

System Testing with
Object-Oriented Programs
Dissertation

Dafydd Vaughan
327039
dafydd@dafyddvaughan.co.uk

January 14, 2007

Abstract

Today's software systems have become increasingly complex with more risks associated with failure. This has meant that a greater importance has been placed on testing to ensure the software works correctly and matches the specification. This document looks at software testing and new technologies and techniques that can be used to both improve the efficiency and rigour of testing software.

Contents

Outline	4
1 Introduction	5
1.1 Why Test Software?	5
1.2 Document Aims	6
2 Background	7
2.1 Levels of Software Testing	7
2.1.1 Requirements Testing	8
2.1.2 Design Testing	8
2.1.3 Module Testing	9
2.1.4 Unit Testing	9
2.1.5 Integration Testing	9
2.1.6 System Testing	9
2.1.7 Acceptance Testing	10
2.2 Object Oriented Programming	10
2.3 Unified Modelling Language (UML)	10
2.3.1 The History of UML	11
2.3.2 UML Diagrams	11
3 System Testing	14
4 Automating Testing	15
4.1 Example	15
4.2 Requirements Analysis & Low Level Design	16
4.3 Activity Diagrams	16
4.4 The Use Cases	17

<i>CS339 Testing Dissertation - Dafydd Vaughan 327039</i>	3
4.5 Expanding the Use Cases	18
4.6 The Final Steps	20
5 Conclusion	21
Glossary	22
References	24

Outline

In recent years, software testing has become an increasingly important part of the development of new systems. Many high profile failures of new systems - some even fatal - have created a bad image of the software development industry. This image, together with the growing use of computers in safety-critical and business-critical areas have forced developers to place a greater emphasis on testing software to make sure it works correctly and successfully matches the requirements of their customers. New testing techniques and practices including international standards for specifying how systems should work - such as Unified Modelling Language (UML) - have been established.

Despite these huge improvements, the time taken to complete the testing of software to a satisfactory level is increasing. This is, in part, due to the increased detail of the testing taking place, but is also caused by the growing complexity of today's software. Further research is being conducted into automating some of this testing process to enable quicker software testing to take place.

The first part of this document (section 1) looks at some of the problems faced by the software testing industry and why we test new software. Section 2 looks at some of the background topics such as the different levels of software testing, object oriented programming and the Unified Modelling Language (UML). Section 3 looks in more detail about testing completed systems to make sure they satisfy their specifications, and Section 4 looks at one of the ways in which UML can be used to automate the process.

1 Introduction

1.1 Why Test Software?

Until recently, comprehensive software testing has been considered as an unimportant part of software development[5]. Businesses viewed the process of fully testing software an unnecessary expenditure in both time and money. Developers saw the process as boring and repetitive. These views meant that software underwent only as much testing as was necessary to launch the product. The testing that was performed usually happened at the end of the development cycle. It therefore, only consisted of checking the software worked, and not that it matched the requirements of the project.

This attitude towards testing meant that software development gained a negative image. In addition to this, a large number of systems produced did not completely match their original specification and a large amount of money and time was spent making changes to these systems to correct these problems. These delays undermined the credibility of the system even when it was finally delivered.

One example of the problems caused by a lack of testing was seen in 1999 when the UK Passport Agency introduced a new computer system in two of its six offices. This system aimed to replace its older systems, making the handling of passport applications easier. It was planned to be rolled out to all six offices before the busy season in mid 1999[11]. However, after being implemented in two of the offices - Liverpool and Newport - the system failed, resulting in a huge backlog of applications. The situation was made worse by customers who started to lose confidence in the system. The public began making applications earlier than usual, increasing the backlog. By the end of June 1999, applications were up by 29 per cent.

The cost of dealing with this high profile failure amounted to around £12.6 million, not including approximately £9 million in lost business borne by the developers of the system.

Another good example of the potential seriousness of this attitude are the incidents from the Therac-25 radiation therapy machine. In these cases, there were at least 6 deaths and other serious injuries caused by massive radiation overdoses[9]. These events have been attributed to a fault in the control software of the device that had not been picked up during testing. Accidents like this in safety-critical systems have potentially lethal consequences. Comprehensive testing the software is essential to prevent these problems.

