The Dynamics of Software Testing

Abstract

In the modern information based society, failure of software systems can have significant consequences. It has been argued that increased attention to testing activities during the software development process can mitigate the probabilities of system failure after implementation. However, in order to justify investments in improved testing, the economic impacts of improper testing should be identified. In this paper, we propose a systematic approach to the evaluation of the economic impacts of software testing. The main factors affecting software testing are identified, and a computer simulation model is developed to investigate different testing scenarios. Usefulness of the suggested approach is demonstrated through several exploratory simulations. The results prove the utility of the System Dynamics modelling approach in building better understanding of the impact of software testing. Implications for software development practitioners, researchers, customers of software products and software support organisations are also discussed.

Keywords: Software testing, economics of software testing, System Dynamics, software development process.

Introduction

Information technology has become an integrated and ubiquitous element in all kinds of human activity. The Internet grew in less than two decades to achieve the status of the largest information repository in human history. Computers, interconnected by complex and interdependent networks, are running software applications that control air traffic, satellite positioning, banking transactions and hospital emergency rooms' equipment. With this increased dependence on information systems, technology failures might have disastrous effects. Such failures may result from both the hardware and software elements of the system, but while hardware design and manufacture has accumulated an admirable track record of reliability and dependability, software reliability has attracted much less attention.

The literature is inundated with reports on large scale disasters attributed mainly to software failures. In some of these cases, human lives were lost (Beynon-Davies, 1999), (McKenzie, 1994). Several authors argue that such disasters would have been avoided if the information system involved was designed and developed in a more careful manner (Parnas *et al.*, 1990), (Herzlich, 2005).

Different approaches have been proposed to address the software quality issue. While they differ significantly in the techniques they suggest, the overarching aims are very similar: preventing software failure by detecting and removing faults as early as possible in the software development lifecycle. Proposed solutions include: testing tools and methodologies, software development techniques, project management disciplines and training and development schemes. The field of software testing in particular grew substantially in the last decade. Researchers and practitioners within this field are developing innovative methods for ensuring the reliability, dependability and trustworthiness of software.

Despite the continuous development in the area of software testing, some may argue that the proposed methods and techniques were not adequately embraced by software developers. This may be attributed to two issues: firstly, no concrete evidence exists to prove the beneficial effects of IT expenditure on organisational productivity and performance. For example, in the USA, a few companies can offer evidence to demonstrate that IT is one of several factors involved in an increase in productivity, but at an aggregate level, there is little, if any, evidence of a positive impact (Carr, 2003), (Rai and Patnayakuni, 1997). Secondly, testing is perceived by many as a complex and expensive undertaking. This argument is likely to continue to grow as software applications are becoming larger and more complex (Whittaker, 2000).

In order to make the case for software testing, concrete evidence should be provided to convince software developers and their customers of the value of testing, and of the magnitude of negative consequences that may arise as a result of poor quality software. The Research Triangle Institute (RTI) conducted a study for the National Institute of Standards and Technology (NIST) in 2002 to estimate the impact of inadequate software testing on the United States economy (NIST, 2002). Its findings were that these costs range from \$22.2 to \$59.5 billion, or 0.06% of the U.S. GDP.

While this study provides significant results, the methods it adopted have several limitations. These methods depend on a variety of assumptions. One is that the respondents can make quantitative judgements concerning alternative prior choices (for example, what would we have saved by finding every error at the stage in which it was made). This requires the formation of a quantitative opinion, not about something the respondents have done, but about something that they have not done.

In addition, the RTI study examines only two segments of two industry sectors. The first is automotive and aerospace manufacturing and the second is financial services. The two sectors chosen and the particular technologies studied are well suited to quantification. Other sectors and technologies would arguably be less so. The study calculates the national impact of inadequate software testing by attributing the results for automotive and aerospace manufacturing to U.S. manufacturing industries at large and the results of financial services to the service sector as a whole. Then it aggregates the two in proportion to their relative contribution to the U.S. GDP. It does not suggest how the resulting figure might be disaggregated to identify the contributions made by each individual industry.

Other difficulties in calculating the impact of software testing arise from the fact that quantitative data about testing is not usually systematically kept by software development companies. Such data would cost significant amount to collect, maintain and apply. Most firms are also reluctant to disclose such information to researchers, on grounds of company confidentiality.

Moreover, software testing is an integral part of the software development process. Therefore, isolating the factors that might affect testing effectiveness might not be practically possible. Different approaches to software development require different testing methods and techniques. This limits the utility of any generic approach to calculate the effectiveness and impact of software testing.

In this paper we suggest a system dynamics approach to address some of these issues and to aid investigation into the expenditure of software testing, the costs of inadequate testing and the productivity gains and savings that would be generated by investments in testing. This approach would provide researchers and software development and testing practitioners with better insights into the value of testing and enable them to test different hypotheses about the most appropriate testing methodologies within specific contexts. The developed model can be utilised to make the case for software testing and to clarify any assumptions made about its impact.

