Towards a consistent legal framework for understanding software systems behaviour

a thesis in support of the degree of LL.M. by dissertation

at the University of Strathclyde Centre for Law, Computers and Technology.

June 30, 1999

Les Hatton, Ph.D., C.Eng., F.B.C.S.

Declaration of author's rights:

Dedication

This thesis is dedicated to my long-suffering family, Gillian, Leo, Felix and Isabelle who yet again have had to watch me tapping away at a machine for hours on end. I hope it's all worth it in the long run.

I would also like to thank Professor Ian Lloyd for supervising this work and providing help, encouragement and above all understanding when I got stuck.

Table of Contents

Author's Preface

The relationship between software and the law is currently a rather uneasy one. For example, the essentially intangible nature of software has led to an understandable reluctance by the courts to attempt to categorise it as either goods or as a service, and yet the explosive growth of software with its frailties as well as benefits in consumer society suggests that this may only be putting off the inevitable. This thesis was written from the point of view of a computer scientist increasingly concerned by this situation in the hope that it will help provide a reasonable bridge between the legal perspective and the software engineering perspective. What indeed can the reasonable man or woman expect of software ?

The strategy used is to describe software engineering from a computer scientist's point of view in order to explain just how and why software engineering is of considerably lower quality than other more mature engineering disciplines. This is followed by a detailed discussion of various aspects of law as might occur in a legal text book but with discussion of implications for software engineering inserted where relevant. Following on from this, certain aspects of particular relevance are developed in detail.

The thesis is constructed according to the following plan:-

Chapter 1:

> In this chapter, software engineering will be described from the perspective of the computer scientist. Software engineering is, by the standards of traditional engineering such as civil and mechanical engineering, a very imperfect discipline. In other words not only is the state of the art depressed with respect to other more mature engineering disciplines, but there is also a much greater natural variation in offered products. It is very important for the legal practitioner to understand these limitations and variations to preserve

a reasonable viewpoint, otherwise software engineering may suffer and the many benefits which accrue along with its problems would be lost. It is in nobody's interests for poor engineering to evade liability but the current state of the art must always be in contemplation. It simply isn't very good yet and it will not be for some considerable time.

Chapter 2:

This chapter presents a conventional current legal view of a number of areas including the position of the consumer and liability in software engineering such as can be read in a number of sources. It is somewhat speculative because there is very little case law to act as guidance yet, so it simply describes the issues and then adds comments based on a computer science perspective, addressing such issues as reasonableness and consistency. It should be contrasted carefully with Chapter 1.

Chapter 3:

This chapter compares in some detail the judgements in two recent influential cases, that of *Saphena Computing v. Allied Collection Agencies* and *St. Albans v. ICL*. Since these represent arguably the only significant civil case law presently available, these judgements are worthy of a detailed investigation. This chapter compares their consistency and their relevance to future litigation.

Chapter 4:

In this chapter, a number of interesting aspects will be pursued in much greater detail, for example, the perennial Software as Goods or Service argument will be studied. The underlying model for trying to resolve some of these difficult issues will be that it is important for the law to be consistent with its interpretation of the nature of software

across all points of contact between the two. Here then, case law in copyright, criminal and other aspects of the law will be collected and analysed to look for points of both consistency and inconsistency. As a result, this chapter can be viewed as source material to help guide legal practitioners in how future legislation might develop for software, in order to preserve a consistent interpretation.

In this chapter, the point will also be made that the contract is of unusual importance in software engineering. In normal life, contracts are written not to be used. In other words, the contract clearly defines the obligations of each of the parties in order that conflict can be avoided. They are written in the expectation that one party is very likely to be able to supply what the other party is contracting for. However, as described in chapter 1, it is *commonplace* for software projects to fail to some degree. Given that the normal remedy of rescission is unfortunately of little value in such cases, software contracts should be written from the point of view that the supplier is *very likely to encounter difficulties* in supplying and more constructive avenues should be described in the contract so that both parties get something more useful than mere rescission in the likely event of a partial failure. Total failure, (which is also depressingly likely in software engineering), of course leaves little option for a constructive solution for both parties.

Chapter 5:

This short chapter brings the thesis to a conclusion suggesting areas of further work.

The thesis is supplemented with an Appendix, which takes a quick look at recent developments in the US pertaining to the Universal Commercial Code (UCC), which in essence provides a set of implied terms to standard contracts and contrasts the UK position.

Finally, the thesis is written in the third person but the footnotes represent personal additional commentary and are therefore written in the first person to reflect my 25 years experience as a professional computer scientist. This experience of course weakens my legal viewpoint (in spite of reading furiously) but it greatly helped to identify a number of legal inconsistencies which are very likely to lead to conflict as time passes.

Les Hatton
December, 1998
lesh@oakcomp.co.uk

Chapter 1: Software engineering from a computer science perspective

It is sometimes difficult even for professional engineers in other disciplines to understand the vicissitudes of software 'engineering' and so in this chapter, the author will attempt to explain its history, its successes and most importantly its failures, in the hope that the legally-trained reader will get some perspective on this most difficult and still immature of engineering areas. Key points will be highlighted by surrounding in a box as:-

This is a key point.

The author will make the point that any software system other than the most trivial is overwhelmingly likely to contain faults. Many of these faults could have been avoided but were not because of generally poor practices. However, a significant number of software faults are unavoidable because of the intellectual complexity inherent in software development. These faults will fail in generally unpredictable ways throughout the life-time of the software. Attempts to correct them will frequently fail, in turn injecting new faults and it is very likely that any software system will exhibit unexpected behaviour throughout its life-time. Amongst all this bad news is the fact that software controlled systems are not only considered beneficial, but all of society absolutely depends on them now and has for some 10-15 years. In other words, they are here, they are annoyingly fragile and they can't be avoided. The law must find an appropriate balance between punishing developments which are demonstrably sub-standard, (and there are plenty of these), and not punishing developments which are highly beneficial but exhibit unavoidable failure, otherwise there will be a strong danger of throwing out the baby with the bathwater which is to nobody's advantage. It is hoped that this chapter makes the distinction clear enough to guide a legally-trained reader.

Digital convergence

Before embarking on a short history of software engineering, it is perhaps appropriate to start with a discussion of digital convergence. In essence this describes the process whereby a particular form of expression, artistic or otherwise becomes digitally represented or *digitised* (i.e. represented as a sequence of 0s and 1s). This process is spreading across many previously independent areas blurring the distinction between them. This has a particular impact on the law.

The law has become accustomed to dealing with artistic works over the last 100 years or so. In particular, the law of copyright embraces an artistic work although quite rightly without seeking too seriously to define one, as for example, under what circumstances a pile of bricks is considered to be an artistic work[1]. Artistic works include *inter alia*, pictures and pieces of music *in whatever format*. This is particularly relevant today as digital technology matures to the point where what was previously the province of analogue media, (i.e. continuous media), is now routinely expressed digitally.

Digital music

The human ear has a maximum dynamic range of between around 30 Hz. and 20,000 Hz., (1 Hz. = 1 Hertz = 1 cycle per second). Adults have a rather smaller range of between around 30-40 Hz. and around 10-12,000 Hz. Even in nature, sounds have a natural upper limit of frequencies set by the physical mechanism by which they are produced. A plucked string, oscillating vocal chords, an earthquake are all governed by restrictions placed by the laws of physics. Suppose a typical maximum frequency is F. Now, there is a very famous theorem in signal processing known as the Whittaker-Nyquist-Shannon theorem, (see for example, [1]), which states that if a signal of maximum frequency F continuously varying in time, (i.e. with a value at every point in time), is represented by its values at discrete

[1] It appears to be an artistic work when it is in the Tate, which begs the question as to whether the walls holding the roof up are also artistic works.

points of time separated by 1/(2F) seconds, (a process known as *sampling* or *analogue-to-digital (A/D) conversion*), the signal is **exactly** reproduced.