With such high profile failures the need to develop comprehensive testing systems has grown momentum. The introduction of computers into more and more safety-critical systems such as the aviation and medical industries has made test-

ing essential. The numerous high profile failures due to incorrect specifications has led to businesses questioning how new systems are developed. They have begun to realise that the cost of software testing outweighs the potential costs of making changes after production as well as the potential loss of money and life should safety-critical or business-critical systems fail.

Recently, other problems have arisen with software testing. Computer software is becoming bigger and much more complex. This has meant that it has become more complicated to test and the time taken to complete each level of testing has increased. However, research and new technologies such as modelling languages could potentially help solve this problem. Testing research is now looking at ways of automating some of this process to speed up the time taken to test software and create more efficient tests.

These changed attitudes have meant that testing has become much more important and is a growing industry. In some cases, testing is considered more important than the code itself. So, how can the industry develop cost effective comprehensive testing with increasingly complex code? Could the testing be accomplished through programs themselves, thereby automating some of the process?

1.2 Document Aims

The following pages of this dissertation will look at the levels of testing that take place during the development cycle. We will consider how object oriented software changes the way system testing is performed. We will also look at how this object oriented software makes use of modelling languages such as UML to quickly develop test cases and reduce the time taken to complete this level of testing.

2 Background

2.1 Levels of Software Testing

The testing of software involves a lot more than just ensuring a product appears to work at the end of its development. Testing should take place at several levels within the development cycle to ensure that not only the product works correctly but the product matches what the customer expected and meets all the requirements successfully.

The major levels of testing include:

- Requirements Testing
- Design Testing
- Module Testing
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

This section will give a brief overview of these different levels of testing and where they fit into the development lifecycle.

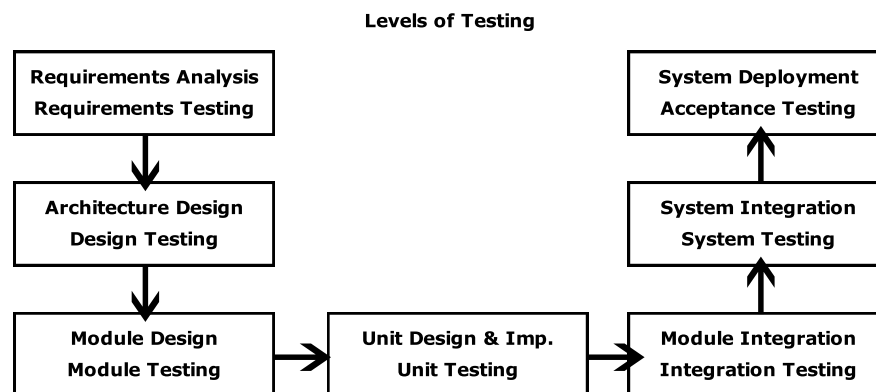


Figure 1: Levels of Software Testing

When software is produced, it ideally progresses through seven levels of development - known as the development lifecycle (shown in figure 1). At the start of the project, the developers work along side the customer to develop a 'Requirements Specification'. This specification outlines exactly what the program should be able to achieve, how it is supposed to achieve it, and even what hardware it should be run on. The second stage of development involves designing the system architecture and the 'modules' of code which will need to be produced. Once this design is completed, stages three and four involve creating these modules and fitting them together to produce 'units' of code.

The fifth stage of development will take these units and merge them (along with any 3rd party code) to produce a complete system. This stage is called integration and is the last stage that involves working with the code of the software. The second to last stage of development will see the software installed on the system that will be used by the customer, and finally the completed software will be deployed so it can be used.

Each stage of development has a level of testing associated with it to ensure the stage has been completed successfully, these are discussed in the following sections.

2.1.1 Requirements Testing

Requirements Testing is the first level of software testing and should take place while producing the requirements specification from which the product will be produced[13]. One of the biggest problems with developing software is that it is very often designed based on wrong, vague or missing lists of requirements. This is one of the major reasons why new computer systems fail to meet expectations.

Requirements testing is intended to help reduce the possibility of software being developed that does not meet the expectation of the customer. This level of testing also has the added benefit that it will reduce the large costs of both time and money involved in changing the final software should it not meet all the requirements.

