size and fix all bugs is approx 1100 time units. At 90% toleration the total time to reach completion is 420 time units. Were this a real process, the conclusion might be drawn that in order to finish work and remove all faults in the shortest possible time, you should make no attempts to fix faults until the target size of code has been completed, and then make no new work and only fix faults.

Tolerance of bugs in code	Max bugs in code	Time to reach target size	Time finish code and remove all bugs
25%	100	900	1100
50%	200	420	610
90%	220	190	420

Table 6. Results from simulating Systems Dynamics model Monte 8, varying the willingness to tolerate bugs, k

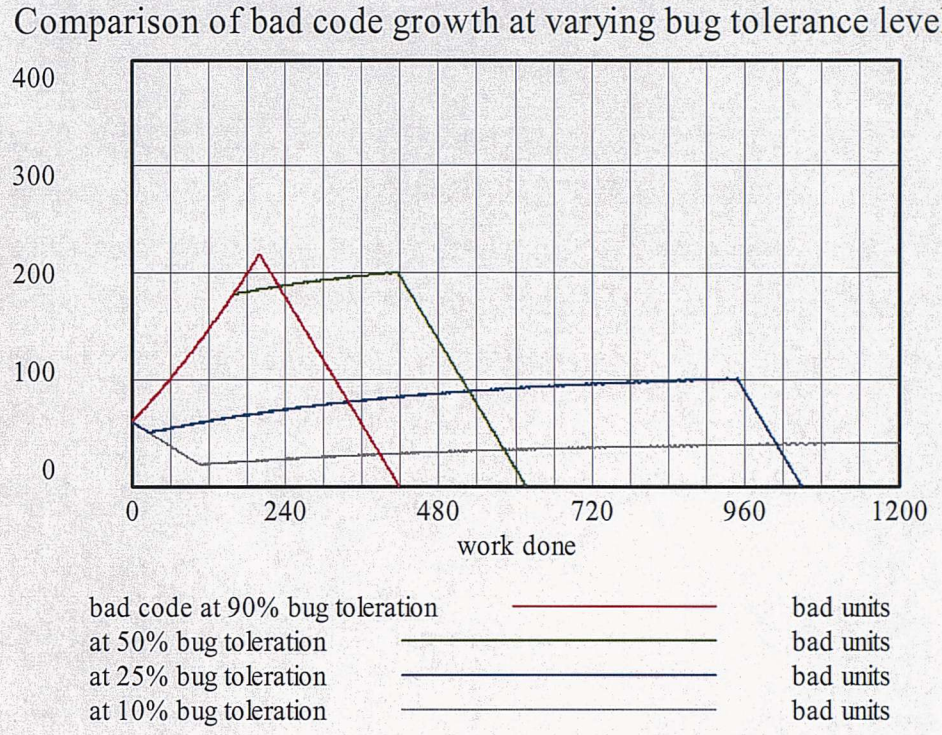


Figure 55. Systems Dynamics simulation of different quality assurance policies

The Systems Dynamic Model follows the behaviour of the Monte Carlo model until the code has reached its target size. After this point, the Systems Dynamics model shows a more rapid rate of reduction in bad code and therefore a shorter time to completion than the Monte Carlo model. The behaviour of the two models is not yet in good correspondence.

5.4.5 Evolution 5, Monte 11

Monte 11 evolves the behaviour of fixing bugs in the Systems Dynamics model closer to the Monte Carlo model, Figure 58.

On examining the flows for making and removing bugs in Monte 8, it becomes clear that the remove bugs flow does not follow the behaviour that, as the proportion of completed code increases, so does the tendency to make a bug. In the *remove bugs flow* this should become a tendency *not* to fix a bug. It may be

represented as the difficulty of removing bugs as the proportion of completed code increases.

In order to represent this properly, the *remove bugs* flow was changed to differentiate the behaviour before the code reaches its target size, when the *make buggy code* flow suppresses the effectiveness of code fixing, and the behaviour afterwards. The variable *tendency to make a bug* wraps up the difficulty factor in the previous model, Monte 8, (*0.3 + (0.65*proportion of code complete*)), and defines the minimum level rate of defects as *aa*, and the variable rate as *bb*.

tendency to make a bug=

*(aa +(bb*proportion of code complete))*

The new definition of the remove bugs flow is:

remove bugs=

IF THEN ELSE("code done?"=1, (1-tendency to make a bug),

IF THEN ELSE(willingness to tolerate bugs= 0, 1, 0))

~ *bad units/Month*

The code flows are similarly changed to correspond more closely to the simple process:

make buggy code=

IF THEN ELSE("code done?"=1, 0, tendency to make a bug)

~ *bad units/Month*

make good code=

IF THEN ELSE("code done?"=1, 0, (1-make buggy code-remove bugs))

~ *code unit/Month*

A Systems Dynamics model Figure 56, of effort allocation and Figure 57, (Vensim causes strip feature) show more clearly the effect of defect creation and removal on the productivity of the process. Effort must be allocated between making new code and defect removal before code is complete. The productivity of effort for new work is further reduced by the effort spent on new bugs.

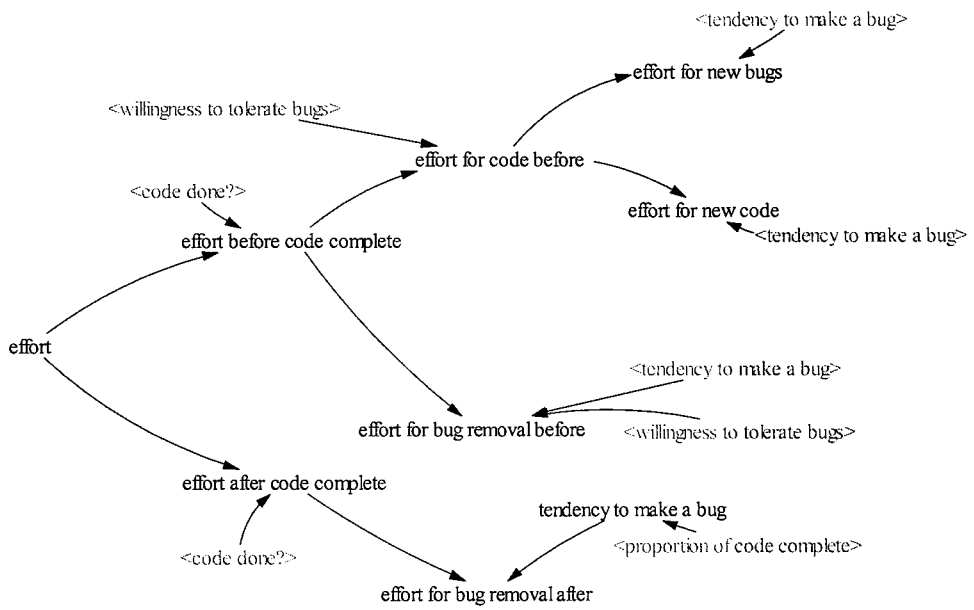


Figure 56, Systems Dynamics model of effort allocation

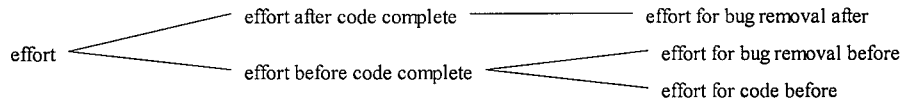


Figure 57. Causes strip for Systems Dynamics model of effort allocation

Work in Monte 8 is represented by adding together all of the code flows; however, because the tendency to make a defect varies, the total code flows at each tick of time is not equivalent to a work unit, understating the amount of work done. In

Monte 11, I have added a *Work Done* stock, so that at each *dt*, one unit of work is added. This is a closer representation of the Monte Carlo model, where at each tick of time, a work unit is added.

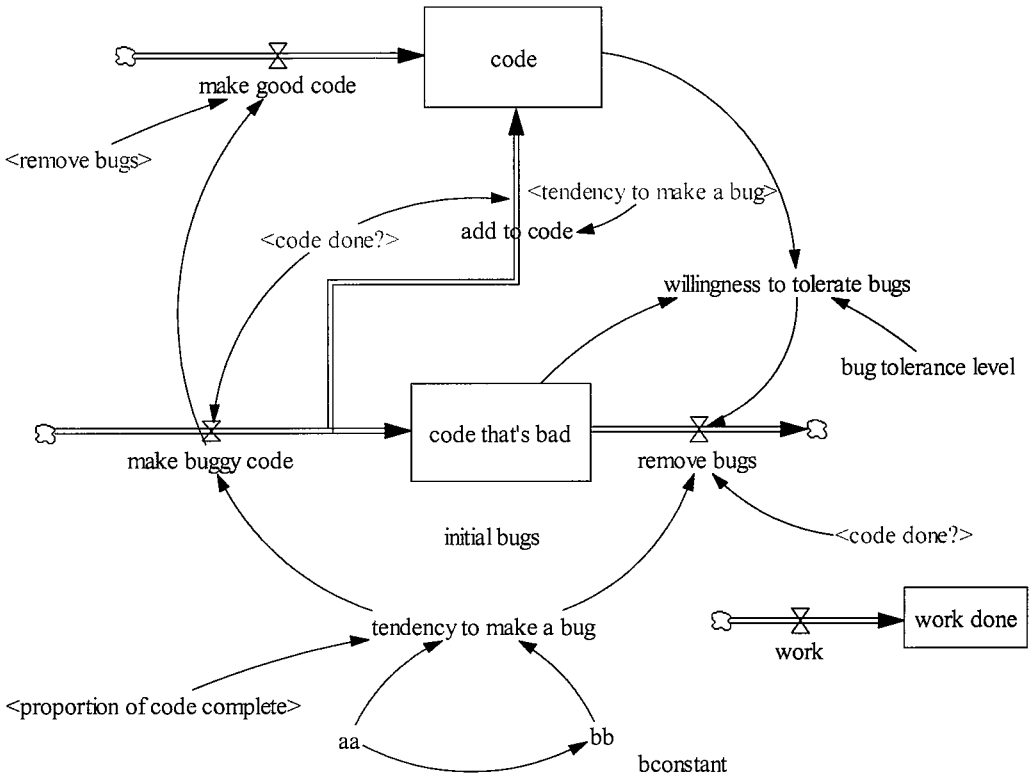


Figure 58. Systems Dynamics model, Monte 11

5.4.5.1 Causes strip for code and bad code

The causes strip feature in Vensim allows us to see which flows and variables create any particular stock. Figure 59 shows the flows and variables that create defects.

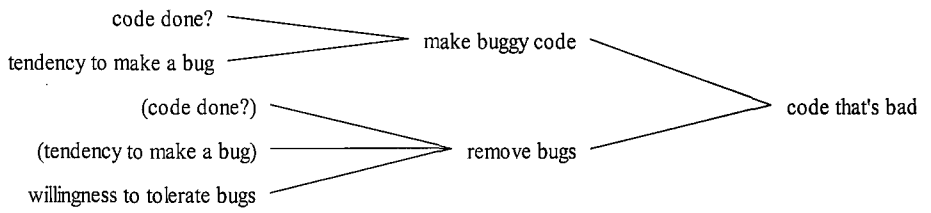


Figure 59. Systems Dynamics causes strip for defective code, Monte 11

Code that's bad is caused by two flows, *make buggy code* that creates defects and *remove bugs* that depletes the stock of defects. We can see two feedback relationships that control the flow *make buggy code*, firstly goal seeking behaviour to stop the flow after achieving a target size (*code done?)* and secondly the *tendency to make a bug*, dependent on the proportion of code completed so far. Remove bugs is also controlled by *willingness to tolerate bugs*.

The causes tree for code, Figure 60, shows that it is caused by two flows, which both add to the stock, *make buggy code* and *make good code*.

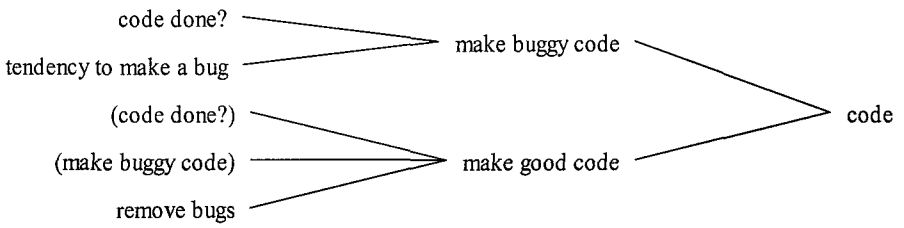


Figure 60. Systems Dynamics causes strip for code, Monte 11

5.5 Exploring the Simple Process with Systems Dynamics

Two aspects of the process affect its dynamic behaviour, firstly the willingness of the process to tolerate bugs in the code, and secondly the efficiency of the production process in terms of making new code and fixing bad code, defined in the model as the tendency to make a bug.

5.5.1 Simulation 1 varying the willingness to tolerate defects in the code

The graph, Figure 61, shows the growth of bad code for policies of tolerating 10%, 25%, 50% and 90% defects in code. Figure 62 shows an equivalent graph for code growth.

The results from simulating the model now show bad code fixing at a rate closer to the results from the Monte Carlo model. Without the 'noise' generated by the probabilistic choice, it is easy to see how the bug tolerance policy affects the rate of bad code production and fixing and of code completion.

The graph for bad code against work at the 25% tolerance level shows an initial decrease in the level of bugs because the bugs present in the code at the beginning of the simulation are already in excess of the tolerated level. Thus work is allocated to reducing the bugs to below the tolerance level before any new work is done.

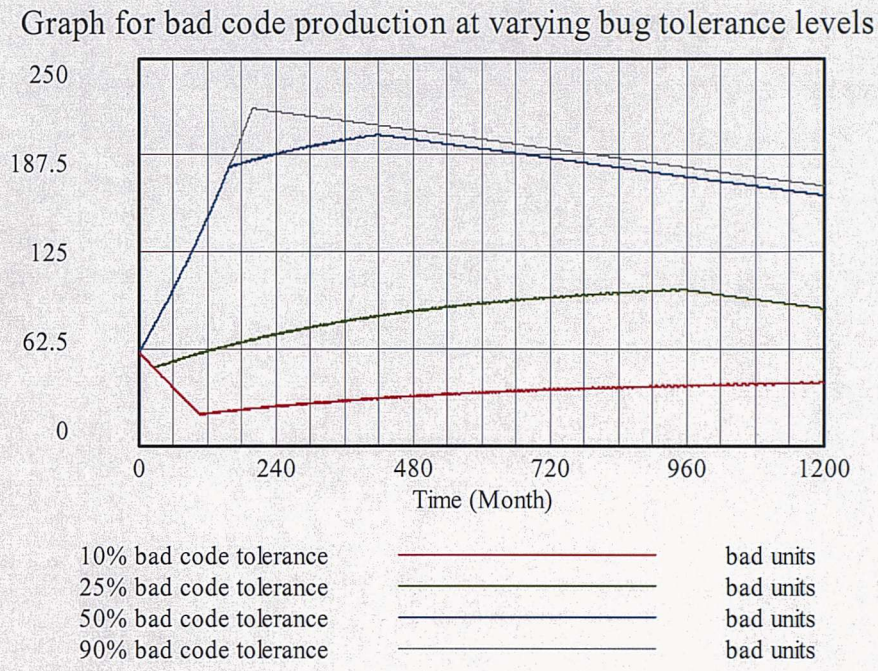


Figure 61. Systems Dynamics simulation exploring the effects of different defect toleration policies on bad code production

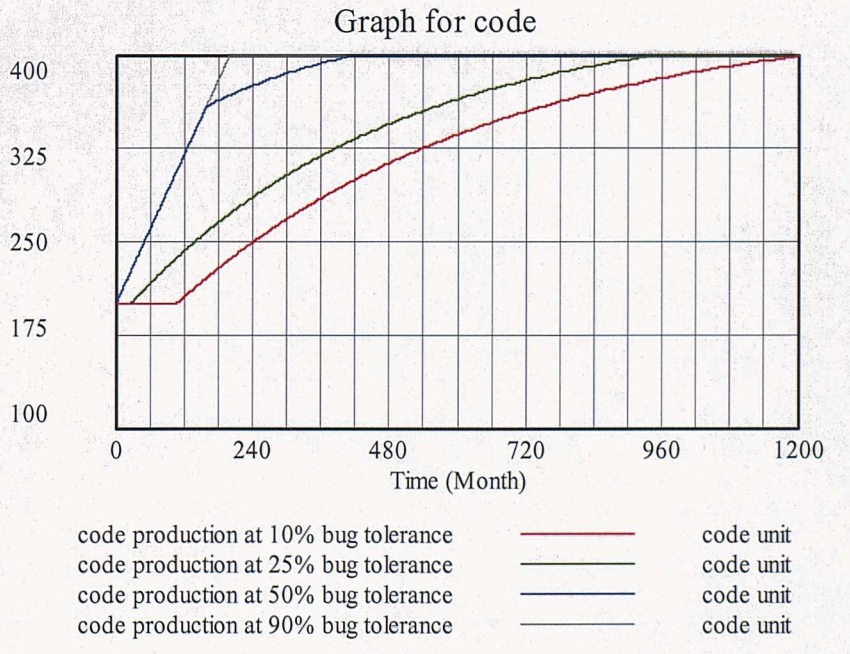


Figure 62. Systems Dynamics simulation exploring the effects of different defect toleration policies on code production

Code growth and bug growth show an asymptotic approach to the maximum. The maximum proportion of bad code is approx 25%, reached as the code reaches target size after 854 ticks (equivalent to work units), The bad code reduced to 22% by continued bug fixing until the end of the simulation.

A more extreme policy of tolerating only 10 % bugs shows a low rate of bad code growth but the bugs are never completely removed and the code only reaches the target size at the end of the simulation time bound. It takes six times longer to reach target size than if a policy of tolerating 90% bugs is used but only 10% of the code is bad.

The graph for bugs against work at the 50% tolerance level shows an inflection at the point where the tolerance level is exceeded and work to correct bad work begins. At this point, new work slows until the target size is reached.

At the 90% tolerance level, the code is almost finished before the tolerance level is exceeded. Fixing bugs begins when the code reaches its target size.

In contrast to the model, Monte 8, where a high bug tolerance caused an early completion of the code with all bugs removed, simulating Monte 11 shows a different behaviour. In this model, if the policy of 90% bug toleration is followed, although the target size is reached rapidly, the proportion of bad code is high (more than 50%) and only reduces to 42% of code by the end of the simulation. The policy of 50% bug toleration fares little better, the time to reach target size doubles and the proportion of bad code is still 40% at the end of the simulation.

5.5.2 Simulation 2, varying the efficiency of the process

In the last simulation experiment, we looked at how the policy of bad code tolerance affected the completion of code and the quality of the code in terms of the proportion of code that was bad. During these experiments the constants *aa* and *bb* that set the minimum and maximum efficiency were set not varied (0.3 and 0.95). In this set of simulations, we vary *aa* and *bb*. The graph Figure 63, shows

the range of code production behaviour when we vary the process efficiency for the same willingness to tolerate bugs and initial defects. The grey graph line shows code growth with high process efficiency. Initial defects are removed quickly because the defect removal process is more likely to fix a bug. The growth of defects is low, Figure 64, with a maximum of 65 because the tendency to make defective code increases only slightly as the proportion of completed increases. All defects are removed after 290 work/time units, again because of the efficiency of the defect removal process. In comparison the red graph line shows a process with low process efficiency; the minimum rate of defective code is 50% which rises to a maximum 95% when code is near completion. In this case defects rise to 100 and the defect removal process fails to remove the bugs before the end of the simulation.

The simulation was set up with the following values:

Graph name	Graph colour	Minimum rate of defects, aa	Maximum rate of defects, bb
Monte11-a5b95	red	0.1	0.1
Monte11-a5b7	green	0.5	0.7
Monte11-ab	grey	0.3	0.95
Monte11-a1b1	blue	0.5	0.95

Table 7. Monte11 simulation constants

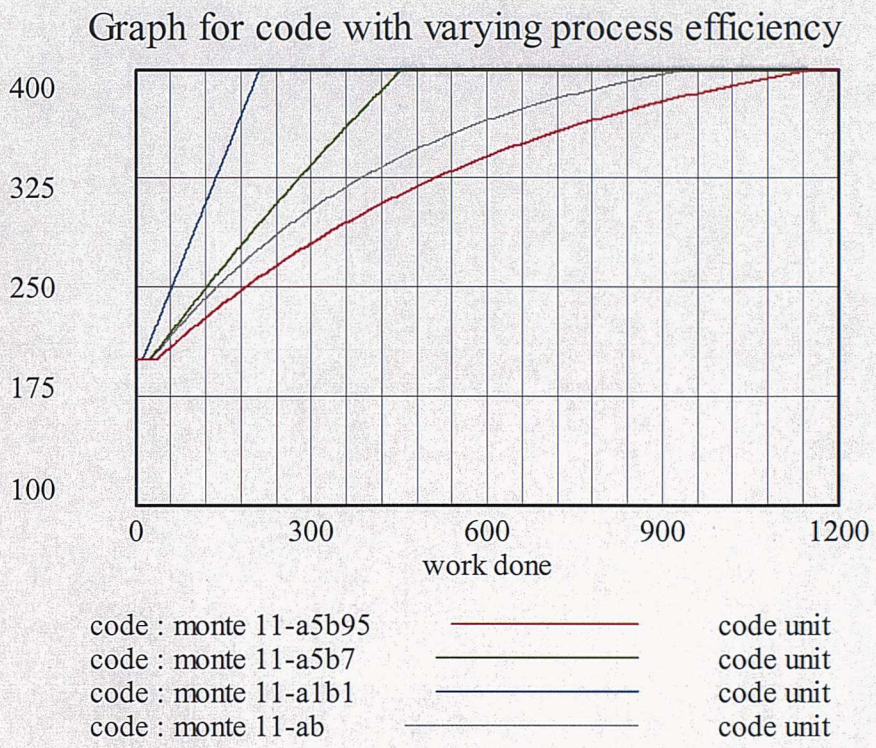


Figure 63, Systems Dynamics simulation results for code production with varying process efficiency

A graph of code growth from the same set of simulations shows the effects of varying the process efficiency on code production. As we have seen from the defects graph (blue graph, aa = 0.1 and bb = 0.1), a highly efficient process completes the code quickly, whereas the inefficient process (red graph, aa = 0.5 and bb = 0.95) takes nearly six times as much work to complete a product of the same size and shows an asymptotic approach to code completion.