As an example, if we sample a piece of music, (which is effectively limited to sounds no higher than 20,000 Hz. because we couldn't hear them anyway), and we sample it (i.e. measure its value digitally) every 1/(2x20,000) second, or every 0.00025 second, the result is **indistinguishable** from the original analogue (continuously varying in time) signal. In other words if we converted the digital sample signal back into a continuously varying signal in time, (a process known as *digital to analogue (D/A) conversion*), the result would be absolutely indistinguishable from the original. All of the information present in the original signal before sampling would be present in the reconstructed signal.

This process has reached its current zenith not only with the de facto standard digital recording of music, but also in the production of music with MIDI, (Musical Instrument Digital Interface). In MIDI, any stream of music is converted in real-time to a sequence of 1s and 0s. For example, when a musician plays a MIDI keyboard or a MIDI guitar, the signal from the instrument is not a continuously varying voltage level as with older technology, but a series of 1s and 0s in a specific format known as MIDI.

In essence then, we have a found a non-unique (there are various ways of doing this) representation of 1s and 0s which are precisely the same as the original piece of music.

Digital Art

Although it took a little longer than in the case of music, precisely the same has happened with graphic works of art such as pictures. The human eye has a spatial bandwidth limitation just as the ear has a temporal bandwidth limitation. In other words, the eye cannot see finer detail than a certain point, (depending on light and colour, the fine detail of a digital image of around 200 dots per cm. cannot be seen) just as the ear cannot hear finer

temporal detail than a certain point. Precisely the same Whittaker-Nyquist-Shannon theorem holds and it is perfectly possible to produce a digital representation of a great work of art, which, *apart from the medium* on which the original work of art is represented, is essentially indistinguishable from the original, except with a microscope. If the image is subjected to D/A conversion back to an analogue image, it would be absolutely indistinguishable.

This has a number of rather disturbing corollaries. *For example, photographs should no longer be admissible as evidence*. The technology has existed for some time now to scan an image from a photograph, (the scanning is typically done at 500-1000 dots per cm. to guarantee precise reproduction), modify the image using a number of digital image processing techniques such as spatial deconvolution, dithering, anti-aliassing and so on, and replay the image back onto photographic film. The resulting film should not be taken as evidence but would appear as a perfectly viable photograph and it is very difficult to determine whether it has been tampered with. (This process has featured in a number of films, for example, the film based on the Michael Crichton book, *Rising Sun*, although it may well be that the public do not realise that it can be done. A more recent example is the film *Titanic*, wherein many of the scenes associated with the ship itself are built up from a mixture of analogue clips and digital simulations, merged together into a composite which is very impressive indeed. Only the closest inspection can reveal the slightly awkward nature of the moving figures in the computer graphics. In another 5 years or so, this distinction will effectively disappear also. It is quite likely that within 10 years, a new Humphrey Bogart film could appear with an entirely digital Bogart *persona*). This should prove intriguing to copyright lawyers.

As a final comment on this area, it should also be pointed out that there are many different ways of representing the same picture in digital form. These are called *digital formats* and some common examples include

TIFF, GIF, JPEG, MPEG, Base64, PICT and so on. Each of these is different and yet the same picture would be produced when each of these formats is correctly *rendered* by an appropriate display program.

Digital speech

Precisely the same situation occurs with speech. The technology for converting speech into 1s and 0s has existed for some time. Its decomposition into *phonemes* permits another digital representation of the same thing.

Signal encryption

This represents a slightly different situation. The need to make 'plain text' transmitted by a communications line secure in some sense has been around for a long time. In W.W.II, the Enigma encryption machine converted plain text into scrambled plain text on a character by character basis. The information in both sets was precisely the same but the latter was entirely inaccessible without a correctly configured Enigma machine or its equivalent. Nowadays, the vast bulk of transmitted text is already encoded in 1s and 0s usually using the standard ASCII code, (for example in this code, the space character is represented by the binary code '00100000'). For encryption, this digital representation is converted into another digital representation using some kind of encryption technology such as the RSA public-key system.

... and software

What has this all got to do with software ? The most important issue from the point of view of software is that the law has evolved at least partially to tackle some of the issues posed by a digital representation of an artistic work without getting too bogged down in the physical medium. However, in each of the above cases, the resulting digital representation is simply a stream of 1s and 0s. *Without knowing how the stream arose, it is impossible to divine what they correspond to*. Now a piece of software when viewed on disc or down a communications medium is simply another stream of 1s and 0s. Not only that, *it is perfectly possible to define*

transformations which can map between different representations, for example,

> Any computer program can be converted into a MIDI sequence playable as a piece of computer music, (this would however rely on the court's known

reluctance to decline whether something was musical[2]). As a matter of interest, to complicate things further, the same is true for any digital representation of a picture. In other words, a picture could be played. By the converse, a piece of music could be displayed as a picture, and it is perfectly possible, (although unlikely), that the digital manifestation of a piece of music or a picture when fed into the control system of a computer could cause the computer to misbehave dangerously. For example, an error in a MIDI sequencing program could cause an anomalously loud note to be sent at a high frequency causing hearing damage to the wearer of a set of headphones. The digital convergence is thus complete.

This fundamental convergence between entirely different entities needs to be borne in mind whenever the relationship of digital information with the law is being considered and a consistent viewpoint is being sought. This theme will be returned to later for example, when the issues of copyright and 'goods v, services' are discussed.

The nature of software

The essential nature of software then is that it cannot be weighed, touched or easily quantified. Like literary and musical works, it is a product of the human intellect. Unlike literary and musical works, it has no direct effect on humans, rather it causes a programmable computer system to behave in some way and this may then have a direct effect on humans.

Software evolves like a mathematical proof although rarely with such rigour. It has its own family of notations, known collectively as *source code*, which the computer programmer writes directly as prose. Each notation is known as a programming language, of which there are hundreds. Some of the more common ones include Ada, used in many defence and aerospace software systems, C and C++, used in most consumer PC software and also in embedded control systems for devices as disparate as medical scanners

[2] This would save the embarrassment of deciding whether a vocal work based on the Major 9th lasting for some 75 minutes by Karl-Heinz Stockhausen was musical or not. (My own opinion is that it is as musical a work as a pile of bricks in the Tate is an artistic work).

and cars, Fortran, used for example in nuclear engineering and Java used in Internet programming. Chronologically preceding these so-called *high-level* or expressive languages, was assembly language, a so-called *low-level language*. In essence, the lower the level, the more source code is required to achieve a typical action. Each programming language is supposed to represent a complete, consistent logical framework in which a set of desired actions, or *algorithm*, can be expressed. However, even in this apparently simple goal, computer scientists have failed and no programming language has ever been free of ambiguity in some form or other, a point which will be discussed later. To understand the implications of ambiguity in written prose, natural languages abound with it. Consider for example the various meanings of “Fruit flies like a banana.”.

Source code is measured in lines of code. An example is given in Figure 1.1 which shows 14 lines of a network switching telephone system. The overall system comprises some 3 million such lines. A simple inadvertent transposition of just two letters can correspond to a catastrophic change in behaviour of the program. For example, the underlined line of code should not be there, it was introduced inadvertently in a simple modification to the original source code in January 1990. The effect of this inserted line of code is that when a sending switch is out of service and if there is a residual message stored, the processing of the incoming message is incorrectly skipped. This leads to the database being left in an inconsistent state. This caused a propagation of errors which were not handled, swamping first the originating network and then all the connected networks. Had the upgraded software only been introduced in one network, the older version would have handled the problem correctly in surrounding networks, but the corporate faith in the update was such that all caution was thrown to the winds and all networks were upgraded simultaneously.