Method and Approach

The growing interest in software testing led many researchers and practitioners to invest significant effort in investigating the value of testing and how it contributes to the organisation’s competitive advantage. However, these works have raised more questions than answers. Questions such as the following remain to be answered:

- What are the economic costs of inadequate IT systems testing infrastructure?
- What is the realistic cost reduction from feasible improvements to IT systems testing to the economy as a whole and to specific industry sectors?
- What costs do users of IT systems incur as a consequence of inadequate IT systems testing?
- What costs are incurred by IT development and support as a consequence of inadequate IT systems testing?
- Are there any significant differences in these costs between different industry sectors?

The problem was approached on the basis of a systems viewpoint. Our understanding of the nature of software development led us to believe that, in practice, the processes involved are characterised by delays and feedback, based on a set of relationships (structure) between the activities involved. This implied the need to adopt an approach that would enable us to determine the structure of software development processes and to identify the relationships between variables, and to use the understanding provided by doing this to guide the investigation into the nature and values of these relationships.

We suggest that System Dynamics would be the most appropriate methodology to achieve these aims. Developing a System Dynamics model of software testing based on a “stocks and flows” view, and supported by one of the available software simulation packages, would enable the behaviour of the system to be simulated and, crucially, to conduct true “what if?” experiments by altering the values of constituent variables or “policies” and demonstrating how this affects other values within the model. This functionality would, we maintain, provide invaluable insight into answering the questions listed above, while maintaining a high level of flexibility to adapt to different situations and contexts. The following section describes the model development process.

Model Development

System Boundary

This paper is intended to investigate the impact of software testing. Therefore, the main focus of the model will be confined to the software development phases, because software errors and defects are usually introduced during development. Testing is also an integral part of the software development activities.

These phases include software design, coding and testing. The majority of errors in software development occur during these phases (Nelson, 1974). We do not address the introduction of errors arising from a failure to correctly derive the requirements catalogue. Requirements elicitation is excluded from the system we are considering, as the people involved in software development usually do not have sufficient control over the requirements elicitation phase. In addition, some software projects start without clearly defined requirements. By excluding the requirements elicitation phase, the developed model will be more generic and applicable to a wider range of software development projects.

Another phase that is considered external to the system under consideration is the maintenance and support phase. This was also due to the lack of control of the software development team on activities performed in this phase.

Model Structure

Following the main focus of this research on software testing, the testing phase of the software development lifecycle will be the central element of the developed model. Software testing includes all the activities that are performed during software development to detect the maximum amount of errors within the software at an early stage in order to produce appropriate fixes with minimum cost.

Other elements of the system should be modelled to provide the required inputs into the software quality assurance process. These will include the software development activities were errors are actually generated. Generated errors are also greatly influenced by the expertise and skill level of the software development team (Belford et al., 1977), therefore the human resources aspect of the project should be taken into consideration. The effort dedicated to software testing and quality assurance has a substantial impact on the behaviour of the system (Lehman, 1980), and aspects of planning and control of the development project should be modelled to count for that impact.

The first integrated system dynamics model for software development projects was developed by Abdel-Hamid and Madnick (1991). The model was intended to explain the interacting factors involved in the software development process. It was the first model to reflect the complexity of software development processes. However, Abdel-

Hamid’s model had several limitations and we have also identified certain issues that we aim to address.

Firstly, Abdel-Hamid's model made significant assumptions that, though valid 15 years ago, should be reconsidered today to reflect the developments arising from research into different areas of software development. For example, the model made many assumptions about the Error Introduction Rate and the Error Detection Rate in the Software Testing section. The estimations of these values reported in the literature at the time when the model was developed varied substantially, and required major compromises on the accuracy of the values used. However, the area of software testing has witnessed considerable progress since then, and the recent results, if properly incorporated, should yield better reflection of the reality in the model.

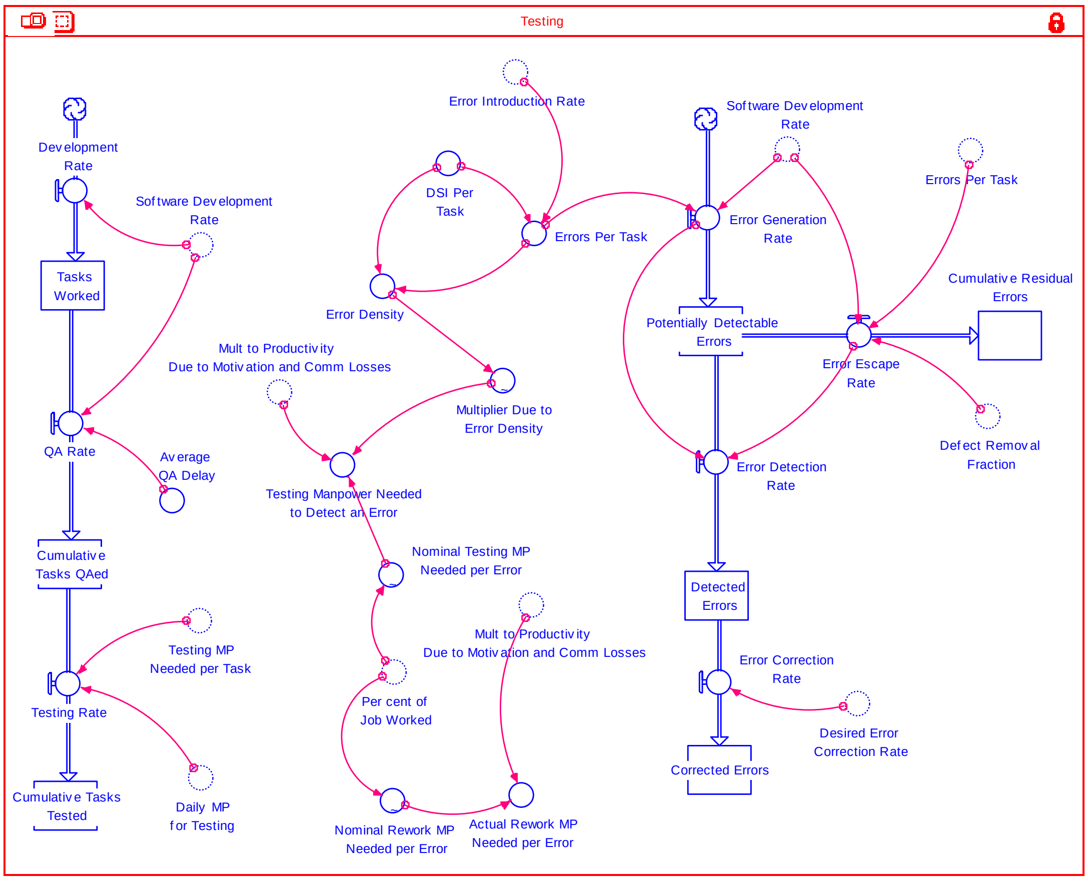
Secondly, the model reflects only one, then popular, software development lifecycle: the waterfall model. Most of the fundamental assumptions for the model development are based on this approach, which raises many questions about its applicability and validity to other approaches to software development. It is not clear, for instance, how the model copes with changing user requirements or system specification.