2.1.2 Design Testing

Design Testing is performed to ensure that the architecture of the system is designed correctly to meet all the requirements specified in the previous level[14]. If this design is correct, the developers can then begin to produce the code for each module of the system to perform the desired functions. This level of testing is very important as an error introduced at this level could be very expensive and time consuming to resolve later in the development cycle.

2.1.3 Module Testing

Once the modules designed above have been constructed by the project developers, they will need to be tested to ensure that they work correctly[1]. This stage of testing is called Module Testing. These modules are tested using the test cases and data developed during the previous stage of the project. This level aims to reduce the number of incorrect modules passed on to the unit development stage - thus reducing the overall number of errors which could potentially occur later in development.

2.1.4 Unit Testing

Unit Testing is intended to check that groups of modules work together correctly. This level of testing occurs before all the units are integrated into the final system and just concentrates on ensuring that small groups of modules (units) work together. The purpose of this level of testing is again to reduce the possibility of errors being passed onto the next level of software development.

2.1.5 Integration Testing

Integration Testing is performed once all the modules and units have been combined together to produce the final complete system[4]. This level of testing is performed to ensure all the units of the system work together as expected and interface issues are minimised. This level of testing is extremely important as units are usually produced independently and this is the first occasion in which they are joined together. It is also an important level of testing when using 3rd party components which have not been tested in previous levels due to copyright issues. These components will be used for the first time when integrated into the full system.

Integration testing is the final stage of testing in which the inner workings of the program (the code) is examined. All future levels are considered Black box and look only at the results produced.

2.1.6 System Testing

System Testing is concerned with ensuring the final system matches the specification layed out in the requirements at the beginning of the project. This is the final stage of testing before the customer sees the completed program. Further details of system testing can be found in section 3.

2.1.7 Acceptance Testing

Acceptance Testing is the final level of testing that will take place in the software development cycle. This stage will take place after the software is released to the customer and is intended to ensure that the software satisfies the requirements set by the customer[10]. If the software passes these tests, it will be accepted by the customer as final and complete. Unlike other levels of testing, acceptance testing is performed by the customer - not the developer - and ensures that the software they are receiving is fit for purpose and satisfies their needs. All the previous levels of testing are intended to increase the likelihood that the software produced will pass this level of testing. Should the software fail here, it could be very expensive to make the required changes to bring it inline with the specification and will also undermine confidence in the system which could even lead to delays from customers refusing to use it.

2.2 Object Oriented Programming

Object oriented programming is a way of developing software from groups of 'objects'[12]. The use of objects instead of classical procedures makes object oriented programming more logical as its use of objects usually relate to groups of functions. For example, in a library system you might have the objects 'customer', 'book' and 'loan'. The 'customer' object would contain all the variables related to a customer - such as name, address, contact number and number of books on loan - as well as methods (actions) such as 'send overdue notice' which sends the customer an overdue notice.

One of the biggest advantages of object oriented programming is code reuse. Objects can easily be reused in other applications, cutting down the development time. This however also presents a potential problem for testing. Regularly in large systems, 3rd party components and objects will be used to provide some of the functionality. In many cases, it is not possible for developers to test these modules as the code is not available to them. Instead, they must rely on the 3rd party adequately testing the component. Integration testing becomes very important in this situation as it is the first time that the developers can test the module and its interfaces.

Object oriented software also affects other aspects of testing such as the composition of the modules and units. In object oriented software, modules are more likely to be methods and units are likely to be objects or groups of objects. In classic programming this distinction is less obvious.

2.3 Unified Modelling Language (UML)

UML is the international standard for designing, specifying and modelling computer systems. In this section we will outline what UML is and the history of this language.

2.3.1 The History of UML

Prior to 1994, there was no standard method or notation with which to model a computer system. Many books had been written about the design and analysis of software but no standard had emerged. Of these different methodologies, three stand out. Grady Booch a developer from Rational Software had a methodology based on Ada; Jim Rumbaugh, who led a team of researchers at General Electric, developed a method called Object Modeling Technique (OMT). Ivar Jacobson also developed a method based on his experience with telephone switches for Ericsson[3]. All of the methods of the time were considered similar but with very different notation which caused confusion in the industry. However talk of standardization was avoided by each of the developers.