Shortly afterwards and within around 15 minutes of this ‘fix’ being inserted, *this led to the loss of all long-distance telephone lines in the United States for several hours*, the cost of which was estimated by some

sources as being around $1.1 billion. Even relatively simple systems would today have up to around 100,000 lines of source code, whereas the most complex systems might have 10 million lines of source code or more, (for example the European Fighter Aircraft onboard computer systems).

It is useful for the practising lawyer to contemplate an industry in which a single inadvertently inserted line in the specification of behaviour of a system comprising several million such lines can have such an enormous impact, given that legal contracts for example frequently have simple typographical mistakes. Roughly speaking, one clause in a legal contract contains the same semantic content as one line of software and a typical legal contract might be 10,000-100,000 times simpler, containing only 50-100 clauses and even then could be considered relatively difficult to understand. To expand the analogy further, coupling between clauses in legal contracts is known to make things much more complex. For example, it is common to insert a clause to the effect that 'this contract comprises the whole of the agreement between the parties privy to this contract'. Of course, this cannot be true because all contracts are subject to implied (i.e. additional) terms for reasonableness as defined in the Unfair Contract Terms Act 1977. In other words, every legal contract inherits properties from at least one statutory document. Not only this, but in the case of dispute, a court may imply terms into a contract based on two well-known bases:- a) to make the contract have business efficacy or b) if the parties had been asked by an officious bystander what would happen in a given event, they would have answered in the required sense "of course"[3]. The complexity invoked by this leads to many extremely complex legal arguments and yet is still trivial in comparison to the word-processor which is being used to write this thesis which comprises several hundreds of thousands of lines of source code closely interacting with an operating system involving some millions of lines. Both have exhibited numerous

[3] Quoted directly from the words of Staughton LJ in *Saphena Computing Ltd. v. Allied Collection Agencies Ltd*. (Court of Appeal), (1989).

failures in the time the author has been using them. Given the stupendous complexity of such a system, it is hardly surprising.

```
...
switch(message)
{
case INCOMING_MESSAGE:
      if (sending_switch == OUT_OF_SERVICE)
      {
            if (ring_write_buffer == EMPTY)
                  send_in_service_to_smm();
            else
                  break;
      }
      process_incoming_message();   /* skipped */
      break;
      ...
}
do_optional_database_work();
...
```

Figure 1.1 A small fragment of code responsible for one of the most expensive software failures in history so far. The underlined line should not be there - it was introduced inadvertently during a software upgrade. The resulting catastrophic effect on the system is described in the text.

The programmer's source code, also known as *human-readable*, (a misnomer as humans can read and understand any of the codes used in a machine, although some with greater facility than others), cannot typically be understood by a computer however. For this to occur, the programmer's source code is translated into object code, (also known as machine-readable), using another piece of software known as a *translator*. The most common form of translators are known as compilers and interpreters. Compilers and interpreters are themselves immensely complicated pieces of software usually requiring hundreds of thousands of lines of source code

to delineate all their actions. So the input to the translator is source code and the output is object code. Object code has no natural line-based structure and is therefore measured in units known as bytes. A byte is 8 bits, each of which can either be true or false, (represented by the digital values 1 and 0). Today, even a simple installation of an office automation system with a word processor, presentation package and a spreadsheet can easily consume 100 megabytes or 10^8 bytes.

In essence, and without loss of generality, this is stored in magnetic form on some medium, for example, a floppy disc, a hard disc, a tape or cartridge, a CD-ROM and numerous other media. In this form, the software now appears to be intangible, but is no more intangible than a piece of music for example, with which most people are familiar. The magnetic medium is then inserted into a computer just as a music CD is inserted into a CD player, and the internal electronic mechanisms of the computer cause a series of actions to take place; devices might be controlled, a computer screen could be filled with graphics and so on. We say the software is now *executing*. The act of executing a piece of software actually copies it from its original medium to a temporary storage medium inside the computer known as RAM, (Random Access Memory). This act of copying has attracted much interest from specialists in copyright. The act of execution of a piece of software *always* involves some form of copying. To complicate things further, copying also takes place during a process known as *defragmentation* whereby the computer's operating system moves things around its discs for housekeeping purposes such as efficiency.

The analogy with music is very close and will be referred to later. Of course when we put a music CD in a CD player, the internal electronic mechanisms of the CD player *execute* it and the result is music. In this case, we say the music is *playing*. The analogy with computer software is even closer because most computer CD storage devices, (CD-ROM), will also play musical CDs, so the physical representation, especially with digital music, is essentially identical. As we will see later, the close

relationship between music and software in many aspects will be enlightening from a legal point of view. The notion of copying even appears in musical CDs as it is the basis of *anti-shock technology*. A CD resists shock by copying the next few seconds of music into RAM built into the CD player. When the CD player is jarred sufficiently to disturb the laser, the CD automatically switches to its RAM copy whilst the laser settles down. Only if the period of settling down exceeds the capacity of the RAM is any interruption of music observed This copying is closely analogous and physically identical to the copying which takes place in a computer prior to the execution of a piece of object code.

The phases of software development

A typical software system will ideally go through the following recognisable phases:

a) Requirements

b) Design / Specification

c) Implementation

d) Unit testing

e) System and integration testing

f) Release

g) Maintenance

and increasingly likely as the years go by,

h) Litigation

Entire phases can be missed out and in other cases, sub-phases may be inserted, but this sequence captures the ideal model without any loss of generality. The steps a) - f) are known as the *development phase*, which is everything which takes place before the software system is formally

given or *released* to the intended customer for the first time. It is common for computer scientists to organise these phases using various models such as the waterfall and V- models, [2], but this is frequently window dressing given the parlous state of much software engineering. The V-model is instructive in how it encourages various kinds of verification at different stages in the life-cycle as shown in Figure 1.2

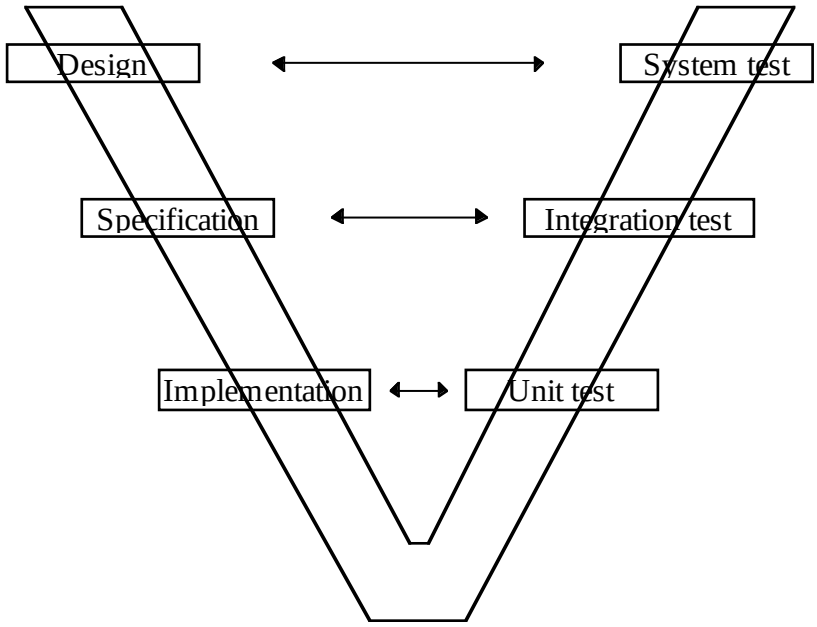

Figure 1.2 The V-model of software engineering. Design activities on the left hand side of the V are associated with corresponding test activities on the right hand side. The model is idealised in the sense that software development is rarely this tidy.

