Liability for Defective Software in South Africa

LLM Dissertation

University of Cape Town

Andrew Marshall

BA LLB (UCT)
Attorney, Notary and Conveyancer of the High Court of South Africa

October 2005

Table of Contents

1. Introduction 1
2. Types of Software 2
   2.1. Operating Systems 3
   2.2. Applications 3
      2.2.1. "Desktop" or "Stand-alone" Applications 4
      2.2.2. "Distributed" Applications 4
      2.2.3. Databases 5
      2.2.4. E-commerce Web Sites and Web Applications (Web Apps) 5
      2.2.5. Critical and Expert Systems 6
   2.3. The Hardware / Software Boundary 6
3. The Software Development Process 7
4. Types of System Failure 11
   4.1. Syntax Errors 12
   4.2. Run-time Errors 13
   4.3. Error Handling 13
   4.4. Logic Errors 15
   4.5. Architecture Design Faults 16
   4.6. “Business” Errors 16
   4.7. Gravity of Errors 17
5. The Law of Delict in Context 17
   5.1. Wrongfulness 18
      5.1.1. Negligent Misstatements / Misrepresentations 21
      5.1.2. Relational or Indirect Harm 22
      5.1.3. Defective Structures and Products 22
   5.2. Negligence 23
6. Law of Contract in Context 24
   6.1. Breach 25
      6.1.1. Positive Malperformance 25
      6.1.2. Negative Malperformance 25
      6.1.3. Prevention of Performance 26
   6.2. Remedies 26
      6.2.1. Specific Performance 26
      6.2.2. Cancellation 26
      6.2.3. Damages 27
7. Application of the Law to Defective Software 27
8. Establishing Liability through the Software Lifecycle 30
   8.1. Requirements Analysis 31
      8.1.1. Parties 32
      8.1.2. Actions 32
      8.1.3. Possible Points of Failure 33
      8.1.4. Damages 34
      8.1.5. Establishing Liability 34
   8.2. Design 36
      8.2.1. Parties 36
      8.2.2. Actions 36
      8.2.3. Possible Points of Failure 36
      8.2.4. Damages 37
      8.2.5. Establishing Liability 38
   8.3. Construction 41
      8.3.1. Parties 41
      8.3.2. Actions 41

8.3.3. Possible Points of Failure 41
8.3.4. Damages 42
8.3.5. Establishing Liability 42
8.4. Verification and Validation (Testing) 44
8.4.1. Verification 45
8.4.2. Validation 46
8.4.3. Parties 47
8.4.4. Actions 47
8.4.4.1. Black box Testing 49
8.4.4.2. White box Testing 50
8.4.4.3. Unit Testing 51
8.4.4.4. Integration Testing 51
8.4.4.5. Usability Testing 52
8.4.4.6. Function Testing 52
8.4.4.7. System Testing 53
8.4.4.8. Acceptance Testing 54
8.4.4.9. Measuring Testing 54
8.4.4.10. Cost of Testing 56
8.4.5. Possible Points of Failure 57
8.4.6. Damages 57
8.4.7. Establishing Liability 57
8.5. Installation 63
8.5.1. Parties 63
8.5.2. Actions 64
8.5.3. Possible Points of Failure 64
8.5.4. Damages 64
8.5.5. Establishing Liability 65
8.6. Operation 66
8.7. Maintenance 66
8.8. Retirement 67
9. Conclusion 67
10. References 69

----------oooOooo----------

1. Introduction

“There is no such thing as error-free code.”

This is neither the favoured mantra of a software development guru, nor the refrain of the chairman of a multinational software company, but the consensus of those at the coalface of software development the world over.

Perhaps this axiom is somewhat pessimistic: a short program of a few dozen lines would probably contain no errors. A complex accounting package of several hundred thousand lines of code may, by some extremity of chance, actually perform as intended under all circumstances. Extreme simplicity and Acts of God aside, the software industry works on the assumption that all software contains errors. While every effort is made to minimise errors in software, it is inevitable given the complexity of modern software that failure will occur from time to time, whether they be serious or not.

This paradigm raises interesting questions for the lawyer. There is, as far as I am aware, no other discipline that operates on this basis. The heart surgeon will operate on the basis that he follows a certain procedure, and if the result is not as intended, it is not because of his actions or omissions, but rather because of the unwelcome interventions of Mother Nature. The engineer building a complex machine, similarly, operates on the basis that his machine, once it has been tested, will perform as intended. As it turns out, their actions may not conform to expectations, but they operate on the presumption that they will. It is only the software developer who, because of the complexity of software code, as well as the unpredictable environment in which his software will be executed, admits that the chance of his creation not performing as intended under all circumstances is so high that he must assume there are faults in it.

I use the term "Failure" here in the widest sense, meaning simply that the software does not perform as intended. This would cover all contingencies from a fault that causes execution of a computer program to terminate with the loss of all data, right down to an irritating but harmless glitch where an accounting package occasionally displays the incorrect currency symbol. This definition is equivalent to that given under international standard ISO/CD 10303-226 for "Failure" as the lack of ability of a

component, equipment, sub system, or system to perform its intended function as designed.[1] Where I refer to "legally significant failures" I mean failures that cause damage.

I will also later examine the position where the developer has misunderstood what his client wanted and built software that does not have the functionality the client required.

The distinction in computer operations between hardware and software is well understood, and while I will be examining the liability for defective software, the reader would do well to bear in mind that software is often inextricably bound to the hardware that it serves.

As the tasks which computers are used for become more important and complex, so the danger that defective software will cause damage either to the users of computers or to third parties increases. In this dissertation, I propose to discuss such defects, and the damages that may arise from them. In doing so I shall briefly discuss the wide variety of software applications that exist, and describe the software development process. I will then go on to describe the types of errors that software is subject to. After discussing the pertinent law in general (with reference to international precedent), I will apply the law to each phase of the software development process. In doing so, I will display a bias towards examining the delictual implications of defective software because, as will become apparent, the contractual aspects are usually dealt with in terms of the contract between the parties. In reality an action in delict will often be excluded by contract, but I will assume for the purposes of this paper that no such waiver exists.

I worked for some years in software development, and uncited observations of the workings of the software industry are my own.

2. Types of Software

Given the enormous range of uses to which computers are put, it is difficult to clearly categorise the areas of software development, but for our purposes, software can be

[1] I use the terms "error" and "failure" interchangeably due to their apparent equivalence in the relevant literature.

categorised as set out below. While it is worth bearing in mind that use of these distinctions may be meaningless in five years' time, it is nonetheless important to place in context the discussion of the software development process.

2.1. Operating Systems

When a computer leaves the "factory floor" it is incapable of performing any operations. It needs software to instruct it to carry out such operations. Other than the BIOS (Basic Input / Output System) which is commonly stored in memory (ROM) on the computer motherboard and allows for the system to interact with its devices in a rudimentary manner, the first "layer" of software between the hardware and interactions with the user and the outside world is the Operating System (OS). The operating system allows the computer to interact with its devices and peripherals, such as the monitor used to display data, the keyboard used to input data, printers, the Local Area Network (LAN) and so forth. The operating system also controls and maintains the File System, allowing users to store and retrieve data on the computer's storage devices, such as hard drives and "floppy" disks. Perhaps most importantly, the operating system allows for software applications to be "run" on the computer, by allowing the application to make use of the computer's resources.[2]

Note that only "general purpose" computers will require an operating system, as this allows them the flexibility to handle an almost infinite variety of tasks. Certain computers will be designed specifically to perform a limited function, and any software used by them may interact directly with the hardware without use of an operating system.

2.2. Applications

Applications are really the reasons why computers are made in the first place, being the set of instructions that allows computers to carry out the tasks that make them so useful. While the types of applications are obviously legion, is important to have a basic grasp of system architecture, which is the way in which the applications interact with the computers on which they run.

---

[2] Webopedia "Operating System", available at http://www.webopedia.com/TERM/o/operating_system.html (accessed 22/10/2005); Walters p10

2.2.1. "Desktop" or "Stand-alone" Applications

As the appellation suggests, these are applications that run on a single computer. They are usually simple in nature, and do not interact with any other computers by way of a network whether local or remote (for instance by way of the Internet). Examples of such applications are word processors, spreadsheets and "small business" accounting packages that are not designed to interact with a central database.

2.2.2. "Distributed" Applications

These are application applications where resources and processing are shared between more than one computer. Such applications can be as simple as an office accounting package where the accounting data is stored and accessed on one Desktop computer, but can also be accessed by other computers on the same network. At the other extreme are bespoke applications used by multinational companies to administer their operations, the data for which can be stored (and replicated) between database servers in multiple locations across the globe, and accessed from thousands of individual workstations worldwide over the Internet using an advanced authentication system. Much of the processing on such a system may be carried on by software running on servers, and accessed by users through their desktop computers.

While the simple system described above will almost certainly be a generic product installed by the office "techie", the second more complex instance could have been written specifically by a software development house (possibly with that company outsourcing critical parts of the operation), installed worldwide by the company's local employees (or possibly local contractors) advised by the software development company. There may be separate maintenance agreements for each location in which the software is installed, as well as an agreement with the software development house for regular upgrades to the application.

Quite clearly in the second example, the possibility of damage arising from negligence is considerably greater. Firstly, the sheer number of people involved in developing, installing and maintaining the application increases the chance of errors creeping in the along the way. Moreover, the complexity of rolling out an application

of such a scale naturally provides greater scope for disaster. Most large systems are distributed nowadays.[3]

2.2.3. Databases

While most business applications make use of databases of some kind, and could therefore fall into both of the categories already mentioned, it is important to make one or two comments about databases in general. The storage and use of information, especially information of a sensitive nature, by a software application immediately leads to security of that data becoming an issue. Steps must therefore be taken where any application makes use of databases containing sensitive information to ensure that a reasonable level of security is maintained, as loss or misuse of such data could lead to liability.

Databases are usually developed by third parties. Developers design a database structure using one of these pre-existing databases to allow their application to store and manipulate the information that they require. Databases are thus one of the most important types of "third party" software which will be described below.

2.2.4. E-commerce Web Sites and Web Applications (Web Apps)

E-commerce web sites include online banking web sites, online vendors (whether of "hard" goods such as groceries and books or of software and other electronic resources), and providers of online services, such as share trading or financial analysis. Web applications simply use a Web browser as a terminal, for example online e-mail "clients" such as Microsoft Hotmail, translation services or virtual file storage facilities. Although also a subset of distributed applications discussed above, these applications stand out for two reasons, the first being that because the applications are served over the Internet by means of the World Wide Web, they are accessible to a huge number of potential users, and secondly and related to the first point, the fact that the services are so widely accessible means that they are accessed through disparate systems, making them especially vulnerable to error, allowing possible theft of personal data.

---

[3] Sommerville 240 *et seq*

2.2.5. Critical and Expert Systems

An expert system is one which uses the knowledge and expertise comparable to that which an expert would possess in order to advise users, the likelihood being that the users will rely upon the information given out by the system.[4] The expert system does not only provide information, but draws opinions as an expert human would. It can either assist users directly, or assist an expert in providing an expert service. Thus such systems would include both a Web-based medical diagnostic system accessed directly by users, and a system used by a civil engineer to assist in calculating the loads that materials used in constructing a bridge can carry.

Critical software is defined by IEEE/ANSI[5] as software whose failure could have an impact on safety, or good cause large financial or social losses,[6] such as software then would include flight traffic control systems, aircraft fly-by-wire systems, and software operated by banks which processes and stores records of financial transactions.

The terms are not mutually exclusive, and for example navigation systems could be critical, while at the same time for example automatically plotting courses based on weather conditions and fuel consumption, which would make them expert systems.

2.3. The Hardware / Software Boundary

I have stated the distinction between hardware and software as if it were a clear one. It is however quite possible that functionality that could be embodied in software is in fact "hardwired" into the hardware, or is stored in "read-only memory" (ROM) in such a way that it is available to the hardware as soon as the device, whatever it is, is turned on ("firmware"). Drawing a distinction between the two for legal purposes is however beyond the scope of this inquiry and I will analyse defective software on the assumption that there is a clear distinction between hardware and software.

---

[4] Gemignani 121, Miyaki 124

[5] The "Institute of Electrical and Electronics Engineers" (www.ieee.org) and the "American National Standards Institute" (www.ansi.org) respectively

[6] IEEE Std 610.12-1990 – "IEEE Standard Glossary of Software Engineering Terminology"

3. The Software Development Process

What processes are used to develop software? Software development is often considered a branch of engineering, and it certainly takes as much planning and skill to develop a complex software product as it does to deliver on a civil engineering project for example. In the infancy of the discipline, in the 1960s and early 1970s, it was found that many large software projects were failing – the specific problems were that the software was unreliable, late or over budget. There was no question as to the competence of the people involved in writing this software; the weakness at that stage lay in the management approaches used. Since then, many software management schemas have developed, all of them tailored to the specific challenges posed by software development.[7] The aspects that separate this discipline from that of other engineering disciplines are:[8]

1. Software is intangible and it is thus difficult to track the progress of the project.

2. There are very few standard software processes. In other words, while an engineer working on a bridge knows that certain engineering processes will leads to certain results, the short history of software development makes it difficult to predict what problems a certain approach in writing software may cause.

3. There is rapid advance in the technology which is manipulated by software, as well as the tools available to computer programmers in developing software. As a result, it is difficult for software managers to build up a body of experience, because what was applicable to one project may not be applicable to the next. The sheer scale of possibilities available compounds this problem.

While I have no intention of enumerating one or more of the common software project management schemas in detail, a brief comment on the general process is in order.

---

[7] Sommerville 72

[8] *ibid.* 72

It is very rare indeed for a software project to involve writing computer code only. Rather, a software development project follows a life cycle, a common manifestation of which can be described in the following steps:[9]

1. **Requirements Analysis:** the needs to be addressed by the software and the features to be included in the software are defined, in consultation with the prospective users of the software. Once these are defined in detail, they are expressed in the functional specification, which in essence sets out what the completed software should do.