This changed in 1994 when work began to unify these three methods to create UML. In 1995, Booch and Rumbaugh merged their two modelling techniques to create the 'Unified Method version 0.8'. Later, Jacobson joined the pair and Unified Method was merged with his object oriented technique to create 'Unified Modelling Language 0.9'[8]. In early 1997 UML was proposed to the Object Management Group (OMG) as its standard for modelling information technology systems. The Object Management Group is a not-for-profit organisation that develops and maintains specifications for enabling applications to work together in the computer industry. The OMG accepted UML as their standard in November 1997.

Several revisions to the standard have taken place since it was adopted with the latest version - UML 2.0 - being released in October 2004. UML has now been accepted as an international standard by the International Organization for Standardization (ISO) - number ISO/IEC 19501:2005.

2.3.2 UML Diagrams

Unlike other languages, UML is a modelling language that consists of graphical diagrams to specify the system. The latest UML standard - UML 2.0 - consists of 13 different diagrams split into 2 categories - structural diagrams and behavioral diagrams. In this section we will briefly describe each diagram and how they are used to specify a system.

The structural diagrams below are used to show the structure of the system and detail how objects relate to each other. This category consists of six diagrams - Class, Component, Composite structure, Deployment, Package and Object

diagrams.

Class Diagrams

These diagrams show the composition of the classes and interfaces (and their relationships) that make up the system being modelled. Very detailed class diagrams can be used to quickly generate code.

Component Diagrams

These diagrams show how the full system is split into several independent subsystems. These diagrams group several classes together and provide an interface for accessing them. As a result, they provide a Black box view of the implementation.

Composite Structure Diagrams

These diagrams provide a link between class diagrams and composite diagrams but do not show the same level of detail. They show how elements are combined to show complex relationships. This type of diagram is new to UML 2.0.

Deployment Diagrams

These diagrams show how the system is configured at deployment and show how the components of this software are linked to the hardware that will execute it.

Package Diagrams

These are an extension of the class diagrams and show how classes and interfaces are grouped together.

Object Diagrams

These diagrams are very similar to class diagrams but instead show how the instances of a class are related at a point in time.

The behavioral diagrams below are used to show how elements within the system behave and the operations they can perform. This category consists of seven diagrams - Activity, Communication, Interaction overview, Sequence, State Machine, Timing and Use case diagrams.

Activity Diagrams

These diagrams show the flow of the system from one activity to another. They are similar to classic flow charts but provide more detail. These diagrams are also used to help automate testing and will be discussed in more detail later.

Communication Diagrams

These diagrams are similar to interaction overview diagrams and are used to show the elements involved in system interactions. Unlike sequence diagrams, they do not show detailed sequencing or timing.

Interaction Overview Diagrams

These diagrams are similar to activity diagrams but show the objects that are involved in performing each activity.

Sequence Diagrams

These diagrams are the most common behaviour diagrams used in UML and are based on Interaction diagrams. These show the type and sequence of the messages passed between system objects during execution.

State Machine Diagrams

State diagrams show the different states in which part of the system can reside during execution and the transitions between them. The part of the diagram shown can be a single class, a group of classes or the entire system.

Timing Diagrams

These diagrams are similar to sequence and interaction diagrams but concentrate on the timing specifications of the message - such as how long the system has to respond to a message once it is received.

Use Case Diagrams

These diagrams show the functional requirements of the system. They provide a view of the system that focusses on use needs. Use Case diagrams are an important part of object oriented system testing and will be used in later sections to show how to create test cases.

3 System Testing

System testing is one of the levels of testing that comes just before the customer of the product gets to see the final version. This level of testing is concerned with ensuring that the program matches the specification that was drawn up at the beginning of the project[6] One of the most important points to make about system testing is that unlike integration testing, it is not concerned with how the system works. Instead, it is more concerned with the results produced. For this reason, system testing is considered to be "Black box". This means it just looks at what happens on the outside but cannot look on the inside.