Following the development phase, the software system enters what is euphemistically known as its *maintenance phase*. Maintenance comprises essentially three components:-

- Corrective maintenance

  This relates to the correction of *defects* or departures between actual and expected behaviour. Actual behaviour is also known as its *functionality* and does not normally include *performance*, or the speed at which the actual behaviour is achieved. Note that expected behaviour is defined by the requirements. If there are no requirements, there are no defects.

- Adaptive maintenance

  This relates to modifications to the software which do not affect its functionality, but would affect other attributes such as its performance.

- Perfective maintenance

  This aspect relates to extending or otherwise modifying the original actual behaviour to model the intended user's evolving notion of what the software system should be able to achieve. This occurs either because the very imperfect initial requirements stage fails to clarify all the intended goals or because the intended user changes his or her mind about what the software should do based on actual experience using it. In other words, this latter possibility could not generally have been foreseen at the requirements stage.

Together the development phase and the ensuing maintenance phase make up the *software life-cycle*, the life span between the original idea and the point at which the software is no longer used.

The distribution of cost across the development stage and the three components of maintenance is somewhat surprising as indicated by Figure 1.3.

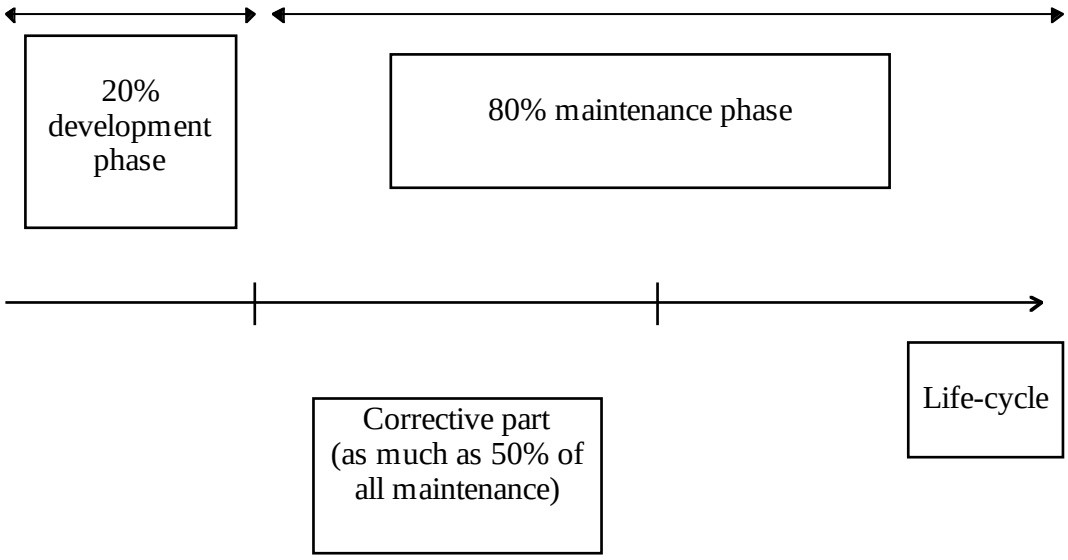

Figure 1.3. This diagram combines together two widely accepted maintenance statistics, [3], [4], leading to the conclusion that corrective activities may consume as much as twice as much resource as the original development.

These results are summarised graphically in Figure 1.4.

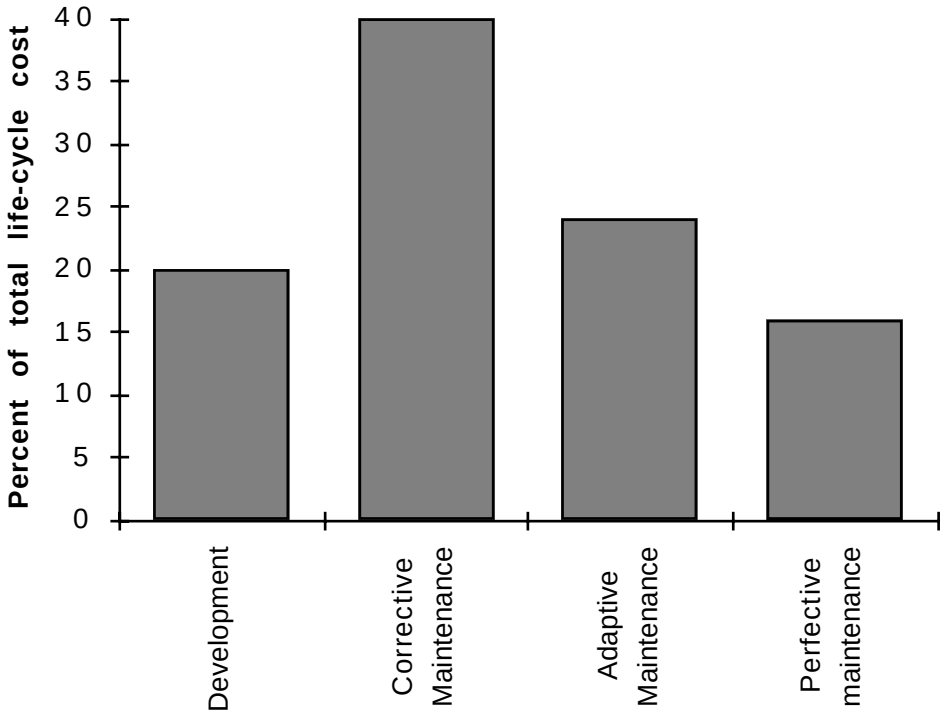

Figure 1.4: Shows the approximation distribution of costs over the entire software life-cycle.

In other words, for every pound spent on development, as much as two pounds are spent in fixing defects in the software, *after it has been given to the user*. This may indeed come as a surprise to the typical consumer as it appears to represent a level of unreliability which would be considered unacceptable in any other sphere. However, in spite of considerable dissatisfaction, such as that experienced with PC software, (see for example, [5]), nothing has so far changed, although it seems clear that the consumer would not accept levels of reliability typical of PC products in a washing machine for example, as evidenced by Table 1.

Operating System	Period of measurement	Hours used	Total defects	Total reboots	Defect frequency
Windows'95/AST P90 Pentium	April - June 1996	23	31	5	1 per 0.7 hours
Macintosh 7100/66 OS 7.5.1	July 1995 - June 1996	467	146	68	1 per 3.2 hours
Unix Sparc, HP various OS	Nov 1985 - June 1996	> 5000	< 5	1	< 1 per 1000 hours
Linux, Slackware on Pentium P90	April 1995 - June 1996	26	0	0	Not yet defined.

Table 1: This table indicates defect rates of various popular operating systems in the author's *personal* experience spread over the last ten years. The author keeps an Excel spreadsheet record of each operating system and log each defect as and when it occurs when the bugs in Excel itself permit.

Software reliability

It is worthwhile exploring why PCs appear to have worse levels of reliability than other systems. In principle, it is very simple. Every system has a trade-off point between reliability and availability as exemplified by Figure 1.5. Here, the Rate of Occurrence of Failure (ROCOF) is plotted against usage time, i.e. testing time, during development. As the system fails, faults are

identified and corrected so that the system ROCOF hopefully falls[4]. Systems never attain perfection with zero ROCOF so that they have to be released at some stage at some level of reliability, whatever their intended use. The point is that the release time is an economic decision balancing the maintenance and possibly legal costs of releasing a system with defects against the economic benefits of beating the competition to market. For a safety-related system, these would hopefully work in favour of reliability giving a release point considerably to the right on the Usage time axis. However, reliability is not a selling point for PC software. Users seem mostly happy to put up with it failing frequently provided there are lots of features to play with. Consequently, it is economically sensible to release software early on the Usage time axis to maximise the marketing opportunity. The massive success of companies like Microsoft bears mute testimony to this strategy. PC software will only get better if reliability becomes a user requirement.

[4] Regrettably, it doesn't always.