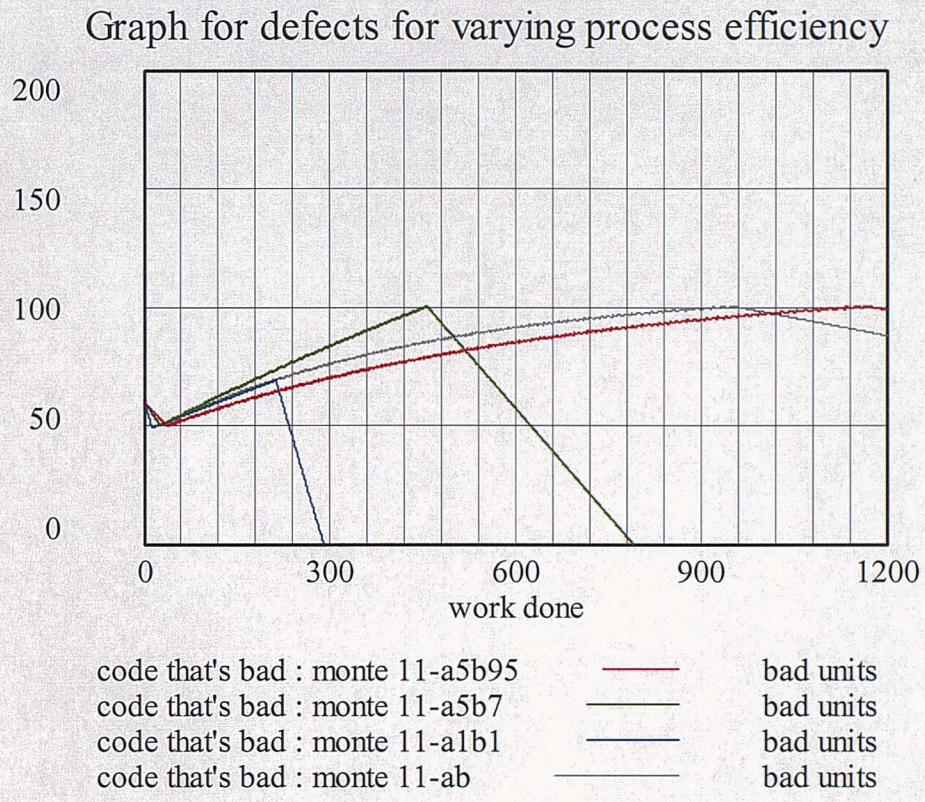


Figure 64. Systems Dynamics simulation defect production with varying process efficiency

5.6 Conclusion

In this chapter we showed an evolutionary method of Systems Dynamics modelling where feedback at each evolution from comparing the behaviour and structure of the model against real world behaviour is the dynamic that generates a closer correspondence in the next evolution of the model.

In this example, we have defined a simple software development process to provide the real world behaviour to be modelled. We defined the simple process as a set of probabilistic choices of activities that could be simulated using Monte Carlo methods to provide qualitative and quantitative visibility of the behaviour.

Comparative simulation results from the Monte Carlo model show that there are still differences between the last evolution of the Systems Dynamics model and the Monte Carlo model in terms of the quantitative results from simulations although the qualitative correspondence is strong. Table 8 shows results from Monte Carlo simulations for variable willingness to tolerate defects, k. The results are average values from ten simulation runs, and comparable results from the Systems Dynamics model simulations.

The results from the Systems Dynamics model show fairly close correlations between maximum and residual defects and work to complete target size for k = 10% and k = 25%. The results for higher tolerations of defects show the difference between the behaviour of the two models widening.

k	Maximum defects		Residual defects		Work to complete N, 400		Final Quality, P	
	Monte Carlo	Sys Dyn.	Monte Carlo	Sys Dyn.	Monte Carlo	Sys Dyn.	Monte Carlo	Sys Dyn.
0.10	38	40	36	40	1200	1200	0.91	0.9
0.25	92	95	76	80	852	950	0.81	0.79
0.50	153	190	120	150	550	460	0.70	0.6
0.90	190	22	146	170	346	310	0.64	0.59

Table 8. Results from Monte Carlo and Systems Dynamics simulations of the simple process, varying k.

The behaviour of the two models when assessed qualitatively seemed to be in close correspondence, but when we examine the results from simulations of both models we can see discrepancies in quantitative behaviour.

The questions that we can examine to bring the models into quantitative correspondence are:

- Are there modelling errors in the Systems Dynamics model?
- Are there failures of understanding of the behaviour of either the Monte Carlo model or the Systems Dynamics model?
- Does the Monte Carlo method more closely represent the simple process than the Systems Dynamics model?
- Is the difference caused by the different abstractions in the modelling paradigms?

One clear difference is that in the Systems Dynamics model, we are representing activities as flows; the Monte Carlo method model represents the activities as probability distributions. In some ways, the Monte Carlo representation is easier to understand; we can visualise a software implementer either making good or bad code according to process efficiency or using the quality measures from the code to decide when to remove defects. It is more difficult to abstract from this visualisation of a single action for every tick of process time to flows of code stocks that increase or reduce at each tick of process time. The Systems Dynamic simulation results are closer at the extreme values of willingness to tolerate defects and process efficiency. The Systems Dynamics uses the exact values of these constants whereas the Monte Carlo model randomly selects from a probability distribution; this may be the cause of the qualitative differences between the results from the models in the two paradigms. We should sample values for these constants to produce values that more closely represent the probability distribution used in the Monte Carlo method.

We have shown how we can use evolutionary model building to produce a Systems Dynamics model of the simple process, with successive models making an evolutionary step closer to correspondence to the real world process.

Figure 65 shows the evolutionary path to the successful final model, but there were alternative paths that could have been followed, and models that were discarded. Unsuccessful models are shown on the diagram with a red cross.

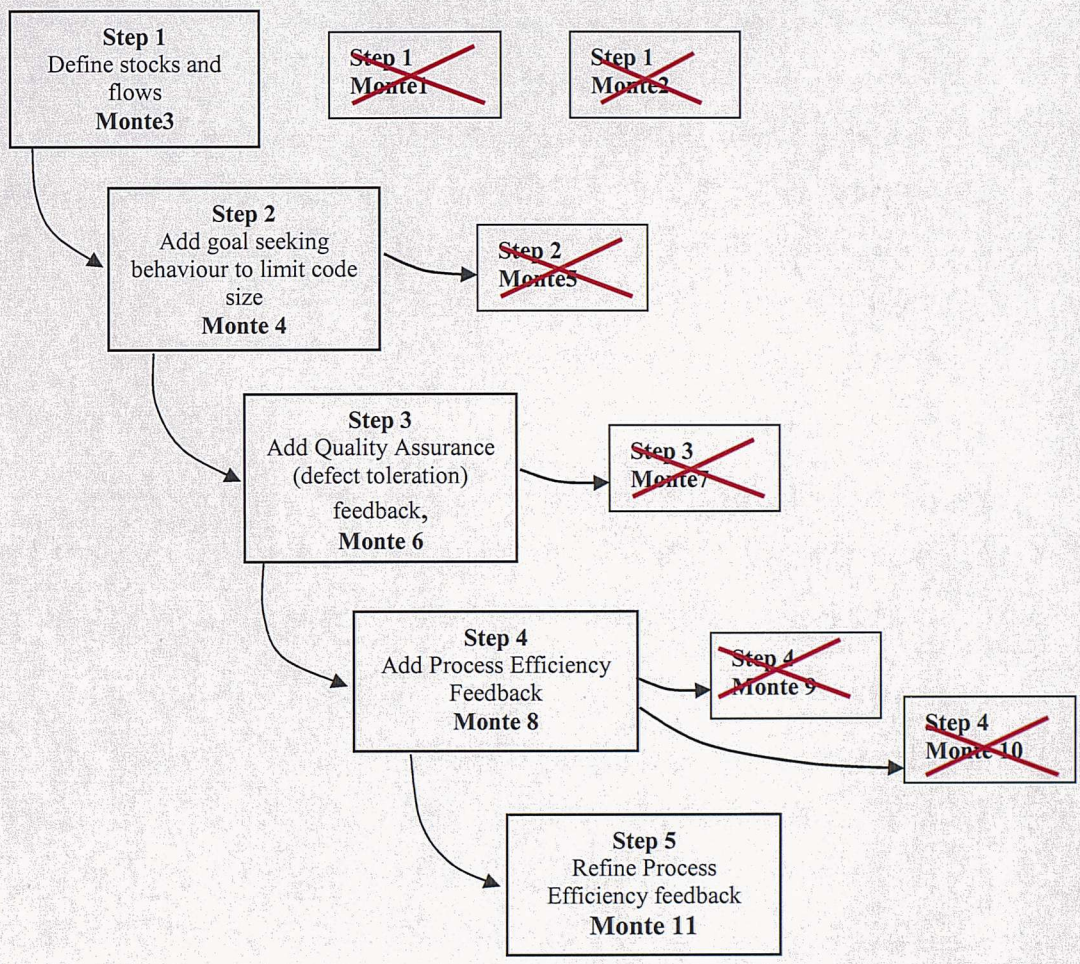


Figure 65, Evolutionary path of Systems Dynamics models of the simple process

The equivalent process to natural selection to identify the successful models is the check for correspondence between the Systems Dynamic Model and the Monte Carlo Method model of the simple process. At each evolutionary step quantitative and qualitative analysis of the results of simulations identify the model with the closest correspondence, which becomes the next evolution.

Careful analysis of the real world process, together with analysis of qualitative results from the real world process and simulations of the model provide the basis for understanding the problem domain. Qualitative analysis can be used to identify similar patterns of behaviour; for example, whether code production stops when the code has reached its target size, or defect removal continues when code production ceases. Correspondence of overall patterns, for example, asymptotic approach to maximum values, can indicate that feedback loops have been identified. However, analysis of comparative quantitative simulation data is necessary to enable the precise feedback and control mechanisms to be understood and replicated. The Monte Carlo method simulations of the simple process provided qualitative results for comparison with qualitative results from the Systems Dynamics Model.

The investigation shows how systematic, evolutionary modelling using qualitative and quantitative data enables a model to achieve close correspondence between the model and the real world behaviour of which it is an abstraction.

In this set of simulations, the simple process provides no real data; quantitative and qualitative data was provided by a model of the process modelled by Monte Carlo methods. This shows that we were able to use an alternative model representation of the simple process to provide external visibility into dynamic behaviour to provide a basis for Systems Dynamics modelling.

Chapter 6

Simulation Experiments in Modelling Software Processes using Components

A prime requisite is to use only a small sound and rigorous set of symbols or building blocks... it is important that they are fully understood in isolation before describing how they are linked in methodological terms for model construction and analysis – [Wolstenholme 1990]

The motivation for this chapter is to show how components of software development processes may be identified and modelled using System Dynamics.

- To develop models of software processes using process components
- To build incremental models exploring the effects of feedback relationships.
- To show that Systems Dynamics can be used in a systematic way to examine the causes of software process behaviour and predict the behaviour of those processes.

In this investigation I have used the controlled experimental method described by Zelkowitz and Wallace as simulation; modelling the behaviour of the environment

for certain variables and using the results to examine the validity of an hypothesis [Zelkowitz and Wallace 1997].

6.1 Components

The term 'component' has been used to describe both elements of software and also of processes but we should examine what it means to use the term. In software terms component has been used to describe anything from a GUI button to a complete system, without any agreement on the meaning of the term except in an individual context (for example Microsoft COM). The looseness of the definition varies from 'a useful package' to a fully standardised plug in part. However, for the term to become useful, it is important that we have a definition that is well understood and is not limited to the context of a particular implementation architecture. Recent discussions within the object oriented community have focussed on this issue and this definition of component has emerged from the work of Szyperski [Szyperski, Gruntz and Murer 2002]. A component is:

- A unit of functionality
- A unit of deployment and replacement
- Composable
- Usable by other software elements whose authors are unknown to the component's authors
- A software element conforming to a defined component standard
- All of these, but used as black box, targeted at execution (whether as binary, byte or source code.

An alternative view suggested by Bertrand Meyer [Meyer 2000] is similar, but adds criteria for specifications of functionality and dependencies and leaves out the criteria for conformance to a defined component standard:

- May be used by other software elements (clients)
- May be used by clients without the intervention of the component's developers
- Includes a specification of all dependencies (hardware and software platform, versions, other components)
- Includes a precise specification of the functionalities it offers
- Is usable on the sole basis of that specification
- Is composable with other components
- Can be integrated into a system quickly and smoothly.

A component defined from these viewpoints, has a specification, has an implementation, can be composed, can be deployed and may conform to a standard. For a component to be useful and composable, the following requirements can be considered to be essential; to be reusable, extendable, evolvable and reliable. We should also think about how components defined in this way should be used in order to produce successful systems. When we compose components, 'plugging them together', they are only pluggable to the extent that they satisfy the specifications of what we plug them into; just because it is possible connect a component to our system doesn't mean that it makes sense to do so. The completed system must satisfy its requirements [Henderson 1998].

What can we take from this into the area of software processes and, further than that, into software process modelling? We have already discussed that in the same way that software systems are evolutionary, software processes are evolutionary systems in their own right. We want our processes to be reusable, evolvable, reliable and extendable as described in Watts Humphrey's process improvement model for U, W and A processes [Humphrey 1990].

If we consider our processes in component terms, we can evolve processes by evolving individual components, by adding new components or replacing

components and we can change how we connect the process components together, for example, connecting them in series or in parallel or a combination of these .

In order to improve the way we implement process evolution, we use modelling and simulation to understand and predict the behaviour of processes and process improvements. In the same way that our product can be structured into components, giving us benefits of reusability and an evolutionary capability, our process model may be similarly structured, so that we can model components and then compose the components to make the process model.

6.2 An example of using components to build a Simple Process

In this set of models we will describe a simple process component that produces software development products. These may be lines of code, components from which larger components will be built or any other software deliverable. We will build the simplest possible process that can be justified.

We will show how the model evolved and how each structure of the model can be justified in terms of the observed behaviour.

6.2.1 Simple 1

The first model of a process, Simple 1 Figure 66, shows a stock of software components that is increased by an activity, *input rate*, and depleted by an activity, *output rate*. This models a scenario that an implementer receives a stock of components at the input rate, works on the components at the output rate; thus depleting the stock of components to be worked on. In this case there is a limitless supply of components that can be received; designated by the cloud.

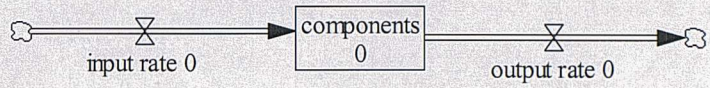


Figure 66. Simple 1, modelled in Systems Dynamics

If the *input rate* equals the *output rate*, then the stock of software *components* will maintain its initial level, in a steady state. If the *input rate* exceeds the *output rate*, then the level of components will grow. If the output rate exceeds the input rate, the stock decreases. (Figure 67)

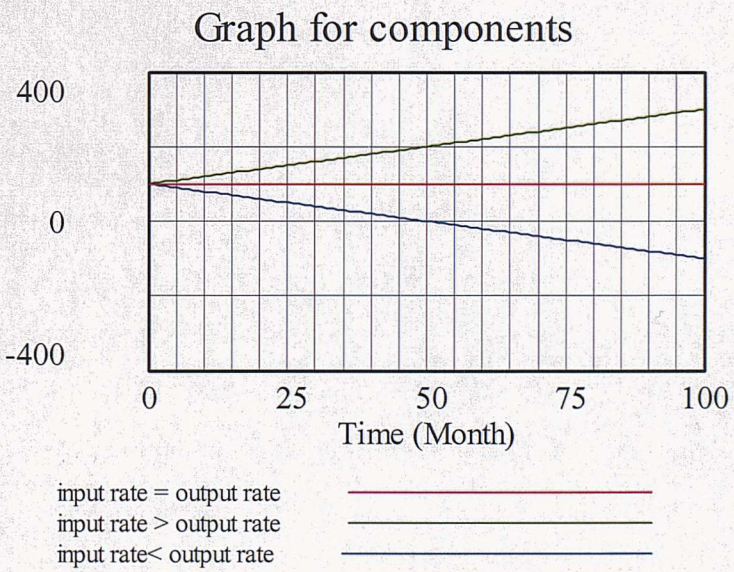


Figure 67. Graph of software components stock from Simple 1

If this simple process were to be considered as a process component, one of the properties that we would want from our component is composability; we should be able to connect one simple component to another, either another simple component or any other process component from our defined set. Simple 1 does not provide easy composition; For example, in a serial composition, an output rate would have to be connected to input rate, which is not possible without an intervening stock. (Figure 68).

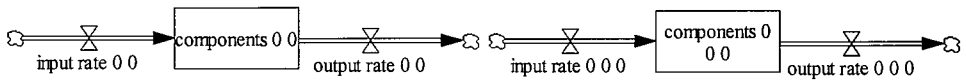


Figure 68. Serial composition of two simple 1 process components

6.2.2 Simple 2

The second simple model, Figure 69, shows another interpretation of the development process. We have a stock of software components that we must transform by an activity, *work rate* into a stock of finished components, *finished.* The software components might be requirements that we must transform into component build plans, or components to be assembled. This is a conserved flow.

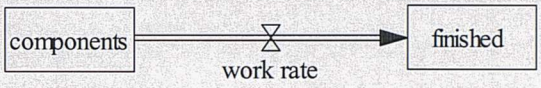


Figure 69. Simple 2 modelled in Systems Dynamics

The following graph, Figure 70, shows the effects on components and finished components when we simulate the model with an initial stock of 100 components and a work rate of transforming 2 components for each tick of time, t. After time, t=50 all of the components have been transformed, and the stock of finished components, finished = 100. (The modelling tool allows stocks to be come negative, unless explicitly prevented from doing so, hence the continuing growth of finished components after t =50).

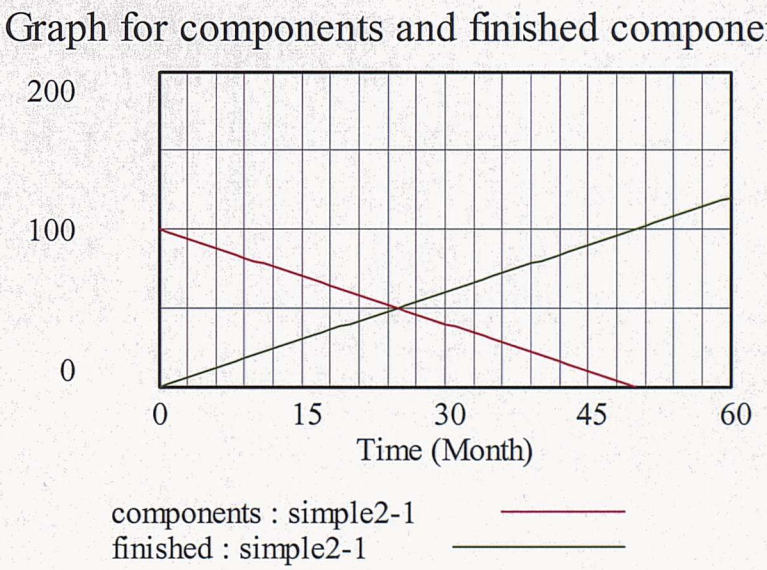


Figure 70. Simple 2, graph of component and finished components

As a process component, Simple 2 does not provide easy composition. If we wish to compose simple 2 process component instances in series, we must connect two stocks together, *finished 2* and *components 3*; this is not possible without some

intervening structure, for example, a flow. Inserting an additional flow to connect the output stock of one software component to the input stock of the other would mean that the composed model was not the same as the two process components added together.

6.2.3 Simple 3

We want our process component to be the simplest unit that structurally represents a development process and that will be composable and reusable. Feedback from comparisons of Simple 1 and Simple 2, suggests that the component model must be evolved further in order to achieve a closer correspondence to the simple development process.

Simple 1 has two flows, increasing software components (*input rate*) and decreasing software components (*output rate*), corresponding to work activities. Simple 2 has only one flow, corresponding to doing work, which decreases the stock of work to do. The Simple 2 work flow is a closer conceptual representation of the simple process than Simple 1.

Simple 1 has one stock, corresponding to software components. Simple 2 has two stocks; the software components that are to be built from and the components that are produced by the work activity. Simple 2 appears to be a plausible conceptual representation of the simple process, but in our simple process, we have a stock of software components that must be transformed by the work activity; in a series of simple process components, the finished software components become the stock of work to do for the successor simple process. This indicates that only one stock is necessary to represent the simple process.

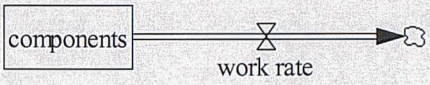


Figure 71. Simple 3 modelled in Systems Dynamics

Simple 3 (Figure 71) represents the simple process as a stock of software components that must be built from and the work activity that transforms the components into a product. The finished product is not represented as a stock but as a cloud, or sink. Simulation of Simple 3 shows the stock of components decreasing by the work rate at each tick of time as the product is made, Figure 72.

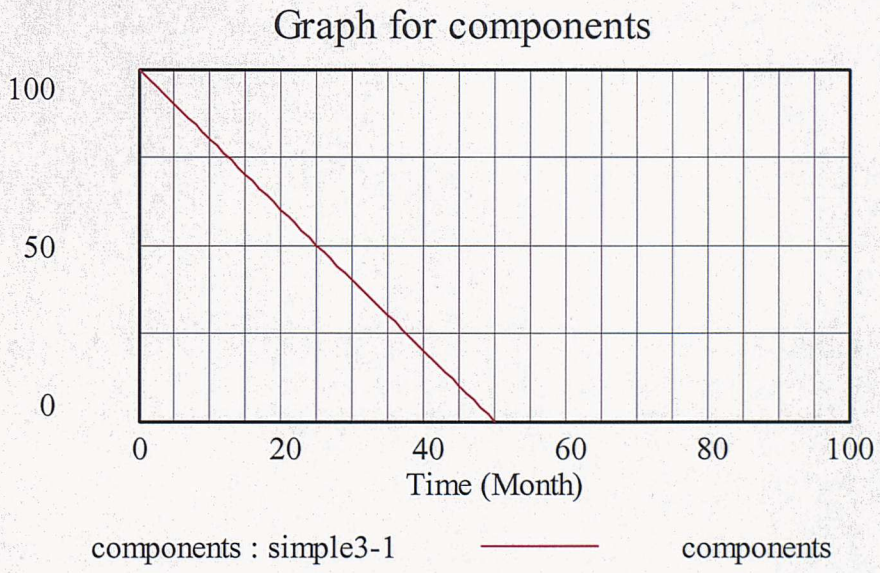


Figure 72. Simple 3 simulation of components

6.2.4 Composing Simple 3 Components

We can create a process in a systematic way easily, by composing Simple 3 process components. In the following example, Figure 73 we have a process composed of two Simple 3 process components connected serially.

The second process structure has an identical structure to the component; the stock from the second process component replacing the 'cloud' sink from the first component. No process structure has to be added or removed. We can continue to add further process components in the same way.

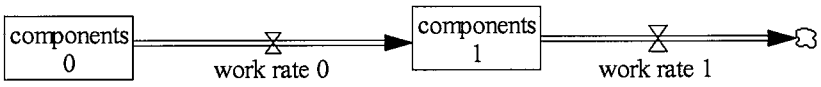


Figure 73. Simple 4, Two Simple 3 process components connected in series

6.3 Experiments in building systems development process models using components

In the following two sections of this chapter we will describe two experiments in building Systems Dynamics models of development processes composed of simple repeating process structures

- The first experiment investigates differences in perceived against predicted rates of completion of a product.
- The second experiment investigates the effect of reworking faulty software components on project schedule.

The two experiments investigate the effects of process behaviour on the completion of a product.