   Some hold that the functional specification should not contain the non-functional specifications[10], which should be listed separately, but for simplicity's sake I will use the term "Functional Specification" for the document that describes all the requirements of the software.

2. **Design:** this is where the software is planned; flow charts can be used to track the proposed execution of the code, data structures are defined, the Graphical User Interface is designed and the composition of the various modules is specified. The programming language to be used in writing the software will be decided upon in this phase, as well as the architecture.

3. **Construction:** the design is used as the basis for the writing of the actual software code, as well as building databases to be used by the software, and incorporating any third party software.

4. **Verification & Validation (Testing):** this is a complex phase of the process, is time-consuming, and is the step that is most critical in avoiding errors in software. There are a huge variety of testing methods, but a common procedure would be to test the individual elements of software on their own and then gradually widen the scope of the tests to check the interaction of the software components with each other, then with the operating system and

---

[9] *ibid.* 45

[10] Non-Functional specifications are the parameters that the software must operate within, apart from the actual functionality that it must provide, like the number of users who must be able to access the system at any given moment, compatibility issues that may need addressing, speed of response expected of the system and so forth. See Sommerville 102

other applications that it will be executed with, as well as any peripheral devices that it may use or possibly interfere with.

5. **Installation:** the complexity of the installation process will obviously differ widely depending upon the software involved: off-the-shelf software will be designed to be easily installed by the user, while bespoke (custom made) software will often need the attention of the software developers[11] to install, or at least that of a third party expert which is in such operations.

6. **Operation:** again the operation of the software will be entirely the responsibility of the user for off-the-shelf software, but for bespoke software (and more complex generic software), there will often be an agreement between the software developers and the client in terms of which the developers will train the client (including its employees) in the use of that software. Liability could attract should this not be properly carried out.

7. **Maintenance:** the same is true in this case, where there will often be an agreement between the software developer (or a third party) and the client in terms of which the system is maintained, possibly throughout its operational life. This is often the most expensive part of the lifecycle, and for some systems may amount to several times the development cost.[12]

8. **Retirement:** the retirement of the software may certainly mean the retirement of all versions of the software and the development from scratch of an entirely new software product. It is also common, however, for retirement to involve the phasing out of one version of software in favour of the next.

Because these stages "cascade" into each other, this model is often referred to as the "Waterfall" model.[13] However, these steps are not necessarily followed in a linear fashion, but can "feed back" on each other, hence the cyclical nature of development. For example, after the testing phase, the process may pass back to construction to fix errors found in the testing phase. Further, as the needs of users of software

---

[11] In this paper I will use the term "developer" to refer to the entity which constructs software, be that entity an individual or a multinational company.

[12] Sommerville 607

[13] *ibid.* 45

change, so aspects of the software may need to be revisited. The process would thus feed back to the beginning for a new version of the software which would then go through the steps as before.[14]

The difficulty from an analytical point of view is to hit upon a development approach which allows legal analysis of each phase. While there are several development models, the "Waterfall" model is both the most intuitive and the most commonly used. Different types of project will place a different emphasis on the various stages. For example, off-the-shelf software will have a requirements analysis which is very different to that undertaken for bespoke software – rather than determining the needs of a particular client, the needs of a particular group of potential users is assessed. Similarly, the installation, operation and maintenance stages will be very different for bespoke software then for off-the-shelf software, where this is likely to be left as the user's responsibility to a greater extent.

An example of an alternative model is the so-called "prototyping" model. The basis of this approach is to build a prototype of the final system so that the client can see it working, and can thus request additional (or changed) functionality and improvements based upon the prototype. This is greatly beneficial in allowing the client to clarify its requirements.

There are two types of prototyping. The first is so-called "throwaway" prototyping, where the prototype is used simply as a guide to obtain a specification for the final software to be developed. The second kind is evolutionary prototyping, where the system is "released" to the client, and then improvements are made to the system when it is already in use by the client. This approach has its advantages where it is difficult to come up with a clear specification, for example where an artificial intelligence system is being built.[15]

Prototyping presents difficulties from the legal perspective, as most software development contracts set out specifications of what the functionality of the software to be delivered will be. If a prototyping approach is being used, the specification is developed "on-the-fly", making it difficult to establish at a later stage what the duties of the developer were.

---

[14] *ibid.* 45

[15] *ibid.* 176

Prototyping goes hand-in-glove with an approach called "Rapid Application Development" (RAD), which allows these prototypes to be developed quickly. This approach encourages the use of what are called "Computer Aided Software Engineering" (CASE) tools.[16] For example, Microsoft's popular programming language Visual Basic uses what is called an Integrated Development Environment (IDE) which allows developers to write and test their code easily and quickly using a set of useful development tools. Other vendors provide similar environments.

I will be analysing the liability for defective software based upon the Waterfall model for the reasons given, but it is important to bear in mind that there are other approaches available. I would submit however that whatever the development model actually followed, an equivalent can be found in the waterfall model for the purposes of legal analysis.

The software industry uses many divergent terms and classifications. In my description of the software lifecycle, both in this section and later in this dissertation, I have attempted to harmonise these inconsistencies insofar as possible, but I have noted when they intrude on the analysis.

4. Types of System Failure

I have now examined the types of software and the uses to which they are put. The next step is to examine what "errors" are and why they occur.

There are three broad classes of software programming errors, being syntax errors, logic errors and run-time errors. One could draw a distinction between a "fault" and an "error", the first giving rise to the second, but for our purposes, "error" will suffice.

As described above, software is in essence the set of instructions followed by computers to allow them to perform the tasks that we want them to perform. Because computers are binary systems, they can only "read" the software instructions in binary ("machine") code, which is represented as 1s and 0s. While there are people who can read and write binary code, lesser mortals are forced to write the software in one of the many software languages which are available, which allows code to be

[16] Walters 233

written which is readily understandable by human beings, and then "compile" (or "assemble" in the case of an assembler language) the code into binary form.

4.1. Syntax Errors

Each software language has its own "grammar", and should the programmer make a mistake in this "grammar", a fatal error (one that halts execution of the programme) will occur.[17] For instance, the line of code to return (“get a result” in developer's terminology) the current month in Microsoft's Visual Basic 6 programming language would be:

myMonth = month(now)

where “now” and “month” are functions: “now” returns the current date and time, and “month” returns the month of the date given as its argument. “myMonth” is an arbitrary variable that the month value is placed into. If I were to write:

myMonth = month now)

the complier would not know how to render the line in binary, and would raise an error to the developer, who would not be able to compile his code until he had fixed the error. Syntax errors occur at the compilation stage of coding (which is part of the construction phase of the software lifecycle), so they are “ironed out” before the testing phase is reached. As a result, such errors are unlikely to lead to system failure.

Unfortunately, there are important exceptions to this, as not all code is compiled. Such code is often called a “script”, and is commonly used in such applications as rendering web pages, running backup systems, and querying databases. A likely scenario is where a procedure in a script that is rarely used contains a syntax error. The script may run day in and day out for some time, but then when that procedure is called it will suddenly fail, causing the system to fail.

---

[17] Wikipedia "Syntax Error", available at http://en.wikipedia.org/wiki/Syntax_error (accessed 22/10/2005)

4.2. Run-time Errors

These are errors that are not picked up by a compiler that will cause software to fail if they occur when it is executing (at "run-time" as opposed to "compile time"), but which are not always predicted.[18] An example would be:

myResult = 1027 / myNumber

where myResult is the result of dividing 1027 by myNumber. This line of code will work perfectly well until myNumber = 0. As a computer cannot divide by 0, it will raise an error. The developer may not have predicted a situation where myNumber = 0, and the program will fail.

Other common causes of run-time errors are hard-coding file locations, where the software “expects” a file that it uses to be in a certain place in the file system, but, when run on a different computer, cannot find it (this is especially true of applications that link to databases); not predicting the type of data that will be put into variables, resulting in for example text being put in a variable of integer data type; overflow errors, where the bounds of an array are exceeded, and many others.

4.3. Error Handling

No discussion of computer errors would be complete without mention of “error handling”. While a competent developer will predict run-time errors that could occur and write his code so as to avoid them, every developer with any experience realises that there are some situations that they will not predict, and that they must therefore write their code in such a way that it deals with the errors itself without interrupting the execution of the program.

There are many methods for handling errors, the complexity of each of which need not detain us here, but in essence they all direct the computer executing the code to either ignore errors (which could cause a logic error as described below), or to perform some action to handle the error. For example the following function in VB6:

---

[18] Wikipedia "Runtime", available at http://en.wikipedia.org/wiki/Runtime_error (accessed 22/10/2005)

```
1. Function myAnswer (myNumber as integer)
2.         on error goto error_handler
3.         myAnswer = 1027 / myNumber
4.         exit function
5.         error_handler:
6.                 msgbox “Input cannot be zero”
7.                 myAnswer = 0
8. End Function
```

While this may appear complex, it is simple on closer analysis. I have numbered the code lines for convenience.

Line 1 says that if you pass this procedure an integer value (“myNumber”), it will return a value having manipulated the input value in some way.

Line 2 says that if there is any error, the computer must skip the body of the procedure, go to the error handler on line 4, and continue from there.

Line 3 is the same as that used as an example for a run-time error above, except that the result of the computation is fed into a variable that has the same name as the function. This is then the value that the function returns. If there is no error, the computer will skip out lines 5 – 7, returning the value of the function at that point. If there is a run-time error in line 3, execution of that line is stopped, and skips to line 5.

Line 4 instructs the computer to leave the procedure, as there is no need to run the error handler if no error has occurred.

Line 5 denotes the beginning of the error handler, which is only executed if there is an error. Let us assume that myNumber = 0 as in the previous example. A run-time error was raised, and execution passed to the error handler.

Line 6 brings up a “message box” telling the user that only non-zero values can be input.

Line 7 then assigns a value of zero to the function, which then runs on to line 8 and terminates as usual.

Of course this is a very simple example. It is most unlikely that such a problem would be solved by displaying a message box. Most probably execution would be passed to a specific procedure that deals with errors, which would determine exactly what kind of error had occurred and deal with it in a manner consistent throughout the application, or the error would be “passed” back to the procedure that called “myAnswer” for its error handling code to deal with. As an example of how errors are handled though, it will suffice.

4.4. Logic Errors

These errors occur when code executes without raising run-time errors, but gives the incorrect result.[19] If for example you were using a spreadsheet package for your business accounts and you got an incorrect balance because of a “bug” in the program, it would most likely be a logic error. These errors are difficult to find, as execution of the program will not stop, and no log can be created of them as is the case for “handled” errors. A worthy “canned” example like those above would not add much value, as these errors simply result from arithmetic or other errors made by the programmers which do not disturb program execution. Looking at a real-world example would be more helpful: the Millennium Bug would probably have caused a logic error. Having a value of 1900 instead of 2000 in certain calculations would not necessary have caused the application to fail, but would have resulted in the wrong results. Of course there may also have been situations where it would have caused a run-time error.

I mentioned earlier that errors can be handled merely by instructing the computer to ignore them. If the developer is not careful, this could result in a logic error, where incorrect data is used for a calculation.

---

[19] Wikipedia "Logic Error", available at http://en.wikipedia.org/wiki/Logic_error (accessed 22/10/2005)

4.5. Architecture Design Faults

A computer program may in its final form offer all the functionality which was required of it, but yet be unable to deliver it or not be able to deliver it in a useful manner. For example, a system may be required to allow up to a thousand users to access it simultaneously. If poor choices are made in the design phase, it may happen that only a hundred users can access the system simultaneously. This can happen as a result of using a database system that cannot "scale" to handle more than a certain number of requests per second. Other possibilities include a system that is unreliable, or does not interact properly with the client's existing systems. These are examples of non-functional requirements not being met. While not programming errors, these can nonetheless have serious consequences.

4.6. “Business” Errors

Again these are not really errors in the sense that those set out above are: rather they are a result of the developer and his client not communicating properly, resulting in a functional specification that does not reflect the client's requirements. They do not involve any negligence or lack of skill on the part of the developer in writing code, but nonetheless result in the software not performing as the client required.

Complex software must be thoroughly planned before it is written, and planning software is often an extremely tricky task. The client must be interviewed to ascertain exactly what he requires the software to do. Then the developer must plan the software, perhaps returning to consult with the client to get his preferences on particular details, and alternate instructions where his initial wishes prove impractical in the design phase. The developer must then turn a complex plan into even more complex software. This process is difficult enough in the scenario that I have just sketched, where the client and developer are individuals; it becomes far more so when the parties are corporate entities. In that case the “idea” for the software may have originated high up the client’s hierarchy. It would then have been passed to various departments which would use the software for their input. Once this process is completed, the software specification is often passed to the client’s IT department for final polishing before the development company is called in. The people who liaise with the client may not be the developers themselves, but rather customer services personnel of a (hopefully) technical bent. Once the specifications are

decided upon (and usually included as an annexure to the software development contract by lawyers who are not entirely sure what the specifications mean), they are passed to the developers who plan and write the software. Before they begin writing the code, they will probably need to check with the client on certain aspects, and the software plans will be passed down one hierarchy and up the other for examination and approval.

It is not surprising then that a large proportion of software projects fail to deliver or at least fail to deliver on all their requirements.

4.7. Gravity of Errors

The gravity of an error will have a large bearing upon any action for defective software, as it can determine whether damages are suffered and the quantum of damages that may be claimed by the Plaintiff in a subsequent action. While some errors may be irritating, if they do not result in damage, they are not actionable in delict, and should they not relate to the functional specifications of the software, it is unlikely that they will be actionable in contract either.

Software developers will refer to errors that are transient or permanent, recoverable or unrecoverable, non-corrupting and corrupting or combinations of all of them.[20] What is important from a legal perspective is however simply whether failure causes damage.

5. The Law of Delict in Context

Should defective software cause loss, and in the absence of a contract between the software developer or other guilty party and the party which suffered loss, an action may be brought in delict. Moreover, even if there is a contract between the parties, the aggrieved party may elect to bring an action in delict insofar as the conduct of the Defendant is wrongful.[21]

---

[20] Sommerville 377

[21] e.g. *Lillicrap, Wassenaar and Partners v Pilkington Brothers (SA) (Pty) Ltd* 1985 1 SA 475 (A)

I do not intend to make a close analysis of the elements of the law of delict, but it is necessary to stress certain points of law relevant to this paper.