Thirdly, model validation is rather weak. Abdel-Hamid used only one case study to test the validity of his model, and no major efforts were directed into validating the model using data from different projects and within different organisational settings and industrial contexts. Furthermore, the selected case study was a software development project at NASA, which limits the ability to generalise the validation findings due to the specific characteristics of the development environment at NASA.

We used Abdel-Hamid’s model as the conceptual framework for the development of our software testing model as it fairly reflects a significant part of the software development process. However, fundamental differences in our model compared to Abdel-Hamid’s were the results of developing the model capabilities to reflect software testing dynamics in particular and eliminating many of the assumptions made in this context by integrating recent research findings in the area of software testing and its economic impact. We also validated the resulting model using more recent industry data, and incorporated error classifications and impact.

Software Testing Sector

Software errors are inevitably introduced during several stages of the development process. Many classifications of errors exist: coding errors, integration errors, software bugs, to name a few. Residual errors that remain in the software after deployment are the major cause of system’s failure. Software testing includes the activities directed towards the detection and removal of these errors as early as possible in the process so that the number of residual errors is kept to a minimum. Figure 1 shows the Testing Sector of our software testing model.



Testing
Error Introduction Rate
Development Rate
Software Development Rate
DSI Per Task
Errors Per Task
Error Generation Rate
Errors Per Task
Tasks Worked
Error Density
Cumulative Residual Errors
Potentially Detectable Errors
Mult to Productivity Due to Motivation and Comm Losses
Error Escape Rate
Multiplier Due to Error Density
QA Rate
Average QA Delay
Defect Removal Fraction
Testing Manpower Needed to Detect an Error
Error Detection Rate
Cumulative Tasks QAed
Nominal Testing MP Needed per Error
Detected Errors
Testing MP Needed per Task
Mult to Productivity Due to Motivation and Comm Losses
Error Correction Rate
Testing Rate
Per cent of Job Worked
Desired Error Correction Rate
Cumulative Tasks Tested
Daily MP for Testing
Corrected Errors
Nominal Rework MP Needed per Error
Actual Rework MP Needed per Error

Figure 1: Testing Sector

In most software development projects that involve more than one developer, the project is divided into tasks that are allocated to each developer or group of developers. A widely used measure of tasks is the number of Delivered Lines of Source Code (DSI) proposed by Boehm (1981). Each task will contain a number of errors, which can be calculated by multiplying the number of Delivered Lines of Source Code (DSI) in each task by the Error Introduction Rate. Chulani and Boehm developed the COnstructive QUALity MOdel (COQUALMO) to estimate the rates of software defect introduction and removal (1999) (Figure 2).

The Error Generation Rate is calculated by multiplying the number of Errors per Task by the Software Development Rate (which indicates the daily progress of the project, measured by task/day). The Error Detection Rate is determined by the Defect Removal Fraction calculated by the COQUALMO sector. Errors that escape detection eventually become residual errors and add to the Cumulative Residual Errors stock. These are the errors that cause the system to fail after implementation and thus determine the economical impact of testing. Detected Errors can be corrected before implementation. The Error Correction Rate is calculated according to the number of detected errors and the timeframe within which these errors should be corrected.