We will show how the behaviours may be explained by modelling the process in Systems Dynamics.

6.4 Experiment 1, perceived against predicted rates of product completion

One of the difficulties of predicting development schedules is reporting progress against plan. In order to give visibility into a process, we use product and process measurements (for example, time and resource) to report actual progress against a schedule for completion. We may adopt measures for units of product, for example, KLOC (thousands of lines of code), number of components, or other deliverable work products. We may adopt other measures for process, time in days or months, resource in number of people allocated to the project.

The simplest completion behaviour is a straight line graph for completion of type, $y = mx + c$ where y is product size, x is project time and m effort. A more sophisticated plan may expect the behaviour to be an asymptotic approach to completion [Lehman and Ramil 2002].

The usual project management technique we use to provide visibility of progress is to break work down into units and report on the completion of these units of work. By doing so, we can improve our measurement of progress, but however fine the granularity of the work break down structure, this means that we will still under report the actual amount of work complete at any moment. This is because however well structured the reporting method, it is difficult to evaluate completeness of work and what work should be reported against progress towards product completion. There will always be some work units that are complete, some in a partial state of completion and some that have not been started at any moment that we choose to report. At each reporting milestone, the tendency is to report

only completed work to be passed on to the next stage in the process. This has the effect of

- under reporting progress towards completion
- under reporting work done
- overstating effort consumed by reported complete work
- lowering perceived productivity.

By the end of all stages in the process, all work is complete, but intermediate reports of progress and effort are inaccurate.

In a process with 4 stages and five reporting mile stones, we might expect that at the first milestone, the product will be 25% complete, at the second, 50% complete and so on, until the final stage when the product will be 100% complete.

Earlier in this thesis in Chapter 4, we described the Cellular Manufacturing Process Model, (CMPM) where the work of a production cell is to assemble software components into a product, 'gluing components together'. This work includes producing build plans, unit and integration testing and delivery. At each reporting period, some components may be complete; other components may be partially assembled and may range from 0% to 99% completeness. If only complete components are reported, partially assembled products which have consumed effort will not be counted.

Work units may be partially complete because we have not yet expended sufficient effort to complete them. The work on components may not be serial; we do some work towards a number of components, all of them partially completed. We have concurrent completion of components, but a serial model of reporting. Because of the difficulty of counting completeness, as an alternative process measure, we may count resource applied for completion.

In reality we are measuring what can be passed on to the next stage, or effort expended, rather than measuring the work completion.

We will show how this behaviour may be modelled in Systems Dynamics using the systematic, evolutionary approach already described, but also how the process may be modelled by composing identical process components.

Let us examine a software development process with a number of stages; at least two and possibly many. The work at each stage may be different. In the first stage we may be gathering requirements, in the second designing a build plan. In the third assembling software components and so on until the product is ready to be shipped. Or the work at each stage may be the same, each corresponding to an evolution of a product, or component.

The first task is to identify a simple process component structure for a software development process with several reporting mile stones. If we abstract from the specific activities carried out and artefacts produced, we have a process that at each stage, some units of work are received; the stage completes its part of the work on those units, and hands on the work to the next stage. We can describe this as a simple process component repeated for each stage in the process.

We will describe how the process component was evolved and used to build processes with different numbers of stages and reporting milestones and how these models were used to explore perceived progress behaviour.

6.4.1 Perceived Progress Model 1

The first model shows an abstract representation of the process modelled in Systems Dynamics, Figure 74. The model abstracts from separate process stages with reporting milestones to a single stage with simple stocks of work items and activities that increase or deplete them.

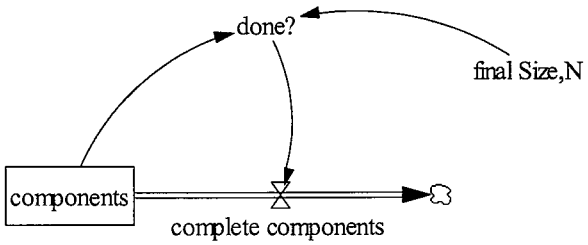


Figure 74. Systems Dynamics model of abstract process

In this example, the process has a stock of raw software components to be assembled. Work activities produce an item of completed work and drain the stock of work to be done. An abstract view of the process is that the work activity, *complete components,* builds our product from a stock of software *components*. The process has a goal seeking behaviour; the process stops when the product has reached its *final size, N.*

As we are interested in qualitative behaviour, we may choose arbitrary values for the variables and constants in the model for simulation purposes. In this case, simulating the model with an initial stock of 100 components and a work rate of 2 components per tick of time shows the behaviour of components over time is a straight line, Figure 75. At the end of time T = 50 weeks, all of the components have been completed. If we were to impose a reporting milestone at t = 25 weeks, 50 components would be complete, and 50 components would be remain to be completed.

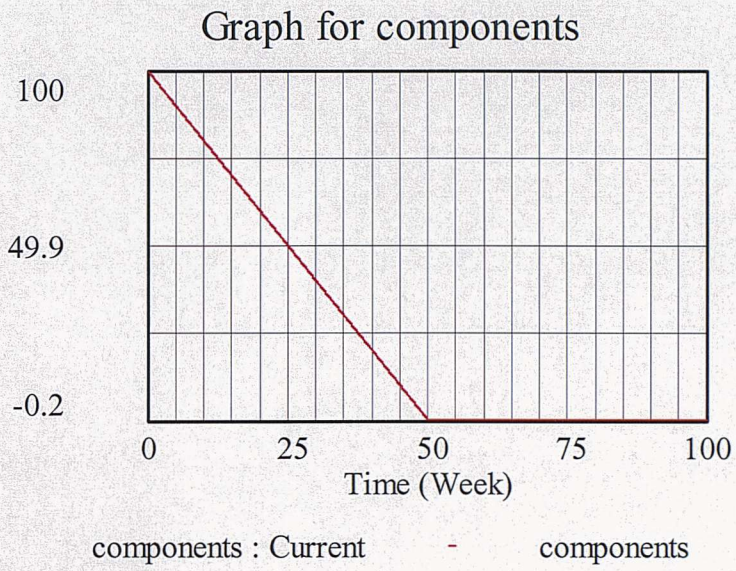


Figure 75. Graph showing software component completion from simulation of abstract model

6.4.2 Perceived Progress Model 2

The first model, a single process component, shows only the stock of work items to be built from; the work activity depletes the stock of software components at each tick of time but the model does not show completed products. The second model is an evolution to identify stocks of completed product, equivalent to a milestone. We could just add a stock, *finished*, of completed software components to the work activity in ad-hoc manner, as shown in Figure 76.

A simulation of the ad-hoc model gives the expected result that the stock of completed components increases at the same rate that the stock of work to be built from decreases, Figure 77. When the stock of work to be done is empty, at time T, all of the work items have been transformed and the stock of completed work has reached its final size.

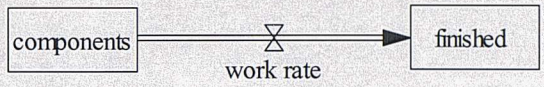


Figure 76. abstract model with ad-hoc completion milestone

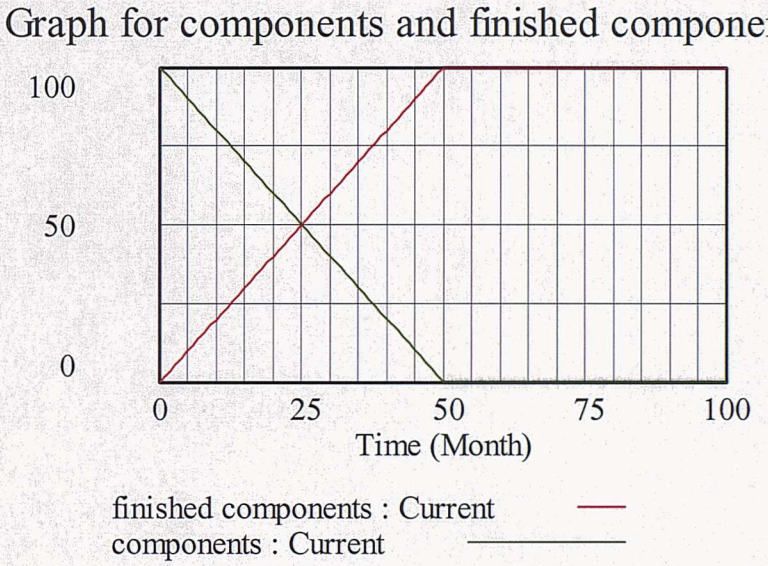


Figure 77. Graph of completed software components from ad-hoc model simulation

Whilst this model corresponds to the expected behaviour, the model does not repeat the process structure, and therefore does not fulfil our criteria for modelling in a systematic way, using repeated components of models.

The second abstract model shows a second milestone added to the model in a systematic manner, Figure 78. The second milestone has an identical structure to the first process component. The components are composed serially together; the stock from the second process component replacing the 'cloud' sink from the first

component. Milestone 2 represents the completed product from the *complete components* activity. The product is finished when all of the software components have been transformed, and *milestone 2* has reached its *final size, N 0.* Composing the two identical structures, also adds another work activity, *complete components 2.* In our process, there is no second work activity; rather than removing the activity, as this would mean that our model is no longer composed of identical structures, we will set the activity flow rate to zero.

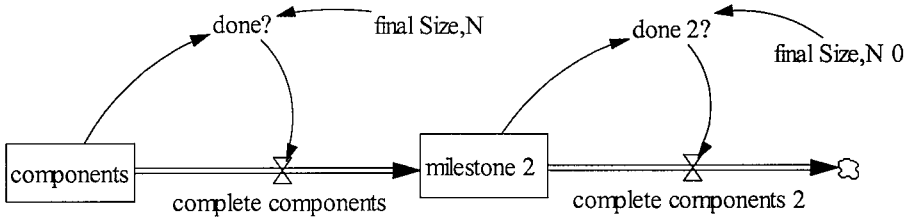


Figure 78. Systems Dynamic model composed of two identical process structures

We can show that this has the same effect as adding a *finished component* stock to the complete components activity by simulating both models and comparing the results from the two versions. The simulations show that the stocks of work to be completed and finished work have the same behaviour as shown in Figure 77 and Figure 79 (there is a rounding difference in the stock size).

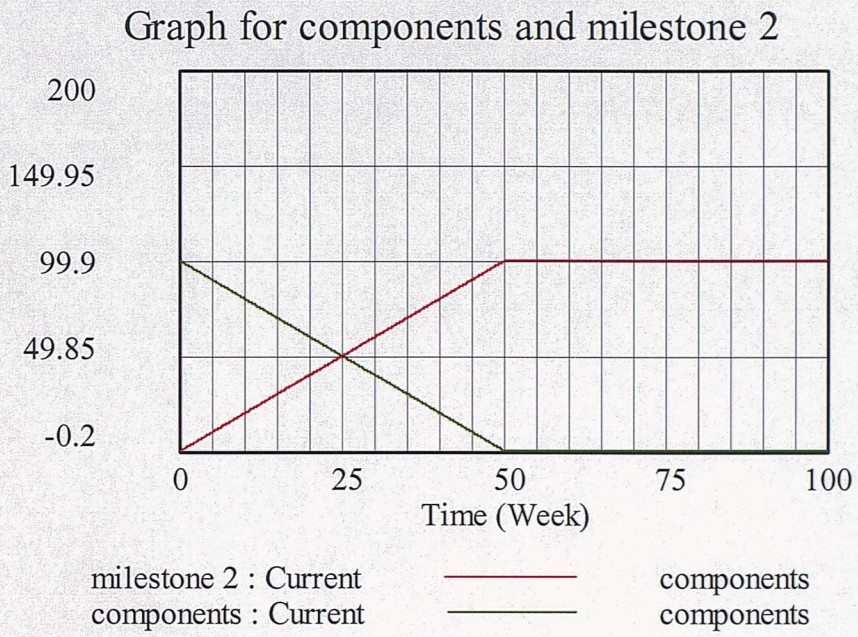


Figure 79. Graph of software component completion from simulating a two component abstract process

We now have a mechanism for making a process with any number of reporting milestones to show an increasingly fine granularity, using simple repeating process components, serially composed. The process can be stopped by setting the final work activity rate to zero.

6.4.3 Perceived Progress Model 3

The process described so far, models the perception of a perfect process; progress towards the completed product is directly proportional to the work activity applied to the work to be done, and all of the work is reported.

The behaviour that we wish to examine is that the completed work reported at each milestone is understated.

This may be described as a delay in reporting, because eventually all work will be reported as complete or it may be described as proportionate effort, where the effort has been applied proportionately to all of the components to be built from, rather than each product in turn.

Each process stage has the characteristic that as the work nears completion it becomes easier to complete software components and send them on to the next stage.

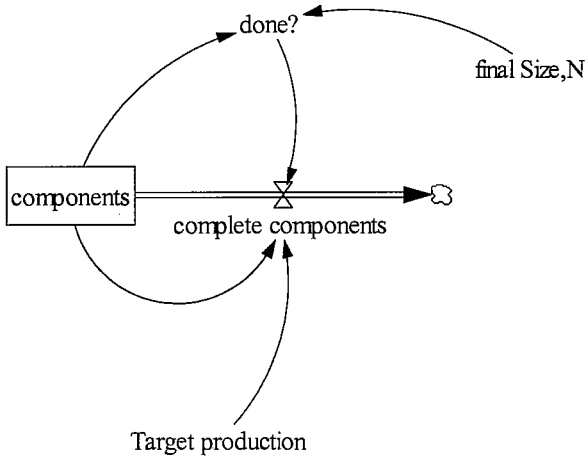


Figure 80. Evolved Systems Dynamics abstract model 3 with feedback

We can evolve our simple abstract model to model the perceived rate of component completion instead of a predicted linear completion by adding feedback from the software components remaining to be built to the work completion activity, Figure 80. This indicates that the work activity is applied proportionately to the work that is required to be done. This is shown by the feedback loop from the stock of work to be done, components, to the *complete components* activity.

The *complete components* work activity now relates to the perceived completion rate.

Examining qualitative results from simulating the abstract process, Figure 81, shows us that the *complete components* activity has a graph with an asymptotic approach to a zero rate.

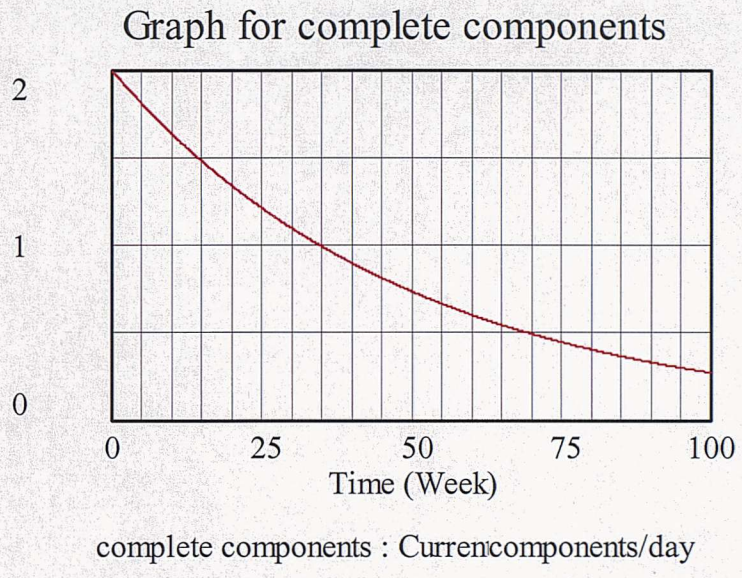


Figure 81, graph for complete components activity from simulation of model 3

6.4.4 Perceived Progress Model 4

As in the previous example of composition, we may refine the abstract process to show completion of software components by repeating the abstract process component structure. We add the second milestone by replacing the output 'cloud' or sink of the work activity, complete components, with an identical process component, Figure 82. The process finishes at the second process component; the work is not passed on to another stage, so we may set the work activity of the second process component *complete components 2* to zero. The model now has the structure of a process that has one stage with a start milestone and a completion milestone.

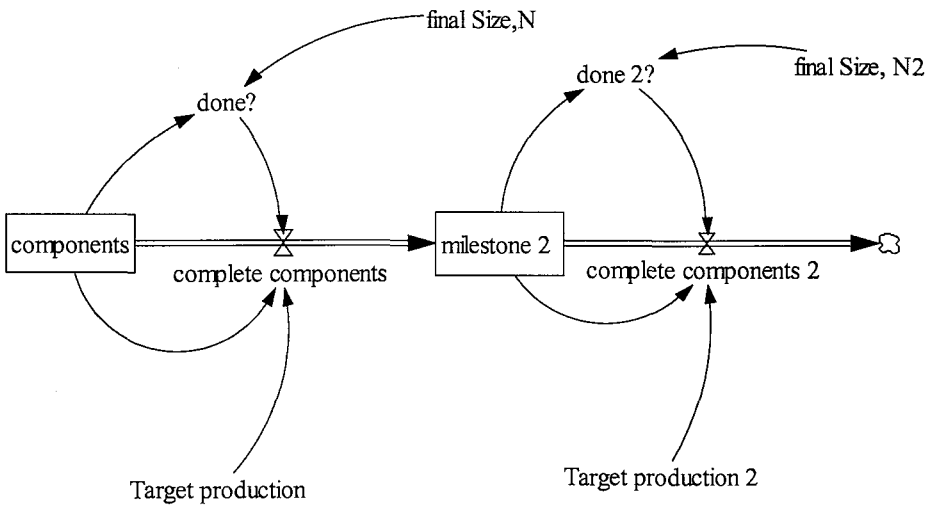


Figure 82. Systems Dynamics model of one stage process composed of two process components

When we simulate the process, the graph for perceived completion of product at milestone 2 (Figure 83) now shows an asymptotic approach to the final size, rather than a straight line. Components are reported as complete later.

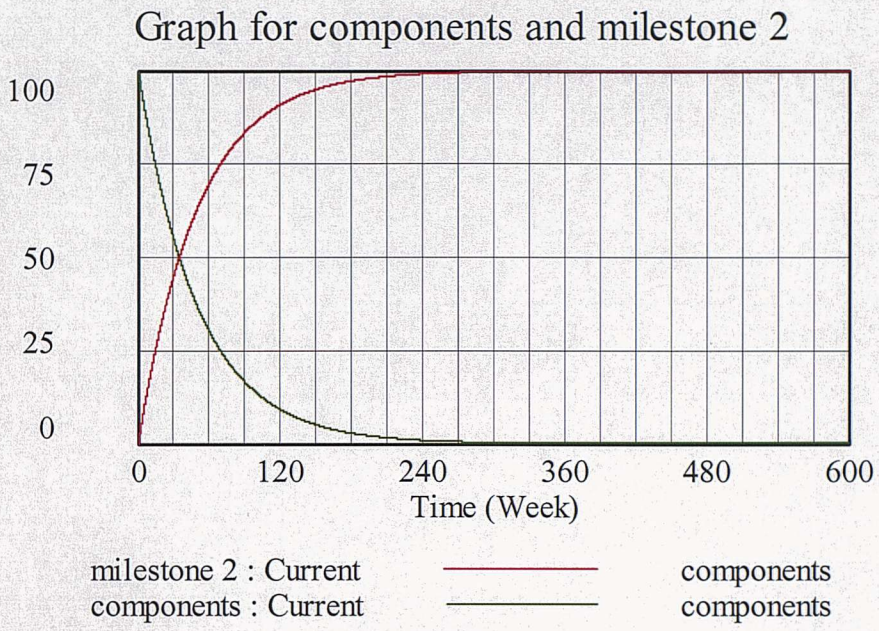


Figure 83. Graph of simulation results from one stage process composed from two process components

We can add more reporting milestones in the same way, replacing the terminal work activity cloud with another process structure, and setting the final work activity rate to zero. In this way we can build processes with, 2, 3, 4, and 5 milestones, equating to 1, 2, 3, and 4 stage processes. As a further evolution, now that we have conserved flows, unnecessary *done?* tests have been removed from the models, and the final zero rate activity has been hidden. Figure 84 shows four process models built in this way from identical process components, with increasing numbers of stages and reporting milestones.

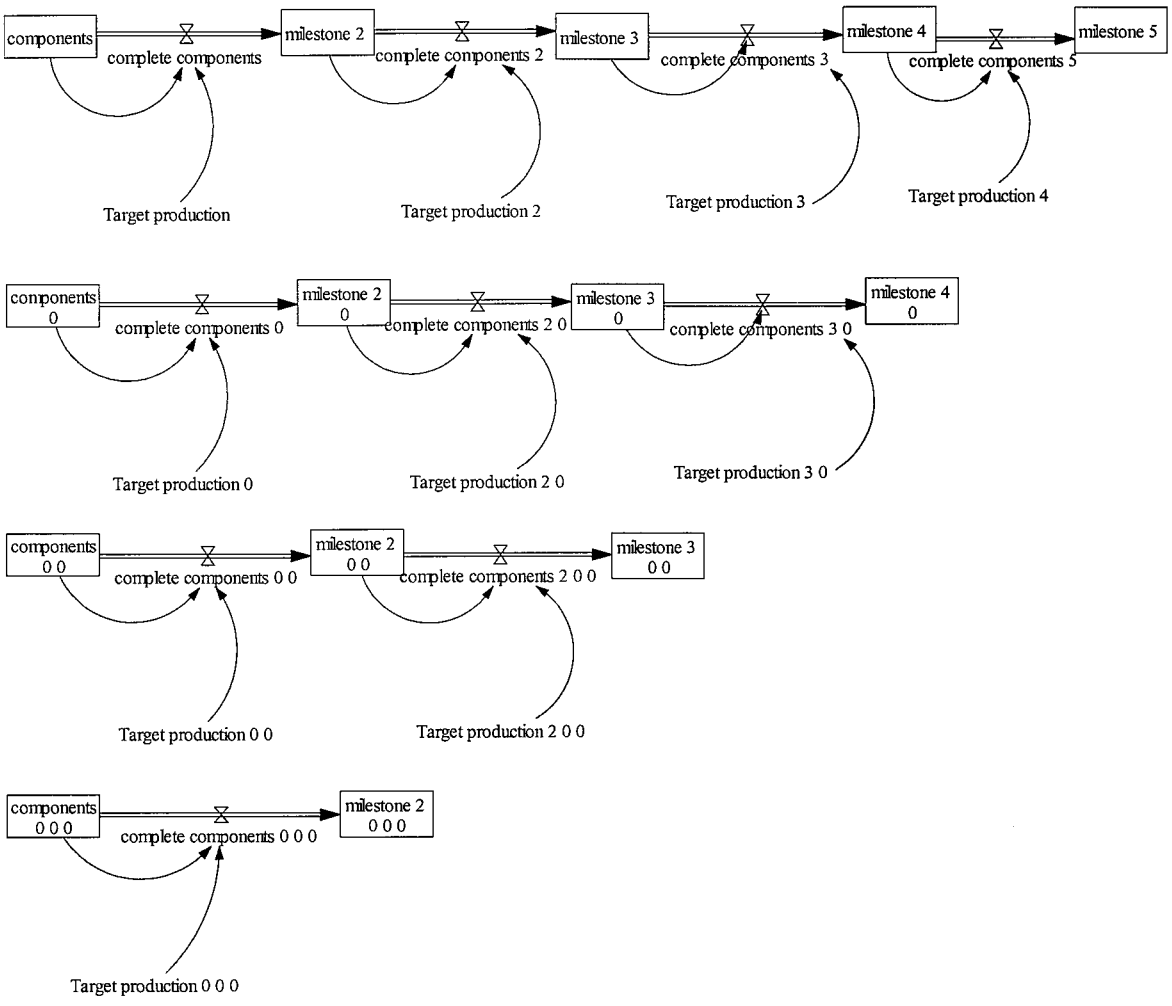


Figure 84. Systems Dynamic models of 1, 2, 3, and 4 stage processes composed of process components

Simulating each process allows us to compare the behaviour at the completion of each process and also at intermediate reporting milestones. A comparison of the behaviour of each model shows S shaped growth of the completed product.