5.1. Wrongfulness

While discussing wrongfulness it makes sense to bear in mind the potential damages that can be suffered as a result of defective software. It is of course quite likely that software controlling a device may result in damage to a corporeal or to physical injury. For example, a navigation system linked to electronic maps may malfunction, resulting in a ship navigated by this system to run aground. The software controlling a robotic device in a manufacturing plant may include functions which allow the robotic device to be shut down for safety reasons when personnel perform maintenance; defects in this functionality may result in death or injury to the personnel. Notwithstanding, it is far more likely that any loss will not result from physical damage. For example the vendors of a dial-up Internet service may provide software which amongst other functions monitors the usage in a small office, ensures that the connection is terminated when not being used, and notifies the network administrator of any abnormal use of the facility. Should such a system fail, it may result in a connection being maintained (and charged for) when the connection is not used, with the associated cost to the client. Again, an accounting package may be used to generate the financial results of a company. If the results are incorrect as a result of a defect in the software, and these results are relied upon to make a critical business decisions, the results can be catastrophic, though purely economic.

The law of delict in South Africa has three legs: the *actio legis aquiliae*, the *actio iniuriarum* and the action for pain and suffering, it is the *actio legis aquiliae* that is relevant here, being the action for patrimonial loss.[22] This action was under Roman Law available only for damage to corporeals.[23] There was no remedy should damage be suffered without actual damage to a corporeal – so-called "pure" economic loss.

The ambit of the *actio legis aquiliae* has been extended over time to include pecuniary loss arising from damage to person or property. The courts also allow an action for negligently caused pure economic loss. It is trite law that damage negligently caused to person or property is *prima facie* wrongful, as is pure economic

[22] Neethling 8

[23] *ibid.* 9

loss caused intentionally. The position regarding negligently caused pure economic loss is less clear however. The position of the courts seems to be that the onus of proving wrongfulness in the case of negligently caused pure economic loss should fall on the Plaintiff,[24] who will have to convince the court that policy considerations dictate that the act or omission on the part of the Defendant is wrongful on the facts of the case.

The general principle accepted by our courts is that wrongfulness can be established either as result of an infringement of a subjective right, or in a breach of a legal duty to avoid damage.[25] In the case of pure economic loss, the courts establish wrongfulness on the basis of a legal duty.[26] Whatever the approach, the test for wrongfulness is the legal convictions of the community (*boni mores*). In the specific context of pure economic loss, the courts have set out a number of criteria to be considered in determining whether or not the conduct is wrongful:[27]

- If the Defendant knew or anticipated that his conduct could cause damage to another, the courts are likely to hold that his conduct was wrongful, as it did in the case of *Coronation Brick (Pty) Limited v Stratford Construction Co (Pty) Ltd*.[28] In this case, the Defendant, a civil engineer, damaged electrical cables which carried electrical power to the Plaintiff's factory. There was no contractual relationship between the parties. As a result of the physical damage to an unrelated party's cable, the Plaintiff was unable to produce bricks and suffered pure economic loss as a result. The court accepted that the Defendant knew of the existence of the electrical cable, and that it should have foreseen that disrupting the cable would cause the damage that it did.

- If the Defendant could have taken steps to prevent the damage the Plaintiff suffered, these steps will be taken into account along with their probable

---

[24] As opposed to the case where the act or omission is *prima facie* wrongful, in which case the Defendant bears the onus of disputing wrongfulness

[25] Neethling 295, *Coronation Brick (Pty) limited v Stratford Construction co (Pty) Ltd* 1982 4 SA 371 (D) at 379

[26] Neethling 297

[27] *ibid.* 297 *et seq*

[28] 1982 4 SA 371 (D)

success, the expenses involved in taking these steps, weighed against the gravity of the damage suffered by the Plaintiff.

- If the Defendant professes to possess professional skill, it has been held that he has a duty not to cause financial loss to others. So for example in the case of *Standard Chartered Bank of Canada v Nedperm Bank Ltd*[29], Standard Chartered Bank relied upon a financial report given on a third party by Nedperm, and suffered pure economic loss as a result of this reliance. The court held that the negligent misstatement on the part of the Defendant was wrongful. An interesting point is raised in this regard by Burchell[30] in which he suggests that distinguishing between services rendered by "professionals" and others could fall foul of the constitutional right to equality.

- The higher the degree of risk of economic loss, the greater the need for protection from that risk and hence the stronger the argument that the defendants act or omission was wrongful in the circumstances.

- The greater the extent of loss, the smaller the chance that the court will find the defendant's actions or omissions wrongful. The reason for this is that if the extent of loss is overwhelming, the defendant cannot be expected to bear the multiplicity of actions which could follow.

- The courts have also taken other factors into account, including whether or not the Plaintiff was in a position to mitigate the loss, and whether the Defendant was in a position to insure against the liability that could attract under the circumstances.

Courts are wary of determining wrongfulness in cases of pure economic loss largely because of the spectre of a "multiplicity of actions". Moreover they are concerned that if the facts of a certain case point to liability for pure economic loss in favour of an "unascertained class of potential victims",[31] there will be no end to the liability of the Defendant, with undesirable social results. In other words, if the Defendant will

---

[29] 1994 (4) SA 747 (A)

[30] Burchell 107

[31] *Shell and BP SA Petroleum Refineries (Pty) Ltd v Osborne Panama* SA 1980 3 SA 653 (D), confirmed on appeal 1982 (4) SA 890 (A)

incur liability in respect of too many Plaintiffs relative to the nature of the conduct giving rise to the action, the conduct will be held not be wrongful. The distinction can be seen in the case of Coronation Brick,[32] where only a limited number of parties could have suffered loss as result of the conduct of the Defendant.

Burchell[33] distinguishes four classes of negligently caused pure economic loss, three of which are useful for our purposes.[34] These are briefly:

5.1.1. Negligent Misstatements / Misrepresentations

The situation contemplated here is that of a professional giving advice or opinion which is relied upon to the financial detriment of its client. An example is the case of *Standard Chartered Bank of Canada v Nedperm Bank Ltd*[35] alluded to above.

The courts have established several guidelines in determining whether a duty existed to furnish correct information to a particular person (and hence whether wrongfulness existed):[36]

- Where there is a statutory duty to furnish correct information.

- Where a contractual relationship exists between the parties. The parties may either undertake that the information given is correct by warranty, or such a duty may arise from the surrounding contractual relationship itself.

- Where a person provides information in his official capacity in a public office such as a notary or auditor.

- If by virtue of his occupation a person is in the exclusive possession of certain information.

---

[32] *supra*

[33] Burchell J, "The Odyssey of Pure Economic Loss" (2000) A10 *Acta Juridica* 99

[34] The fourth relates to proprietary interest in damaged property.

[35] *supra*

[36] Neethling *et al* 303 *et seq*

- If a person claims to possess particular professional knowledge or competence by virtue of his profession and furnishes information in his professional capacity.

An important exception to this is the case where an existing contractual relationship exists between the parties and where breach of contract led to pure economic loss. In that case, the courts are reluctant to allow an action in delict, and the wronged party will have to pursue an action in terms of the contract.[37]

5.1.2. Relational or Indirect Harm

An example of this is the *Coronation Brick* case, where there was no relationship between the Plaintiff and Defendant, but the Plaintiff nonetheless suffered harm as a result of the Defendant's conduct causing damage to the property of another.

5.1.3. Defective Structures and Products

This is the situation where the party who provided a product is held liable by the person for whom he constructed or provided the product (as a possible alternative to an action in contract), or by a third party for damages caused by defects in the product.

In many foreign jurisdictions, notably the United States and the United Kingdom, manufacturers are strictly liable for damage caused by defective products, and thus the jurisprudence relating to damages caused by defective software in those jurisdictions concentrates heavily on the distinction between a good and a service. If the software is held to be a service, then negligence must be established in tort.[38]

In South African law the distinction is not nearly so pertinent, as the general principles of the Aquilian action apply whether the damage is caused by a good or a service. In determining wrongfulness, the general rule is that damage caused by a

---

[37] *Lillicrap, Wassenaar and Partners v Pilkington Brothers (SA) (Pty) Ltd* 1985 1 SA 475 (A) at 499-500

[38] for a discussion of the distinction as compared with South African law, see Alheit K "Contractual liability arising from the use of computer software: notes on the different positions of parties in Anglo-American law and South African law" (2000) 33:1 *Comparative and International Law Journal of Southern Africa* 26

defective good is wrongful as it constitutes a breach of a legal duty to prevent harm. There is untried academic opinion on the exact nature of wrongfulness[39], but it seems that if a product causes damage through being defective, then the act or omission concerned is wrongful.

5.2. Negligence

The test for negligence in South African law is definitively set out in the case of *Kruger v Coetzee*[40]

"For the purposes of liability culpa arises if –

(a) a *diligens paterfamilias* in the position of the Defendant –

(i) would foresee the reasonable possibility of his conduct injuring another in his person or property and causing him patrimonial loss; and

(ii) would take reasonable steps to guard against such occurrence; and

(b) the Defendant failed to take such steps."

Negligence must be established in any action in delict for liability to attract, with certain exceptions which are not relevant. Should the conduct of the Defendant not meet the standard set out above, negligence is present. A court would however not apply the above test as stated in the case of an action arising from defective software. Because computer software is written by experts, the court would apply the test slightly differently.

In the case of a "specialised" Defendant, the court will apply the test of the "reasonable expert", in which an expert with expertise relevant to the case is substituted for the reasonable person in the above test. The leading case is that of

---

[39] Neethling *et al* p322

[40] 1966 2 SA 428 (A) at 430

Van Wyk v Lewis[41] involving medical negligence where it was held that "…the Court will have regard to the general level of skill and diligence possessed and exercised at the time by the members of the branch of the profession to which the practitioner belongs."[42] This test has been widely applied including, for example, the construction industry.[43]

A weakness with this approach is that different levels of expertise and experience exist within the branches of any profession, especially one with such diverse (and constantly changing) skills as software development. If a company entered into a contract to develop a certain computer program, then such a "median" level of skill might be acceptable, because the company would be under an obligation when developing software to ensure that its employees had the requisite skill. However, if the work was performed by a software developer working alone, it is difficult to see how one would arrive at a useful standard.

A related issue is that enshrined in the maxim *imperitia culpae adnumeratur*, the upshot of which is that liability will attract if someone purports to be an expert, but knows all should reasonably know that he lacks the required knowledge or expertise.[44]

Finally, the question arises whether the notion that software is inherently and unavoidably defective can be reconciled with our negligence jurisprudence. I believe that it can, as it is the reasonableness of the actions or omissions related to producing the software, rather than the nature of the software itself that is to be analysed in determining negligence.

6. Law of Contract in Context

Should there be a contract between the party that will be using the software and the developer involved in its creation, and should the software be defective, there may be an action for breach of contract.

---

[41] 1924 AD 438

[42] at 444

[43] *Randaree and Others NNO v W H Dixon and Associates and Another* 1983 (2) SA 1 (A)

[44] Neethling p137

It is unfortunate that most case law relating to defective software turns around the interpretation of the contract for the development of the software. The court uses the specification for the software to be developed included in the contract as the yardstick against which it measures the performance of the developer, and it is the general principles of the law of contract that are applied to such problems. As result, while such cases relate to defective software, the reasoning is the same as it would be in the case of any contract governing a complex project. Nonetheless, I include the following discussion for the sake of completeness.

6.1. Breach

There are several "modes" of breach of contract, three of which are relevant here, being positive and negative malperformance and prevention of performance. [45]

6.1.1. Positive Malperformance

This form of malperformance takes place when the party concerned performs under the contract, but does not deliver the standard of performance agreed between the parties. In the context of this discussion, this would for example take place where a computer program was delivered on time and on budget, but where it did not meet the specifications as set out in the contract in some way.

The South African law was not clear on the issue of fault and especially negligence in this context. It has been suggested both that fault is not at issue if the contract does not demand a standard of reasonable care, and that where performance is inadequate, negligence is assumed.[46]

Should below par performance be delivered, the aggrieved party can reject the performance and claim damages, or accept performance and claim damages being the amount required to remedy the defective performance.

6.1.2. Negative Malperformance

This occurs where the performance in terms of a contract is not made timeously, or is not made all. An example of this would be where a developer underestimated the

---

[45] This is trite law – see van der Merwe *et al* 235 *et seq*

[46] van der Merwe *et al* 253

amount of time that it would take to deliver on a certain project, resulting in the software being delivered late.

6.1.3. Prevention of Performance

One of the parties can beach the contract by preventing the other party from rendering performance. A good example of this in the context of software development is that dealt with in the *Co-operative Group* case discussed below.

It is important to distinguish between these types of malperformance because of the different effects that errors will have. Mistakes made in writing code (and not picked up during resting) will typically result in positive malperformance, while errors in planning the project will typically result in negative malperformance. Prevention of performance in the software context will most often be caused by the client not furnishing the developer with required information in the requirements analysis phase as described below.

6.2. Remedies

Upon breach of contract, the aggrieved party can have recourse to three remedies:[47]

6.2.1. Specific Performance

A court may, where appropriate, make an order that the party that is in breach is to perform in terms of the contract.

6.2.2. Cancellation

If breach of contract is material, the aggrieved party may (after giving notice to the other party to place it *in mora* if necessary), cancel the contract. It is common for contracts to contain a *lex commissoria* (cancellation clause), which will govern the circumstances in which is the contract can be cancelled. Such contracts will typically contain a punitive damages/liquidated damages clause which will govern the damages available in the case of cancellation.

---

[47] van der Merwe *et al* 272 *et seq*

6.2.3. Damages

Should the aggrieved party have suffered loss as a result of the other party's malperformance, damages are available as a remedy.[48] In assessing the quantum of damages, the court will attempt to put the aggrieved party in the position he would have been if the contract had been performed as intended by the parties ("positive interesse").[49]

7. Application of the Law to Defective Software

Now that I have analysed both the technical issues in broad terms and the pertinent law, it remains to see how liability should be established where there is a software failure. The South African law is still in its infancy in this regard – our courts have not handed down any significant judgments in this area. We are forced therefore to look beyond our borders for guidance on this question. Even this guidance is however rather limited.