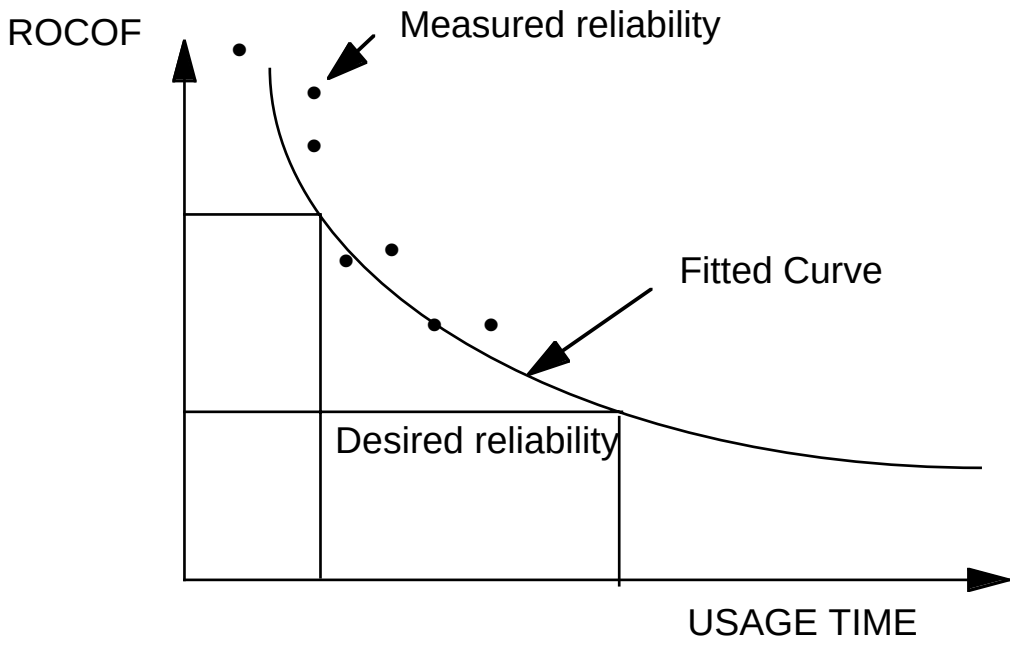

Figure 1.5: Reliability improvement with time of a well-behaved product. As time goes by, corrective maintenance feedback improves the reliability. The point at which the product achieves sufficient reliability for its intended purpose depends entirely on what that purpose is. There is a trade-off between earlier availability and the risk of higher unreliability.

Regrettably this simple economic truth is not well understood by system integrators and PC levels of reliability are already found in some consumer systems by virtue of the fact that these systems actually used PCs to control other systems. For example, on a flight on Singapore Airlines two years ago, the inflight entertainment system crashed 3 times in a 14 hour flight. When the cabin manager was questioned, he stated that it was controlled by a PC and that this was a frequent occurrence. Designing a system to stay functional for 14 hours around a component with a mean time between failures of a couple of hours is clearly a highly dubious methodology although the attractiveness of off the shelf hardware had clearly overridden such considerations. More worrying is the fact that more critical systems such as the latest generation of medical scanners is being controlled by Windows NT, a system with a not much greater reliability. This issue will rapidly become more widespread given the observation that the amount of software in consumer electronic appliances is *doubling* every 18-24 months, [6], an example of which can be seen for example in the commercial aircraft industry as shown in Figure 1.6.

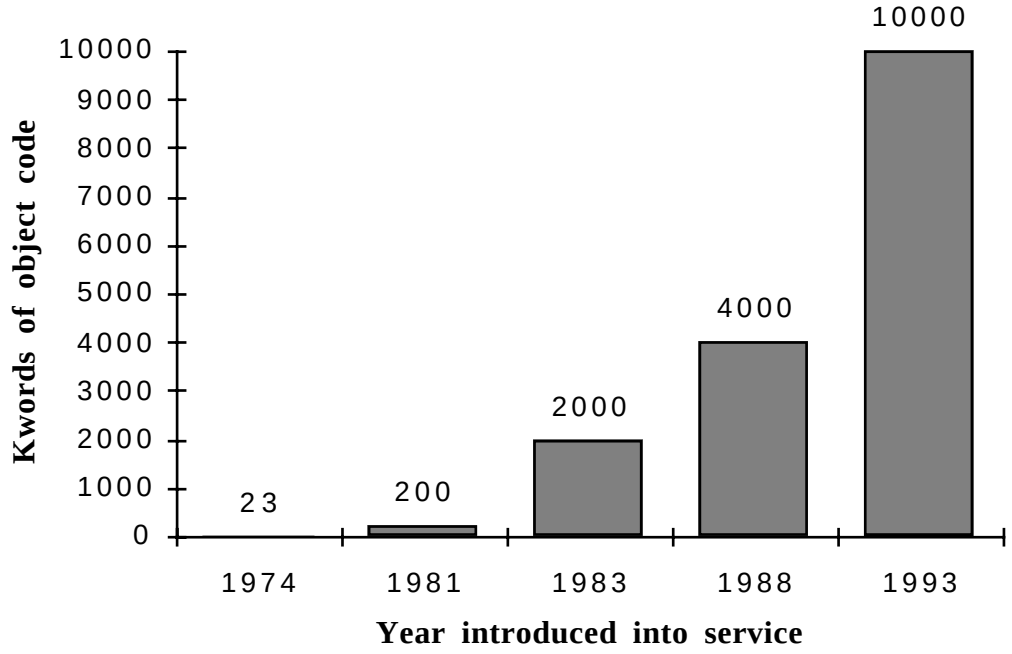

Figure 1.6 Graph illustrating the rate of increase in software measured in Kwords (1000 words) object code, onboard the Airbus AXXX series as described by [7].

Against this backdrop, the density at which defects occur within a software system, (known as the defect density and usually measured in defects per 1000 lines of code or defects / KLOC), does not seem to be improving particularly as evidenced by Figure 1.7 which shows the defect density reported in software measurement at NASA Goddard in the United States over a 15 year period.

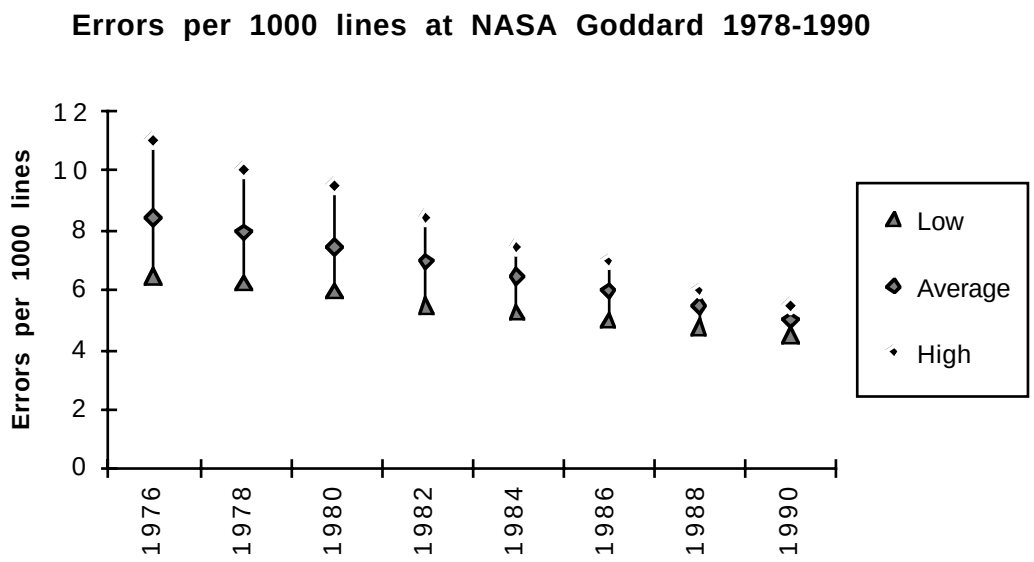

Figure 1.7 Graph published in the December 1991 special issue of Business Week showing the slow decrease in software defect density at NASA Goddard. The curves marked High, Average and Low are the defect densities of the best, average and worst components respectively of each generation.