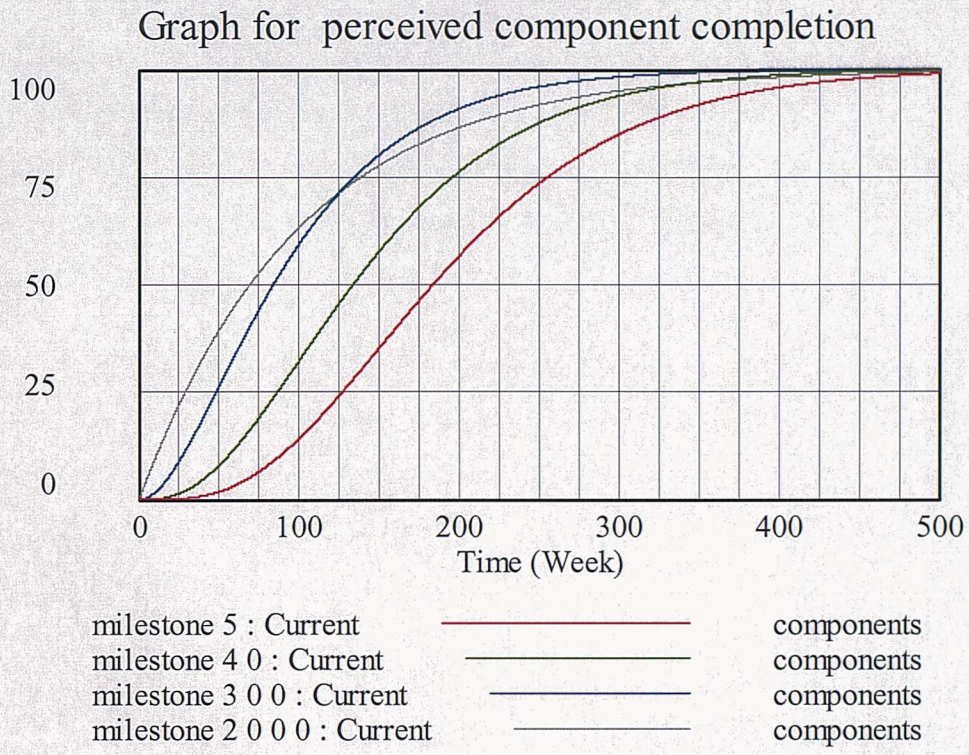


Figure 85. Graph of component completion for four processes with different numbers of milestones

The graph, Figure 85, shows results from simulating four processes:

- grey line (milestone2000), shows a process with one completion milestone
- blue line (milestone 300) shows a process with two completion milestones
- green line (milestone 40) shows a process with three completion milestones
- red line (milestone 5) shows a process with four completion milestones

Each graph shows the growth of the final milestone in each of the four processes. Each graph shows an S shaped completed components growth. The greater the number of milestones, the more pronounced the S-shape. As the number of

milestones in the process increases, the early deflection away from the expected straight line graph increases. The first part of each S shape shows a lower rate of component completion, as more components are only partially complete or not yet started, and fewer can be passed to the next stage. As the work activity continues over time, more components are complete and can be passed to the next stage, showing an increased rate of completion.

If we examine one process in detail, taking for our example, a process with four stages and five milestones. The expected completion rate is shown by the green graph, Figure 86; the perceived completion rate is shown by the red graph. The lower perceived rate of completion shown by the deflection away from the predicted line can be clearly seen.

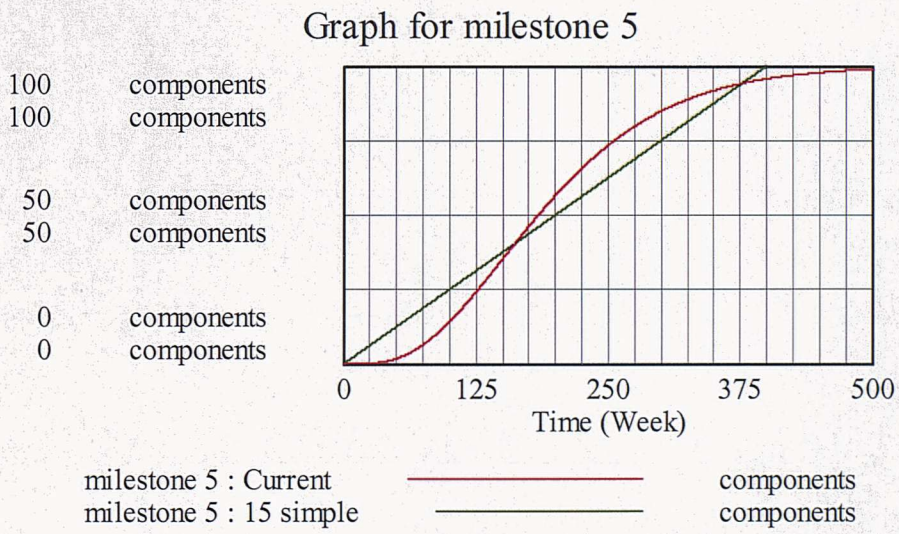


Figure 86. Graph of perceived and expected component completion for a five milestone process

The growth and depletion behaviour of individual stocks, Figure 87, shows how software components are moving through the milestones of the process.

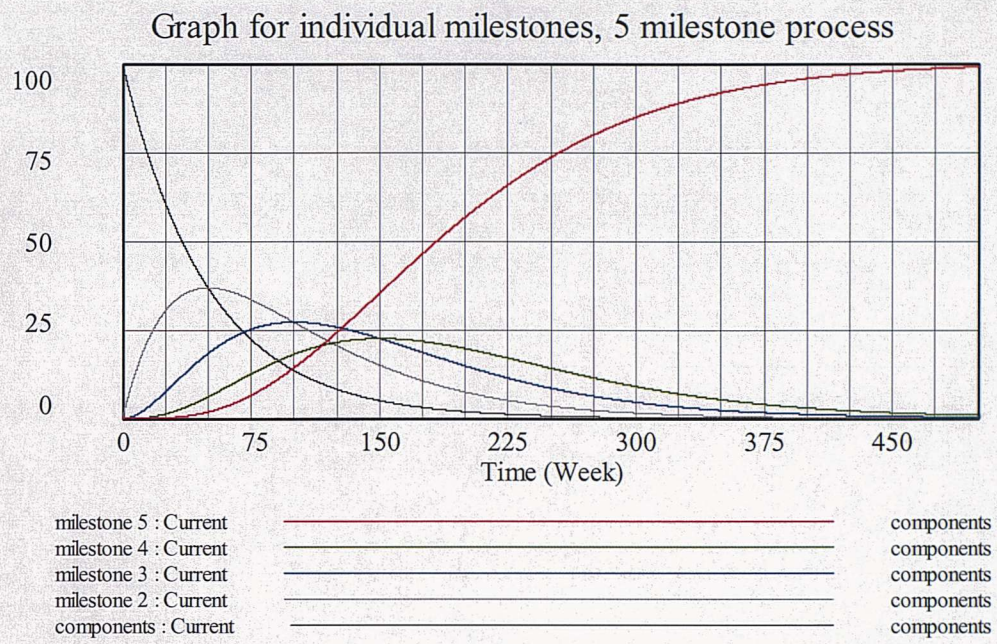


Figure 87. Graph showing components moving through 5 milestones

The behaviour shows the perception that at the beginning of a project there is less than expected progress to be seen, at the end of the project, the approach to completion is asymptotic. We have demonstrated that this behaviour can been replicated by a systems dynamic model composed of simple, repeated process structures.

6.5 Experiment on Rework

The second experiment described in this chapter investigates effects of rework on the completion of a product.

In the first experiment we examined one of the difficulties of predicting development schedules in reporting progress against plan. One of the other difficulties is predicting how much effort will be required to complete a work unit and the subsequent effect on schedule when some of work products will be reworked, incurring additional effort expenditure. The amount of rework required

may be assessed as the required quality of the software component, and also as one of the factors of the productivity of the process

As described in the previous experiment, the simplest completion behaviour is a straight line graph for completion of type, $y = mx + c$ where y is product size, x is project time and m effort. A more sophisticated plan may expect the behaviour to be $y = mxb + c$ where b is a rework factor.

One project management technique we use to provide visibility of progress is to break work down into units and report on the effort expended on the completion of these units of work. Because it is difficult to assess, often the rework element of effort expenditure is not visible in product and process reporting measures. How much of the effort expended is attributable to new work and how much has been expended on reworking is difficult to evaluate. If all of the effort is attributed new work, the completeness of the product and progress towards schedule completion will be overstated. The likely effect is schedule and cost overrun.

We will examine a development process in which there are successive stages; at least two and possibly many. In each stage some software components are received, transformed by the work activities of the stage and passed as products to the downstream, successor stage for the next work activity. The work at each stage may be different. In the first stage we may be gathering requirements, in the second designing a build plan. In the third, assembling software components and so on until the product is ready to be shipped. Or the work at each stage may be the same, each corresponding to an evolution of a product, or component. The process is not perfect and produces a product that contains errors that must be discovered and corrected.

We will assume that each stage some errors escape to be discovered by the downstream stage for which the producer is a supplier. The downstream consumer stage is unable to complete the work on these faulty components and cannot correct the defective component itself, but must return the component to the

producer stage for reworking. It is also possible that the faulty unit is successively referred back to earlier stages until the error can be fixed.

The products of software development processes are unlike other manufactured products, the rework is done by the same process that carried out the original work process and is not error free; we may not only fail to fix the original error but even introduce new errors.

We will show that the behaviour may be explained by modelling the process in Systems Dynamics using simple repeated process structures.

The first task is to identify a simple component structure for the process, by abstracting to the simplest possible structure that will model the behaviour.

If we abstract from the specific activities carried out and artefacts produced, we have a process that at each stage, some units of work are received; the stage detects errors in the some work units and returns them to the supplier stage for rework. The stage completes its part of the work on those units, and hands on the work to the next client stage. We can describe this as a simple process component repeated for each stage in the process.

6.5.1 Rework Model 1

The first model shows a simple, abstract representation of a development process with rework, modelled in Systems Dynamics, Figure 88. The model abstracts from the separate process stages and represents the process as a single stage with simple stocks of work and the activities that increase and deplete them. We will represent the work to be completed by the process by the stock, *tasks* and show the completed work as the stock, *tasks 2*. The work that transforms *tasks* into *tasks 2* is modelled as the work activity *complete tasks*. We will represent the process activity that discovers faulty work as a flow, *return bad work* from the completed work, *tasks 2* back to work to be completed, *tasks*. The proportion of completed work returned by the *return bad work* activity is determined by the constant,

percentage bad work. Thus the stock of work to be done is depleted by the work activity and increased by the flow of returned bad work. Conversely, the stock of completed work is increased by the complete tasks activity and depleted by the *return bad work* activity.

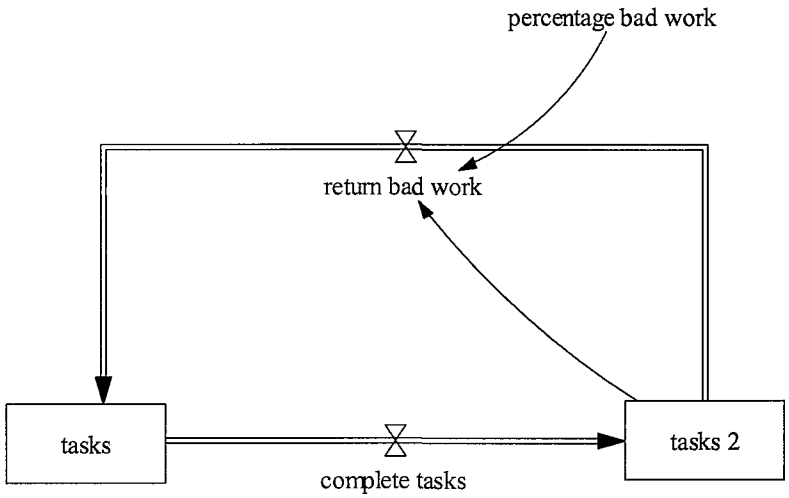


Figure 88. Abstract development process with rework

The abstract model in this experiment does not provide an easily composable process structure. If we were to build a refinement of the abstract model by adding a second identical structure, we would need to connect two stocks together, which is not possible without an intervening flow. However as the stock of completed work, tasks 2, is the stock of work to be completed by the successor stage; the process structure may be represented with a single stock. A better abstraction of the process model shows the process with a stock of work to be completed, *components*, that is depleted by work to *complete tasks* and by work to discover and *return defective work*, Figure 89.

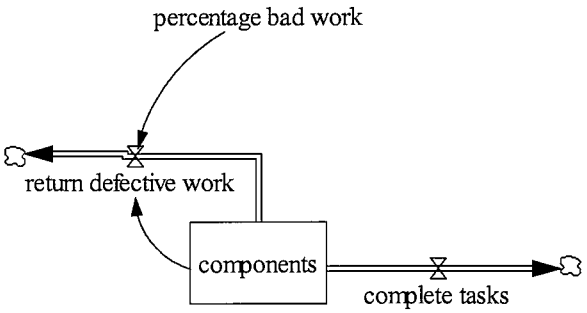


Figure 89. Abstract process component

The model is an evolution of the same simple, abstract process component described in the first experiment, Figure 89, shown here for comparison. In the case of the rework component, there is no structure for stopping the process at a target size, Figure 90.

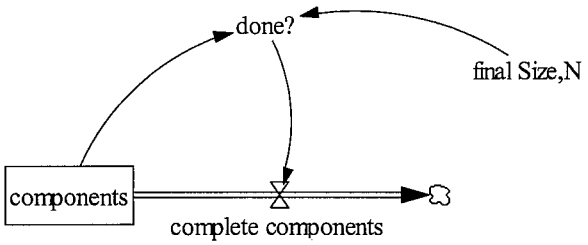


Figure 90. Abstract simple process component modelled Systems Dynamics

6.5.2 Rework Model 2, abstract single stage model composed of two process components

We can remodel the single stage abstract process using two process components composed together, Figure 91. We add the second milestone by replacing the output 'cloud' or sink of the work activity, *complete tasks*, from the first milestone with an identical process structure. The *return defective work 0* activity cloud of the second milestone is replaced by the stock of work *components* to be completed in the first process structure. The process finishes at the second milestone; the work is not passed on to another stage, so we may set the *complete tasks 0* activity of the second process component to zero. Similarly, the first process component has no preceding milestone so we may set its *return defective work* activity to zero.

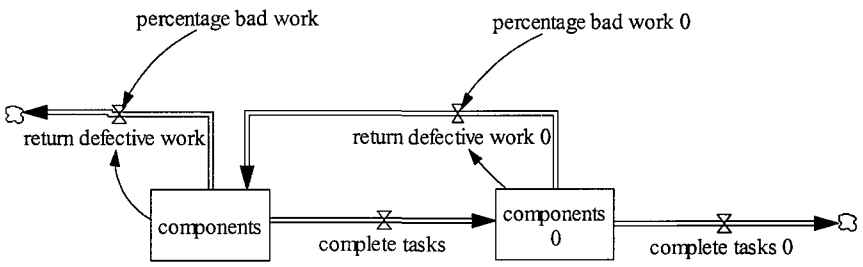


Figure 91. Systems Dynamic Model composed of two rework components

The model now has the structure of a process that has one stage with a start milestone and a completion milestone. The single stage makes both good and defective work, and has a process for discovering and reworking defective work.

We can show that the two versions of the abstract model have equivalent behaviour by simulating both models with the same set up variables and comparing the results. The simulations show that the stocks of work to be completed and finished

work have the same behaviour as shown in Figure 92 and Figure 93. Both have an asymptotic depletion of the stock of software components to be completed and an asymptotic increase in completed components

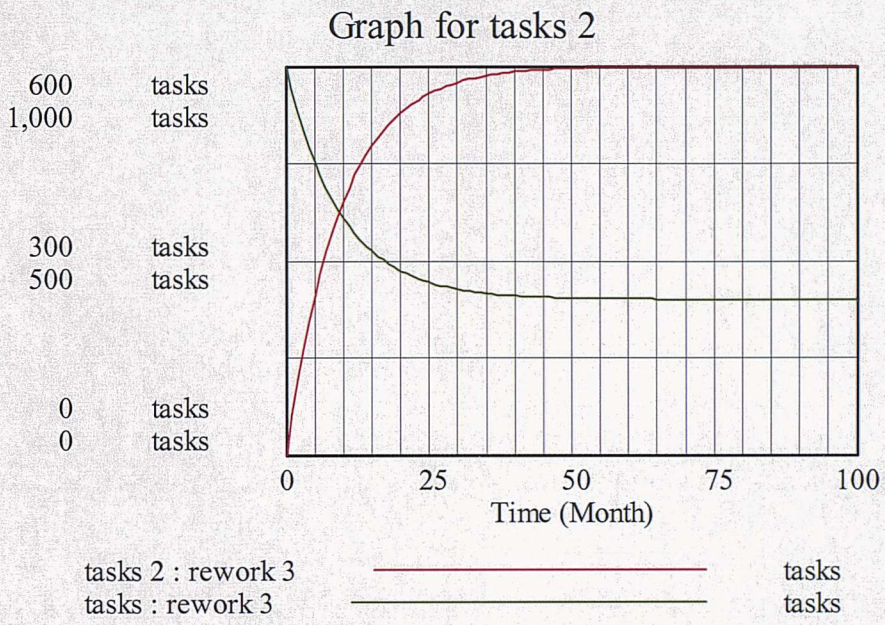


Figure 92. Simulation results from the first version of the abstract model

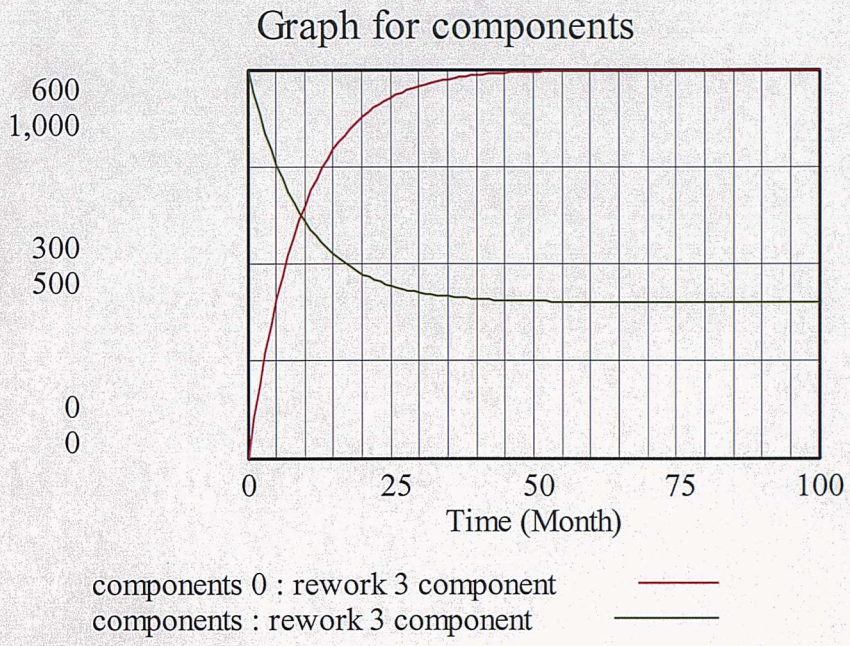


Figure 93. Simulation of abstract process composed to two process components

The abstract model composed of identical process components and the first abstraction of the process are in good correspondence.

6.5.3 Rework Model 3

We can add more reporting milestones in the same way, replacing the terminal work activity cloud with another process component, setting the final work activity rate to zero, and connecting the defective work return activity to the preceding milestone stock of work to be done. In this way we can build processes with, 2, 3, 4, and 5 milestones, equating to 1, 2, 3, and 4 stage processes. Figure 94 shows a model of three stage process with four milestones built in this way from identical components. As in the models from the previous experiment, we have hidden the zero rate flows.

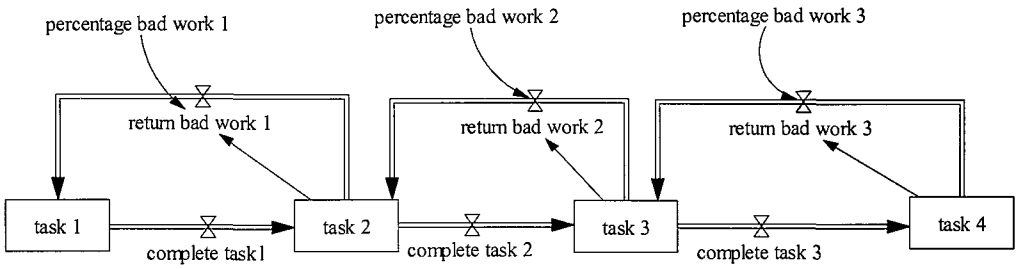


Figure 94. Systems Dynamics model of a three stage, four milestone rework process

6.5.4 Simulating rework model

We can simulate the model to show the effects of a range of different rework rates, but keeping the numbers of tasks and task completion work rates the same.

- set the same for each stage, at 0% and 10 %.
- set differently for each stage in a simulation, increasing from 5% to 20% and decreasing from 20% to 5 %

6.5.5 Simulation 1, 0% rework

The first simulation, Figure 95 shows that when the rework rate is zero for all stages in the process, the model behaves exactly as the simple model described in the previous experiment Figure 78, Figure 79. The stock of completed software components increases at the same rate that the stock of work to be built from decreases. When the stock of work to be done is empty, at time T, all of the work items have been transformed and the stock of completed work has reached its final size.