Most disputes where there is a measure of negligence in the development of software, unfortunately for our purposes, are handled by application of the law of contract as discussed, leading to a dearth of good case-law. There are few cases dealing with the problem from a delict or "tort" angle. As discussed above, the question of negligence is not explored in an action for breach of contract, the question raised being quite simply whether the performance rendered satisfied the specifications set out in the contract.

In the English case of *Co-operative Group Ltd (formerly Co-operative Wholesale Society Ltd) v International Computers Ltd*,[50] a software developer was sued for breach of contract for having delivered allegedly defective software. As the contract did not adequately describe the specifications of the software to be built, the judge was obliged to attempt to construct a yardstick of what reasonable performance would have been. Unfortunately, on the facts this proved very difficult to do, as the Plaintiff had failed to cooperate with the Defendant in rendering its performance by not furnishing it with its requirements, and other important information.[51]

---

[48] *ibid.* 296

[49] *ibid.* 302

[50] [2003] EWHC 1 (TCC)

[51] at ¶ 260

This case is, unfortunately, typical of the difficulties involved in distilling negligence issues from a judgement in contract involving the development of software. The discussion relating to the contractual obligations and the (mis-) communication between the parties tends to muddy the waters and makes it very difficult to determine the obligations of the software developer; and when these obligations have been established, whether the developer performed its work to an adequate standard. It is important to note however that the court readily accepted the premise that bespoke (in that case) software will always contain faults.[52]

We must thus turn our attention to cases where a third party who was not privy to the contract between the software developer and the client suffered loss, or where the victims were injured by generic software. As discussed above, the strict liability regimes for defective products in many of the jurisdictions that we are accustomed to draw precedent from makes comparison largely unhelpful.

Not surprisingly, it is the expert and critical systems which have attracted the most attention. It is strange that a strong body of case law has not emerged internationally resulting from failures in such systems. An example of a critical system error which was caught in time was one involving the F-16 fighter aircraft. This was one of the first aircraft to make use of a fly-by-wire control system. There was a fault in the software which was found during testing which would have caused the aircraft to flip over every time it crossed the equator.[53] Further and more serious examples are those of the NASA Mars lander which crashed on landing on the planet Mars. The crash caused by controlling software shutting off the landing engines prematurely.[54] A MV–22 Osprey, which is a hybrid fixed wing/helicopter aircraft, suffered a fatal crash during testing when software which was designed to switch from primary systems to backup systems failed to perform properly when a faulty hydraulic line burst.[55]

Even more serious incidents were those involving the Therac-25 machine in the USA during the mid-1980s. This was a particle accelerator used in the treatment of

---

[52] at ¶ 264

[53] Miyaki 122

[54] Hoglund 11

[55] *ibid.* 12

cancer, and which had software controlling its operation. The software would control the dosage of radiation (and the kind of radiation, whether x-ray or electron) depending upon user inputs. In certain rare circumstances involving the movement of the cursor on the screen (controlled by operation of the "up" key), a large overdose would be given. To generate x-rays, an electron beam was aimed at a target, which would then produce the x-rays. A far more powerful electron beam was used to generate the x-rays than that which would be applied directly to the patient. The software error led to a failure by the machine to lower the target in the way of the electron stream under certain circumstances, resulting in the patient getting up to a hundred times the required to dosage of electrons in circumstances where x-rays were called for. Several patients were badly injured and one died as a result of excessive doses of radiation being administered. Operators became aware of the problem and what sequence of keystrokes caused it, and the defect was subsequently remedied. Interestingly, they solved the problem in the interim by taping over the "up" key (though there were no subsequent incidents).[56] Unfortunately for our purposes, this matter never came before a court, being dealt with by the regulatory authorities.

In the case of *Coastal State Trading, Inc v Shell Pipeline Corporation*[57] the dispute involved the alleged erroneous delivery of crude oil. The Plaintiff (Coastal State Trading) was a reseller of oil which was pumped through a pipeline by the Defendant. The Plaintiff entered into an agreement with a company called Basin Inc. to on-sell crude oil to it. The Plaintiff then advised the Defendant that it was transferring its rights to receive oil to Basin. A company related to Basin Inc., Basin Refining, advised Shell that it was the company to which the oil should be transferred, and Shell proceeded accordingly. Basin Refining went into bankruptcy soon thereafter, leading to a substantial loss for the Plaintiff.

Shell used a computer system to control its delivery of oil. The owner of the oil would send a "datagram" to Shell advising Shell that it was reselling the oil to another party. This information can be entered into the system and delivery made accordingly. In the event that the oil was sold on by its consignee, that further party could then follow the same procedure. The system was designed to pick up on any inconsistencies or gaps between the parties to such transactions. It was the complexity involved in

---

[56] Bohan 141

[57] 573 F.Supp. 1415 (S.D. Tex. 1983), aff'd 788 S.W.2d 837 (1990)

having oil traded several times in this way before physical delivery that necessitated the introduction of a computer system in the first place. Under the circumstances, Basin Refining had traded the oil to another party, and so the software should have picked up the discrepancy between Basin Inc. and Basin Refining, allowing delivery of the oil to be halted in time. The software failed to do this.

The Plaintiff alleged that its loss was therefore as a result of negligence on the part of Shell Pipeline. The Court *a quo* apportioned 50 percent liability to Shell Pipeline and 50 percent to the Plaintiff. Shell had alleged that the software was a gratuitous aid to traders, and was not designed to be relied upon, but the court held that Shell had given the impression that the system was an adequate safeguard, and that third parties had relied upon this impression. This was upheld on appeal.

It appears that third parties did not directly interact with this computer system: entries were made into the system by Shell employees on the basis of "datagrams" sent to them by oil traders. It is thus not the defective software which is the point here, but rather the fact that Shell had certain procedures in place which third parties relied upon to their detriment. The fact that these procedures involved computer software is incidental. What would have been more interesting for the purposes of this paper would have been if Shell had brought an action against the company which developed the software for Shell based upon Shell's existing procedures (assuming that the software was not developed in-house). Shell's negligence therefore was not in developing a defective software product, but rather in relying upon one unreasonably in its interactions with third parties.

As is apparent from the above, there is an unsatisfying dearth of case law to which to look for guidance. No doubt this will change in the years to come, but for the moment the only guidance where a third party is affected by defective software is that provided by the general principles of the law of delict.

8. Establishing Liability through the Software Lifecycle

I have now examined in broad outline the various classes of software that are developed, and the processes used to develop, implement and maintain software. I have briefly explored the theory behind software failure, but the theory only answers part of the inquiry. To establish liability for defective software, it is important to establish the stage at which the failure was caused, as this will determine who is to

be held liable. Moreover, the complexities of software development may be such that a court would hold that the actions or omissions of the relevant party will not attract liability under the circumstances.

One must ascertain who the parties were, the nature of the actions or omissions involved, and the causal link between these actions or omissions and the damage suffered by the Plaintiff. There is no better schema for obtaining this information than analysing the software lifecycle as described above. In doing so, mindful of the requirements for establishing liability which will be discussed later, I will in each phase of the lifecycle examine the following elements: the parties involved in that phase, the actions taken (or which should be taken), possible points of failure (though this can of course not be exhaustive), likely damages arising from such failures, and finally examine establishing liability for faults in each phase.

As discussed, the "Waterfall" model of the software lifecycle is used because it is the most intuitive, and other models have analogous phases. Secondly, the steps described are not necessarily strictly linear, the testing phase for example feeding back on the construction phase when faults are found.

The standards found in the industry are described in broad terms here, and no attempt is made to analyse the reasonableness or otherwise of conduct during a particular phase of the software lifecycle relative to an industry standard (though standards are referred to for context). The reason for this is that not only is there great debate in the industry around what reasonable standards are, but the technology underlying these standards changes so quickly that an analysis based on such standards would become rapidly irrelevant. I have thus attempted to distil what I feel is useful and what is likely to remain constant enough to make my conclusions valuable in the long term.

8.1. Requirements Analysis

Any software development project will start with a requirements analysis of some sort. This is the phase during which the developer determines the needs to be

addressed by the software. Once the requirements analysis has been completed, its results are used to compile the functional specification.[58]

8.1.1. Parties

The simplest model here is that of a developer discussing directly with a client what its needs are and drawing up the functional specification itself. However, it may happen that a consultant is employed by the client to carry out this step. The consultant will (hopefully) be experienced in systems analysis, and will then either contract the developer to write the software, handing the functional specification over to it, or will deliver the functional specification to the client in its report, and the client could then hire the developer itself. It is also quite likely that a large company will have its in-house IT department perform the requirements analysis.

8.1.2. Actions

The requirements analysis may take different forms depending upon the circumstances. Should bespoke software be envisaged, the analysis will be conducted with the client for whom the software is being developed and should involve extensive interviews with all stakeholders to ascertain exactly what is required. This is a critical stage as a misunderstanding here can undermine the entire process. Once the consultant is satisfied that it understands the client's needs, it will analyse the problem to determine what functionality is required. This analysis will include an inspection of the client's existing systems to determine whether the proposed software will be duplicating, superseding or possibly even conflicting with existing systems, as well as a determination of what other non-functional requirements need to be met, such as required response times and maximum number of concurrent users. The consultant will then report back to the client. This report may be revised several times, and the functional specification will result from the process.

If on the other hand the developer intends to produce off-the-shelf software, the first step may be more akin to a consumer needs analysis: research will be done into

---

[58] As explained above, I have defined this to include both functional and non-functional requirements for the sake of convenience.

what consumers want out of a certain class of application, and what features would give software an edge over its competitors (if any).

If a prototyping development model is being used, the initial requirements analysis will be more superficial than where a more traditional approach is used.[59]

8.1.3. Possible Points of Failure

The first and most obvious point of failure here is that the parties misunderstand each other. It is very common for clients to have no concrete idea of what they really want, possessing only a vague idea of what the problem actually is (a concern that contributed to the rise to the prototyping approach). They often rely upon the consultant or developer to act as a sort of seer and anticipate their needs before they appreciate them themselves. This is particularly true if the client's contact person with the consultant is not technically inclined. The consultant is thus often left in the difficult position of first having to find out what the problem is before being able to posit a solution.

The consultant cannot be held entirely blameless however. Having worked on many projects, the consultant may have preconceptions as to what the solutions to certain problems should be, or even what the problems actually are in certain situations. It is therefore not unheard of for the client to say one thing and the consultant to hear quite another entirely. The result of these misunderstandings may very well be a functional specification that provides an inadequate solution to the client's problem. This could result in the "Business Errors" discussed earlier.

Of course it will not necessarily be misunderstandings that lead to an inadequate functional specification. The consultant may quite simply be negligent. Whatever the case, should the project proceed to the next step and the functional specification be found wanting, the process will have to start again, with the resultant expense. Should the project continue without interruption, the resulting software may suffer from several defects, possibly becoming prematurely obsolete, or addressing only a portion of the client's needs thus necessitating development of a further system.

---

[59] Sommerville p95 *et seq*; Wikipedia "Requirements Analysis", available at http://en.wikipedia.org/wiki/Requirements_analysis (accessed 22/10/2005)

8.1.4. Damages

The client may suffer damages for the following reasons:

1. The project may be late if the requirements analysis has to be repeated at a later stage, which may very well cause productivity problems if the system was a critical one anticipated to be available on a certain date.

2. The project may very well be over budget as result of the delays caused by reworking the requirements analysis.

3. Should the resulting software prove to be inadequate, a new system may have to be developed at great cost to the client, or the existing system may require adaptation.

It is possible that third parties would suffer damages as a result of negligence at this stage, but unlikely as discussed below.

Any losses incurred will naturally be worse the later in the software lifecycle the defects causing them are detected. This is why the verification process, discussed below, should ideally commence as soon as possible.

8.1.5. Establishing Liability

Of the errors described above, it is the "business" errors that will originate in the requirements analysis phase. It is less likely that architecture errors will be traced to this phase, as will be seen.

It is more than likely that any failure in this area will be dealt with in contract as it is unlikely to manifest in such a way as to have detrimental effect upon third parties. The reason for this is that it is merely the functional specification, not the design that is determined upon at this stage, and it is most unlikely that a third party action in delict would arise from this phase of the software life cycle. In other words it is determined at this stage what the software must be able to do, and in what circumstances, not *how* it will do it. Failures resulting from this stage are thus far more likely to be those of requested functionality or capacity not being offered by the final software, rather than the unexpected failure that is likely to cause loss to third parties. An exception to this could be failure in capacity for a critical system. The

capacity of a system should be determined in requirements analysis phase (though how that capacity should be delivered is determined during the design phase), and should the consultant underestimate for example the number of concurrent users that should be supported by a share trading system with a result that traders are unable to buy or sell shares at a critical time, the loss to third parties could be considerable.

While a client would certainly have a remedy against the consultant / developer under contract law, a delictual remedy would be unavailable due to the restrictions placed on such actions by the *Lillicrap* case as discussed.

An action could possibly be brought in delict against the consultant / developer by a third party on the basis that the incorrect analysis amounted to a negligent misstatement - it would probably be able to establish wrongfulness on the basis of the professional standing of the consultant / developer, subject to the other factors set out in the discussion on Delict. These factors are applied on a case-by-case basis.

In theory there should be no difficulty in establishing negligence on the part of the consultant or developer should it be remiss in its analysis at this stage. However, given the complexities of working out the client's needs as I have set out above, this may be easier said than done practically. In the case of a misunderstanding between the parties, the Defendant could at the very least claim contributory negligence on the part of the client. It may also be able to establish that the assistance given it by the client in its analysis was so unhelpful or confusing as to make an accurate analysis difficult, or even to amount to breach of contract by prevention of performance. The Plaintiff could possibly rejoin that the consultant with its claimed expertise should not have delivered an opinion if it was not able to ascertain the facts adequately, or that with adequate experience it should have been able to ascertain that the client was uncertain of its needs.