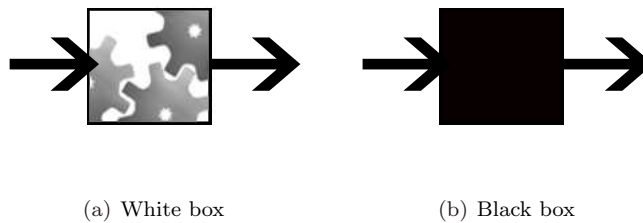


Figure 2: White Box and Black Box Testing

Another point to make about system testing is that it should take place in a set up that accurately reflects (as much as possible) the system in which the product will be deployed. This way, errors caused by the customers setup can be found and fixed prior to release.

On the surface, it would appear that system tests for object oriented systems would be no different to system tests for non-object oriented systems. While this essentially is the case, since system testing is black box and not concerned with how the program works, the process of actually generating the test cases **is** different. Specifications for object oriented software can be very different from non-object oriented software. For example, object-oriented systems can be modelled in UML (as seen in 2.3) and use 'Use Case' and 'Class' diagrams. These diagrams are then used to produce the test cases. This process is discussed in the next section.

4 Automating Testing

The automation of testing is currently a very active research area. Computer systems have become extremely complex and the need to subject software to rigorous testing has become essential. However, these factors have also dramatically increased the time taken to perform comprehensive tests to a satisfactory level - the more complex the system, the more tests are needed to be certain the system works correctly.

In business terms, the more time spent testing the software, the higher the development costs and the later the product will be made available to the customers. It has become essential to reduce the time taken to test the software without reducing the effectiveness of the tests. Test automation is a possible answer to this.

Software developed in object oriented languages can be modelled in UML and this specification can then be used to automatically generate the tests to be run, test cases and create test engines. It is hoped that this will significantly cut down the time required to test new software.

In this section we will look at the use of UML Use Case diagrams, Activity diagrams and Interaction diagrams to help automate part of the process of system testing.

4.1 Example

In the following sections we will refer back to a standard example for consistency. The example we have chosen to use is that of a simple currency converter[7] (see figure 3). This currency converter can convert US Dollars to Brazilian Real, Canadian Dollars, European Union Euros and Japanese Yen.

The currency converter has the following requirements:

- The user can input an amount into an input box
- The user can select the currency to convert to
- When selecting a currency, a flag is displayed for that currency
- Clicking a 'compute' button outputs the equivalent amount into an output box
- There is no limit on the number of conversions that can be performed



Figure 3: Currency Converter User Interface

4.2 Requirements Analysis & Low Level Design

The first stage of the development lifecycle (as discussed in 2.1) is to produce the Requirements Analysis of the project. This is the detailed specification of what the software should be able to achieve and in object oriented programming, can be produced using UML. This UML specification will most likely consist of many class, use case, activity and state chart diagrams that map out how the system will operate.

These simple diagrams will enable the developers to assess the basic test requirements so they can set aside resources to perform the system tests later in the project[2].

As the project continues, and the system architecture is designed, further detail can be added to the UML specification. These extra details can be used to produce the test cases that will be used to perform the system tests.

4.3 Activity Diagrams

The first stage in generating the test cases required to test the system is to create a UML Activity Diagram. These diagrams show the behaviour of the system from the point of view of an actor (a participant in the system). An activity diagram should be created for each actor. Figure 4 shows the activity diagram from the standard example described in 4.1 from the point of view of

the user.

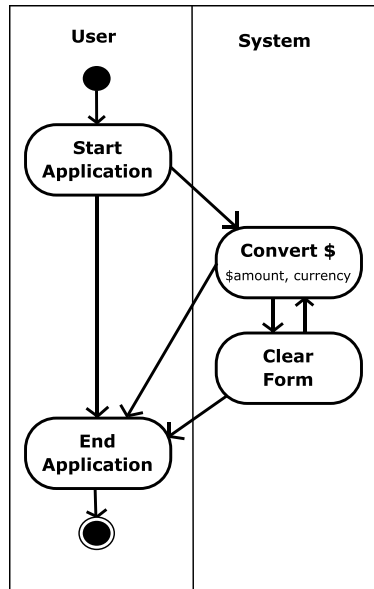


Figure 4: Activity Diagram

From this diagram, it is possible to create a tree diagram showing all of the possible routes through the system. This tree diagram can be created by performing a depth first search[2] of the activity diagram. As our example is very simple, there are not many paths that can be taken, however, on a more complicated system there could be hundreds if not thousands of possible routes.