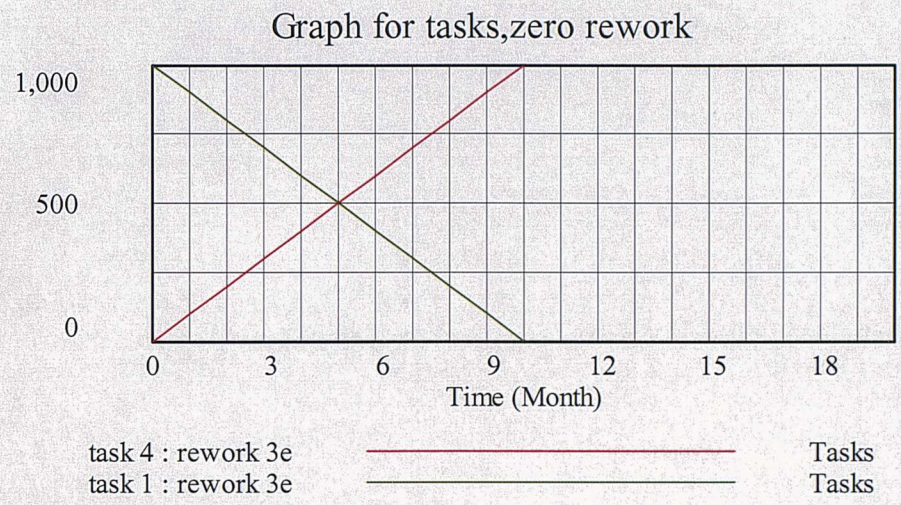


Figure 95. Simulation 1, graph of work to be completed and completed work with zero rework

6.5.6 Simulation 2, Rework at 10%

Figure 96, shows the results from simulating the model with 10% rework rate for each stage. The final stock shows an asymptotic approach to completion whilst intermediate stocks show S-shaped growth behaviour. The time taken to complete production increases.

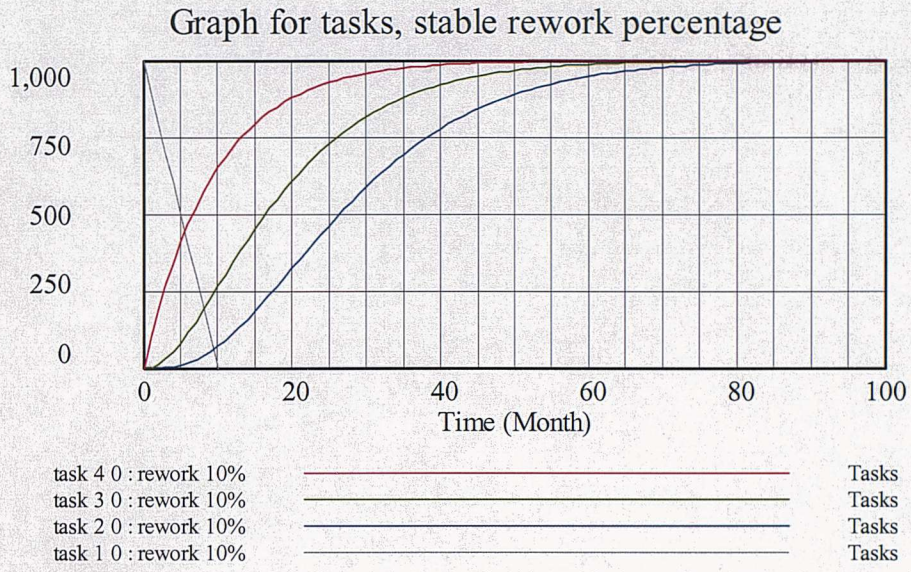


Figure 96. Simulation 2, graph of work to be completed and completed work with 10% rework

6.5.7 Simulation 4, decreasing and increasing rework

Figure 97, shows the behaviour of stocks of work tasks when rework percentages decrease throughout the process, starting at a rework rate of 20% and the final stage has a rework rate of 5%. This is analogous to a process where faults are discovered early in the process, resulting in lower rework requirements later in the process. The final stage again shows an asymptotic approach to the final size and intermediate stages show S-shaped growth patterns, similar to those shapes seen in Figure 85.

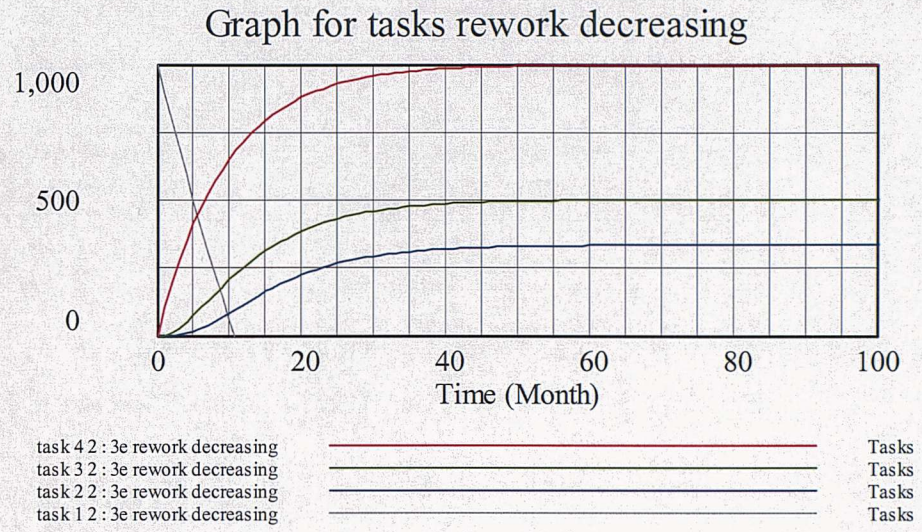


Figure 97. Simulation 3, graph of work to be completed and completed work with rework decreasing

Figure 98 shows the results if rework percentages increase during the process, analogous to a process where faults are detected late in the process.

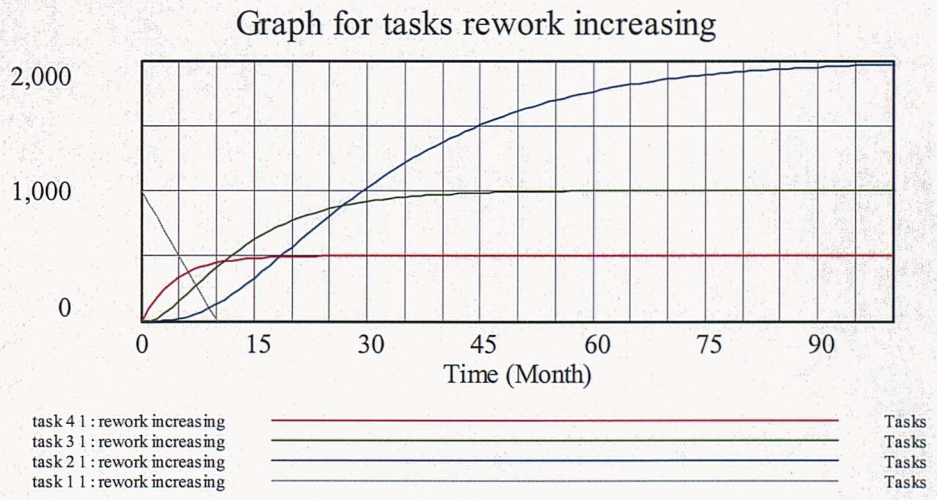


Figure 98. Simulation 4, graph of work to be completed and completed work with increasing rework

We may add additional structures, increasing the complexity of the model by, for example,

- adding a delay to the return of defective stock for rework, representing a delay in the time for discovering errors
- preventing stocks from going negative.

However, the underlying behaviour remains the same; an asymptotic approach to final stage production and S-shaped intermediate stage growth.

The simulations show growth in number of software components that are produced when the process produces faulty work; this is because, reworking a faulty component creates another component. The process stock counts both the original faulty component and the reworked component; this reveals that a faulty process increases the number of components that need to be produced in order to create a product of fault free components of the target size. This is seen as an increase in the work required to produce a fault free product. The number of software components required to achieve a product of size, N is a measure of the efficiency of the process.

6.6 Conclusions

We have used Systems Dynamics to build models from simple process components that explore two plausible reasons for a commonly observed process behaviour. Although both models are of a staged process, in one model we are examining the effect of under reporting completed work and in the other, the effects of reworking faulty work units.

The process models are simplistic, and qualitative; for example in the rework model we have made arbitrary choices for the rates of defective work, and have not differentiated between error rates for reworked software components and new work.

In both cases simulation of the models produces a behaviour that shows an asymptotic approach to the completion of growth targets, an S- shaped growth pattern, consistent with our observed process behaviour.

Using simple process components to build a model of our processes we find that we can find two explanations that reproduce our observed real world behaviour

6.6.1 When our real world project exhibits this behaviour

On the basis of these qualitative models, if we believe that our process under reports progress, we may introduce finer grained reporting structures to improve our ability to predict completion at each stage and final completion, but if the real cause is the poor quality of incoming work, the improved reporting structure will make no improvement to the predictability of the project completion; even worse, the underlying causes of our predictability problems will remain undetected.

If we believe that the problem is due to quality control, one process improvement measure we could adopt would be to introduce a new quality assurance programme. This would increase our project costs for testing and quality control. But if the real problem is insufficiently fine grained project management and reporting, we will increase our project costs without improving our project completion predictability.

Both behaviours have as their fundamental cause under reporting of work done, but causes of the under reporting are different and lead to different process improvement solutions.

This shows that in order to underpin process improvements with modelling and simulation tools we need to have a better understanding of process component behaviour in the domain of software development otherwise process evolution will have little effect on the outcome of project goals.

Chapter 7

From Qualitative to Predictive Quantitative Models

Geoff Coyle [Coyle 2000] suggests that qualitative models may be sufficient to understand the dynamic behaviour of the feedback relationships within the process; that they are 'good enough'. However, in his argument for limiting systems dynamics models to qualitative modes unless there are demonstrable benefits, he uses results of simulations of models to investigate their flaws and quantification of the soft variables used. Without an attempt at quantification, the dynamics of the systems are difficult to understand, and the model cannot be validated.

This suggests that we should examine how far should we go in terms of quantification; what is good enough?

In Chapter 4, we showed a Systems Dynamics model of CMPM, the Cellular Manufacturing Process Model, developed in an ad-hoc manner. The model was simulated using subjectively derived data, including 'soft' variables such as productivity. The ad-hoc CMPM model appears to reproduce some, but not all of the observed behaviour of the real world process.

Whilst this model is useful for exploring the behaviour of the real life system, and increasing understanding of the feedback relationships causing observed behaviour, the model needs to be validated against data from the real world process. The model cannot be used as a tool to predict the behaviour of the real world process unless the behaviour of the model and the real world are in good

correspondence, as we demonstrated in Chapter 5, Evolutionary Model Building. In this set of models we had high visibility into the modelled process and a means of capturing data from Monte Carlo simulations, enabling us to have confidence in the correspondence between the process and the Systems Dynamics model.

However, quantification in the real world is difficult. Coyle describes Systems Dynamics models that have 38,000 variables; clearly quantification of such models would be both expensive and difficult, and one would question how the quantitative data could be collected or evaluated in the case of soft variables. Even if we avoid producing over complicated models, collecting data to support quantification of our models is difficult.

7.1 Ethnography and Quantitative Data.

We have quantitative data supplied for an investigation of CMPM, collected from a systems integration project covering a period of fifteen months [ICL 1999]. The first twelve months of quantitative data was a retrospective re-interpretation of historical data captured for a project reporting system based on a different cost and schedule model. The quantitative data is not source data, but an interpretation of source data in terms of the requirements of the new process model, made by project staff with expert knowledge of the data domain. Interpretation requires not only an understanding of what the source data represents and how its data capture system imposes a view of the data, but also an understanding of the model underlying the new data requirements. The final three months of data were interpreted specifically in CMPM format.

The historical data was presented in three different interpretations derived by the organisation from their time recording; the interpretation process was prone to transcription error.

There may be other constraints that affect the quality or fitness for purpose of the interpreted data:

- Confidentiality of commercial data
- Gaps in source data, a mismatch between what has been captured historically and what the new model for the data requires
- How much effort has been made available to interpret the data
- The perceived value that the organisation places on the purposes of the data use
- Perceived value of the new model for the data
- Personal motivations, resource contention, strength of organisational loyalty
- Authority within the organisation to require access to data.

The sociological theory, Ethnomethodology [Garfinkel 1967] may provide us with an explanation and understanding of how data is interpreted in this way. The theory suggests that individuals construct a framework of order in a social world, which in this case is working in a systems development organisation, by a psychological process called the 'documentary method'. An individual will attempt to organise experiences into a coherent pattern by selecting certain facts from a social situation which seem to conform to a pattern, and then using the pattern to make sense of the facts. An established pattern will be used as a framework for interpreting new facts. From this ethnomethodological viewpoint; the data interpreter will attempt to make sense of new facts and requirements from the research investigation into the new process model (for example, CMPM) within a framework of their existing experience of process models and development practices within their organisation. They will interpret the data from the definitions of the data and from how they believe the interpreted data will be used by imposing a pattern from their

experience and their knowledge of the organisational and the data domain. [Rodden, Rouncefield, Sommerville and Viller 2000] This will affect

- What is selected
- What is not selected
- How the source is interpreted using internal project knowledge to fit the new data model
- Effort applied to accuracy.

Using an ethnomethodological insight, we might suggest the different viewpoints from which two experts within the investigation might apply the documentary method in order to interpret data to satisfy the requirements of the investigation.

Domain expert,

- Higher organisational loyalty
- Resource available constrained
- Higher domain and existing data model knowledge
- Lower knowledge of new data model
- Lower identification with new model and purposes of the new model
- Imposes pattern from their own domain experience on new requirements

External process expert,

- Lower organisational loyalty
- High identification with new requirements and model,
- High understanding of new data requirements and purposes for data
- Lower domain understanding
- Imposes pattern of beliefs from data requirements to try to make sense of sources

People will impose a pattern from their own domain and experience on ideas and requests from outside their known experience in order to make sense of them. The responses to those requests have to be evaluated on that basis.

Understanding the effects of the documentary method on interpretation of source data is important when evaluating the suitability of data for quantifying systems dynamic models for validation and prediction.

The data user must find ways of evaluating the data that minimise the effects of interpretation. Beliefs constructed about results from the data should be based on an ethnomethodological awareness.

7.2 CMPM Project data

As discussed in Chapter 4, CMPM, the Cellular Manufacturing Process Model [Chatters, Henderson et al. 1998] proposed by Peter Henderson is an advanced process strategy based on components that uses concurrency and distribution to reduce cycle times. (As a reminder, in CMPM, networks of semi-autonomous producing cells co-operate to produce a complex large-scale system. The model views development as a manufacturing activity where large scale systems are built from components, which may be a mixture of self built components, re-used components from the producers own asset base and from bought in components.)

The organisation collaborating in the investigation provided project data in order to explore and validate CMPM. The supplied data was derived from the organisation's project management reporting system, based on an existing project process model which records time spent by project staff allocated to a project, less time spent on holidays, sickness, training etc.. The data is commercially confidential, so the data was derived from the source data by project staff and anonymised.

7.3 Measurements from the process

In Chapter 2 we examined models of the quality of software products in terms of external attributes desired by the stakeholders in the product (both users and producers) and internal measures of product and process that indicate the degree to which the external attributes have been achieved. We also referred to the Representational Theory of Measurement [Fenton and Pfleeger 1997], which indicates that the data we use to describe and measure internal attributes must properly represent attributes of the observed entities and that measurement must be consistent and preserve the relationship observed between entities. The rules for consistency in measurement provide a basis for interpreting the data.

In a real process, the measurements need to be simple and either easy to collect or collected automatically. [Chatters, Henderson et al. 1998],

7.3.1 Time ET, Effort Measures W (raw), and Team size,N

Raw effort is comparatively easy to collect, counted in days worked by a person assigned to a task. In order to define it, we must first define what we mean by a day, in CMPM Elapsed Time, E , is calendar days per month less public holidays and weekends. We must also define which people should be counted as project staff.

W = (Elapsed Time in days * people assigned to project) – outage days

where outage days are days not spent on task, for example, recorded sickness, holidays, training

Whilst capturing raw effort data is reasonably easy, capturing effort W, broken down by work type, (problem solving, testing, building) and allocating the effort to either products or product lines is much harder.

7.3.2 Size, S

In Chapter 4, we discussed the problems of achieving representative size measurements in a COTs integration project. In the third phase of the CMPM investigation, S was derived from quality independent cost drivers (building systems, installing products, regression testing, producing project infrastructure, making glue and in-house components by counting process artefacts, measured in standard units (SIU's). This had the benefit that they directly relate to the work done to integrate a component. The measurement depends on the number of supplied components, the amount of in-house development, and the number of incremental builds, not the size of a supplied component. In terms of the representational theory of measurement, this is a more representative way of measuring the size of the output component in terms of the work required to complete it, rather than an internal component dimensions. The problem with this definition is that if the process artefacts change between projects, the representativeness would be lost.

This size data was not available in the historical data, but the measures were defined for the last phase of the investigation. From that point, although they could not be retrospectively applied, data were collected for ongoing development.

7.3.3 Quality Measures, Q and P

Q is a measure of the quality of supplied components and data is based on internal and external problems solved in assembly.

What counts as a problem? What problems should be counted? For component assemblers and integrators, they may perceive their work in 'gluing' components together, as problem solving; therefore problems do not get perceived or distinguished separately from work. Problems that should be counted may be pejoratively misperceived as 'errors' indicating faulty work. There is a tendency to avoid counting these, unless they become highly visible by causing an obvious project delay, or can be attributed elsewhere.

External problems with suppliers outside the organisational boundary are more likely to be counted as there will be documentary evidence to support the problem and blame is not an issue. In the CMPM investigation quality attributes were estimated as 'soft' attributes designed to evaluate the additional project effort required to compensate for imperfect supplied components. For example, one of five attributes for *Q*, attribute A is defined as:

> attribute A: The impact of the product and development process characteristics are fully understood.

Similar attributes were used to estimate P, the quality of the outgoing components.

7.3.4 CMPM Historical data

Historical data from the first 9 months of a systems integration project was used to redefine cost and schedule results from the existing process model in the context of a CMPM interpretation, to enable cost and schedule estimation comparisons. [ICL 1999]

The source data was interpreted at CMPM component level and further interpreted at product version level. Where the source data model didn't map directly to the CMPM model, the project staff interpreted the source data using their internal project knowledge.

The CMPM data model is as follows:

W = f (Q,P,S)

W = effort; S = size

Q = input quality of supplied components

P = delivered quality of system

Q and P are 'soft' variables based on a subjective assessment of component quality

The time recording system provides effort, W, data in man days. The project staff reported the number of errors in externally supplied components, and rated the supplied components, which supports the measurement of Q. However, there are no source data for measurements of S.

The historical data gives a count of the number of components in each release, but the source of the numbers or what they represent is not clear. There are big variations in effort per component, within the same product and release, so without further information to establish a representative measure, the number of components reported in this data cannot be used to calculate Size, S.

However, we can use effort data to represent S by assuming a relationship between effort, W and size, S.

$S = f(W)$

The source data has been interpreted into the following CMPM representation ,Table 9. The product has three software components, SPA, SPB and SPC, a hardware component, HPA which used two versions of hardware x and xx. Each release of a product version has a systems design component SD, an integration component IP, and a release management component, RM.

Component	release a	release b	release c	release d	release e
Software					
SPA	a	b			
SPB	a			d	
SPC	a	b	c	d	
Hardware					
HPA	x	x	x	xx	xx
Integration					
IPA	a	b	c	d	
IPB					e
Release Management					
RMA	a	b	c	d	
Systems Design					
SD	a	b	c	d	

Table 9. Matrix of components and releases

The following table, Table 10, shows how project staff allocated effort, W, to components and releases.

Component	W release a	W release b	W release c	W release d	W release e	Total for component
Software						
SPA	548.5	31.2		659.26		1239
SPB	261.5			803.9		1065.4
SPC	217.5	15.1	137.	325.03		694.9
Hardware						
HPA	196	16.8	155	53.3	26.7	447.8
Integration						
IPA	630	63	250	825.64		1768.8
IPB					766.7	766.7
Release Management						
RM	125	8.4	67.5	27		227.9
Systems Design						
SD	117	17	44.5	37		215.3
Total W	**2095**	**151**	**654**	**2731.13**	**793.4**	

Table 10. Matrix of components and releases showing Effort, W in man days

Because the data source is time sheets generated for a different purpose, these are difficult to interpret for the requirements of CMPM. The data interpreter will attempt to fit their usual understanding of project data and their organisation into a new CMPM view, using their judgement of how effort should be allocated. These judgements include how resource and time should be allocated over a number of components where the project staff does not differentiate them. For example, Software Project B (SPB) releases *a* and *d*, Table 11, shows how the data interpreter has attempted to resolve apportioning work for each release of the software, where the project staff probably reported only that they worked on SPB, note how the first four months of each release have identical effort allocations W and team size N. The SPB project team may have been seven people for the first four months.

			W	T	N
SPBa	01-Aug-97	31-Aug-97	57.75	21	3.50
	01-Sep-97	30-Sep-97	67	22	3.50
	01-Oct-97	31-Oct-97	72.25	23	3.50
	01-Nov-97	30-Nov-97	64.5	20	3.50
SPBd	01-Aug-97	31-Aug-97	57.75	21	3.50
	01-Sep-97	30-Sep-97	67	22	3.50
	01-Oct-97	31-Oct-97	72.25	23	3.50
	01-Nov-97	30-Nov-97	64.5	20	3.50
	01-Dec-97	31-Dec-97	92.4	21	5.50
	01-Jan-98	31-Jan-98	98	21	5.00
...	...	...	...	...	...

Table 11. SPB cell resource allocation

The hardware component, HPA appears to have two versions of hardware, one used in releases a, b c and d, and the second used in releases d and e, Table 4; however, the source data does not easily fit a CMPM representation of the product version. This difficulty is underlined by differences in interpretation of the release structure between the three versions of the data which cannot be resolved.

The interpreted data shows two integration projects, which attempt to resolve the mapping of the product versions into a CMPM structure. Where software components are not shown in releases in the matrix, it is not clear whether they were not present in the release, or whether they were left out of the data because assembly effort was not allocated to them (a re-used component and not a new instance of the component).

> 'Consequently, the cells were redefined for the subsequent incremental releases. A single cell for each incremental release represents the multidiscipline activities of the integration team. This change... requires only one category of cell ("systems integration").'[Chatters, Henderson and Rostron 1999]

Release a	release b	release c	release d
Simmer a			
Simmer b			
	Simmer c		
		Simmer d	
			Simmer e

Table 12. Second interpretation of CMPM structure

The same data reinterpreted into a second view of the CMPM model, Table 12 (there are some discrepancies in the reported effort, W_a) shows simmer releases *a* and b assimilated as *release a*. Simmer *c* becomes release *b*, simmer *d* becomes release *c* and simmer *e* becomes release *d* [Chatters, Henderson et al. 1998]. The actual and estimated values for W, effort, T, elapsed Time and N team size are shown, Table 13.

Release	W_a	T_a	N_a	W_e	T_e	N_e
Release a	2284	149	28	1320	60	28
Release b	440	84	7.9	576	40	18
Release c	2758	213	15	1512	140	14
Release d	793.4	213	4.2	640	140	4.5

Table 13. Second interpretation of source data into CMPM structure

The following data shows values obtained for product releases in the third phase of the investigation, Table 14.