What test will the court apply in determining if the actions in drawing up the requirements analysis were negligent? It would certainly apply that of the reasonable expert. Given the plethora of specialisations within the IT industry, the reasonable expert here would need to be the reasonable systems analyst or similar, subject to the reservations expressed above.

8.2. Design

As is the case with all complex undertakings, software should be planned before it can be written. What the developer essentially does in this phase is to take the functional specification and use this to develop a design of the software and related materials like databases.

8.2.1. Parties

The party involved in this phase will be the developer, though it may of course happen that the consultant who was involved in the requirements analysis works with the developer in the design of the software. The developer may subcontract certain aspects of the development to third parties. For example, where software will make use of a database, the developer may use a third-party database developer to design the appropriate data structures. If certain aspects of the system require other specialist expertise, this may also be outsourced.

8.2.2. Actions

Design is governed by the contents of the functional specification and what is designed will differ widely depending upon the nature of the software. Common tasks will include determining what modules will make up the system, how they will be structured and what functionality will be addressed by which component, what the GUI will look like and how it will operate (critical for the usability of the software), and how data processed by the software will be manipulated and stored (the variables that will be used to store the data, which databases will be used to store it, the database structure and so forth). The programming language to be used will be chosen during the design phase[60] (though the choice will often be dictated by the nature of the project and, especially in the case of smaller software companies, the languages that the developers are familiar with).

8.2.3. Possible Points of Failure

A poor design will in extreme cases lead to software that fails catastrophically. If the architecture is poorly planned, the final system could be unstable. Moreover, poor

[60] For an indication of the number of available programming languages, see http://www.99-bottles-of-beer.net (accessed 24/10/2006)

planning could lead to compatibility problems with existing systems. However, it is more likely that such a shortcoming will lead to software that does not fully meet all its functional or non-functional requirements in a more benign way. This can take the form of the software not offering functionality that was envisaged in the functional specification, but it may also happen that the architecture chosen or other planning faults lead to software that delivers the functionality, but does not have the capacity to deliver it adequately (an example of a non-functional requirement not being met).

For example, a distributed accounts package which may be accessed by hundreds of people at different locations of the same time may use such a cumbersome database that it simply cannot cope with having a more than a dozen people using it at the same time. While it may provide all the functionality envisaged, it is of little use if it cannot deliver this functionality on the scale required.

Should the data structures prove to be inappropriate, this can have profound long-term consequences. The best example of this of course is the year 2000 bug, where the data structure restricted the storage of year the data to two figures. While the developers who wrote much of the affected software 30 years ago cannot be held liable for not predicting the problem (they did not anticipating the affected software to be in use for as long as it was), it is an excellent example of the kinds of impact that a poorly planned data structure can have.

It is also at this stage that error handling is usually planned, and should this be badly designed, it can be very disruptive to redesign the error handling at a later stage, presuming that the fault is detected at all.

8.2.4. Damages

As with a poor requirements analysis, a poor design may need to be reworked. If this should happen at the end of the design phase, the effects would not be as serious as where the defects in the plan are established further into the process, where the damages will be more significant. The effect of this clearly increases as the project proceeds. The damages will be those associated with any project being delivered late or over budget.

Should poor design not be discovered before the software is delivered, or should it be decided that it would be more cost-effective to roll-out deficient software and to

begin the planning process over, the result may be unworkable or deficient software (the former is unlikely where the fault is discovered prior to roll-out, as it would not be decided to continue should the software most likely not work).

The software may not offer functionality that was agreed upon, and there is nothing unique to software development in that. What is unique however is where the software does not perform adequately in conjunction with the hardware that it was designed to execute on. In that case, the client may be forced to purchase further hardware in order to have the software perform at an acceptable level. The client may also have to commission further software (or make changes to the delivered software) to allow this solution to work. An example of this would be way the software is designed to be run on one server, which proves to be deficient. In that case, software will need to be obtained to allow two or more servers to be "clustered", allowing the workload to be shared between them.

A point that is often overlooked is the importance of planning the graphical user interface (GUI). With most modern applications relying heavily upon an intuitive interface between the user and the application, a clumsily designed GUI can significantly handicap usability of an application.

As described above, it is less likely that the types of failures leading to damage to third parties will emanate from this phase, but it is still quite possible. Not only could damages of a purely economic nature occur, as in the case of a millennium bug-type scenario, but should for example a flight traffic control system have a poorly designed interface that is confusing to its operators, leading to a mid-air collision, the damages would clearly be of a far wider ambit. They would include damages suffered by aircraft passengers (or their dependents), the owners of the aircraft, and any parties on the ground affected by the crash. The lack in capacity in the stock broking system discussed above could also be caused by faulty design.

8.2.5. Establishing Liability

It is in this and following stages that serious software defects are likely to have their genesis, though given the fact that it is relatively early in the software lifecycle, the likelihood is high that such faults will be picked up in testing. It is worthwhile bearing in mind that while the phases of the software lifecycle are treated separately, it can be difficult to determine where a fault first manifested itself. It may happen that a

person planning a software project relies upon the fact that during the construction and formal testing processes, the design will be checked and rechecked, with the result that any minor defects will be picked up. Naturally this does not excuse the person responsible for design from liability for negligence at this stage.

Again, the test for negligence to be applied by the courts would be that of the reasonable expert. In my view in the particular context of avoiding defects in software, the reasonable expert would have to pay attention to the following during the design phase:[61]

- **The nature of the software**

  Whether the software is critical or expert software as opposed to software where defects are not likely to give rise to damage is an important consideration. In considering the question of negligence therefore, a court in determining what steps the reasonable expert in the place of the Defendant (designer) would have taken to guard against errors in the software resulting in damages should impose a far more onerous burden upon the designer of software which controlled the dosage given to patients by cancer radiation therapy equipment, or software used for air-traffic control than it would in respect of a small-business accounting package or, even further down the scale, a computer game.

- **Users**

  Another factor is that of the expected users of the software. This is obviously tied to the nature of the software itself, as expert or critical systems are more likely to be used by trained individuals. Thus a critical system which will be used by people who are not experts in the field or trained to operate computers needs to be designed in such a way as to be as robust as possible to guard against damage caused by misuse. Moreover, attention should be given to the kinds of misuses that users make of the software, and guard against them accordingly.

  For example, if a user of the Windows operating deletes files, he is asked to confirm that this operation should be performed by clicking on a "Yes" button.

---

[61] My thinking was influenced by Alvin S Weinstein "The Performance Audit: Minimising Software Liability (Part I)" (1988) 29 IDEA: The Journal of Law and Technology 127 p130

On the other hand some database management systems, which are designed to be used by experts, give no such warning even though the consequences of a mistake would most likely be more dire.

- **Environment**

  The environment in which the software will be used is also important to take into account. For example, the designer may have overlooked compatibility issues in our radiotherapy software example. This incompatibility, between the software and some other software commonly installed on computers used for medical purposes, may cause the software to cease operating from time to time. As no radiation would be administered to the patient under these circumstances, the error is not a serious one from the legal perspective (though rather embarrassing for the software developer). However, should this type of ever manifest itself in a fly-by-wire software system, it would be hazardous in extreme for the occupants of the aircraft involved.

  In practical terms, the more chance there is of any factor interfering with the operation of the software, the greater the duty upon the designer to guard against such interference, subject naturally to the other factors.

It is an open question whether defective design would give rise to an action in delict as a negligent misstatement or a defective product (which would allow for a concurrence of actions). I would attempt to distinguish it from the provision of professional services as contemplated in *Lillicrap* on the basis that in software development the design is closer to the construction phase than for example the drafting of architect's plans is to the construction of a building, and that its proper home is an action arising from a defective product. The architect's planning phase would be closer to the requirements analysis phase of the software lifecycle. On that basis, there would be a concurrence of actions, the delictual action to be brought on the basis that the design led to a defective product.

If this argument were successful, a third party would probably have an action for defective product if damage were caused by defective software. Depending on the facts of the case though, it could be argued that the developer could not have anticipated that a software failure would damage a third party, or that the extent of the loss (in the case of a really calamitous failure) was such as to preclude wrongfulness for policy reasons.

As to defences, there can be no question of contributory negligence on the part of the client at this stage of the software cycle, and this is also the case for the construction and testing stages.

8.3. Construction

In this phase, the developer and any contracted third party creates the relevant software with reference to the design.

8.3.1. Parties

Naturally the developer will be the major player in this stage, but as with the design phase, it may have subcontracted aspects of the task. A third-party tasked with designing a database structure may produce the database itself. It is also common for developers to make use of third-party software. For example, the developer of a website may use an application that loads advertisements on to designated areas of web pages served by the Web server, and keeps track of the number of "hits" on those advertisements. The developer of a database of an application may use third-party software that allows his application to communicate with the database, which is itself third party software.

Apart from third-party software incorporated into developed software, some software is actually itself developed using third-party software – the CASE tools described earlier.

8.3.2. Actions

While a complex process, the construction phase is essentially that of building the software according to the design. In doing this, the source code of the various modules will be written, any third-party software integrated into the programme, databases built and so forth.

8.3.3. Possible Points of Failure

As this phase involves writing the code that the computer will ultimately execute, the possible points of failure are almost infinite. Not only could the developer stray from

the design, but may make careless programming errors that if left undiscovered could cause failure later.

Because the construction phase follows the design, any errors made in the design will manifest in the construction. Apart from those errors resulting from faulty design, which we have discussed above, the construction phase is also where any syntax errors and runtime errors will appear, as well as logic errors. It is, in other words, the phase in which careless mistakes are most likely to be made: while a grammatical error in the design is unlikely to have an adverse effect, a careless mistake in construction can easily lead to a runtime error.

8.3.4. Damages

A discussion of damages here is superfluous, as almost anything that can go wrong could be caused at this stage. The circumstances leading to damage are extremely varied, with any loss that can be caused by defective software potentially having its origin here.

As hinted above, it is in the construction phase that the faults that cause damage to third parties are most likely to have their origin. This is because the types of faults that cause serious failure tend to be mistakes in writing code. While it is perfectly possible for failure to be caused by a system architecture (finalised during the design phase) not able to handle a certain workload, in my opinion such faults are far less likely to cause damage, and are moreover more likely to be picked up before the software is released.

8.3.5. Establishing Liability

Construction is the implementation of the design, thus in determining negligence it is in my view not necessary to have regard to factors such as the purpose of the software. This is properly dealt with in the design and, as I shall show, the testing phases. The construction phase involves taking the design and building it in as competent a manner as possible under the circumstances.

Assuming an action in delict on the basis that the software constitutes a defective product (I don't see any trouble with establishing wrongfulness in principle here, though the factors enumerated above will have to be applied on a case-by-case

basis), it is difficult to apply a "reasonable expert" test to this phase for a number of reasons, in addition to those raised in the discussion of the "reasonable expert" test above. Firstly, the reasonable programmer would, while certainly making a reasonable effort to avoid making coding errors, be working on the assumption that his code will be tested. Moreover, on a practical note, given the multiplicity of programming languages which are available, as well as databases and other ancillary technology that a computer programmer might specialise in, it is difficult to construct a "reasonable expert". Each programming language has its own syntax, and given that errors made in the construction phase will be specific to the programming language or other technology being used, the practical difficulties for the courts to apply the test of the reasonable expert to the construction phase would be enormous. In the United States, for example, there have been suggestions that computer programmers should be certified, in the same way as doctors, lawyers or engineers are, so that they would be liable for malpractice suits. While the American jurisprudence in this area is very different from our own, this is an interesting perspective. It is worthwhile noting however that the writer who proposed the measure was only in favour of developers of critical or expert systems being so certified.[62]

Despite the difficulties set out above, a court must still attempt to establish a standard to measure the conduct of the developer against. Establishing such a standard would be far easier were the negligence such a blatant nature, that every programmer of any experience would know not to act I that way. An example would be the division-by-zero run-time error, which will produce the effect discussed earlier in almost any programming language.

If third-party software has been used by the developer and this software proves defective, this may cause the developer's own software to fail. Under these circumstances, the developer may choose to join the third-party in any resultant action, or to pursue it subsequently.

In the case of this third party software, there are several points to consider. Should the software be widely used, as is often the case, other developers may have discovered its shortcomings, which would then almost certainly be published in chat

---

[62] Perlman D T, "Who pays the price of computer software failure?" (1998) *24 Rutgers Computer & Tech. L.J.* 383

rooms and other online resources. The third-party vendor would probably also make these shortcomings known, if it was aware of them, in its accompanying documentation. Where the third-party vendor has itself published this information, it would be almost impossible to prove fault on its part if this information was available at the time that its software was included in the developer's product. Where these shortcomings have been published by the developer "community", the third-party vendor would at the very least be able to argue that there was contributory negligence on the part of the developer in not appraising itself of this information. Should the shortcomings be published on a website or other resource that the developer is known to use, or be a well-known shortcoming, this argument may defeat the developer's action entirely.

I would submit that the same argument applies where an IDE is used, though this is of course subject to the licensing agreement which would almost certainly contain a waiver of liability.

8.4. Verification and Validation (Testing)

While the public image of a software developer is of someone sitting in front of a computer terminal writing computer code constantly, the reality of software development is somewhat different. A large amount of time is spent ensuring that the software performs correctly. This phase is the most important as far as liability is concerned. It is after all of no consequence for our purposes whether developers in writing code make careless errors if these errors are picked up in the validation stage (unless of course such errors result in the project being delivered late or over budget). As will be seen, faults in the requirements analysis and design phrases are detected through verification, faults in the construction by both verification and validation.