In order to ensure the appropriate allocation of resources to undertake the required testing and rework activities, man power requirements for the detection and correction of errors should be determined. The man power needed to detect an error is affected by several factors. First of all, error type has a significant role. The effort to detect an error changes as the project progresses simply because errors change from design to coding errors (Abdel-Hamid and Madnick, 1991). This behaviour is reflected in the graphical function: Nominal Testing Man Power Needed per Error. Error density also affects the required man power for error detection as the higher the error density in particular software the less effort it requires to detect one. Similarly, the required man power for error correction depends on the type of error being reworked (for example, design errors are much more demanding to correct than coding errors). This is also reflected in the model through the Nominal Correction Man Power Needed per Error graphical function. In addition, Man power requirements for both error detection and rework are affected by the communication overhead caused by the increase in the number of team members. When the team becomes larger, the difficulty of communications among team members increases, thus reducing the actual productivity of the team.

COQUALMO Sector

This sector models the COQUALMO approach to the calculation of error introduction and removal rates (Chulani and Boehm, 1999). The COQUALMO model identifies 21 defect introduction drivers that affect the Error Introduction Rate (Figure 2). These drivers are grouped into four main categories: Platform, Product, Personnel and Project. Chulani suggested numerical values for each of these drivers (Chulani, 1997).

These drivers are used to calculate the Quantity Adjustment Factor (QAF). The Error Introduction Rate can be determined by multiplying the QAF by a Nominal Error Introduction Rate (the number of errors without the impact of the quality adjustment factor) (Chulani and Boehm, 1999).

COQUALMO estimates the number of detected errors through the Defect Removal Fraction. This value is derived from three major profiles of testing activities, namely: Automated Analysis, People Reviews and Execution Testing and Tools. Each of these profiles has 6 levels, indicating the effectiveness of its contribution to defect removal. Table 1 explains these profiles and the six levels associated with each:

Rating	Automated Analysis	People Reviews	Execution Testing and Tools
Very Low	Simple compiler syntax checking.	No people review.	No testing.
Low	Basic compiler capabilities for static module-level code analysis, syntax, type-checking.	Ad-hoc informal walkthroughs. Minimal preparation, no follow-up.	Ad-hoc testing and debugging. Basic text-based debugger
Nominal	Some compiler extensions for static module and inter-module level code analysis, syntax, type-checking. Basic requirements and design consistency, traceability checking.	Well-defined sequence of preparation, review, minimal follow-up. Informal review roles and procedures.	Basic unit test, integration test, system test process. Basic test data management, problem tracking support. Test criteria based on checklists.

High	Intermediate-level module and inter-module code syntax and semantic analysis. Simple requirements/design view consistency checking.	Formal review roles and procedures applied to all products using basic checklists, follow up.	Well-defined test sequence tailored to organization (acceptance / alpha / beta / flight / etc.) test. Basic test coverage tools, test support system. Basic test process management.
Very High	More elaborate requirements/design view consistency checking. Basic distributed-processing and temporal analysis, model checking, symbolic execution.	Formal review roles and procedures applied to all product artifacts & changes (formal change control boards). Basic review checklists, root cause analysis. Use of historical data on inspection rate, preparation rate, fault density.	More advanced test tools, test data preparation, basic test oracle support, distributed monitoring and analysis, assertion checking. Metrics-based test process management.
Extra High	Formalised specification and verification. Advanced distributed processing and temporal analysis, model checking, symbolic execution. Consistency-checkable pre-conditions and post-conditions, but not mathematical theorems.	Formal review roles and procedures for fixes, change control. Extensive review checklists, root cause analysis. Continuous review process improvement. User/Customer involvement, Statistical Process Control.	Highly advanced tools for test oracles, distributed monitoring and analysis, assertion checking. Integration of automated analysis and test tools. Model-based test process management.

Table 1: Defect Removal Parameters (Chulani and Boehm,1999).

The COQUALMO model predicts the number of non trivial errors that are generated and detected during the software development process. Chulani (1999) indicates the importance of classifying errors in terms of their impact, which aligns with our aim of estimating the economic impact of software failure. She identified three categories of errors: critical (causing a system crash or unrecoverable data loss), high (causing impairment of a critical system function with no workaround solution) and medium (causing impairment of a critical system function but with a workaround solution). Other researchers proposed similar approaches to software error classification (Wagner & Seifert, 2005).

In our model, we follow COQUALMO's suggestion to classify Residual Errors into three categories: critical, high and medium. For each type of these errors, the economic impact is determined based on the cost to the business of the consequence of this error. This impact is industry-dependent and the model should allow enough flexibility to enter values that match the industry in every specific scenario. After the economic impact per error type is identified, the total value can be aggregated to give an indication of the likely consequences of a certain level of testing during the software development process.