Release	W_a	T_a	N_a	W_e	T_e	N_e
Release e	362	148	3	205	87	2.5
Release f	303	125	4	293	82	2.9
Release g*	2127	221	12	1819	189	12

Table 14. Third phase CMPM data interpretation

- W_a is based on a revised estimate as the project had not completed.

This suggests that in this phase the CMPM process has been abstracted from a network of producer cells to a single black box cell.

7.4 Relationships between project data and Systems Dynamics models

As long as we are aware of the inconsistencies between the interpretations, the data can be used to investigate process and model behaviour, however without representative size data, it cannot be used to quantitatively validate the Systems Dynamics models for use as predictive tools.

7.4.1 Ad-hoc Systems Dynamic model of CMPM

We investigated the Stella ad-hoc CMPM model Chapter 4, Figure 27, with quantitative data values from Table 12 and Table 14. There are problems with quantification; the model and the CMPM implementation diverged and the model was not 're-synchronised', or brought back into correspondence with the implementation, for example, some areas of greatest divergence are input and output metrics and the definitions of Q and P.

The Systems Dynamics model definition of output quality is based on the number of defects remaining in the product and its 'completeness', whereas the implementation assesses P by comparison with an output checklist to evaluate completion of requirements. The implemented CMPM process uses an assessment of incoming component quality, Q, based on the suppliers ability to supply high quality components. This soft evaluation is neither used in the CMPM model, nor is there a variable relating to supplier capability.

There are few data points available for the largest project, release g, Table 14; we have data points at elapsed time, T = 40, T = 63 and T = 229 (estimated), limiting the value of comparisons of completion behaviour. *Releases e* and *f* were produced concurrently; *release e* has 6 data points and *release f* (completed one month later)

has 7. The graph of effort, for releases e (S_rel_e), and f (S_rel_f), over elapsed time, produced in Mathcad [Mathcad 1999] is shown below, Figure 99.

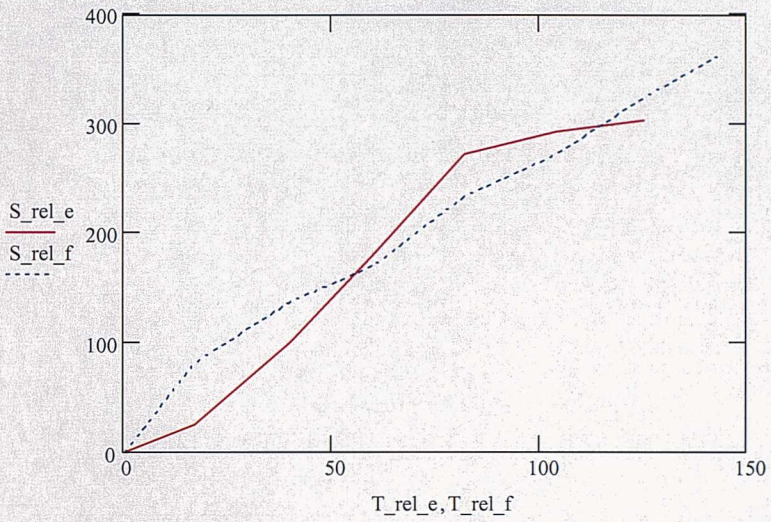


Figure 99. Graph showing growth of effort for release e and f (S_rel_e, S rel_f) over time T

In comparing the two graphs, *release e* shows an S-shaped completion shape whereas *release f* shows an asymptotic completion.

We can examine the correspondence between the Systems Dynamics model and the CMPM implementation, by comparing the results from simulating the Systems Dynamics model of CMPM and results from the CMPM implementation.

Initially, we simulate *release e* using the model calibrated as in Chapter 4, setting the variables as shown in Table 15.

The resulting simulation (sim 1) Table 16, shows little correspondence in completion time (schedule); the product is complete and all defects are removed in 48 days. This should ensure a perfect product where P = 1. However, the output quality from the simulation is P = 0.32.

Simulation variables	Sim 1	Sim 2	Sim 3	Sim 4
Percentage effort for tasks	60	60	60	60
Percentage effort for quality (defect removal	40	40	40	40
Tasks per man-day	0.7	0.7	0.7	0.7
Defects per man-day	0.3	0.3	0.3	0.3
k	0.32	1	1	1
Defects per task	1	0.1	0.1	0.1
b		0.02	0.02	0.003
b_defects		0.05	0.05	0.005
Initial defects	50	50	50	200
Target size (W, release e)	362	362	362	2127
effort (Team size N)	3	3	3	12

Table 15. Simulation initialisation values

For the second and subsequent simulations (sim 2 and sim 3) the System Dynamics model was changed to weaken the effect of the feedback loop from completed tasks to task completion, (as the product nears completion, it gets more difficult to complete) by a factor *b*. Similarly, the feedback loop affecting defect removal was weakened by a constant *b defects*

For the third simulation, we changed the graph supporting productivity in the feedback relationship. The simulations produced schedule completion times with closer values to the implementation data Figure 100, Figure 101.

In the following table of simulation results, Table 16, the long asymptotic tail of completion has been truncated at approximately 98% of effort completion.

	simdata		Sim 1		Sim 2		Sim 3		Sim 4	
release	T		T	P	T	P	T	P	T	P
e	148		48	0.32	250	0.84	150	0.94		
f	125		34	0.32	137	0.97	113	0.95		
g	221						50	1	300	0.95

Table 16. Table of simulation results

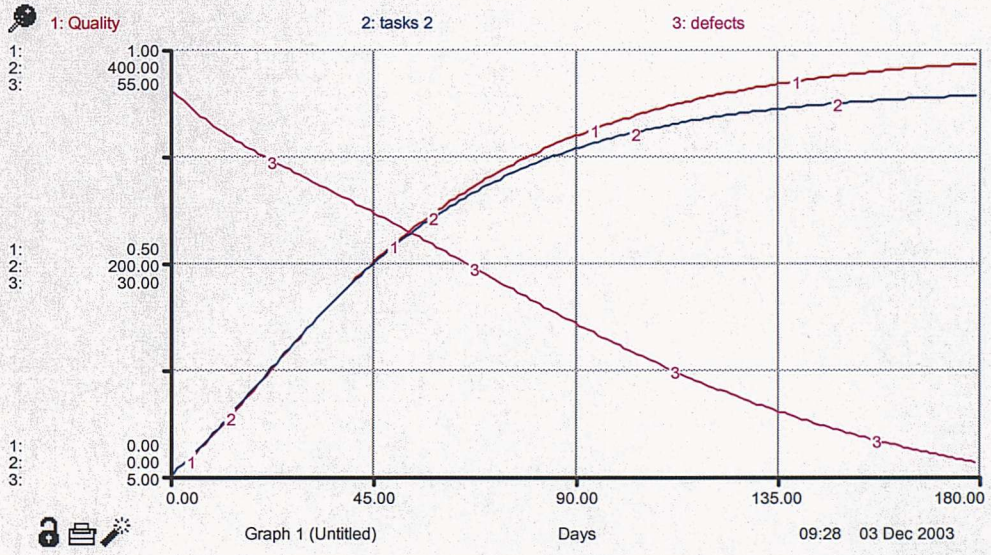


Figure 100. Graph from Systems Dynamics simulation 3 of release e

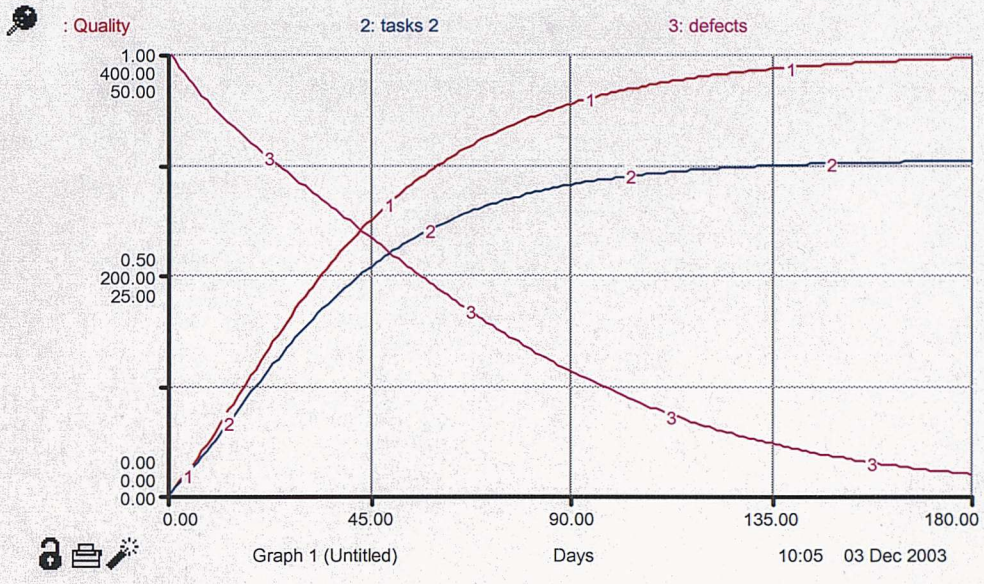


Figure 101. Graph from Systems Dynamics simulation 3 of release f

From the simulations for releases e and f, we may believe that the Systems Dynamics model is now reasonably calibrated for the CMPM implementation. However, if we simulate implementation data for release g, the feedback factors *b* and *b defects* have to be weakened by a factor of 10 to 0.003 and 0.005, in order to achieve an estimate of schedule completion corresponding to the implementation estimate.

CMPM model was created in an ad–hoc, not systematic way, without an evolutionary process that ensured correspondence with the implementation. When data from the implementation is used in model, where there are errors, it is difficult to work out what or where they may be.

7.4.2 Component based Systems Dynamics model used to investigate Hardware Project, HPA

We used Mathcad and Systems Dynamics [Vensim 1988 -1997] to investigate the behaviour of the hardware project, HPA as a CMPM cell, Table 10. We examined the work consumed by the cell over the duration of the project; in this way

anomalies in the attribution of effort to individual releases could be ignored. Using Mathcad, we plotted cumulative effort data for each reporting period against time.

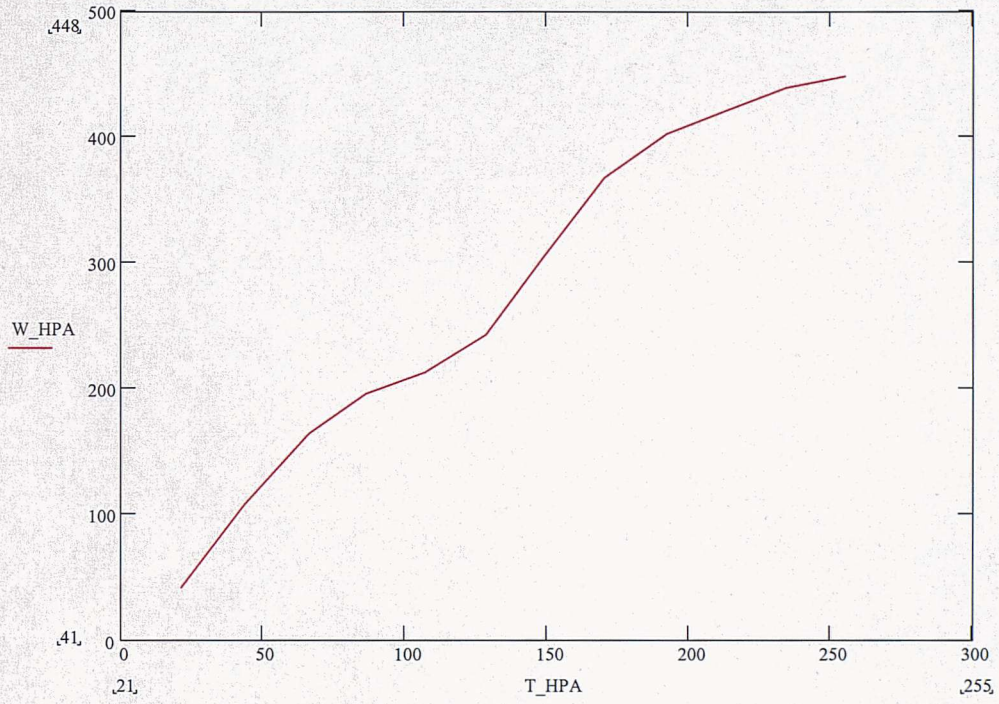


Figure 102, Graph of effort (W_HPA) over time (T_HPA)

The project data records five releases of Hardware Project A, a to e. The graph of effort *W_HPA*, over time *T_HPA*, Figure 102, shows an asymptotic approach to the total effort expenditure, as discussed in previous chapters, however, there is an unexpected inflection between T = 100 and T = 175. On examining the project data, this appears to coincide with the start of HPA release d where a new version of hardware was integrated into the product.

We can investigate the behaviour of HPA using a simple Systems Dynamics model composed of two components (see Chapter 6). We have little visibility into the project, so we will abstract the project to show two completion milestones

milestone 1 and *milestone 2* and one activity, *complete components* . *Milestone 1* represents the initial workload of components to be completed by the cell, and *milestone 2* represents the completed work Figure *103*. In this model, we will assume the relationship of effort, W and components, S, $W = f(S)$.

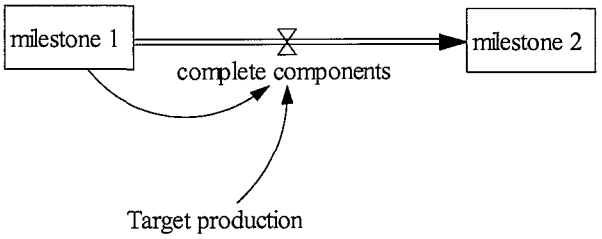


Figure 103. *Systems Dynamics Model of the hardware project cell, showing two milestone abstraction.*

Simulating the model with arbitrary values for stocks of components and flows gives the expected asymptotic behaviour, (Figure 104), but does not correspond to the inflexion seen in the actual project data.

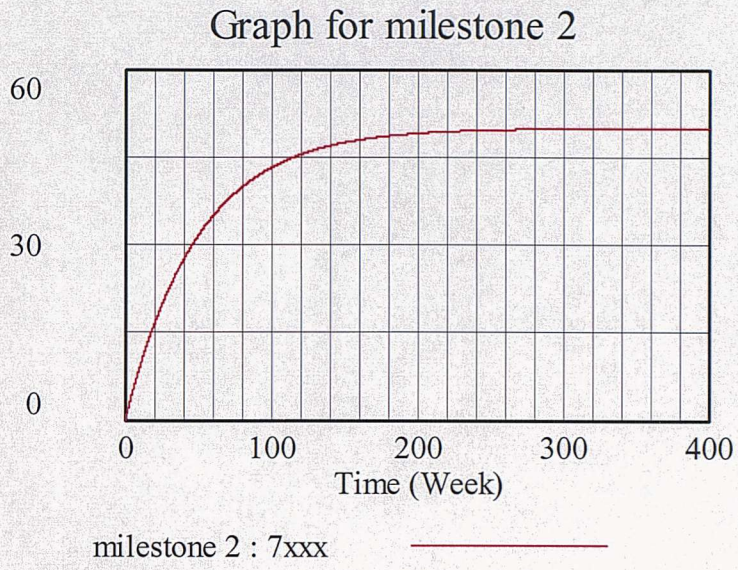


Figure 104. Graph of HPA with two completion,milestones

We can simulate the coincidence of the start of HPA release d, where work on the new version of hardware commenced, by adding another process component to the process model, Figure *105*. The new process component adds a pulse of new *components* of work to do into the process at T = 125, as shown in Figure 106. This represents the additional workload received by the CMPM cell when the new hardware version was introduced.

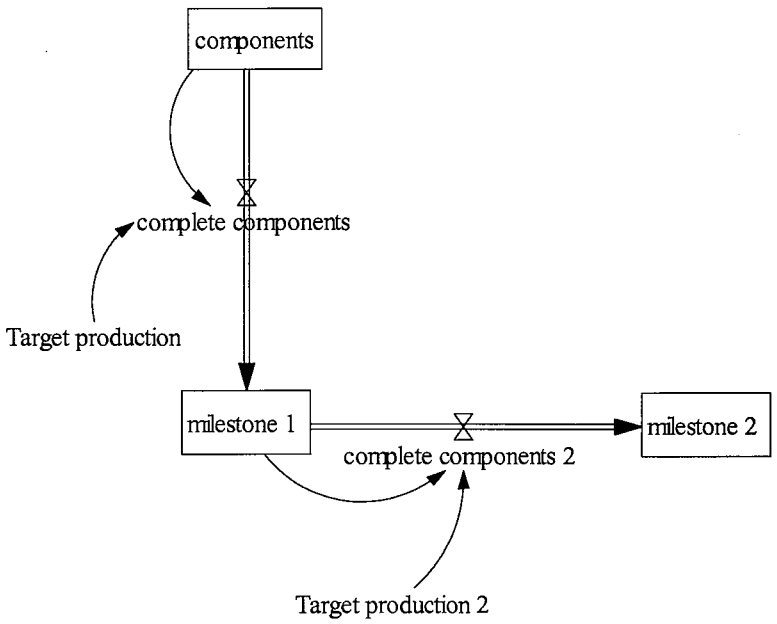


Figure 105. *Systems Dynamics model with three process components, additional hardware requirements at T = 125*

Simulating the model, again with arbitrary values for stocks of work and activities, shows behaviour corresponding to the graph of project data shown in Figure 102. There is an inflexion in the graph of effort at the point where the additional work for the new version of hardware is received. The effect on the work behaviour of the cell is the same as work on a new product with a new set of components to be integrated. The effort for each version of hardware follows the same asymptotic behaviour.

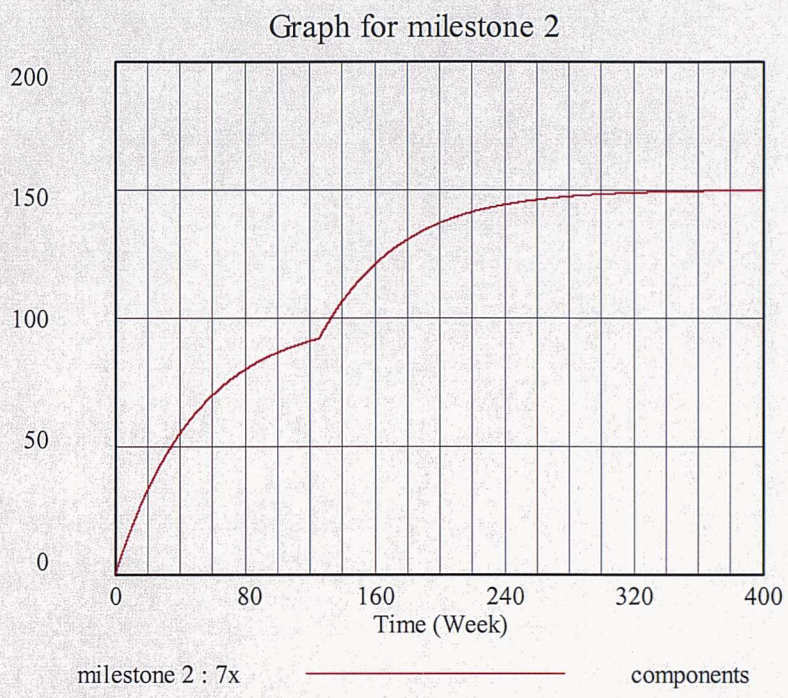


Figure 106. Graph of work done, Hardware Project A, modelled in Systems Dynamics

The Systems Dynamics model composed of simple process components has allowed us to suggest a plausible reason for the observed effort behaviour of the hardware project cell.

Chapter 8

Summary, Conclusion and Future Work

8.1 Summary

We have shown that systems development processes can be described as evolutionary systems; in the same way that software systems must evolve to meet new needs, process models must evolve to meet the needs of the developer and the system domain. Unless steps are taken to evolve the process to meet new needs, the developer's satisfaction with the process model declines when the model no longer meets their need to respond to technological and cultural domain changes in order to meet market expectation.

In chapter 2 we provided a background for the study. We examined definitions for software quality models, incorporated as the ISO universal single model. We discussed the effects and necessity of evolution in software, investigated by Lehman in the Feast projects [Lehman and Stenning 1996] and Henderson et al. in the SEBPC project [Henderson 2000]. We suggested that the processes that develop software were also subject to the evolutionary laws proposed by Lehman; declining quality (VII) and feedback (VIII). In order for software to evolve successfully, the processes by which software is developed must also evolve successfully. Warboys [Warboys, Greenwood et al. 2000] concurs that evolution is necessary to maintain competitiveness and the ability of an organisation to change and adapt to new threats and opportunities.

The ISO single universal model includes maintainability as an external attribute of software quality, however, Lehman's Laws for Evolutionary Systems and research

in the SEBPC project suggests that the definition may need to include the ability of the software to evolve, 'evolvability'.

We compared software process models from the early life cycle models (Waterfall and V), to iterative models, (Spiral and Win-Win Spiral) with newer incremental and iterative models that attempt to free evolutionary growth (Microsoft Synch and Stabilise, agile methods) and distributed models (Open Source).

We showed that measuring processes enables producers to predict schedule and effort (COCOMO, COCOCOII and COCOTs) and their ability to produce quality software (CMM and SPICE).

The measurement models themselves need to evolve, for example COCOMO and II become less satisfactory as process models evolve to use components, leading to development of COCOTs. However, they need to evolve further to support the dynamic control of processes required by higher levels of CMM and SPICE.

We examined Humphrey's work on the relationships between Universal, World and Atomic process models and the process improvement model implicit in the Capability Maturity Model, CMM and SPICE. We showed that process modelling and simulation support Humphrey's process improvement model [Humphrey 1990].

In chapter 3, Understanding Process Behaviour using Modelling and Simulation, we provided a study of the need for modelling to support the understanding of complex interactions of behaviours found in software processes and described methods for modelling and simulating them. In the light of our background studies into evolution and process improvement, and the importance of being able to understand the dynamics that cause behaviour, we restricted our investigation into those methods that allow feedback to be explicitly modelled in a dynamic representation.

The study showed that methods providing a graphical representation of the model and simulation enable greater understanding of behaviour and better communication of that understanding. We examined both continuous and discrete methods of dynamic modelling, showing that at a strategic level, continuous methods such as Systems Dynamics are most appropriate because they model how the process structure affects its behaviour and the process outcome, abstracting from individual entities.

We described how Abdel Hamid and Madnick [Abdel Hamid and Madnick 1991] used Systems Dynamics successfully to model a software development process (with a text based modelling representation), and carry out experiments on different planning, resourcing and quality assurance policies. We described two extremes in the level of abstraction of behaviour; Abdel Hamid and Madnick modelled at a low level of detail, concerned that too high an abstraction would leave out vital aspects of real world behaviour, whereas Wernick and Lehman [Wernick and Lehman 1998] showed that simpler models enabled understanding and insight to be retained.

We showed that it is easily possible to over-complicate models in an ad-hoc model building method so that clarity is lost and the models become impossible to evolve.