Testing can be divided into two broad forms – verification and validation. Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.[63] Validation, which is what most people think of as "debugging", is the

[63] Kit 29

process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.[64]

8.4.1. Verification

There is more to testing software then ensuring that the code itself operates as required. As alluded to above, is quite possible to produce software that "works" perfectly well, but does not meet the requirements of the client – a perfect landing at the wrong airport so to speak. This is where software verification comes in. In other words, verification is a "human" inspection of the software being built[65] which concentrates on the project documentation rather than the software code itself. Clearly then the sooner verification is begun the better: if the requirements analysis is incorrectly performed and the software development process begins based upon an error at that stage, the expenses arising from the error, both to the developer and the client, increase the longer the error remains undetected.[66]

On the other hand, if there are budget or time constraints to the project, it might make sense to restrict verification to the core functionality of a program – ensure that the most important aspects of the software are as the client wanted them. Further, as the greatest pay off is obviously verification of the requirements analysis – the so-called "requirements verification", any skimping on verification should ideally take place later in the project.[67]

As implied above, verification should ideally take place throughout the software life cycle, ensuring at all stages of the project that the software being developed fits the requirements of the client or of the functional requirements if generic software is being built. The methods used in verification are really those dictated by common sense. The best practice seems to be drawing up a checklist at every stage where verification is carried out, and running through that checklist in the course of the testing.[68]

---

[64] *ibid.* 30

[65] *ibid.* 57

[66] *ibid.* 62

[67] *ibid.* 62

[68] *ibid.* 63

The nature of the tests depends very much upon the phase of the project being tested. For example, where the functional specification is being tested, the overriding factor will be determining how successfully the requirements analysis has been translated into the design. Where the code itself is being tested, the code will have to be compared with the design. At this stage, the code itself may be read through with reference to a language specific checklist, depending which language was used to write the code.[69] Note that this is not the same as validation, where the code is tested in operation.

8.4.2. Validation

There has been a trend for increasingly complex software, as advancing technology allowed for more powerful computers to be produced at a reasonable cost. While the 1983 version of Microsoft Word had approximately 27 000 lines of code (LOC), by 1995 the same computer program was 2 million LOC.[70] Further examples of approximate lines of code are 14 million for the software operating the International Space Station, 7 million for the Boeing 777 airliner, 1.5 million for the Linux operating system, and 14 million for the Windows XP operating system.[71] Software that has been rigorously tested will have, on average, five faults per thousand lines of code.[72] The more complex the software, the greater the chance that a fault in it will cause an error.

Further aggravating factors are increasing efforts to allow software to operate across as many different "platforms". The goal is to allow the same code to be run on any computer (no matter what operating system it uses), cellular telephones, personal digital assistants, and in the future probably microwave ovens and refrigerators too! While these advances are to be encouraged, they greatly increase the scope for software failure.[73] Further, the fact that most computers nowadays are connected to the Internet gives it further scope for trouble through vulnerability to security breaches, malicious code, and the need to interact successfully with other systems remotely.

---

[69] *ibid.* 70

[70] Hoglund 14

[71] *ibid.* 15

[72] *ibid.* 14

[73] *ibid.* 18

8.4.3. Parties

It is often the case, especially in small organisations, that the developer will test his or her own code. The problem with this approach is that the developer is forced to wear two "hats": one as the developer and the other as the tester. Because as a developer he or she has an emotional involvement with the work done, there is an unavoidable tendency for someone testing his or her own work to overlook faults that a disinterested party would pick up.[74] Another approach for the small organisation is for developers to test each other's code. While this is a better approach, it is difficult for a developer to simultaneously learn the rather specialised skills required of the tester.[75]

A large organisation can afford the luxury of having a dedicated tester or testers, either within the development team or constituted as a separate department. The exact structure will obviously depend upon the size in the nature of the organisation.

It is of course possible, for the ultimate in impartiality, to employ a third-party tester, which would have implications for the question of liability. This is usually only done for so-called "high – level" testing described below.[76]

8.4.4. Actions

I do not propose to attempt an exhaustive examination of one or other of the many software validation schemas. There are several reasons for this. Firstly, such a detailed examination would be beyond the scope of this inquiry, involving a technical and the analysis that is not helpful for our purposes. Secondly, as the type of software being developed changes, so the methods required to test it will change apace. For example, an application with a traditional "command line" interface will have this interface tested in a very different way to an application using a Graphical User Interface (GUI). Similarly, different kinds of software will require different types of testing, making it very difficult to come by a hard and fast rule. Finally, it is my intention to distil the essential principles of validation, so as to apply a test of

---

[74] Kit 167

[75] *ibid.* 167

[76] *ibid.* 96

reasonableness to the conduct of a tester. Given the wide range of applications that can be written, and the wide range of circumstances that applications are used in, the test for negligence will be whether the tester took reasonable steps against software failure in the circumstances. I must thus come to such principles as unencumbered as possible by technicalities and the requirements of current technology.

Having said that, in order to get a sense of how the testing process works, I will paint in broad brushstrokes what the validation process entails. This process is informed by the way the software is written at the moment, and it will become obvious that such a process is subject to change over time as technology changes. My sense however is that the categories of testing set out below will remain fairly stable, while the specific tests they contain (which are not described in any great detail) will change.

Modern software is usually built in a modular fashion, meaning that code which is written to perform a certain function will be developed in such a way that it can "stand alone", so that should a particular function need to be updated at a later stage, instead of reworking the entire source code, merely a small portion of the code need be reworked and "slotted in" to the object code. This style of programming has been ubiquitous for some time, as it saves programmers much duplication. While programmers will differentiate between merely "procedural" code, where all the source code is compiled into a single object code, and "object based" code, where the code that implements different functionality is compiled into separate "objects" which can interact with each other so as to constitute a system in combination, the essential idea is the same in isolating similar functionality into different "blocks".[77]

In a complex system, where many of these components interact, it is important that not only the components themselves be tested, but their interactions with each other be scrutinised.

The most commonly used testing model is a commonsense application of this programming structure. Firstly, individual components which implement particular functionality are each tested in isolation ("unit testing"); then the components are combined and tested as a whole ("integration testing"). These first two stages are

---

[77] See Walters 208 *et seq*

often referred to as "low level" testing.[78] Thereafter "high-level" testing, including "system testing" may be carried out which does not test the components or how they fit together, but the system as a whole.[79]

There are two broad testing strategies, being Black Box and White Box testing. These are not methods used to test the code of such, but rather frameworks within which different sets of methods are used. So for example "Black Box testing techniques" will refer to the testing methods used with a Black Box strategy.

A black box strategy assumes that the tester is ignorant of the internal program structure of the component – hence its name. Methods used in black box testing thus check the performance of the component against its design. White box testing on the other hand requires knowledge of the code used to write the component.[80]

Note that within each of the categories of testing set out below there are many separate types of tests which are beyond the scope of this enquiry.[81]

8.4.4.1. BLACK BOX TESTING

The advantage of black box testing is that (in theory at least) the tester is not biased towards the code, and can thus give a more objective analysis. The tester should in fact never be the developer who actually wrote the code. Unfortunately, Black Box testing can only test for what the component is supposed to have in it – if the developer has inserted extra material for whatever reason, the tester will not pick it up.[82]

The tester is given the specification for the component, setting out the inputs that it is signed to accept, and the outputs that it is designed to deliver. This method is hence also known as "functional testing".[83] The tester will then input data into the

---

[78] Kit 93

[79] *ibid.* 96

[80] *ibid.* 79

[81] see Kaner C "Software negligence and testing coverage" (updated at http://www.kaner.com) (1995) 2:2 *Software QA Quarterly* 18 for a list of 101 such tests

[82] Kit 80

[83] Wikipedia "Black Box Testing", available at http://en.wikipedia.org/wiki/Black_box_testing (accessed 24/10/2005); Kit 81

component in isolation from the rest of the system, and study the outputs. The tester will not only use expected inputs, but will input data that is unusual or invalid, to check that the component is able to deal with such inputs without error. Several approaches exist to assist the tester in doing this, including "equivalence partitioning" and "boundary value analysis", the precise nature of which lie mercifully outside the ambit of this enquiry.[84]

8.4.4.2. White box Testing

Also known as 'structural testing', because the testing is done with knowledge of the structure of the software, this strategy is usually used for unit testing.[85] This testing strategy involves the testing of the actual code, rather than merely how the program behaves. Because the tester can see the code he is testing, a different set of testing methods come into play. This strategy is particularly useful when one wants to ensure that every statement in the component is executed at least once during the testing process.[86]

Some common examples are line testing, where the tester attempts to execute every line of code, branch testing, where the possibilities of every conditional statement are tested, and path testing, where every possible path through the code is followed. The proportion of the code covered in each test is referred to as "line coverage", "branch coverage", and "path coverage" respectively. These tests become more improbable in order. Line testing can be a reasonable burden for the tester; branch coverage is exponentially more complex. Path testing is certainly the ideal; if every possible path through the component has been executed, then every statement must also have been executed. As the number of paths through even a small component is very large however, and increases exponentially with the size of the component (in particular where the code contains loops), path testing is only feasible for relatively small components.[87] Note that while commonly-used as measures of "testedness", these are just a few of many available tests.

---

[84] *ibid*; Kit 83 *et seq*; Sommerville 443 *et seq*

[85] Sommerville 448

[86] *ibid.* 449

[87] *ibid.* 450

If strictly adhered to, this strategy will not detect missing functions, because the tester does not have to refer to the functional specification and thus won't know what is missing.

8.4.4.3. UNIT TESTING

This is the first of the low-level tests. Let us assume that a certain company is developing an antivirus software package. This software will be installed by its users, and will scan for and detect computer viruses based upon definitions of viruses which are downloaded from the company's website from time to time. A certain component of the antivirus software would be responsible for applying the virus definitions to files as they are opened on the user's computer, scanning these files and ensuring that they are not infected with computer viruses. The software would be entirely made up of components such as this which interact with each other. When testing this software package, the tester would first test each of these components individually before testing the system as a whole. This is unit testing. It is important to distinguish here between verification, where the code for the component will be compared with the design, and validation which is what is being done here. The tester will in essence be trying to "break" the component – trying to make it behave in a way that is inconsistent with the design, or which is obviously erroneous.

It is most likely that White Box testing would be used to validate the component, and would use such techniques as described above. The tester then has knowledge of the inner workings of the code and can use this knowledge to anticipate the best data to use as inputs to test the functionality of that component.

8.4.4.4. INTEGRATION TESTING

This is also a low-level test. As the name suggests, this process tests the integration of the various components to software into the whole system. There are two broad approaches: incremental integration where components are added one by one and the system is tested after each addition until all the components have been added, and non-incremental integration where all the components are combined at once and tested.[88] The incremental approach makes it easier to detect errors arising from the

[88] Kit 94

interactions between the components.[89] The ability of the components to interact is what is what is being tested, not their internal functioning, so a black box testing methodology will be used here.

8.4.4.5. Usability Testing

The remaining tests are high-level tests. As its name suggests, this tests the software for ease of use. The purpose here is to ensure that the software suits the user's expectations both as regards ease-of-use and existing work styles. Both the operation of the system and the user documentation is reviewed.

Such testing is most effective when the intended users of the system are involved in the testing, and such testing begins as early as possible in the development process. Usability testing is validation not verification because the tester interacts with the system itself, not with the planning and documentation. There is a plethora of usability tests available, depending upon the nature of the software.[90]

A rather extreme example of where usability issues can be important is that of the USS Vicennes, a US Navy warship which in 1988 shot down a commercial airliner over the Persian Gulf. The crew of the ship had misidentified the airliner as a fighter aircraft. The reason they gave for the specification was not a software error as such, but rather that the screen display of the radar system was misleading.[91]

8.4.4.6. Function Testing

Function testing attempts to pick up the divergence between the functional specification and the functionality actually delivered by the program. This can be begun as soon as enough of the software has been built for functionality to be tested. The methods used follow from this – the functional specification is analysed to determine what functionality the program should have, and the program is then Black Box tested to ensure that it in fact delivers the functionality as planned.[92]

---

[89] Sommerville 452

[90] Kit 96

[91] Hoglund 12

[92] Kit 100

8.4.4.7. SYSTEM TESTING

System testing has been described as the test to show that the program will pass Acceptance testing (the final test, usually carried out by the client, described below). It is not the same as function testing, as it is the nonfunctional elements that are tested, such as its ability to handle a certain workload, or to recover from an error. (Having said that, all the high-level tests other than acceptance testing are sometimes included under the rubric of System testing).

The following tests could form part of System testing:[93]

- Volume testing: where anticipated demands on the software in terms of volume of inputs and required outputs are made to ensure that it can meet its planned requirements. In the example of the share trading system above, the inability of the software to support more than an unacceptably low number of users simultaneously would be picked up at this stage.

- Load/Stress testing: attempts to find the limits of the software's ability to meet requests within given time periods. This would obviously need to take account of the hardware that the program is operating on.

- Security testing: checks if the level of security required of the program has been met – an attempt is made to challenge whatever security measures have been built.

- Resource Usage testing: checks whether the program uses acceptable levels of system resources (memory, disk space, Internet bandwidth etc).

- Configuration testing: checks that the program operates as it should with the hardware that it is supposed to run on and the software that is to be run in the same environment when configured as required.

- Compatibility testing: if the program is required to be compatible with other systems this will be tested here.

---

[93] *ibid.* 101 *et seq*

- Installability testing: checks that the program can actually to be installed successfully, and checks for situations that could lead to errors resulting from installation.

- Recovery testing: this ensures that if the system fails it can be successfully recovered. For example if computer software used by bank fails for whatever reason and once it is restarted at is found that records of transactions have been lost, this is a serious problem. This test checks that no data would be lost in such circumstances.

- Reliability/Availability testing: checks that the program offers the level of maintenance and "down" time (or less) specified for it.

8.4.4.8. Acceptance Testing

Where bespoke software is written, acceptance testing will typically be done by the client to satisfy itself that the software performs as the client required. It is often a requirement of software development contracts that acceptance testing be passed before the software is "signed off". If the software is generic by nature, the developer may use Alfa or Beta testing, and sometimes a combination of the two to ensure that the software meets the requirements that it set for itself.