In other words, although the difference between good and bad has shrunk by a factor of 4 over a period of 15 years, the underlying defect density of the best components of any generation has barely changed. This, together with the previous observation that the amount of software is doubling around every 18-24 months leads directly to the unpleasant conclusion:-

The number of defects in consumer electronic devices will double approximately every 18-24 months.

This certainly seems to have the makings of a crisis to come, but the most important point from a legal view is that:-

All software contains defects and significant amounts of software contain large numbers of defects. A considerable number of these could have been avoided by better practice, but not all, as certain categories of defect appear to be by nature unavoidable, and in some cases at least, unforeseeable. It is the nature of the beast with its current state of development.

That software defects are indeed visible in consumer electronic devices can be seen from the author's own observations detailed below. In each case comments have been added as to why the failure is most likely due to software.

	Nature of problem	Comments
1.	New digital answering machine has to be rebooted approximately every 3 months through, “micro-processor lock-out due to software failure”.	Previous analogue tape based machine did not fail in 5 years of use. Each time the digital machine fails, all the stored messages are lost. *The user manual contains a fault-tree which confirms that this is a software failure*.
2.	FAX machine has been rebooted once in 18 months.	Very similar behaviour to item 1.
3.	ABS braking system on a contract hire Land Rover Discovery failed every 3 months for a year until the computer controlling system was finally replaced, (at a cost of £2000), after consultation with Rover engineers.	The failure caused complete loss of brakes necessitating stopping the vehicle with the hand-brake and in the third case, driving off the road to avoid hitting the car in front. *Although difficult to pin down, this looks like a requirements fault.*
4.	New ‘fuzzy logic’ based washing machine displays the wrong panel lights. It shows washing when spinning and vice versa.	Well, really ! *This is certainly a requirements fault.*
5.	Electronic sliding roof in hired Vauxhall Omega Estate oscillated until operating button held in for two seconds to reset it.	In this case, a small footnote at the back of the operating manual warned of this possibility. *Resetting is a standard recovery for software failure.*

6,	Land Rover radio requires two presses rather than one press of a positive action software controlled on/off switch about 10% of the time in order to turn it off.	The same radio also displays occasional random volume variations, (also software controlled), and if a traffic RDS interrupt occurs during a pause, the radio jumps to a random state. *The nature of these has the unmistakeable reek of software.*

Table 2: A catalogue of computer and in most cases software related consumer electronic product problems experienced by myself in the last two years.

The production of software

This is a suitable point to return to a description of the means whereby software is produced as described by the phases a) - f) at the beginning of the last section.

The requirements phase

One or more people gather together and attempt to define exactly what the software will do when it is loaded and its execution started within a suitable computer. The group will ideally consist of engineers who will produce the software, managers who will manage its development, marketing people who will sell it, testers who will help verify that the software achieves its intended goals and users who will actually use it. In practice, this ideal situation is rarely achieved. The group may consist of one person having a 'great idea' in the comfort of their study. Different sub-groups of people may be absent, for example, the testers, and not infrequently representatives of the intended users. There may be so many people present that agreement on the intended goals is impossible and the meeting acrimoniously degenerates into a series of mutually contradictory goals.

If the reader is surprised by this seemingly chaotic and arbitrary situation, *requirements capture* as it is known is one of the great unsolved problems of computing science. It is subjective, rarely supported by any

kind of automation, wayward in its nature and frequently incomplete and inconsistent. Regrettably, it is also inextricably linked with the perceived quality of any resulting software. Mistakes at the requirement phase are usually the most expensive to put right and are frequently never satisfactorily corrected.

Design / Specification

In the design and specification stage, engineers take the original requirements and try to produce a design which will satisfy those requirements. Again, this is more of an art than a science and there are a plethora of different design methodologies, no one of which has become dominant. Designs are generally not supported well by automation and are fundamentally difficult to verify, although some success has been obtained using essentially manual techniques known as *design inspections*, whereby a design will be independently inspected by a few people who will then report their findings back to the original designers. From the design, a specification can be written in a form suitable to be turned into source code by the engineers.

Implementation

In the implementation phase, the specification for the software system is turned into source code. In many cases, it may also reference source or object code produced by third-parties, a mechanism known as *re-use*. Many defects are introduced at this stage due largely to the inadequacies and fashion-based nature of programming languages. There is much lively debate amongst computer scientists about the ideal programming language, but from the point of view of defects in the resulting system, the differences seem far less obvious, [8]. Various techniques have been promoted to reduce the number of defects introduced at this stage, by far the most successful of which is the *code inspection*, akin to the design inspection described above. However, such is the immature nature of software engineering that comparatively few companies bother.

It should be noted that at this stage, ambiguity essentially disappears. The nature of programming language compared with design methods is that the engineer must make his or her mind up about the intended action of the software even if the specification does not seem clear. Ideally, ambiguities should be resolved with designers, but this frequently is not done in the inevitable rush to produce a system which accompanies most software developments. The resulting behaviour is known colloquially amongst programmers as an 'undocumented feature'. The word undocumented would be more appropriately replaced by inadvertent.

Unit testing

In unit testing, each component is tested as it is produced, usually by its developers. The source code is translated into object code automatically by the use of software tools generally supplied with the computer as described earlier, and then executed and its actual behaviour compared with the intended behaviour. It has long been recognised that, largely due to inadequate education and the attitude that testing isn't as 'exciting' as development, that engineers are not very good at testing their own code and many defects are traditionally missed here which should have been detected. Furthermore comprehensive testing can be expensive and is frequently truncated to get the software system 'out of the door'. To gain some idea of just how inadequate testing is in practice, studies such as that done by [9] show that on average, *only around 40% of the lines of source code in any system are actually exercised by any of the tests carried out before the software is delivered to its intended users*. In other words, on average, the majority of a system has never been tested in any form before release.

System and integration testing

At this stage, individual components are glued together such that their individual functionality combines together to produce some kind of system which collectively obeys the original requirements. Implementation defects

found at this stage are expensive to correct as the component must be returned back to the implementation stage. In some cases, design defects are found and the system as a whole may have to be returned back to the design stage for appropriate re-design. This may even be done deliberately using the mechanism of *prototyping*, whereby a system is implemented in a skeletal form to alert the designers to any fundamental deficiencies.

Release

At this point, the software is released to its intended user. Defects found here can be very expensive indeed. For example, a defect in a car's braking system could lead to a recall of many thousands of cars[5]. The object of software testing is to minimise the number of defects which are released. Elimination is generally believed to be either impossible or commercially unachievable.

Taken together, these phases may appear somewhat involved and error-prone to the reader. This is indeed the case as can be seen by data produced on the relative frequency of achieving a satisfactory conclusion to a software development project. Figure 1.8 is an example of such data. This data was produced by the Audit Office of the United States Department of Defence covering the development of some $140 million worth of military avionics projects in the period 1985-1990.

[5] When I first wrote this, I was thinking hypothetically, however this has now happened. On 22 May, 1998 in the USA, it was reported ([10]), that coding problems for anti-lock brakes may affect 4 million GM pickups and cars. These have experienced very long stopping distances leading to 15,000 complaints in last 3 years. Robert Lange, director of safety engineering at GM, said they have now ruled out corrosion and are focussing on computer algorithm problems. He also stated that he thought fewer than **1 million** of the trucks would be likely to be affected by a recall - a contention that federal officials are expected to dispute vigorously.