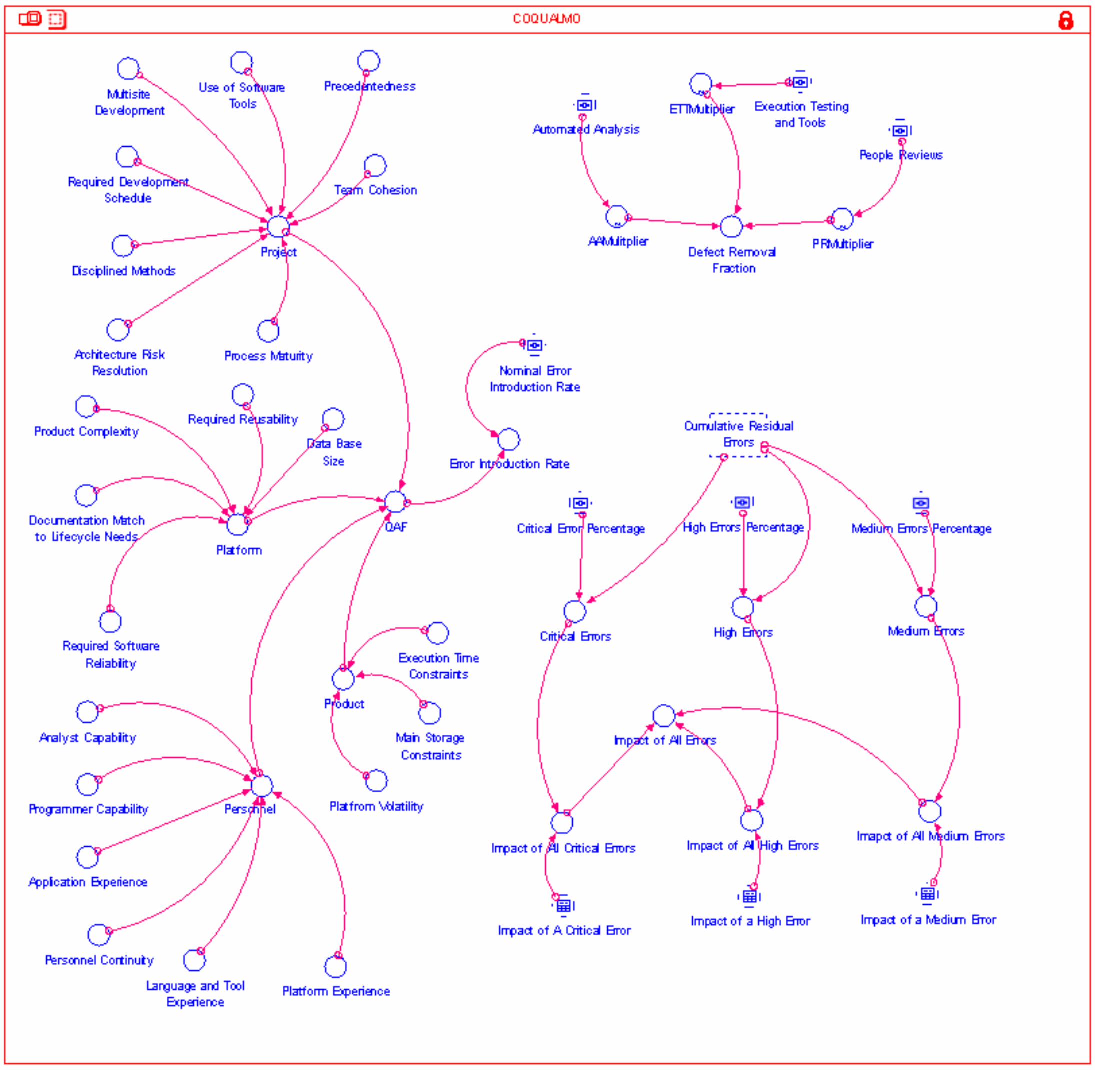


Figure 2: The COQUALMO Sector

Model Validation

In order to establish a certain level of confidence in the developed model, its behaviour must be validated against data collected from real projects. Unfortunately, detailed quality information about software development projects is rarely reported. However, few researchers have collected and reported such information in order to support software quality research. Two aspects of our model required validation: the software development process and the software testing elements. We used the same dataset reported by Abdel-Hamid and Madnick (1999) to validate the software development process as our model builds on their work. The data was collected form the NASA DE-A project. Our model produced similar results based on the specific project characteristics (Figures 3, 4). The slight differences can be attributed to the updates made to the error introduction and error removal rates, which will certainly result in slightly different distribution of the available workforce.

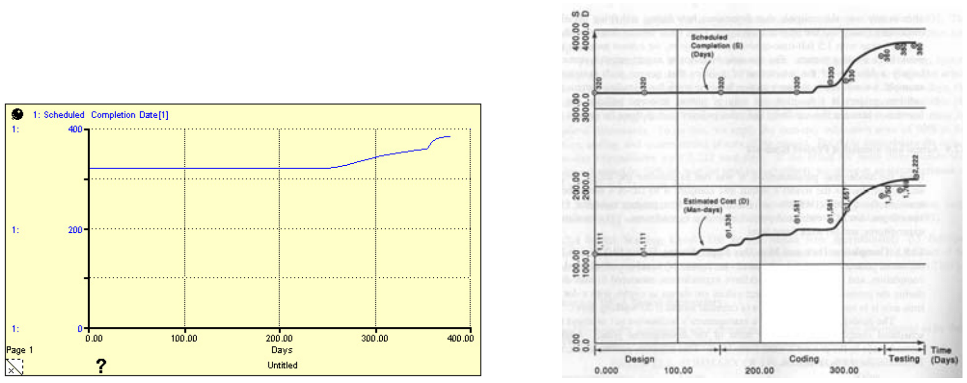


Figure 3: Scheduled Completion Date: Our Model (left) Abdel-hamid's (Right)