Chapter 4, The Cellular Manufacturing Process Model, presents a process modelling case study of CMPM, proposed by Peter Henderson and investigated at ICL [Chatters, Henderson et al. 1998]. We used Systems Dynamics to model the process in an ad-hoc manner, abstracting the model to the activities of completing component tasks and removing errors. The model showed the interaction between interdependent goals of cost (effort) and output quality as we vary policies of resource allocation between activities and process quality, and allowed us to understand the dynamics of the process.

By simulating the model we could reproduce predictions of effort and quality growth behaviours of the process. The model had a graphical representation that

allowed feedback in the process to be investigated and the feedback effects to be better understood. Through simulation, again with graphical representation, we were able to demonstrate the relationship between the quality of supplied components Q, and the output quality of the product P.

The model lacked correspondence with the implementation of CMPM and was not re-synchronised. As a result, there was a divergence of the metric definitions, explained in chapter 7. The case study showed the difficulty of retaining model and implementation synchronisation when the modelling method is ad-hoc, rather than systematic and designed for evolution.

In chapter 5, Evolutionary Systems Dynamic Model Building, we use evolutionary model building to investigate behavioural congruence between models of a process in different paradigms. We used a simple software development process as a case study and examined the effects of resource allocation policies on the schedule and quality of the product. The simple process produces software that contains defects; it has policies that control defect removal activities that depend on the perceived quality of the software in production. The process was modelled firstly by Monte Carlo methods and secondly, using Systems Dynamics.

We chose the process because, in process terms, it is relatively simple and yet the structural dynamics create complex behaviour the outcome of which, in terms of quality and schedule, is intuitively difficult to predict. The process has simple, visible activities and policies; it has precisely defined behaviour, we have a formal description of the system in the form of a model defined by Monte Carlo methods, and simulating the Monte Carlo model provides quantitative results to validate the models.

The Monte Carlo model was defined by a set of probabilistic choices describing the process activities of making code and removing defects, and policies for choosing between the activities.

We described systems dynamics models created using an evolutionary modelling method, where successive models brought closer correspondence to the simple process.

In this experiment, we described a sequence of five systems dynamics models in the evolution of the simple process model. As described by Warboys, each new model was the evolutionary offspring from an earlier model, and was a response to new requirements (refining the model abstraction), or discovery of non-conformances [Warboys, Greenwood et al. 2000]. Graphical representation of the model and simulation allowed convergence of the simple process and Systems Dynamics model through qualitative and quantitative comparison, showing the importance of qualitative and quantitative correspondence.

The final model shows a close behavioural correspondence to the simple process, assessed by qualitative comparison with the simple process and the Mathcad Monte Carlo model and also by quantitative comparison, whilst retaining abstraction and simplicity.

The modelling method follows the improvement model suggested by Humphrey for process improvement and extended in chapter 2, Figure 13 (model, compare, feedback, continue from the beginning)

The motivations for Chapter 6, Experiments in Modelling Software Processes using Components, were to show how components of software processes may be identified and modelled using Systems Dynamics and how these components may be used to build process models in a systematic, incremental, evolutionary way.

We conducted two experiments to show how abstract software development processes composed of simple, repeated components may be modelled and simulated to investigate their behaviour.

The first experiment examined perceived against predicted rates of product completion that may cause inappropriate schedule or resource allocation decisions

in process planning and control. The second experiment investigated the effects of reworking defective work on the completion of a product.

In each experiment we demonstrated that a simple process component could be identified, modelled and simulated with arbitrary values. In each case, we used the simplest possible process component that could be justified. The component was composed with other components to build process models with repeated process structures; the resulting models were simulated.

The simulations provided plausible explanations for the observed process behaviour, in both cases an asymptotic approach to the completion of growth targets.

We showed that whilst we may build models composed of repeated components, a better understanding of process component behaviour causality in the software development domain through quantitative simulation analysis is required before we may have confidence that the models will enable us to underpin process evolution.

Simple components with well understood behaviour that can be combined to form a process model will allow process modellers to have confidence about the causes of observed behaviour and propose process changes that will improve process outcomes.

In Chapter 7, From Qualitative to Predictive Quantitative Models, we discussed factors that affect our ability to build predictive quantitative systems dynamic models, using examples from the CMPM case study.

We found that ethnomethodology may provide an explanation of how frameworks of order from social world of the software development organisation and the process designer affect the capture of data for quantitative analysis.

We showed that understanding the effects of the ethnomethodolgical documentary method on interpretation of source data is important when evaluating the suitability of data for quantifying systems dynamic models for validation and prediction.

We showed that quantitative comparison and analysis revealed that the ad-hoc Systems Dynamics model of CMPM, whilst superficially showing behavioural correspondence with the CMPM model, diverged from the implementation.

We further showed that we could build a model composed of one of the simple process components identified in chapter 5 that could explain the product growth behaviour of the CMPM hardware project cell.

The experiments supported the understanding gained during the work in chapters 5 and 6, that qualitative and quantitative correspondence between the abstract model needs to be maintained at each evolutionary step.

8.2 Conclusion

In this thesis we have suggested that process designers may reduce the risks and increase the benefits of introducing new processes by improving their understanding and prediction of the effects of change. We have suggested that modelling and simulation allows us to examine and improve our understanding of the processes that produce software, and further, that the use of modelling and simulation is essential to achieving the goal of building effective, flexible, and evolvable processes.

Failure to understand the evolutionary nature of software and to build processes that enable successful evolution increases the risk that process changes will not achieve their predicted benefits and make only marginal improvements.

We have shown, through a case study investigating CMPM, that just using modelling and simulation is not enough to ensure that process improvement benefits will be achieved. Ad-hoc modelling and lack of synchronisation between

model and real world behaviour at each evolutionary step may cause failures of understanding and possible failures of process improvement.

As a means of reducing these risks, we have demonstrated an evolutionary modelling method that uses quantitative simulation to ensure close correspondence between the abstract model and the real world behaviour. Secondly, componentisation allows us to evolve process models in a more dependable way, by breaking processes down into components that are well understood, with predictable behaviour. Process designers may be better placed to design flexible processes that make good use of complex strategies like distribution, concurrency and feedback if we can develop re-usable process components, with well understood and predictable behaviour in the software development domain. With these methods process designersshould be able to avoid producing models that Nuthman describes as 'plausible nonsense'. [Nuthman 1994]

We were able to show that we must always be able to answer the question,

> 'but what have we modelled?'

8.3 Future work

The future work suggested by this thesis is to:

- identify simple process components in the software development domain,
- validate the process components by quantitative analysis using data from appropriate domains. The Open Source community may prove to be a rich source of data.
- and investigate component composition. In the experiments on using components in this thesis, we showed serial composition, repeating identical process structures, however, in order to support concurrent and distributed process models, we need to investigate other forms of composition.

Appendix 1 CMPM Systems Dynamic Model [Stella 1990 - 1998]

defects(t) = defects(t - dt) + (add_defects - remove_defects) * dt

INIT defects = 0

add_defects = complete_tasks_2*defects_per_task

remove_defects = effort_for_quality*defects*defects_per_man_day*(1-undetected_error_density)

tasks_2(t) = tasks_2(t - dt) + (complete_tasks_2) * dt

INIT tasks_2 = 0

complete_tasks_2 = effort_for_tasks*(target_size_2-tasks_2)*tasks_per_man_day_2*productivity_2*(1-undetected_error_density)

undetected_errors(t) = undetected_errors(t - dt) + (component_error_rate) * dt

INIT undetected_errors = 0

component_error_rate = PULSE(2,1,0)

cumulative_quality = (tasks_2/target_size_2)*(k2/1+defects+undetected_errors)

defects_per_man_day = .3

defects_per_task = 1

effort = 4

effort_for_quality = effort*(1-percentage_of_effort_for_tasks/100)

effort_for_tasks = effort*percentage_of_effort_for_tasks/100

k = .32

k2 = 0.32

percentage_of_effort_for_tasks = 60

Quality = (tasks_2/target_size_2)*k/(1+defects)

target_fraction_2 = (tasks_2/target_size_2)*100

target_size_2 = 303

tasks_per_man_day_2 = .2

undetected_error_density = undetected_errors/target_size_2

productivity_2 = GRAPH(target_fraction_2)

(0.00, 0.895), (10.0, 0.895), (20.0, 0.9), (30.0, 0.895), (40.0, 0.89), (50.0, 0.895), (60.0, 0.835), (70.0, 0.76), (80.0, 0.69), (90.0, 0.605), (100, 0.535)

Appendix 2 Monte Series of Models [Vensim 1988 -1997]

2.1 Monte 3 (Figure 41)

code that's bad= INTEG (

make buggy code,
60)
~ bad units
~ |

add to code=
35
~ code units/Month
~ |

code= INTEG (
make buggy code+make good code,
200)
~ code unit
~ |

make buggy code=
0.35
~ bad units/Month
~ |

make good code=
0.65
~ code unit/Month
~ |

```
********************************************************
        .Control
********************************************************~
                Simulation Control Paramaters
        |

FINAL TIME  = 1200
        ~       Month
        ~       The final time for the simulation.
        |

INITIAL TIME  = 0
        ~       Month
        ~       The initial time for the simulation.
        |

SAVEPER  =
    TIME STEP
        ~       Month
        ~       The frequency with which output is stored.
        |

TIME STEP  = 1
        ~       Month
        ~       The time step for the simulation.
        |
```

2.2 Monte 4 (Figure 45)

add to code=

IF THEN ELSE("code done?", 0, 0.35)
~ code units/Month
~ |

make buggy code=
IF THEN ELSE("code done?", 0, 0.35)
~ bad units/Month
~ |

"code done?"=
IF THEN ELSE(code>="final code size,N",1,0)
~
~ |

"final code size,N"=
400
~
~ |

make good code=
IF THEN ELSE("code done?", 0, 0.65)
~ code unit/Month
~ |

code= INTEG (
make buggy code+make good code,
200)
~ code unit

```
        ~               |

code that's bad= INTEG (
        make buggy code-remove bugs,
                25)
        ~       bad units
        ~               |

remove bugs=
        0.1
        ~       bad units/Month
        ~               |

**********************************************************
        .Control
**********************************************************~
                Simulation Control Paramaters
        |

FINAL TIME  = 1200
        ~       Month
        ~       The final time for the simulation.
        |

INITIAL TIME  = 0
        ~       Month
        ~       The initial time for the simulation.
        |

SAVEPER  =
     TIME STEP
```

~ Month

~ The frequency with which output is stored.

|

TIME STEP = 1

~ Month

~ The time step for the simulation.

|

2.3 Monte 6 (Figure 48)

```
make good code=
      IF THEN ELSE( "code done?", 0, (0.65-remove bugs))
      ~     code unit/Month
      ~           |

willingness to tolerate bugs=
      IF THEN ELSE( "code done?">=1,0,(IF THEN ELSE(XIDZ(code
that's bad, code, 0)>=0.25, \
            0,1)))
      ~
      ~           |

remove bugs=
      IF THEN ELSE(willingness to tolerate bugs=0, 1,0)
      ~     bad units/Month
      ~           |

add to code=
      IF THEN ELSE("code done?">=1, 0, 35)
      ~     code units/Month
      ~           |

make buggy code=
      IF THEN ELSE( "code done?">=1, 0, 0.35)
      ~     bad units/Month
      ~           |

"code done?"=
      IF THEN ELSE( code>="final code size,N",1,0)
      ~
      ~           |

"final code size,N"=
      400
      ~
      ~           |

code= INTEG (
      make buggy code+make good code,
            200)
      ~     code unit
      ~           |

code that's bad= INTEG (
      make buggy code-remove bugs,
            60)
      ~     bad units
      ~           |

********************************************************
      .Control
********************************************************~
```

```
            Simulation Control Paramaters
      |

FINAL TIME  = 600
      ~     Month
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Month
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Month
      ~     The frequency with which output is stored.
      |

TIME STEP  = 1
      ~     Month
      ~     The time step for the simulation.
      |
```

2.4 Monte 8 (Figure 50)

make buggy code=

IF THEN ELSE("code done?">=1, 0, (0.3+ (0.65*proportion of code complete)))

~ bad units/Month

~ |

work done= INTEG (

make good code +make buggy code +remove bugs,

0)

~ work units/Month

~ |

proportion of code complete=

code/"final code size,N"

~ dmnl

~ |

make good code=

IF THEN ELSE("code done?"=1, 0, (1-make buggy code-remove bugs))

~ code unit/Month

~ |

willingness to tolerate bugs=

IF THEN ELSE("code done?">=1,0,(IF THEN ELSE(XIDZ(code that's bad, code, 0)>=0.25, \

0,1)))

~

~ |

remove bugs=

```
        IF THEN ELSE(willingness to tolerate bugs=0, 1,0)
        ~       bad units/Month
        ~               |

add to code=
        IF THEN ELSE("code done?">=1, 0, 35)
        ~       code units/Month
        ~               |

"code done?"=
        IF THEN ELSE( code>="final code size,N",1,0)
        ~
        ~               |

"final code size,N"=
        400
        ~
        ~               |

code= INTEG (
        make buggy code+make good code,
                200)
        ~       code unit
        ~               |

code that's bad= INTEG (
        make buggy code-remove bugs,
                60)
        ~       bad units
        ~               |
```

```
********************************************************
        .Control
********************************************************~
                Simulation Control Paramaters
        |

FINAL TIME  = 1200
        ~       Month
        ~       The final time for the simulation.
        |

INITIAL TIME  = 0
        ~       Month
        ~       The initial time for the simulation.
        |

SAVEPER  =
    TIME STEP
        ~       Month
        ~       The frequency with which output is stored.
        |

TIME STEP  = 1
        ~       Month
        ~       The time step for the simulation.
        |
```

2.5 Monte 11 (Figure 58)

add to code=

```
        IF THEN ELSE("code done?"=1, 0, tendency to make a bug)
```

~ code units/Month
~ |

bb=
bconstant-aa
~ dmnl
~ |

bconstant=
0.85
~ dmnl
~ |

initial code=
200
~ code unit
~ |

code that's bad= INTEG (
make buggy code-remove bugs,
initial bugs)
~ bad units
~ |

initial bugs=
60
~ bad units
~ |

aa=
0.3

~ dmnl

~ |

tendency to make a bug=

(aa +(bb*proportion of code complete))

~

~ |

make buggy code=

IF THEN ELSE("code done?"=1, 0, tendency to make a bug)

~ bad units/Month

~ |

make good code=

IF THEN ELSE("code done?"=1, 0, (1-make buggy code-remove bugs))

~ code unit/Month

~ |

P=

(code-code that's bad)/code

~ dmnl

~ |

remove bugs=

IF THEN ELSE("code done?"=1, (1-tendency to make a bug), IF THEN ELSE(willingness to tolerate bugs\

= 0, 1, 0))

~ bad units/Month

~ |

work done= INTEG (

work,

```
        0)
    ~   work units
    ~       |

work=
    1
    ~   work unit/Month
    ~       |

bug tolerance level=
    0.25
    ~   dmnl
    ~       |

willingness to tolerate bugs=
    IF THEN ELSE(XIDZ(code that's bad, code, 0)>=bug tolerance level, 0,1)
    ~
    ~       |

proportion of code complete=
    code/"final code size,N"
    ~   dmnl
    ~       |

"code done?"=
    IF THEN ELSE( code>="final code size,N",1,0)
    ~
    ~       |

"final code size,N"=
    400
```

~
~ |

code= INTEG (
make buggy code+make good code,
initial code)
~ code unit
~ |

****
.Control
**********~
Simulation Control Paramaters
|

FINAL TIME = 1200
~ Month
~ The final time for the simulation.
|

INITIAL TIME = 0
~ Month
~ The initial time for the simulation.
|

SAVEPER =
TIME STEP
~ Month
~ The frequency with which output is stored.
|

TIME STEP = 1

~ Month

~ The time step for the simulation.

|

Appendix 3 Process Components Series of Models

3.1 Simple 1 (Figure 69)

(Ref: simple1.mdl

```
input rate 0 0 0=
      4
      ~     components/Month
      ~           |

input rate 0 1=
      4
      ~     components/Month
      ~           |

components 0 0= INTEG (
      input rate 0 0-output rate 0 0,
            0)
      ~
      ~           |

components 0 0 0= INTEG (
      input rate 0 0 0-output rate 0 0 0,
            0)
      ~
      ~           |

components 0 1= INTEG (
      input rate 0 1-output rate 0 1,
            0)
      ~
      ~           |

output rate 0 0=
      4
      ~     components/Month
      ~           |

output rate 0 0 0=
      4
      ~     components/Month
```

```
      ~           |

output rate 0 1=
      4
      ~     components/Month
      ~           |

input rate 0 0=
      4
      ~     components/Month
      ~           |

components= INTEG (
      input rate-output rate,
            100)
      ~
      ~           |

components 0= INTEG (
      input rate 0-output rate 0,
            100)
      ~
      ~           |

components 1= INTEG (
      input rate 1-output rate 1,
            100)
      ~
      ~           |

input rate=
      2
      ~     components/Month
      ~           |

input rate 0=
      4
      ~     components/Month
      ~           |

input rate 1=
      2
      ~     components/Month
      ~           |

output rate=
      2
      ~     components/Month
      ~           |

output rate 0=
      2
      ~     components/Month
      ~           |
```

```
output rate 1=
      4
      ~     components/Month
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |

FINAL TIME  = 100
      ~     Month
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Month
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Month
      ~     The frequency with which output is stored.
      |

TIME STEP  = 1
      ~     Month
      ~     The time step for the simulation.
      |
```

3.2 Simple 2 (Figure 71)

(Ref: simple2.mdl)

```
finished 1= INTEG (
      work rate 1,
            0)
      ~
      ~           |

components 0= INTEG (
      -work rate 0,
            100)
      ~
      ~           |

components 1= INTEG (
      -work rate 1,
            100)
      ~
```

```
      ~           |

components 2= INTEG (
      -work rate 2,
            100)
      ~
      ~           |

components 3= INTEG (
      -work rate 3,
            100)
      ~
      ~           |

work rate 1=
      2
      ~     components/Month
      ~           |

finished 0= INTEG (
      work rate 0,
            0)
      ~
      ~           |

work rate 3=
      2
      ~     components/Month
      ~           |

finished 2= INTEG (
      work rate 2,
            0)
      ~
      ~           |

finished 3= INTEG (
      work rate 3,
            0)
      ~
      ~           |

work rate 2=
      2
      ~     components/Month
      ~           |

work rate 0=
      2
      ~     components/Month
      ~           |

components= INTEG (
      -work rate,
            100)
```

```
      ~
      ~           |

finished= INTEG (
      work rate,
            0)
      ~
      ~           |


work rate=
      2
      ~     components/Month
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |

FINAL TIME  = 100
      ~     Month
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Month
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Month
      ~     The frequency with which output is stored.
      |

TIME STEP  = 1
      ~     Month
      ~     The time step for the simulation.
      |
```

3.3 Simple 3 (Figure 73)

(Ref: simple3.mdl)

```
components 0= INTEG (
      -work rate 0,
            100)
      ~     components
      ~           |
```

```
components 1= INTEG (
      work rate 0-work rate 1,
            100)
      ~     components
      ~           |

work rate 0=
      2
      ~
      ~           |

work rate 1=
      2
      ~     components/Month
      ~           |

components= INTEG (
      -work rate,
            100)
      ~     components
      ~           |

work rate=
      2
      ~     components/Month
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |


FINAL TIME  = 100
      ~     Month
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Month
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Month
      ~     The frequency with which output is stored.
      |

TIME STEP  = 1
      ~     Month
      ~     The time step for the simulation.
      |
```

3.4 Perceived Progress Model 1 (Figure 76)

(Ref: 2nd Monte 2)

```
complete components=
      IF THEN ELSE( "done?"= 1, 0, 2)
      ~     components/day
      ~           |

components= INTEG (
      -complete components,
            100)
      ~     components
      ~           |

"done?"=
      IF THEN ELSE( components>="final Size,N",0,1)
      ~
      ~           |

"final Size,N"=
      0
      ~
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |

FINAL TIME  = 100
      ~     Week
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Week
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Week
      ~     The frequency with which output is stored.
      |

TIME STEP  = 0.0625
      ~     Week
      ~     The time step for the simulation.
      |
```

3.5 Perceived Progress Model 2 (Figure 80)

(ref 2nd Monte 5)

```
"final Size,N 0"=
      100
      ~
      ~           |

"done 2?"=
      IF THEN ELSE(milestone 2>="final Size,N 0",0,1)
      ~
      ~           |

complete components 2=
      IF THEN ELSE("done 2?"=0, 0, 0)
      ~
      ~           |

milestone 2= INTEG (
      complete components-complete components 2,
            0)
      ~     components
      ~           |

complete components=
      IF THEN ELSE( "done?"=0, 2, 0)
      ~     components/day
      ~           |

components= INTEG (
      -complete components,
            100)
      ~     components
      ~           |

"done?"=
      IF THEN ELSE( components>="final Size,N",0,1)
      ~
      ~           |

"final Size,N"=
      0
      ~
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |
```

```
FINAL TIME  = 100
      ~     Week
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Week
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Week
      ~     The frequency with which output is stored.
      |

TIME STEP  = 0.0625
      ~     Week
      ~     The time step for the simulation.
      |
```

3.6 Perceived Progress Model 3 (Figure 83)

(Ref: 2nd Monte 3.mdl)

```
complete components=
      IF THEN ELSE( "done?", 0,(2*components/Target production))
      ~     components/day
      ~           |

components= INTEG (
      -complete components,
            Target production)
      ~     components
      ~           |

Target production=
      100
      ~     components
      ~           |

"done?"=
      IF THEN ELSE( components>="final Size,N",0,1)
      ~
      ~           |

"final Size,N"=
      0
      ~
      ~           |
```

```
********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |

FINAL TIME  = 100
      ~     Week
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Week
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Week
      ~     The frequency with which output is stored.
      |

TIME STEP  = 0.0625
      ~     Week
      ~     The time step for the simulation.
      |
```