The difference between Alfa and Beta testing is simply that Alfa testing is carried out by people within the development organisation (who were ideally not directly involved in the development of that particular software), while Beta test testing is carried out by a (more or less careful) selection of end users. Both involve testing the software in the form which the final release will take, or a form as close as practicable to it.[94]

8.4.4.9. Measuring Testing

The term "coverage" is usually used to describe the level of testing which has been done on a system. It can be used in relation to a specific test, as in "line coverage" which is a measure of the number of lines out of the total number of lines in the source code of a computer program that had been tested. It is also common to refer to "statement coverage", which refers to statements in a computer program that have

---

[94] *ibid.* 103

been checked. Thus "85% statement coverage" means that 85 percent of the statements in the source code have been tested. This can be misleading however, as just because a statement has been checked does not mean that possible errors have been eliminated in that line of code. Because a line of code may contain a branch, which directs the execution of the program to a different place in the code depending upon the inputs, the simple statement coverage test or line coverage test does not give an adequate indication of how well tested the system is. While a syntax error may be picked up in this way, problems caused by division by zero for example more than likely will not be. As each test has its own "coverage" measure, it is very difficult to arrive at an accurate measurement of coverage. [95]

"Coverage" can also be used in the wider sense in relation to the entire testing process. Coverage in this context would thus be the degree of 'testedness" of a computer program. Again, given the huge range of tests available, it is difficult to come to any meaningful judgement as to whether a statement relating to coverage is in fact meaningful. Because the types of tests performed will differ both between the type of application and the experience and preference of the tester (or testing team), comparisons are difficult.

There is extensive debate in the industry on what a reasonable standard of testing should be. For example, there is a group called the "Context-Driven School of Testing" which believes that there are no best practices, but that testing is itself a skill which allows the tester to adapt to each particular situation. There is a camp which holds that all testing processes should be carefully planned ("scripted") in advance, which is opposed by the "exploratory" camp which holds that test design and execution should take place simultaneously. There is also considerable controversy on the issue of manual versus automated testing. Some are of the opinion that automated testing is of little value, as the value of human intuition is not to be discounted. On the other hand there are those who hold that automation should be used in all cases.[96]

---

[95] Kaner 3

[96] see eg Wikipedia "Software Testing", available at http://en.wikipedia.org/wiki/Software_testing#Controversy (accessed 26/10/2005)

8.4.4.10. COST OF TESTING

Testing is an expensive undertaking, as it takes up a lot of time and employs highly qualified individuals. As I have discussed, it is axiomatic that all software contains errors, even software that has been rigorously tested over a long period of time. As more errors are found, the effort taken to find the remaining errors becomes greater, as they become "rarer". The cost of finding errors thus increases with every error found.[97] Moreover, even if errors are found, it may not be worth while to fix them. An example of this is the much-publicised release by Microsoft of software with defects in it known to them.

It is apparent from the discussion that testing is a very complex process. Not only that, but the huge number of potential tests available leads to a situation where one could potentially test relatively simple software for an almost infinite amount of time. Not only would all possible tests not be exhausted, but all errors would almost certainly not be found after a long period. Kraner[98] lists 101 unique tests that can be performed, yet doubts that even a high level of testing will find all defects. The level of testing to be performed is thus a balancing act between the need to find errors and the cost of doing so. This balance will obviously shift depending upon the computer system being tested. In the case of a computer game on the one hand, an error would cause customer dissatisfaction, but is unlikely to be a matter of life and death. On the other end of the scale, critical systems such as fly-by-wire software for commercial aircraft must be as fault free as possible. Thus the resource expended on testing such critical systems will of necessity be higher than that spent in testing a system where failure will be less calamitous.

My description of the testing process is by no means comprehensive, but is designed to give some idea of the basic principles of software testing, the complexity involved, and most especially the difficulties involved in striking a balance between depth of testing versus cost and time constraints.

---

[97] See e.g. National Institute of Standards and Technology "The Economic Impacts of Inadequate Infrastructure for Software Testing" (2002) Planning Report 02-3, 4-2

[98] Kaner C "Software negligence and testing coverage" (updated at http://www.kaner.com) (1995) 2:2 Software QA Quarterly 18

8.4.5. Possible Points of Failure

It is important to bear in mind that any failure in this phase means that in effect there are two failures: the negligent act or omission which occasioned the fault in the first place in a previous phase, and the negligent testing which failed to find that fault. As verification could be carried out from the beginning of a project, testing can be seen as accompanying the software lifecycle rather than being a phase on its own.

Apart from the obvious failure of negligence by individual testers tasked with testing the software, the most likely point of failure is one of inadequate planning. As should be clear from the above, and as will be discussed more fully below, effective testing (both verification and validation) requires proper planning that takes account of the project circumstances and balances these against cost and time constraints.

8.4.6. Damages

The damages caused by a failure in testing will be the same as those discussed with regard to the previous phases in which the fault originates.

8.4.7. Establishing Liability

It is the testing phase which is the most important as far as liability for defective software is concerned. While I have examined the preceding phases of the software lifecycle, and while defects will certainly manifest themselves in those phases (in fact they will certainly have their origins there), it is testing that is the most practicable safeguard against defective software. Again I foresee no difficulty in establishing wrongfulness when bring an action under products liability, subject to the factors set out in the discussion of wrongfulness.

I have already covered the distinction between verification and validation, but it is important to stress the fact that verification takes place throughout the software lifecycle up until the release of the software. While the purpose of verification is more important from a contractual perspective to ensure that what is being built is what the client requested, none the less verification is important from a delictual perspective in that it will pick up on flaws in design, for example where the design is compared with the functional specification. Moreover, verification of the code against the design may similarly pick up errors that may otherwise not have been found.

Despite this, the best way to pick up faults that cause legally significant failures is through validation. The trouble with validation, as is apparent from the discussion above, is that it is impossible to finish. There are so many possible validation tests which can be performed on any given software that performing all of them is simply not practicable. Performing full path testing would take such a long time (and be so expensive) for anything other than a component of the lowest level of complexity, as to render software development unprofitable. The question then is what a reasonable level of testing is for a given software program.

Applying the test for negligence as set out above to testing would result in the following:

- the reasonable tester in the position of the defendant
    - would foresee the reasonable possibility of a particular fault causing failure which would result in patrimonial damage, and
    - would have taken reasonable steps to find such a fault, and
- the real tester did not take these steps.

The complexity of software is such that it is most unlikely that a tester would foresee a particular failure occurring. The balance of academic opinion on foreseeability seems to be that a specific consequence or consequences should be foreseeable, not that there was a general risk occasioned by negligence.[99] In the case of software testing, I would submit that it is not a particular failure that the reasonable tester would be required to foresee, but rather a particular class of failures for example unanticipated user inputs causing system failure, or incompatibility with existing systems doing the same. A tester should know from experience what failures are most likely, and testing is performed on a probability basis in any case. To expect a particular failure to be foreseen would be unrealistic.

As to the reasonableness of steps to be taken, the following example demonstrates the tester's dilemma. Let us assume that the tester has a hundred possible tests open to him with which to unit test a particular component. Let us assume further that

---

[99] Neethling *et al* 137

each test takes the same amount of time to complete, and that each test is designed to check for particular different types of fault. The tester can obviously not run all 100 tests, but must strike a balance between the time (and associated expense) of running tests against the possibility of a particular error occurring. As it is almost impossible for anyone to anticipate the exact error that could occur, a cost / benefit analysis must be made based upon what failures the tester considers most likely to occur or to cause the most damage if they were to occur versus the time and cost required to find the faults that would cause those failures.

I would thus restate the test for negligence here as follows:

- the reasonable tester in the position of the defendant
    - would foresee the reasonable possibility of a particular class of failure occurring which would result in patrimonial damage to another, and
    - would have applied a reasonable level and manner of testing to find faults that could cause such a failure, and
- the tester (Defendant) failed to take these steps.

The reasoning that I will use here is similar to that which I used when discussing the design phase and involves examining the nature of the software to determine the level of testing which is reasonable under the circumstances. Of course the above example is simplistic. Different tests take different amount of time and may cost more per hour to perform than others because of the level of expertise required. Moreover, different applications will demand different combinations of tests because of the different architectures that are involved. For example, a server application which does not interact directly with users will not require the same testing of the graphical user interface that a desktop application will.

The level and type of effort to be expended by the reasonable expert (tester) in testing (including both verification and validation) for any given software should therefore be established with reference to the following factors:

- **Nature of the software**

  Clearly more effort should be spent in testing critical and expert software than software which is neither of these. It should be stressed however that this is a continuum and not a simple categorisation. There are certainly different levels of criticality, and the failure of an expert system can cause more or less damage depending on the circumstances. A website offering medical diagnoses is not likely to cause as much damage as an engineering program that miscalculates the amount of re-enforcing steel for a skyscraper. I submit that the reasonable level of testing be proportional to the damage that would result from the worst foreseeable failure of the software. So for example an accounting package, even one which is written for relatively small businesses, will clearly have the greater potential for damage than a computer game. A website allowing users to search for immoveable property for sale would not demand the same level of care as software controlling traffic lights.

- **Anticipated users**

  It is a source of wonder to many software developers the myriad (unanticipated) ways that users of their software find to make it fail through sheer incompetence. A system written for users who are highly computer literate or used to working with computers in their field does not necessarily have to be as robust (or "idiot proof" as many developers would have it) as software which is written for use by the general public. While the former type of application may not fail simply because the users are unlikely to abuse it through ignorance, software which is exposed to the vagaries of the lay user must be, subject to the other factors, more rigorously tested to ensure that it is as robust as possible. The reasonable tester would anticipate the abuses that the system is likely to be exposed to.

  Should the system fail, it will also have different effects depending on the user. For example, an expert system is more likely to mislead a lay person using it than an expert if it provides an erroneous output.

- **Environment**

  The essential point here is that the more likely it is that an extraneous factor will interfere with the execution of the software, the more care must be taken in testing it for its ability to handle such interference.

For example, software which is to be used in conjunction with other computer programmes must be tested for compatibility issues. If the environment that the software will operate in is known, as with bespoke software, the level of testing will depend upon the nature of the other known systems. As off-the-shelf software is rarely critical, its compatibility testing is not as important (and is in any event almost always covered by a shrink-wrap disclaimer of liability). If it were critical, the testing burden would be far greater as the environment in which off-the-shelf software will operate is unknown to the developer.

- **Current state of technology**

  Quite clearly validation specifically can only be as good as the methods that are available to carry it out. As much testing is carried out using tools that assist the tester and can automate many of the testing tasks, the technology available to the tester will have a profound effect on the level of testing which is reasonable. While a certain level of testing may be reasonable in certain circumstances today, in a year's time testing tools may have been developed which make certain types of testing cheaper and quicker, and thus justify a more extensive validation process for a given type of software project. This factor will also have an impact on the reasonableness of the *types* of tests performed.

  It is interesting to note that the EU Product Liability Directive,[100] which imposes strict liability for defective products in certain contexts, allows a manufacturer which would otherwise be strictly liable for its defective products to raise the so-called "state-of-the-art" defence. Should the state of advancement of scientific and technical knowledge at the time that the product was put into circulation have been such that the existence of the defect could not be discovered, the manufacturer of the defective product will escape strict liability. I would submit that this principle, which is the corollary of the principal set out in the first paragraph of this section, applies too: should tools not have been available to allow for the discovery of certain defects at the time when software was developed, the tester should not be held liable for such defects.

---

[100] Directive 1985/374

- **Cost**

  The above factors must then be weighed up against the cost of performing tests to a certain level. Courts will naturally have to make policy decisions as to where the trade off between cost and benefit lies here. Clearly the greater the potential for harm, the more time and money the developer can be expected to expend on testing. Likewise, a court is likely to hold that a lower spend on testing is reasonable in the case of software where the potential for harm is less.

Despite the application of reasonable testing, it is perfectly possible for failures to occur, as the following example demonstrates:

The American telephone company AT&T's telephone network switches are operated by software, and this software had a reputation for being very stable. The company had recently installed new switches, and had updated their software accordingly. This new software contained a function which kept track of the status of other switches on the network. If any switch "rebooted" as result of an error, it would send a message to the other switches "telling" them that it had been unavailable but that it could now accept routed calls. This message would be recorded by the function in question. The flaw in the function involved an interaction with data from incoming calls if the switch was hit by two calls within a hundredth of a second of each other. This interaction caused an irregularity in the data processed by the switch, causing it to reboot. After doing so it would send notifications to all the other switches in the network.

In January 1990 one of the switches on the network rebooted for an unrelated problem (due to the software's general stability such a reboot was relatively rare), and when it became available again it sent notifications of its status to all the other switches. This caused a ripple effect across the entire network as some switches were hit by two calls within one hundredth of a second and rebooted themselves. The more switches that were affected, the more chance there was of the remaining switches suffering the same fate, as the calls which would otherwise have been routed to the affected switches ended up being routed through those that were still available, increasing the chance of two calls being received within one hundredth of a second. As the rebooting process only took four to six seconds, the network was not completely unavailable – only about half of traffic could not be handled. None the

less the consequences were dire, especially as AT&T took several hours to identify and resolve the problem.[101]

The particular flaw was so obscure and the circumstances that resulted in the failure were so unlikely that it is doubtful that such a flaw would have been picked up by even the most fastidious testing process, and given the reputation of AT&T, it is likely that a reasonable level of testing was applied in any event.

8.5. Installation

The nature of the installation of software depends greatly upon the software in question: as we shall see, off-the-shelf software has far less scope for liability in this respect than does bespoke software.

8.5.1. Parties

There are essentially three parties that could undertake the installation of software. In the case of bespoke software, this is likely to be the developer itself, as this would have done the assessment of the client's existing systems, and will thus be in the best position to install in such a way as to cause the least disruption. Naturally this role could also be taken by a consultant if the consultant were involved in the whole project.

In the case of customised software, or complex generic software, the developer could certainly undertake the installation, but it is then more likely that a consultant who has been trained in the installation and maintenance of that software will undertake this task.

Naturally in the case of off-the-shelf software in the vast preponderance of cases it is the user which undertakes installation.