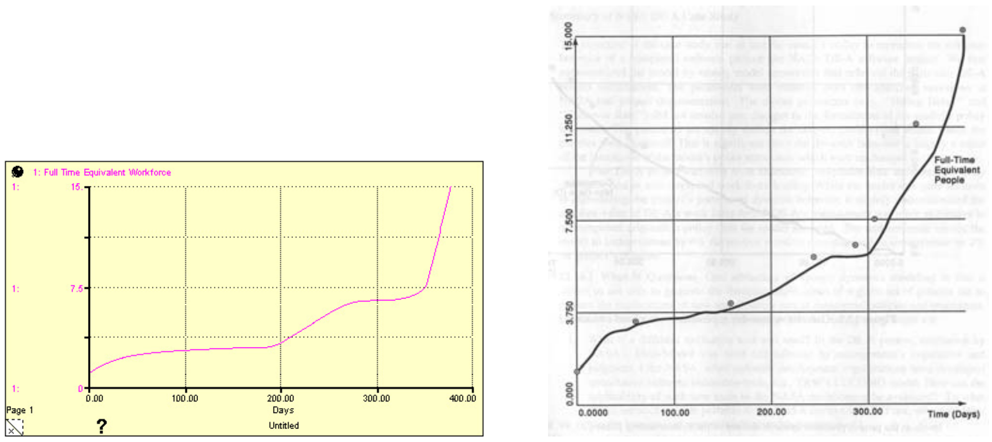


Figure 4: Full Time Equivalent Task Force: Our Model (Left) Abdel-Hamid's (Right)

The validation of the software testing elements of the model was conducted using datasets from the NASA Planetary Rover Robot software project, which was used by Boehm et al. (2004) to test the iDave quality model based on COQUALMO. This project consists of 380,000 DSI and has all its error introduction drivers and testing profiles identified. Our model produced very close results to those reported by Boehm et al. (2004). Table 2 shows the project variables and testing profiles used and Table 3 presents the number of residual errors calculated by our model and the values published by Boehm et al.

Parameter	Value	Parameter	Value
Platform			
Required Software Reliability	Very Low	Data Base Size	Low
Required Usability	Neutral	Documentation Match to Life-Cycle Needs	High
Product Complexity	Neutral		
Product			
Execution Time Constraint	Very High	Main Storage Constraint	Very High
Platform Volatility	Neutral		
Personnel			
Analyst Capability	Very High	Programmer Capability	Very High
Applications Experience	High	Platform Experience	High
Language and Tool Experience	High	Personnel Continuity	High

Project			
Use of Software Tools	High	Multisite Development	Very High
Required Development Schedule	Neutral	Disciplined Methods	Neutral
Precedentedness	High	Architecture/Risk Resolution	Very High
Team Cohesion	Very High	Process Maturity	High
Testing Parameters			
Automated Analysis		Very Low	
People Reviews		Very Low	
Execution Testing and Tools		Very Low	

Table 2: Software Testing Environment for the NASA Planetary Rover Case Study

Number of Residual Error	
Reported	9,216
Simulated	9,155

Table 3: Reported and Simulated Results of Residual Errors in NASA PR Case Study

Scenario Testing and Analysis

After an acceptable level of confidence in the model's behaviour is established, it can be utilised as a testing vehicle to experiment with different scenario options. Such experimentation should aim to answer possible questions about the economic impact of software testing. The results of this exercise can provide invaluable input into the planning process of any software development project. Decisions to include or exclude certain tools and practices can be better justified and taken with higher confidence.

In the following exercise we will use a hypothetical project (SoftWeb) to test different testing scenarios in order to gain some insight into the value of software testing. The project variables fed into the model are summarised in Table 4 below.

Variable	Value
Project Size	100,000 DSI
Time to Develop	400 Days
Hiring Delay	30 Days
DSI Per Task	60

Table 4: SoftWeb Project Variables

The project behaviour was simulated first to determine the impact of implementing People Reviews in the testing activities. The following three scenarios were simulated:

Scenario	Level of People Reviews	Description	Value in the Model
1	Low	No people review	0
2	Nominal	Well-defined sequence of preparation, review, minimal follow-up. Informal review roles and procedures.	2
3	Extra High	Formal review roles and procedures for fixes, change control. Extensive review checklists, root cause analysis. Continuous review process improvement.	5

		User/Customer involvement. Statistical Process Control.	

Table 5: People Review Scenarios

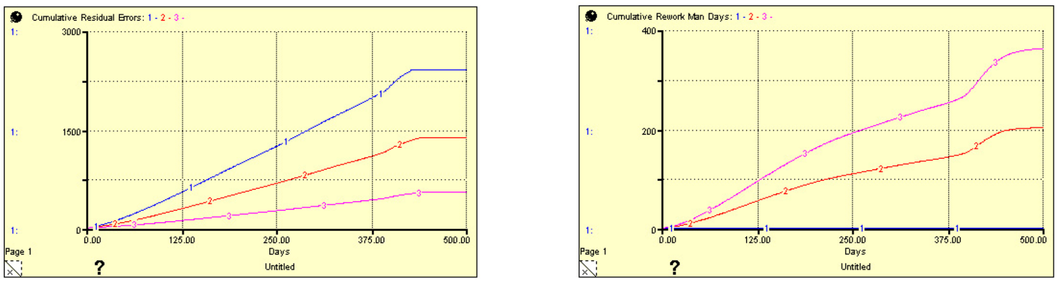


Figure 5: Residual Errors (left) and Cumulative Rework Man Days (right)

Figure 5 shows the results for two variables: Residual Errors and Cumulative Rework Man Days. When errors escape the testing process and become residual errors in the software, they are usually referred to as “defects”. The impact of incorporating people reviews in the testing process is clearly demonstrated by the significant reduction in the number of residual errors (or defects). This number decreased from 2480 defects when no people reviews were used, to 1380 using nominal levels of review and informal procedures, and ultimately reached 554 defects at the highest level of people reviews. The graph also exhibits an increase in the completion time of the project with the incorporation of higher levels of people reviews. Such delay would result in excessive costs and should be appropriately planned for from the outset.