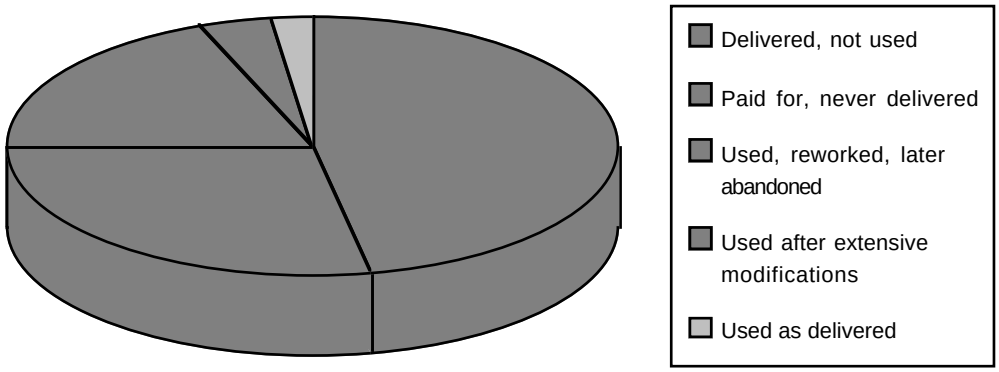

Figure 1.8: The risk associated with software development. Data due to the Audit Office of the U.S. Department of Defence on military avionics software development in the period 1985-1990.

As can be seen, only 10% was delivered in a useable form and only 2% was delivered in a form which could be used without extensive re-work. Again this may come as a surprise to the reader but *failure of software projects to achieve their intended goals is the rule not the exception*. That these data are still relevant is confirmed by the fact that the literature is full of examples of contemporary grand failures such as the London Ambulance Service system, the Taurus project of the London Stock Exchange and so on, [7], [11], [12], [13] , [14]. Many of these failures have all too familiar patterns of management incompetence and over-ambition, [14]. Truly we are not learning the lessons of system failure easily.

Types of software

Software can be delivered in a number of forms and this is particularly relevant to the legal viewpoint. There are traditionally three categories.

COTS

COTS is an acronym for Commercial Off The Shelf. In other words, COTS software is developed by a manufacturer with no formal input from the intended users and it is sold to the user community as possessing certain

functionality. The potential user assesses from the description and perhaps behaviour of this software whether it satisfies a particular need or not and if it does so at a reasonable price, will buy it. The requirements may have originated entirely within the supplying company or may have arisen from a third-party, such as some legislative function.

The important thing is that the intended user did not in any formal way influence the eventual behaviour of the software, although they may of course have taken part in a market survey.

Modified software

In this case, the user may have identified a piece of software as capable of satisfying all of their needs but only with some modification. In this case, a contract may be set up between the user and the producer to alter the behaviour of the product to suit the needs of the user.

Altering the behaviour of a software product can be done either with source code or by object code. In the case of source code, engineers will read the source code and introduce changes necessary to produce the additional functionality. The process is fraught with problems as exemplified by the known fact that modified code has about 3 times as many defects per thousand lines of code as unmodified code. This is particularly relevant to the current Millennium (also known as Year 2000 or Y2K) preparations. To paraphrase Winston Churchill,

> "Never in the field of engineering has so much change been done by so many people to so many systems with so little understanding of the effects of such change"

For example, as quoted by [15], the rate at which defects could be injected by such a process could be as high as one injected defect for every three changes. This is likely to lead to significantly reduced levels of reliability compared with the same systems before Y2K changes. No other engineering discipline has such a high probability of injecting further defects as a result of correcting existing defects and this should be borne in

mind by the legal reader. It is not deliberate, it is simply that we don't know how to stop it happening.

In the case of object code, it is very much more difficult to divine the relationship between the code and the existing functionality. A small number of experts can read object code directly but it is extraordinarily laborious for all but the most trivial of programs and in general it must be *dissembled* using a tool which operates in the opposite direction to the translator or compiler mentioned earlier. In fact, this notation is inaccurate as there is a historic intermediate step which is often forgotten. As this use has crept into existing legislation such as the EC directive [16], the original definitions are indicated in Figure 1.9

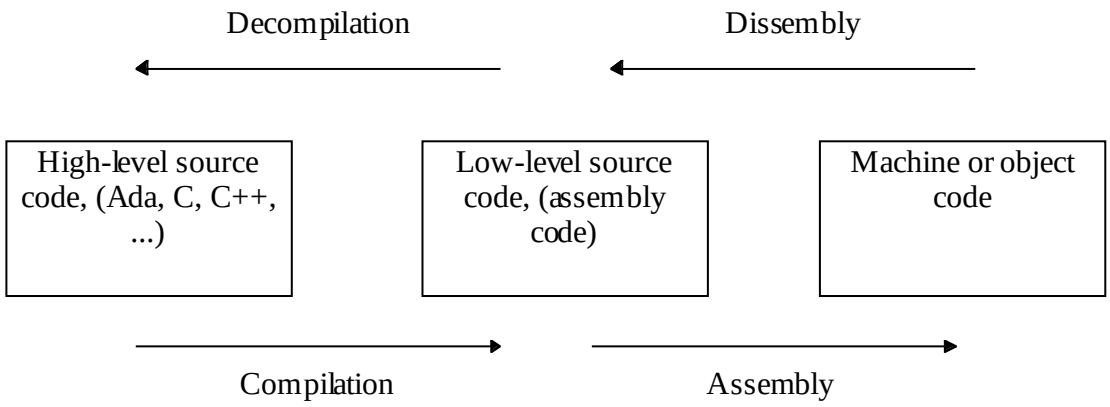

Figure 1.9 The actions of compilation and dissembly.

In either case, the user specifically takes part in the definition of the requirements of the eventual system, perhaps contributing significantly.

Bespoke software

In bespoke software development, the software does not initially exist in any form usually and the user is involved from the beginning in the definition of the requirements for the eventual system. To a greater or lesser extent, the developer may which wish to re-use components of software which have been previously written but the essential nature of this kind of software is

that the user has an intended functionality in mind and wishes to commission a piece of software to carry out this functionality.

In this case, the user is instrumental and plays a central position in the definition of the requirements of the software. Note that even though bespoke, such systems may also use third-party products within them as indeed might COTS or modified systems.

So we see that one of the principle differences between these three types of software lies in the degree to which the eventual user was involved in the *definition* of the requirements. This turns out to be a key point in understanding the differences between the cases quoted in Appendix A.

It should be noted that a segregation of software into three discrete forms is entirely artificial. In practice there is a *continuum* between on the one hand, COTS software and on the other, entirely bespoke software. This continuum is shown in Figure 1.10 along with its relationship to requirements.

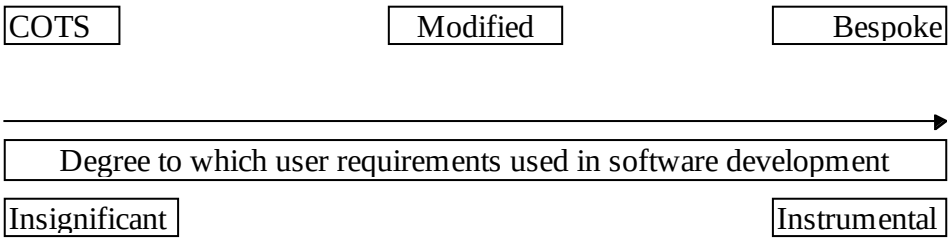

Figure 1.10: Software as a continuum between COTS and bespoke and its relationship with the degree to which user requirements influenced the software development itself.