---

[101] Bruce Sterling B, "The Hacker Crackdown: Law and Disorder on the Electronic Frontier" available at http://www.chriswaltrip.com/sterling/crack1j.html (accessed 27/10/2005)

8.5.2. Actions

In the case of off-the-shelf software, where the programme is being installed on one machine, or across a small network, installation would be a simple matter of installing the software from the relevant media, be they compact discs, compressed file on the file server or whatever. This could be undertaken by the user itself.

At the opposite extreme, simultaneous rollout of bespoke software (or customised generic software like S.A.P.) by a company with branches throughout the world is far more complex. Not only would the installation process itself have to be carefully planned, with the appropriate personnel trained or hired in each location, and the installation media distributed (or, more commonly, made available on the central server for installation over the company's network), but certain other safeguards would need to be in place. For example, in the case of a migration from an old system to an updated one, all data would need to be backed up to ensure that failure in the installation would not lead to loss. Similarly, if an installation of for example an upgrade is taking place while the system is "live", steps will need to be taken to ensure that the new system can integrate into the existing databases seamlessly, or that the existing databases are replicated in such a way as to ensure that there is the minimum system downtime.

8.5.3. Possible Points of Failure

The failure that springs to mind immediately is that of incompatibility with existing systems, but as this issue should have been identified far sooner in the development life cycle, it is not a point of failure which belongs in this phase. Assuming that the software has been properly planned and constructed, there are two chief concerns. The first is that data is lost or corrupted due to a poorly planned installation. The second is that systems are unavailable for an unduly long period due again to the planning or execution of the installation.

8.5.4. Damages

Loss of data can have huge financial implications, whether it be as a result of the expense of reconstituting last records, statutory liability for not maintaining proper records, or the effects on business of the loss of for example marketing information. If the user is in the business of maintaining third party records, such a failure could be calamitous.

Should a faulty installation procedure cause lengthy downtime, this will have productivity implications, especially if the system involved is central to the operations of the user concerned. The issue of downtime becomes even more problematic in the case of a critical system which is used to provide a service to third parties. In that case, the user could attract liability from third parties as a result of the unavailability of the service, whether in contract or (less likely) delict.

8.5.5. Establishing Liability

There will almost certainly be a contractual relationship between any party undertaking the installation and the user. As such, any damages resulting from a failure in installation will be covered in a claim for breach of contract. As set out above, it is perfectly possible that extended downtime will lead to damage to third parties. However, any action arising from this will be brought not against the party undertaking the installation, but the user. Under the circumstances of a third party suffering damage, such a third party will sue the user, which could in turn seek compensation in a claim for breach of contract against the installer. An action by a third party against the installer is unlikely to succeed because it would be difficult to establish wrongfulness by the installer relative to the third party.

In the unlikely event of a delictual claim, the Plaintiff will have to show that the installation was negligently undertaken. The critical aspect of any installation involving a complex system is planning, especially where this system has to be installed in such a way as to prevent interference with existing systems, and to allow for minimum disruption if the installation is of an upgrade. Thus in any action if it were to be shown that the installer had not made proper plans for the installation process where damage has been established, negligence should be clear.

Even if a plan exists, it is quite possible that the plan is not executed properly, whether because the installer failed to hire personnel who were qualified to do the job, or having such personnel, failed to train them or brief them correctly on the details of the plan.

It could of course occur that the party accepting installation is found to be negligent itself in not disclosing properly the details of systems that a new installation will be replacing or the details of systems running on the machines that the new software is

to be installed on. For example, if a company has critical software running on several servers, one of which is the machine on which the new software is to be installed, and neglects to advise the installer that the machine in question cannot under any circumstances be rebooted, if the installer does in fact reboot this machine during the installation process, the installer can clearly not be held liable for any damages that result. Such contributory negligence would then prejudice any claim against the installer.

8.6. Operation

The last three phases, operation, maintenance and retirement will be dealt with in short order. As will be seen, while they take up a significant portion of the software life cycle, their essentials are dealt with elsewhere.

Operation of software is often left to the user, especially in the case of off-the-shelf software. It often occurs that a training agreement is entered into with the developer (or a consultant), in terms of which the developer agrees to train the employees of the user in the operation of the software. This is especially true in the case of bespoke software, but is also done in the case of complex generic software.

Should damage arise as a result of an error in operating the software, the user will almost certainly have no claim. In the case that a training agreement was entered into, and poor training led to damage, any liability will be dealt with in terms of the contract between the parties. A delictual action for negligent misstatement would be precluded by the principles set out in the *Lillicrap* case.[102]

8.7. Maintenance

As discussed above, maintenance can easily be the most expensive aspect of the software life cycle. While, again, the user will be responsible for most maintenance of off-the-shelf software, in some cases maintenance agreements may be entered into either with the software developer or with a consultant who possesses the relevant skills. In either case, I would submit that any damages would be handled along the lines of the principles set out in the discussion of installation: if a third party suffered damage, even if maintenance of the system was undertaken by third party, its claim

---

[102] *supra*

would be against the user, who would then have a claim in contract against the person undertaking maintenance.

The reason why I give maintenance short shrift is that in practical terms the phase simply loops back on the previous phases of the life cycle. If the system needs to be altered or upgraded, any change would itself go through the phases of requirements analysis, design, construction and testing, and to restate the principles here would be a duplication.

There are however aspects to maintenance which are unique. While maintenance may involve making changes to the system, there may be a duty upon someone who was undertaken to maintain the system to take a proactive approach to factors which may challenge the viability of the system. For example, there would probably be a duty to ensure that the system was properly protected against security threats, for example new computer viruses which appear from time to time. Moreover, if the environment within which the software operates requires a change to its functionality, there could be a duty upon the party undertaking maintenance to initiate the process of upgrading the system; failure to do so may amount to negligence. For example, in the case of an accounting package that had as one of its core requirements compliance with Generally Accepted Accounting Practice (GAAP), there would be a duty upon the maintainer to initiate an upgrade in the case of any material change to this standard. Here again, however, the action is likely to lie in contract.

8.8. Retirement

Retirement in and of itself is unlikely to give rise to liability: obviously if a user retires a critical system without making provision for its replacement, that user is itself liable for whatever damages this may cause to itself or third parties. Moreover, problems at retirement typically do not arise from the retirement itself, but rather the software which replaces it. As this has been dealt with adequately above, it warrants no further comment here.

9. Conclusion

I have now traversed the entire software lifecycle and hopefully in doing so shed light on the legal implications of the actions performed in each phase. While I have treated them separately, it must be borne in mind that in a real-world project the phases will

be inextricably linked. They will often feed back on each other, for example where an error is found and operation reverts to the construction phase to amend the code appropriately. The phases will also not necessarily be carried out exclusively - while the above fault is being fixed, the testing phase may continue.

Where an action is brought as a result of defective software it may in practical terms be difficult to determine where in the lifecycle a fault had its origin - an evidentiary problem that is beyond the scope of this paper. Hopefully however my description of the stages will go some way to helping lawyers understand to which stage to look for fault.

It should be clear that in software development, perhaps more than in any other discipline of engineering, it is the testing phase that is of most importance in preventing faults. It is the phase that is the last and best defence against legally significant failures. For this reason I have paid more attention to it than to the other phases, and I expect (and recommend) that this be the case when a court has to make a determination of fault in a matter involving a complex software project.

Software development is very interesting from legal perspective. It assumes a faulty product in the way that no other field does. This poses a challenge to existing legal principle, especially relating to wrongfulness and negligence. This is aggravated by the fact that the field is a new, complex and rapidly evolving one. As yet there is little case law to guide the lawyer, but I trust that this will change as software becomes ever more central to our lives. The principles governing liability when software causes damage will need to be addressed and I hope that in this paper I have made a contribution to that process.

<u>The Tester's Drinking Song</u>
99 little bugs in the code,
99 bugs in the code,
Fix one bug, compile it again,
101 little bugs in the code,
101 little bugs in the code...
(Repeat - and keep drinking - until BUGS = 0)
Anonymous

----------oooOooo----------

10. References

Books

Bainbridge D, "Introduction to Computer Law" (London: Longman 2000)

Bass L, "Products Liability: Design and Manufacturing Defects" 2nd ed (St Paul MN: West Publishing Group 2001)

Ezzell B, "MCSD: Analyzing Requirements and Defining Solution Architectures" (San Francisco: Sybex, 1999)

Hoglund G and McGraw G, "Exploiting Software" (Boston: Addison-Wesley 2004)

Huchinson D *et al* (eds), "Wille's Principles of South African Law" 8th ed (Cape Town: Juta & Co, Ltd 1991)

Jorgensen P C, "Software Testing" 2nd ed (Boca Raton, Florida: CRC Press 2002)

Kit E, "Software Testing in the Real World" (Boston: Addison-Wesley 1995)

Michalson, L "South Africa & the Millennium Time Bomb: a Guide to the Legal Issues" (Cape Town: Francolin, 1998)

Neethling J *et al*, "Law of Delict" 3rd ed (Durban: Butterworths 1999)

Sommerville I, "Software Engineering" 6th ed (Boston: Addison-Wesley 2001)

Sterling B, "The Hacker Crackdown: Law and Disorder on the Electronic Frontier" (New York: Bantam 1992) available at http://www.chriswaltrip.com/sterling/crack1j.html (accessed 27/10/2005)

Van Der Merwe S *et* al, "Contract: General Principles" (Cape Town: Juta & Co, Ltd 1993)

Walters E G "The Essential Guide to Computing" (New Jersey: Prentice Hall, 2001)

Watkins J, "Testing IT" (Cambridge: Cambridge University Press 2001)

Articles

Adams J, "Quality and Title Warranties in Transfers of Computer Software" (1995) 1 *International Trade & Business Law Journal* 35

Alces P A,"W(h)ither Warranty: The B(l)oom of Products Liability Theory in Cases of Deficient Software Design" (Jan 1999) 87:1 *California Law Review* 271

Alheit K, "Contractual liability arising from the use of computer software: notes on the different positions of parties in Anglo-American law and South African law" (2000) 33:1 *Comparative and International Law Journal of Southern Africa* 26

Alheit K, "The applicability of the EU Product Liability Directive to software" (2001) 34:2 *Comparative and International Law Journal of Southern Africa* 188

Bohan TL, "The Performance Audit: Minimising Software Liability (Part II)" 1988) 29 *IDEA: The Journal of Law and Technology* 135

Burchell J, "The Odyssey of Pure Economic Loss" (2000) A10 *Acta Juridic*a 99

Gemignani M C, "Potential Liability for Use of Expert Systems" (1988) 29 *IDEA: The Journal of Law and Technology* 120

Jamieson M, "Liability for defective software" (2001) May *The Journal* 26

Janks D, "Y2K compliance: Lessons from the past – a tale of two horses, the Noah principle and an ostrich" (Aug 1998) 367 *De Rebus* 69

Kaner C, "Software negligence and testing coverage" (updated at http://www.kaner.com) (1995) 2:2 *Software QA Quarterly* 18

Kathrein R R, "Class Actions in Year 2000 Defective Software and Hardware Litigation" (1999) 18:3 *The Review of Litigation* 487

Miyaki P T, "Computer Software Defects: Should Computer Software Manufacturers Be Held Strictly Liable for Computer Software Defects" (1992) 8 *Santa Clara Computer & High Technology Law Journal* 121

National Institute of Standards and Technology "The Economic Impacts of Inadequate Infrastructure for Software Testing" (2002) *Planning Report* 02-3

Owen D G, Madden M S, and Davis M J, "Publications and Products Liability" (November 2000) 141 *Products Liability Advisory*

Perlman D T, "Who pays the price of computer software failure?" (1998) 24 *Rutgers Computer & Tech. L.J.* 383

Preston E & Lofton J, "Computer Security Publications: Information Economics, Shifting Liability and the First Amendment" (2002-3) 24 *Whittier Law Review* 71

Prince J, "Negligence: Liability for Defective Software" (1980) 33 *Oklahoma Law Review* 848

Reece L H III, "Legal theories in actions against software developers for defective software" (1988) 29 *IDEA: The Journal of Law and Technology* 113

Rowland D, "Liability for Defective Software" (1991) 78 *Cambrian Law Review* 22

Weinstein A S, "The Performance Audit: Minimising Software Liability (Part I)" 1988) 29 *IDEA: The Journal of Law and Technology* 127

Cases

Coastal State Trading, Inc v Shell Pipeline Corporation 573 F.Supp. 1415 (S.D. Tex. 1983), aff'd 788 S.W.2d 837 (1990)

Co-operative Group Ltd (formerly Co-operative Wholesale Society Ltd) v International Computers Ltd [2003] EWHC 1 (TCC)

Coronation Brick (Pty) limited v Stratford Construction co (Pty) Ltd 1982 4 SA 371 (D)

GEC Marconi Systems Pty Ltd (t/as EASAMS Australia) v BHP Information Technology Pty Ltd NG733 of 1997, 2003 FCA 50

Kruger v Coetzee 1966 2 SA 428 (A)

Lillicrap, Wassenaar and Partners v Pilkington Brothers (SA) (Pty) Ltd 1985 1 SA 475 (A)

Memorex Telex Pty Ltd v National Databank Ltd [2002] NSWSC 1111 4654/98 SUPREME COURT OF NEW SOUTH WALES EQUITY DIVISION 2002 NSW LEXIS 802; BC200207207 (unreported)

Randaree and Others NNO v W H Dixon and Associates and Another 1983 (2) SA 1 (A)

Shell and BP SA Petroleum Refineries (Pty) Ltd v Osborne Panama SA 1980 3 SA 653 (D), confirmed on appeal 1982 (4) SA 890 (A)

Standard Chartered Bank of Canada v Nedperm Bank Ltd 1994 (4) SA 747 (A)

Van Wyk v Lewis 1924 AD 438

Standards

IEEE Std 610.12-1990 – "IEEE Standard Glossary of Software Engineering Terminology"

ISO/CD 10303-226

ISO/IEC 12119:1994, "Information technology – Software packages – Quality requirements and testing"

----------oooOooo----------

Liability for Defective Software in South Africa

By: Andrew Marshall

LLM Candidate

University of Cape Town

Student Number MRSAND009

Dated: 31 October 2005

I confirm that I am the author of this work.

Andrew Marshall