Costs of additional reviewers and schedule slippages are not the only costs associated with implementing software testing techniques. The primary aim of any increase in testing levels is to discover as many errors as possible prior to software release. However, when these errors are detected, resources should be allocated to correct them. The Cumulative Rework Man Days graph in Figure 5 reflects this behaviour. As the level of people reviews is increased, the number of man days allocated for error rework grew from 0 when no reviews were conducted (as no errors were detected) to 202 at the nominal level and to 362 at the highest level. Both graphs shot upward towards the end of the project, which reflects the typical increase in testing activities before the software release date.

In order to justify the investment in improved testing (by incorporating people reviews in this scenario) and the subsequent costs of higher staffing levels and longer development times, the savings associated with decreasing the number of software defects should be calculated. These calculations require the attribution of a monetary value to each class of software defects. Accurate identification of the economic consequences of software failure due to each class of defect heavily depends on the context in which the software is implemented. Our model provides enough flexibility to adapt these values according to the specific industry in which the software product will be deployed. For the purposes of our scenario, we assume the following distribution of errors and the associated economic impact of the occurrence of each (Table 6).

Classification	Percentage	Impact £
Medium	5 %	100.00
High	38 %	1,000.00
Critical	57 %	10,000.00

Table 6: Distribution and Economic Impact of Residual Errors

The same simulation was repeated with the scenarios described above for the level of people reviews: 1: low, 2: nominal and 3: extra high. The aggregate economic impact of software defects (or residual errors) of each scenario is reproduced in Table 7.

	Scenario 1		**Scenario 2**		**Scenario 3**	
	No of Errors	**Impact £**	**No of Errors**	**Impact £**	**No of Errors**	**Impact £**
Medium	1,373.18	137,318.35	786.83	78,683.41	315.83	31,583.22
High	915.64	915,455.66	524.56	524,556.09	210.55	210,554.80
Critical	120.45	1,204,546.92	69.02	690,205.39	27.70	277,045.79
Total	**2,409.09**	**2,257,320.95**	**1,380.41**	**1,293,444.90**	**554.09**	**519,183.82**

Table 7: Economic Impact of Residual Errors in SoftWeb

The results reported in Table 7 clearly demonstrate the significant savings that could be achieved by incorporating higher levels of people reviews within the software development process. These savings can be weighed against the costs associated with the addition of people reviews in order to make an informed decision about the most appropriate level of testing.

One of the questions that may arise when deciding on software testing tools and techniques is: what is the most effective method of software testing? Project managers with tight budgets could utilise the model to compare the outcomes of several testing options in order to maximise the return on investments from their budget. The following simulation compares three scenarios. In the first scenario, the highest level of automated analysis is utilised, with no people reviews and no execution testing and tools. The second scenario demonstrates the use of the highest levels of people reviews with automated analysis and execution testing and tools set at their very low level. Lastly, in the third scenario, only execution testing and tools is implemented, with no automated analysis or people reviews. The results of the simulation are provided in Figure 6.

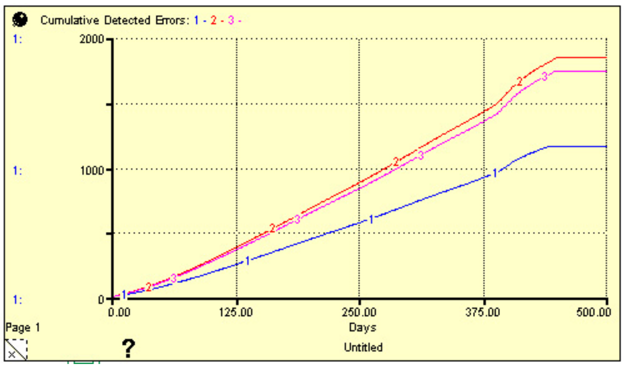


Figure 6: Comparison between different testing strategies

As evident from the graphs in Figure 6, people reviews proved to be the most effective method in the SoftWeb project, detecting 1855 errors. Execution testing and tools closely follows, yielding 1751 errors, while automated analysis proved to be much less effective, and captured 1164 errors only.

Conclusions and Implications

This paper presented the importance of software system testing, and provided some examples of large scale disasters caused mainly because of software failures. It has been argued that such failures could have been avoided if appropriate testing processes and mechanisms are integrated within the software development process. However, there is not enough evidence in the literature to support an objective justification for such claims. Testing is perceived as an expensive extension of the development process, and any investments in testing require a convincing and supported business case.