4.4 The Use Cases

From looking at the activity diagram (figure 4), we can determine a number of activities such as ‘Start Application’, ‘End Application’ and ‘Convert Dollars’. These activities will have been described in the UML specification using Use Case diagrams (see 2.3.2).

Using these diagrams, we can create some High Level Use Cases (HLUC) for each activity that describe what the activity does. These HLUCs show very little detail but provide a very brief overview of the functions of the system. Tables 1,2 and 3 below show examples of the HLUCs for our currency converter

HLUC1	Start Application
Actor(s)	User
Description	The user starts the currency conversion application in Windows

Table 1: High Level Use Case 1

HLUC2	End Application
Actor(s)	User
Description	The user closes the currency conversion application in Windows

Table 2: High Level Use Case 2

HLUC3	Convert Dollars
Actor(s)	User
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country

Table 3: High Level Use Case 3

4.5 Expanding the Use Cases

Now that we have the basic use cases in place, it is now possible to begin adding in some more detailed information. The HLUC designed in 4.4 will now be expanded into Essential Use Cases (EUC). These EUCs show the sequence of actions that take place and the responses from each actor. They make use of Sequence and Interaction diagrams from the UML specification. Each response in the EUC is numbered in the order in which they should occur.

Table 4 below, shows the EUC generated from the High Level Use Case shown in table 3.

These EUCs provide us with a good basis from which to plan the system testing. However, we cannot directly create the test cases themselves from these use cases. Another level of refinement is required once the low level system architecture is designed. At this point, we can use the detailed UML specification of objects and classes to create much more useful Expanded Essential Use Cases (EEUCs).

The EEUCs add much more detail to the use cases such as pre and post conditions - conditions that must be true before the activity can run, and conditions that must be true after the activity has finished. These use cases also provide references to system functions and objects so that test cases can be created. Table 5 below is an example of an EEUC generated from the EUC in table 4.

EUC3	Convert Dollars	
Actor(s)	User	
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country	
Sequence	Actor action	System response
	1. The user enters a dollar amount on the keyboard 3. The user selects a country 6. The user requests a conversion calculation	2. The dollar amount is displayed on the GUI 4. The name of the country's currency is displayed 5. The flag of the country is displayed 7. The equivalent currency amount is displayed

Table 4: Essential Use Case 3

EEUC3	Convert Dollars	
Actor(s)	User	
Preconditions	<<input box>>has focus	
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country	
Sequence	Actor action	System response
	1. The user enters a dollar amount on the keyboard 3. The user click a country button 5. The user clicks <<compute button>>	2. The dollar amount appears in <<input box>> 4. Currency flag appears in <<currency label>> 7. Equivalent amount appears in <<output box>>
Postconditions	<<clear button>>has focus	

Table 5: Expanded Essential Use Case 3

4.6 The Final Steps

Now that we have created the Expanded Essential Use Cases, the next step is to create the test cases - or Real Use Cases (RUCs). To create RUCs, we edit the EEUCs to contain 'real' values - e.g. "Enter **10** into <<textbox>>" instead of "Enter **value** into <<textbox>>". Table 6 shows an example of an RUC for the EEUC in table 4.

The biggest problem with generating real use cases is determining the values to use. The values used must cover as many problem inputs as possible to ensure that the system handles them correctly without causing an error. It is not possible to test the system with every available value as this would take an infinite amount of time. This means that the chosen values must also be as efficient as possible. A further question is - can this process be automated? The answer to this is possibly - this is a topic of continued research and the industry hopes that by automating this step, software will become much more reliable and cost effective.