3.7 Perceived Progress Model 4 (Figure 86)

(Ref: 2nd Monte 13.mdl)

```
milestone 2 0 0 0= INTEG (
      complete components 0 0 0,
            0)
      ~     components
      ~           |

complete components 0=
      2*components 0/Target production 0
      ~     components/day
      ~           |

complete components 0 0=
      2*components 0 0/Target production 0 0
      ~     components/day
      ~           |

complete components 0 0 0=
      1*components 0 0 0/Target production 0 0 0
      ~     components/day
      ~           |
```

```
components 0 0= INTEG (
      -complete components 0 0,
            Target production 0 0)
      ~     components
      ~           |

complete components 2 0=
      2*milestone 2 0/Target production 2 0
      ~     components/Week
      ~           |

complete components 2 0 0=
      2*milestone 2 0 0/Target production 2 0 0
      ~     components/Week
      ~           |

milestone 2 0= INTEG (
      complete components 0-complete components 2 0,
            0)
      ~     components
      ~           |

complete components 3 0=
      2*milestone 3 0/Target production 3 0
      ~
      ~           |

Target production 0 0=
      100
      ~     components
      ~           |

Target production 0 0 0=
      100
      ~     components
      ~           |

components 0= INTEG (
      -complete components 0,
            Target production 0)
      ~     components
      ~           |

milestone 3 0 0= INTEG (
      complete components 2 0 0,
            0)
      ~     components
      ~           |

components 0 0 0= INTEG (
      -complete components 0 0 0,
            Target production 0 0 0)
      ~     components
      ~           |
```

```
milestone 4 0= INTEG (
      complete components 3 0,
            0)
      ~
      ~           |

milestone 3 0= INTEG (
      complete components 2 0-complete components 3 0,
            0)
      ~
      ~           |

milestone 2 0 0= INTEG (
      complete components 0 0-complete components 2 0 0,
            0)
      ~     components
      ~           |

Target production 0=
      100
      ~     components
      ~           |

Target production 2 0 0=
      100
      ~
      ~           |

Target production 2 0=
      100
      ~
      ~           |

Target production 3 0=
      100
      ~
      ~           |

Target production 4=
      100
      ~     components
      ~           |

milestone 4= INTEG (
      complete components 3 -complete components 5,
            0)
      ~     components
      ~           |

milestone 5= INTEG (
      complete components 5,
            0)
      ~     components
      ~           |
```

```
complete components 5=
      2*milestone 4/Target production 4
      ~     components/Week
      ~           |

complete components=
      2*components/Target production
      ~     components/Week
      ~           |

complete components 3=
      2*milestone 3/Target production 3
      ~     components/Week
      ~           |

milestone 3= INTEG (
      complete components 2-complete components 3,
            0)
      ~
      ~           |

Target production 3=
      100
      ~     components
      ~           |

complete components 2=
      2*milestone 2/Target production 2
      ~     components/Week
      ~           |

milestone 2= INTEG (
      complete components-complete components 2,
            0)
      ~     components
      ~           |

Target production 2=
      100
      ~     components
      ~           |

components= INTEG (
      -complete components,
            Target production)
      ~     components
      ~           |

Target production=
      100
      ~     components
      ~           |

********************************************************
```

```
      .Control
*********************************************************~
             Simulation Control Paramaters
      |

FINAL TIME  = 500
      ~     Week
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Week
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Week
      ~     The frequency with which output is stored.
      |

TIME STEP  = 0.0625
      ~     Week
      ~     The time step for the simulation.
      |
```

3.8 Rework Model 1 (Figure 90)

(Ref: rework3.mdl)

```
complete tasks=
      60
      ~     tasks/Month
      ~           |

percentage bad work=
      0.1
      ~
      ~           |

return bad work=
      tasks 2*percentage bad work
      ~     tasks/Month
      ~           |

tasks= INTEG (
      +return bad work-complete tasks,
            1000)
      ~     tasks
      ~           |
```

```
tasks 2= INTEG (
      complete tasks-return bad work,
            0)
      ~     tasks
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |

FINAL TIME  = 100
      ~     Month
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Month
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Month
      ~     The frequency with which output is stored.
      |

TIME STEP  = 1
      ~     Month
      ~     The time step for the simulation.
      |
```

3.9 Rework Model 2 (Figure 93)

(Ref: rework3 component.mdl)

```
components= INTEG (
      +return defective work 0-complete tasks-return defective
work,
            1000)
      ~
      ~           |

complete tasks 0=
      0
      ~
      ~           |

components 0= INTEG (
      complete tasks-complete tasks 0-return defective work 0,
```

```
            0)
      ~
      ~           |

percentage bad work 0=
      0.1
      ~
      ~           |

return defective work 0=
      components 0*percentage bad work 0
      ~
      ~           |

return defective work=
      0*components*percentage bad work
      ~
      ~           |

complete tasks=
      60
      ~
      ~           |

percentage bad work=
      0.1
      ~
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |

FINAL TIME  = 100
      ~     Month
      ~     The final time for the simulation.
      |

INITIAL TIME  = 0
      ~     Month
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Month
      ~     The frequency with which output is stored.
      |

TIME STEP  = 1
      ~     Month
      ~     The time step for the simulation.
      |
```

3.10 Rework Model 3 (Figure 96)

Ref: (rework 3e v4.mdl)

```
done 1=
      IF THEN ELSE( task 3 2>=initial tasks 1, 1, 0)
      ~
      ~           |

initial tasks=
      1000
      ~
      ~           |

initial tasks 0=
      1000
      ~
      ~           |

complete task 2 2=
      IF THEN ELSE(done 1, 0, 100)
      ~     Tasks/Month
      ~           |

complete task1 2=
      IF THEN ELSE( done 0, 0, 100)
      ~     Tasks/Month
      ~           |

done=
      IF THEN ELSE( task 4 2>=initial tasks, 1, 0)
      ~
      ~           |

done 0=
      IF THEN ELSE( task 2 2>=initial tasks 0, 1, 0)
      ~
      ~           |

complete task 3 2=
      IF THEN ELSE(done, 0, 100)
      ~     Tasks/Month
      ~           |

initial tasks 1=
      1000
      ~
      ~           |

return bad work 3 2=
      IF THEN ELSE(done, 0, (task 4 2*percentage bad work 3 2))
      ~     Tasks/Month
      ~           |
```

```
task 2= INTEG (
      +complete task1+return bad work 2-complete task 2-return bad
work 1,
            0)
      ~     Tasks
      ~           |

return bad work 1=
      percentage bad work 1*task 2
      ~     Tasks/Month
      ~           |

task 1= INTEG (
      +return bad work 1-complete task1,
            1000)
      ~     Tasks
      ~           |

complete task 2=
      100
      ~     Tasks/Month
      ~           |

complete task 2 0=
      100
      ~     Tasks/Month
      ~           |

complete task 2 1=
      100
      ~     Tasks/Month
      ~           |

complete task 3=
      100
      ~     Tasks/Month
      ~           |

complete task 3 0=
      100
      ~     Tasks/Month
      ~           |

complete task 3 1=
      100
      ~     Tasks/Month
      ~           |

complete task1=
      100
      ~     Tasks/Month
      ~           |


complete task1 0=
```

```
      100
      ~     Tasks/Month
      ~           |

complete task1 1=
      100
      ~     Tasks/Month
      ~           |

percentage bad work 1=
      0
      ~     Dmnl
      ~           |

percentage bad work 1 0=
      0.1
      ~     Dmnl
      ~           |

percentage bad work 1 1=
      0.05
      ~     Dmnl
      ~           |

percentage bad work 1 2=
      0.3
      ~     Dmnl
      ~           |

percentage bad work 2=
      0
      ~     Dmnl
      ~           |

percentage bad work 2 0=
      0.1
      ~     Dmnl
      ~           |

percentage bad work 2 1=
      0.1
      ~     Dmnl
      ~           |

percentage bad work 2 2=
      0.2
      ~     Dmnl
      ~           |

percentage bad work 3=
      0
      ~     Dmnl
      ~           |

percentage bad work 3 0=
```

```
      0.1
      ~     Dmnl
      ~           |

percentage bad work 3 1=
      0.2
      ~     Dmnl
      ~           |

percentage bad work 3 2=
      0.1
      ~     Dmnl
      ~           |

return bad work 1 0=
      task 2 0*percentage bad work 1 0
      ~     Tasks/Month
      ~           |

return bad work 1 1=
      task 2 1*percentage bad work 1 1
      ~     Tasks/Month
      ~           |

return bad work 1 2=
      task 2 2*percentage bad work 1 2
      ~     Tasks/Month
      ~           |

return bad work 2=
      task 3*percentage bad work 2
      ~     Tasks/Month
      ~           |

return bad work 2 0=
      task 3 0*percentage bad work 2 0
      ~     Tasks/Month
      ~           |

return bad work 2 1=
      task 3 1*percentage bad work 2 1
      ~     Tasks/Month
      ~           |

return bad work 2 2=
      task 3 2*percentage bad work 2 2
      ~     Tasks/Month
      ~           |

return bad work 3=
      task 4*percentage bad work 3
      ~     Tasks/Month
      ~           |

return bad work 3 0=
```

```
      task 4 0*percentage bad work 3 0
      ~     Tasks/Month
      ~           |

return bad work 3 1=
      task 4 1*percentage bad work 3 1
      ~     Tasks/Month
      ~           |

task 1 0= INTEG (
      +return bad work 1 0-complete task1 0,
            1000)
      ~     Tasks
      ~           |

task 1 1= INTEG (
      +return bad work 1 1-complete task1 1,
            1000)
      ~     Tasks
      ~           |

task 1 2= INTEG (
      +return bad work 1 2-complete task1 2,
            initial tasks)
      ~     Tasks
      ~           |

task 2 0= INTEG (
      +complete task1 0+return bad work 2 0-complete task 2 0-
return bad work 1 0,
            0)
      ~     Tasks
      ~           |

task 2 1= INTEG (
      +complete task1 1+return bad work 2 1-complete task 2 1-
return bad work 1 1,
            0)
      ~     Tasks
      ~           |

task 2 2= INTEG (
      +complete task1 2+return bad work 2 2-complete task 2 2-
return bad work 1 2,
            0)
      ~     Tasks
      ~           |

task 3= INTEG (
      complete task 2+return bad work 3-complete task 3-return bad
work 2,
            0)
      ~     Tasks
      ~           |
```

```
task 3 0= INTEG (
      complete task 2 0+return bad work 3 0-complete task 3 0-
return bad work 2 0,
            0)
      ~     Tasks
      ~           |

task 3 1= INTEG (
      complete task 2 1+return bad work 3 1-complete task 3 1-
return bad work 2 1,
            0)
      ~     Tasks
      ~           |

task 3 2= INTEG (
      complete task 2 2+return bad work 3 2-complete task 3 2-
return bad work 2 2,
            0)
      ~     Tasks
      ~           |

task 4= INTEG (
      complete task 3-return bad work 3,
            0)
      ~     Tasks
      ~           |

task 4 0= INTEG (
      complete task 3 0-return bad work 3 0,
            0)
      ~     Tasks
      ~           |

task 4 1= INTEG (
      complete task 3 1-return bad work 3 1,
            0)
      ~     Tasks
      ~           |

task 4 2= INTEG (
      complete task 3 2-return bad work 3 2,
            0)
      ~     Tasks
      ~           |

********************************************************
      .Control
********************************************************~
            Simulation Control Paramaters
      |

FINAL TIME  = 100
      ~     Month
      ~     The final time for the simulation.
      |
```

```
INITIAL TIME  = 0
      ~     Month
      ~     The initial time for the simulation.
      |

SAVEPER  =
        TIME STEP
      ~     Month
      ~     The frequency with which output is stored.
      |

TIME STEP  = 1
      ~     Month
      ~     The time step for the simulation.
      |
```

Bibliography

Abdel Hamid, T. K., S. Kishore and R. Daniel (1993). "Software Project Control: An Experimental Investigation of Judgement with Fallible Information,." IEEE Transactions on Software Engineering **19**(16).

Abdel Hamid, T. K. and S. E. Madnick (1991). Software Project Dynamics, Prentice Hall.

Adams, E. (1984). "Optimizing preventive service of software products." IBM Journal of Research and Development **28**(1): 2-14.

Allen, P. M. (1988). "Dynamic Models of Evolving Systems." Systems Dynamics Review **4**(1-2): 109-130.

Beck, K. (1999). Extreme Programming Explained: Embrace Change, Addison-Wesley.

Belady, L. A. and M. M. Lehman (1972). An Introduction to Program Growth Dynamics. Statistical Computer Performance Evaluation, Brown University, NY, Academic Press.

Belady, L. A. and M. M. Lehman (1985). Program Evolution - Processes of Software Change. San Diego, CA, Academic Press Professional Inc.

Bennett, K. (2000). Software Maintenance and Evolution: a Roadmap. The Future of Software Engineering". A. Finkelstein. Limerick, Ireland, ACM Press: 73 - 87.

Boehm, B. (1988). "A Spiral Model of Software Development and Enhancement." IEEE Computer **21**(5): 61-72.

Boehm, B. (1997). COCOTS- COTS Software Integration Cost Modelling Study, University of Southern California.

Boehm, B. (2000). Spiral Development: Experience, Principles and Refinements. Spiral Development Workshop. W. J. Hansen, Carnegie Mellon Software Engineering Institute.

Boehm, B., C. Abts, B. Clark and S. Devnani-Chulan (1996). The Cocomo II Definition Manual, University of Southern California.

Boehm, B. and P. Bose (1994). A Collaborative Spiral Software Process Model Based on Theory W. ICSP 3, Reston, VA, IEEE.

Boehm, B., J. R. Brown and J. R. Kaspar (1978). Characteristics of Software Quality. Amsterdam, TRW.

Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy and R. Selby (1995). "Cost models for future software life cycle processes: COCOMO 2.0." Annals of Software Engineering **1**: 57 - 94.

Booch, G. (2004). The Limits of Software. Southampton, University of Southampton.

Brooks, F. (1995). The Mythical man Month:essays on software engineering. Reading, MA, Addison-Wesley.

Chatters, B., P. Henderson and C. Rostron (1998). Planning a complex Software and Systems Integration Project using the Cellular Manufacturing Process Model. The European Software Measurement Conference, FESMA '98, Antwerp.

Chatters, B., P. Henderson and C. Rostron (1998). SIMMER: Software and Systems Integration, Modelling, Metrics and Risks (Getting to Level 4). European Conf. on Software Process Improvement (EuroSPI), Goteborg.

Chatters, B., P. Henderson and C. Rostron (1999). SIMMER: Software and Systems Integration Modelling Metrics and Risks, ICL: 34.

Christie, A. M. (1998). Simulation In Support of Process Improvement. ProSim'98, Silver Falls, Oregon.

Cockburn, A. (2002). Agile Software Development, Addison-Wesley.

Coyle, G. (1996). Systems Dynamics Modelling - A Practical Approach. London, Chapman & Hall.

Coyle, G. (2000). "Qualitative and quantitative modelling in systems dynamics: some research questions." Systems Dynamics Review **16**(3, Fall 2000): 225-244.

Cusumano, M. A. and R. W. Selby (1997). "How Microsoft Builds Software." Communications of the ACM **40**(6).

Cusumano, M. A. and D. B. Yoffie (1998). Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft. New York, The Free Press.

Deming, W. E. (1982). Out of the Crisis. Cambridge, MA, MIT Center for Advanced Advanced Engineering Studies.

Dorling, A. (1993). "SPICE: Software Process Improvement and Capability Determination." Information and Software Technology **35**(6/7).

Dröschel, W. and M. Wiemers (1999). Das V-Modell 97. Oldenbourg.

Fenton, N. (1996). "Counterpoint: Do Standards Improve Product Quality?" IEEE Software **13**(1): 23-24.

Fenton, N. and S. L. Pfleeger (1997). Software Metrics: A Rigorous and Practical Approach, International Thomson Computer Press.

Fishman, G. S. (1996). Monte Carlo Concepts, Algorithms and Applications, Springer Verlag.

Ford, D. and J. Sterman (1997). Dynamic Modelling of Product Development Processes, Sloan School of Management, MIT.

Forrester, J. W. (1961). Industrial Dynamics. Waltham MA, Pegasus Communications.

Forrester, J. W. (1980). "Information Sources for the National Economy." Journal of the American Statistical Association **75**(371): 555-574.

Garfinkel, H. (1967). Studies in Ethnomethodology. Englewood Cliffs, Prentice Hall.

Garlan, D., R. Allen and J. Ockerbloom (1995). Architectural Mismatch or why it's hard to build systems out of existing parts. 17th International Conference on Software Engineering, Seattle,Washington, USA, ACM Press.

Glass, R. L. (1982). Modern Programming Practices: a Report from Industry. Englewood Cliffs, NJ, Prentice Hall.

Henderson, P. (1998). Laws for Dynamic Systems. International Conference on Software Re-Use (ICSR 98), Victoria, Canada, IEEE Computer Society Press.

Henderson, P., Ed. (2000). Systems Engineering For Business Process Change. London, Springer Verlag.

Henderson, P. and Y. Howard (1998). "Simulating a Process Strategy for Large Scale Software Development using System Dynamics." Software Process Improvement and Practice(5): 121 - 131.

Houston, D., G. Mackulak and J. Collofello (2000). Stochastic Simulation Risk Factor Potential Effects for Software Development Risk Management. ProSim 2000, London.

Humphrey, W. S. (1990). Managing the Software Process, Addison Wesley.

ICL (1999). Simmer Data. West Gorton.

isee systems (1998). Stella. Lebanon, New Hampshire.

Jones, C. (1986). Programmer Productivity. New York, McGraw Hill.

Kellner, M., R. Madachy and D. Raffo (1999). "Software process simulation modelling: Why?, What? How?" Journal of Systems and Software **46**(2/3): 91-105.

Kreutzer, W. (1986). System Simulation Programming Styles and Languages. New York, Addison Wesley.

Lehman, M. M. (1996). "Feedback in the Software Evolution Process." Information and Software Technology **38**(11 (special issue on Software Maintenance)): 681 - 686.

Lehman, M. M. (1996). Laws of Software Evolution Revisited. EWSPT96, Springer.

Lehman, M. M. and J. F. Ramil (1999). "The Impact of Feedback in the Global Feedback Process." Journal of Systems and Software **46**(2/3): 123-124.

Lehman, M. M. and J. F. Ramil (2000). Towards a Theory of Software Evolution - And its Practical Impact. Principles of Software Evolution, Kanazawa, Japan, IEEE.

Lehman, M. M. and J. F. Ramil (2002). "Software Evolution and Software Evolution Processes." Annals of Software Engineering **14**(special issue on Software Process-based Software Engineering): 275 - 309.

Lehman, M. M. and V. Stenning (1996). FEAST/1: Case for Support Part 2 Feedback Evolution and Software Technology. London, Dept of Computer Science, Imperial College London.

Lehman, M. M. and V. Stenning (1998). FEAST/2: Case for Support Part 2 Feedback Evolution and Software Technology. London, Dept of Computer Science, Imperial College London.

Lientz, B. P. and E. B. Swanson (1980). Software Maintenance Management. Reading, MA, Addison Wesley.

Littlewood, B. (1991). Software Reliability Modelling. Software Engineers Reference Book. J. McDermid, Butterworth Heineman.

Madachy, R. (1996). Systems Dynamics Modelling of an Inspection-Based Process. International Conference on Software Engineering, Berlin.

Mathcad (1999). Mathcad 2000 Professional, Mathsoft.

McCall, J. A., P. K. Richards and G. F. Walters (1977). Factors in Software Quality, US Rome Air development Center.

McConnell, S. (1996). Rapid Development: Taming Wild Software Schedules. Redmond, Wa, Microsoft Press.

Metropolis, N. and S. Ulam (1949). "The Monte Carlo Method." Journal of American Statistical Association **44**: 335 - 441.

Meyer, B. (2000). "Beyond Objects:What to Compose." Software Development.

Moon, J. and L. Sproull (2000). "Essence of Distributed Work: The Case of the Linux Kernel."

Nuthman, C. (1994). "Using human judgement in systems dynamics models of social systems." Systems Dynamics Review **10**: 1-19.

OSDN, O. S. D. N. (2002-2004). SourceForge.

Parnas, D. L. and P. C. Clements (1985). A Rational Design Process: How and Why to Fake It. TAPSOFT Joint Conference on Theory and Practice of Software Development, Berlin.

Paulk, M. C., B. Curtis, M. B. Chrissis and C. V. Weber (1994). The Capability Maturity Model - Guidelines for Improving the Software Process,. Carnegie Mellon Software Engineering Institute, Addison Wesley.

Perry, D. E., N. A. Staudenmayer and L. G. Votta (1994). "People, organisations and process improvement." IEEE Transactions on Software Engineering **11**(4): 36-45.

Pfleeger, S. L. and H. D. Rombach (1994). "Measurement Based Process Improvement." IEEE Transactions on Software Engineering.

Porter, M. E. (1985). Competitive Advantage: creative and sustaining superior performance, The Free Press.

Powell, A., K. Mander and D. Brown (1999). "Strategies for Lifecycle Concurrency and Iteration
- A Systems Dynamics Approach." Journal of Systems and Software **46**(2/3).

Raymond, E. S. (1999). The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly and Associations.

Richardson, G. P. and G. L. Pugh (1981). Introduction to Systems Dynamic Modelling and DYNAMO. Cambridge, MA, M.I.T Press.

Richmond, B. (1990). Systems Thinking: A Critical Set of Critical Thinking Skills for the '90's and Beyond.

Rodden, T., M. Rouncefield, I. a. Sommerville and S. Viller (2000). Social Viewpoints on Legacy Systems. Systems Engineering for Business Process Change. P. Henderson. London, Springer: 151-163.

Rodrigues, A. and J. Bowers (1996). "Systems Dynamics in project management: a comparative analysis with traditional methods." Systems Dynamics Review **12**(2): 121-139.

Royce, W. W. (1970). Managing the Development of Large Software Systems. WESCON, IEEE.

Rus, I., J. Collofello and P. Lakey (1999). "Software Process Simulation for Reliability Strategy Assessment." Journal of Systems and Software **46**(2/3).

Scacchi, W. (1999). "Experience with Software Process Simulation and Modelling." Journal of Systems and Software **46**(2/3): 183 -192.

Scacchi, W. (2002). "Understanding the Requirements for Developing Open Source Software Systems." IEE Proceedings -- Software **149**(1): 24-39.

Scacchi, W. (2004). Understanding free/Open source Software Evolution. Software Evolution. D. Perry. New York, John Wiley and Sons Inc.

Senge, P. (1990). The Fifth Discipline: The Art and Practice of the Learning Organisation. New York, Doubleday Currency.

Simon, H. A. (1996). The Sciences of the Artificial. Cambridge, MA, MIT Press.

Stella (1990 - 1998). Stella, High Performance Systems.

Sterman, J. (1985). "A Behavioural Model of the Economic Long wave." Journal of Economic Behaviour and Organization **6**(1): 17-53.

Sterman, J. (1989). "Misperceptions of Feedback in Dynamic Decision Making." Organisational Behavior and Human Decision Processes **35**(3).

Systems, H. P. Stella.

Szyperski, C., D. Gruntz and S. Murer (2002). Component Software:Beyond Object Oriented Programming - Second Edition, Addison-Wesley and ACM Press.

Vensim (1988 -1997). Vensim Professional32, Ventana: Ventana Simulation Environment.

Vensim (1988 - 1997). Vensim Professional32, Ventana: Ventana Simulation Environment.

Warboys, B. C., R. M. Greenwood and P. Kawalek (2000). Modelling and Co-Evolution of Business Processes and IT Systems. Systems Engineering for Business Process Change. P. Henderson. London, Springer Verlag. **1:** 10-23.

Weiss, D. M. and V. R. Basili (1985). "Evaluating Software development by analysis of changes: some data from the Software Engineering Laboratory." IEEE Transactions on Software Engineering **SE-11**(2): 157-68.

Wernick, P. and M. M. Lehman (1998). Software Process White Box Modelling for Feast/1. ProSim98, Silver Falls, Oregon.

Wolstenholme, E. F. (1990). System Enquiry - A Systems Dynamics Approach. Chichester, Wiley.

Zelkowitz, M. V. and D. Wallace (1997). "Experimental Validation in Software Engineering." Information and Software Technology **39**(11): 23 - 31.