Many factors contribute to the difficulty of collecting and compiling compelling evidence about the economic impact of software testing. These include the complexity of the software development process itself, the large number of interacting factors within this process and industry-dependence of the economic impact of software failure, to name a few. Moreover, reliable data about defects in software projects is very hard to collect. Such information may be considered damaging to the reputation of the software development firm, or it may be protected for competitive reasons.

We proposed a System Dynamics approach to the problem of determining the economic impact of software testing. System Dynamics provides a structured method to examine the nature of the problematic situation from a systematic point of view. It also supports computer based simulation tools that enable the testing of different scenarios to support decision making.

The dynamic model presented in this paper incorporates the system testing activities within the overall software development process, and accounts for the interdependencies between the testing elements and other project factors. The model behaviour was validated using two published datasets about software development projects. Results produced by the model were similar to those reported in the literature, which established an acceptable level of confidence in the model’s behaviour.

The model utility was then demonstrated through a series of scenarios developed to answer several questions related to the impact of software testing. The simulation results of these scenarios revealed useful insights into the importance of testing. Significant economic savings could be achieved by improving the testing methods and techniques within the software development process and the implementation of new approaches. The scenarios also uncovered several issues that should be taken into consideration during the planning phase of the project. For example, in addition to the costs of more staffing to conduct the testing activities, these activities will expose higher numbers of software errors. Additional man power should be allocated to rework the discovered errors, which will incur more costs to the project. Furthermore, increased testing levels will lead to longer development times, as more effort should be allocated to testing and rework activities.

Our contribution has several implications to software development practitioners and researchers, customers of software products and software support organisations. Software development practitioners can utilise the model to explore different scenarios related to their particular project and make appropriate decisions accordingly. Such decisions may include determining the required level of particular testing technique, selecting the most effective testing mechanism, justifying investments in testing based on the economic consequences of lower software quality, planning the appropriate level of staffing during different stages of the project lifecycle and estimating project costs.

Researchers could use the model to investigate the effects of different testing tools and techniques on the overall software development process. This analysis can guide the development of new or improved tools and methods and the prioritisation of software testing research agenda. The computer simulation model offers an invaluable tool to experiment with different settings and options in a very short period of time. Something that can not be achieved otherwise without observing lengthy software development projects, which may cause serious delays to the research project. Moreover, the model can be used to safely test “destructive” scenarios which may lead to disastrous results if implemented in a real project.

The model also provides customers and support organisations with a useful tool to evaluate and compare several software products or development firms when making any purchase decisions. Software users with rigid requirements for reliability may mandate the use of certain levels of testing or particular tools and techniques in their requirement documents and software development contracts. More informed and justified decisions can be made with regard to the software expenditure when the costs and consequences can be more accurately estimated and accounted for.

References

Abdel-Hamid, T. K. & Madnick, S. E. 1991. *Software Project Dynamics.* Englewood Cliffs, New Jersey, USA: Prentice Hall.

Belford, P. C., Donahoo, J. D. & Heard, W. J. 1977. An Evaluation of The Effectiveness of Software Engineering Techniques. *COMPCON '77, Vol., Iss., 6-9 Sep 1977.*

Beynon-Davies, P. 1999. Human Error and Information Systems Failure: The Case of the London Ambulance Service Computer-aided Despatch System Project. *Interacting with Computers 11 6.*

Boehm, B. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall Inc.

Boehm, B., Huang, L., Jain, A. & Madachy, R. 2004. The ROI of Software Dependability: The iDAVE Model. *IEEE Software*, 21(3).

Carr, N. 2003. IT Doesn't Matter. *Harvard Business Review*.

Chulani, S. & Boehm, B. 1999. Modeling Software Defect Introduction Removal: COQUALMO (Constructive QUALity Model). The Center for Software Engineering, University of Southern California, Los Angeles, CA.

Chulani, S. 1997. Results of Delphi for the Defect Introduction Model – Sub-Model of the Cost/Quality Model Extension to COCOMO II. Technical Report, USC-CSE-97-504, Computer Science Department, University of Southern California, Centre for Software Engineering.

Herzlich, P. 2005. *The Need for Software Testing*. Ovum Research: London, UK.

Lehman, M. M.. 1980. Programs, Life Cycles, and Laws of Software Evolution *Proceedings of the IEEE, Vol.68, Iss.9*.

McKenzie, D. 1994. Computer-Related Accidental Death: an Empirical Exploration. *Science and Public Policy 21 4, pp. 233–248.*

Nelson, E. A.. 1974. Software Reliability, Verification and Validation. *Proceedings of the TRW Symposium on Reliable, Cost-Effective, Secure Software*. Redondo Beach, CA: TRW, Inc.

NIST. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing.

Parnas, D. L., van Schouwen, A. J. & Kwan, S. P. 1990. Evaluation of Safety-Critical Software. *Communications of the ACM, vol. 33, pp. 636-648*.

Rai, A. R. & Patnayakuni, N. 1997. Technology Investment and Business Performance. *Communications of the ACM* (40 :7).

Wagner, S. & Seifert, T. 2005. Software Quality Economics for Defect Detection Techniques Using Failure Prediction. In Proceedings of the 3rd Workshop on Software Quality (3-WoSQ). ACM Press.

Whittaker, J. A. 2000. What is Software Testing? And Why is it so hard? *IEEE Software, v.17 No.1, pp.70-79*.