RUC3	Convert Dollars	
Actor(s)	User	
Preconditions	<<input box>>has focus	
Description	The user inputs \$10 and selects the European Community; the application computes and displays the equivalent €7.50 in the <<output box>>	
Sequence	Actor action	System response
	1. The user enters 10 on the keyboard	2. 10 appears in <<input box>>
	3. The user clicks European Community button	4. EU flag appears in <<currency label>>
	5. The user clicks <<compute button>>	7. 7.50 appears in <<output box>>
Postconditions	<<clear button>>has focus	

Table 6: Real Use Case 3

5 Conclusion

Poor system testing has led to the production of faulty products and software which does not perform the task it was intended to do. This, together with the increased complexity of today's software and the risks associated with it failing has led to more rigorous and cost effective software testing being undertaken. Many different testing processes have been created to improve the quality of our software. The introduction of UML as an industry standard helps software developers better specify their systems to ensure they match the requirements of their customers.

However, despite all of these improvements, the process of making software more reliable is not complete. Further changes and improvements to the processes used are being made all the time. UML has recently been the focus of a major upgrade with work already starting on the next enhancement. Further research is also being undertaken, including work on the automation of testing to speed up the process. All of this should mean that software developed in the future should be much more reliable and much better tested than that produced today.

Despite more and more complicated software, the industry looks forward to less strenuous testing. This will come through the adoption of improved testing tools and techniques. The industry will no doubt see better testing, but with less effort.

Glossary

Acceptance Testing	This level of testing is performed by the customer to confirm that the software satisfies all of its requirements., 6, 8
Black box	Black box testing means testing the system while not looking at the the code inside (just looking at the results produced)., 8, 11, 13
Design Testing	This level of testing aims to ensure that the system architecture has been designed to satisfy all the requirements of the specification., 6, 7
EEUC	Expanded Essential Use Cases expand EUCs to include pre/post conditions and references to system functions / objects., 17
EUC	Essential Use Cases extend HLUCs to include actor actions and system responses., 16
HLUC	High Level Use Cases provide a brief overview of system functions., 16
Integration Testing	This level of testing aims to ensure that the complete system works as expected without producing errors., 6, 8
Module Testing	This level of testing aims to ensure that the constructed modules work correctly before being implemented into bigger units., 6, 7
Requirements Testing	This level of testing aims to ensure the requirements which are used to develop the software are accurate., 6, 7
RUC	Real Use Cases replace values in EEUCs with real world values. These can be considered as test cases., 17
System Testing	This level of testing aims to ensure that the completed system matches the specification designed at the beginning of the project., 6, 8

UML	Unified Modelling Language is an international standard for graphically modelling systems in information technology., 3, 5, 9, 10, 13, 14, 16, 20
Unit Testing	This level of testing aims to ensure that complete units of modules work correctly before they are integrated into the full system., 6, 8
White box	White box testing means testing the system while looking at how the code works., 13

References

- [1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, 1982.
- [2] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. In *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 194–208, London, UK, 2001. Springer-Verlag.
- [3] N. Pitman D. Pilone. *UML 2.0 in a Nutshell*. O'Reilly Media, 2005.
- [4] Ishbel Duncan, Dave Robson, and Malcolm C. Munro. Test-case development during OO lifecycle and evolution. *Journal of Object-Oriented Programming*, 11(9):36–40, 44, 1999.
- [5] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31(6):687–695, 1988.
- [6] Mechelle Gittens, Hanan Lutfiyya, Michael Bauer, David Godwin, Yong Woo Kim, and Pramod Gupta. An empirical evaluation of system and regression testing. In *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 3. IBM Press, 2002.
- [7] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2002.
- [8] Cris Kobryn. Uml 2001: a standardization odyssey. *Commun. ACM*, 42(10):29–37, 1999.
- [9] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.
- [10] Grigori Melnik and Frank Maurer. The practice of specifying requirements using executable acceptance tests in computer science courses. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–370, New York, NY, USA, 2005. ACM Press.
- [11] National Audit Office. *The passport delays of Summer 1999*. National Audit Office, United Kingdom, 10 1999.
- [12] Tim Rentsch. Object oriented programming. *SIGPLAN Not.*, 17(9):51–57, 1982.
- [13] S. Robertson. An early start to testing: How to test requirements. *Conference on Software Testing*, 1996.

- [14] N. Scott Strong. Identifying a complete object oriented life cycle for large systems development. In *TRI-Ada '92: Proceedings of the conference on TRI-Ada '92*, pages 166–175, New York, NY, USA, 1992. ACM Press.