The growth of engineering principles

Engineering principles take many years to evolve. The science of bridge building and other civil engineering structures has taken many hundreds and in some cases thousands of years to mature. As long ago as 3500 years, the ancient Egyptians had a unit of measurement - the cubit - which varied by less than 8% in the ancient world. Indeed, they even

accommodated this variation eventually by defining a royal cubit, some 8% longer than the ordinary common man's cubit. They also knew how to construct a right angle using a length of string, a pivotal rod and a marker rod and had technologies for moving and manipulating very large building blocks. The Carthaginian general Hannibal had a natural suspicion of engineers 2000 years ago. When he crossed the Alps, the bridge-building engineers were always sent across first on the heaviest elephants. This practice no doubt concentrated the engineers' minds wonderfully.

In contrast, software engineering is only around 50 years old and although it appears to have made great strides in supporting technology, this however is largely illusory and should be distinguished from underlying maturity in which it has not made great strides. Indeed, it has far more in common with the fashion industry at its present stage of development than an engineering industry. This is evidenced by the arbitrary automation, the regular rejection of tools in place of new untested tools and technologies and the almost breathtaking lack of systematic measurement, one of the primary properties of an established engineering discipline. This is witnessed by data such as that quoted by [17] for example, whereby 85% of all *software tools*, (levels of automation to support software technology), ceased to be used after 3 months. It is quite common to find supporting automation which resolved a particular software engineering problem to terminate at the end of the project which funded it as if the benefits then ceased. This would be analogous to finding a carpenter knocking nails in with a hammer and then when the chair or whatever is finished, knocking them in with a brick because the project has now finished. In other words, tools to support software engineering automation have the usage pattern of a child's toy rather than an engineering tool which will conventionally last an engineer in mature engineering disciplines through his or her working lifetime with only occasional and essentially identical replacement. If a tool solves a problem, you don't throw it away otherwise the problem begins to

occur again. Software engineering is uniquely and regrettably characterised by such *repetitive failure*.

Problem areas in software

From the above discussion, it will be apparent that software engineering is frequently inconsistent, produces an unusually high number of defects in comparison with other engineering disciplines and appears to be improving far more slowly than the speed of its adoption in all manner of consumer and other devices. To guide us in a legal understanding, probably the two most important deficiencies are requirements capture, (which can be used to understand the differences between existing judgements as described in Appendix A), and the rate at which defects appear to the user.

Software requirements capture

It has already been noted that this is quite simply an unsolved problem and it is significant factor in software systems failing to achieve their intended functionality. This situation is unlikely to change for a long time and is likely to be a key factor in legal determination of disputes.

Software reliability

Defects in delivered software appear to be inevitable at the current state of technology, a situation already remarked on in the prescient nature of the ruling in *Saphena Computing v. Allied Collection Agencies, (1985),* where the court noted the comments of an expert witness to the effect that:-

> "Just as no software developer can reasonably expect a buyer to tell him what is required without a process of feedback and reassessment, so no buyer should expect a supplier to get his programs right first time".

Again as we have seen, this situation is likely to get worse rather than better in the near future as consumer electronic software grows rapidly.

Fault versus failure

It is important to realise the distinction between software *faults* and software *failure*. A software fault is a potential problem waiting to happen. They can in general be detected simply by visually inspecting the source code without actually running or executing it. In contrast, a failure is a departure between actual and expected behaviour when the software is actually running in its so-called machine-readable form. *The relationship between fault and failure is generally very complex in that failure stems from at least one fault, but some faults never actually fail*. This situation was aptly demonstrated by [18] who demonstrated after an extensive survey of IBM's fault and failure database that:-

- About 33% of all faults fail less than once every 5000 execution years

- The most common failures, i.e. those occurring less than once every 5 execution years are due to a small minority of all faults (around 1-2%). In other words, repetitive failure is a dominant characteristic of software systems. This is unique in engineering generally.

- Each time a fault is corrected, there is about a 15% change of injecting a fault at least as serious as the one which is being corrected.

It is also in principle impossible to predict whether a particular fault will fail and if it does, the extent of the departure from expected behaviour, for example, whether it will be catastrophic or not. The only relationships which we appear to be able to use are statistical ones. For example, all programming languages in common use have well-known faults, whose presence is due essentially to the fact that the relevant programming standards committee simply could not agree on what happens in every eventuality and therefore simply allowed anything to happen. If this frightens the legal reader, welcome to the club. These faults occur again and again and fail with some frequency. We can even measure the

dependence of commercially released software on these known faults as can be seen in Figures 1.11 and 1.12 below which show these dependencies for the C and Fortran 77 programming languages respectively[6]. These are not toy languages, indeed C is very widely used in everything from operating systems to cars and medical scanning devices and Fortran 77 has been the de facto standard language for nuclear reactor control for a number of years and is only recently being superseded.

The reader may note from Figure 1.12 particularly, the alarming differences that can occur even within the same industry. For example, the nuclear engineering industry is responsible for the worst code ever measured for this effect, (rising to 140 statically detectable faults per KLOC and indicated with an arrow on the Figure), and the very best, (0 statically detectable faults per KLOC), for comparable levels of integrity, which to a legal reader would be equated to a risible relationship between duty of care and standard of care. This enormous difference in the capabilities of engineers working in the same area has long been known in terms of productivity. It is a natural human phenomenon and occurs in other disciplines also but in those disciplines the difference is largely suppressed by effective checks and controls resulting from many years of experience. Software engineering has not yet attained this level of maturity and so individual engineer differences often dominate, obscuring other factors and making progress difficult.

[6] The reader should also note that the apparent repetition of the same category in both diagrams, for example the Utilities category in Fig 1.11, is due to Microsoft Excel's unfortunate predilection for missing out words arbitrarily. In my data, the words *Water* Utilities, *Gas* Utilities and *Electricity* Utilities appear. Excel took offence to these words and excised them from the graph.

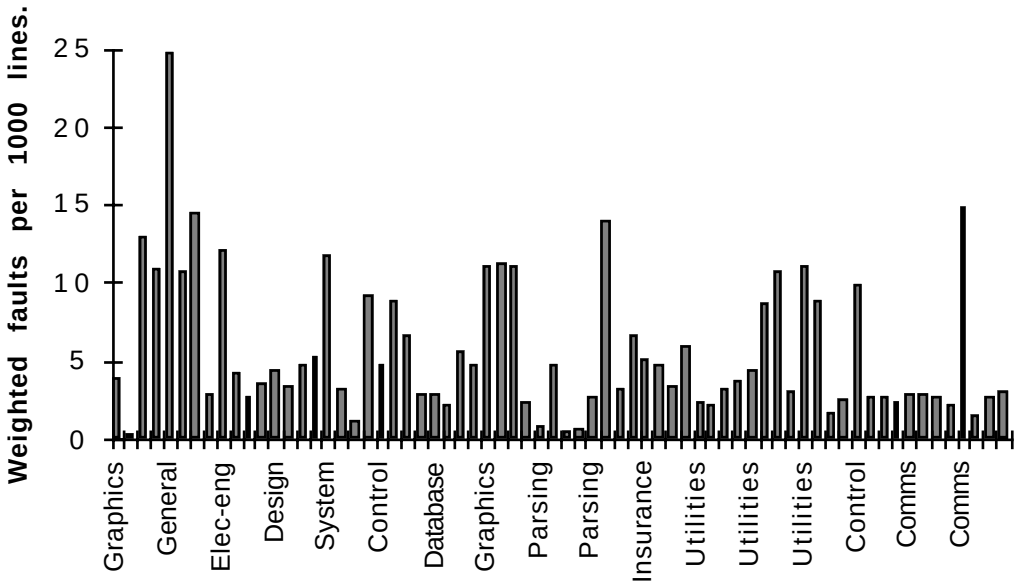

Figure 1.11: The distribution of statically detectable faults per KLOC (1000 lines of code) in commercially released applications around the world measured between 1992 and the present day written in the language C as measured by [19]. The codes were generated in the last twenty years and come from many kinds of applications and industries around the world. Several million lines are represented in these statistics.

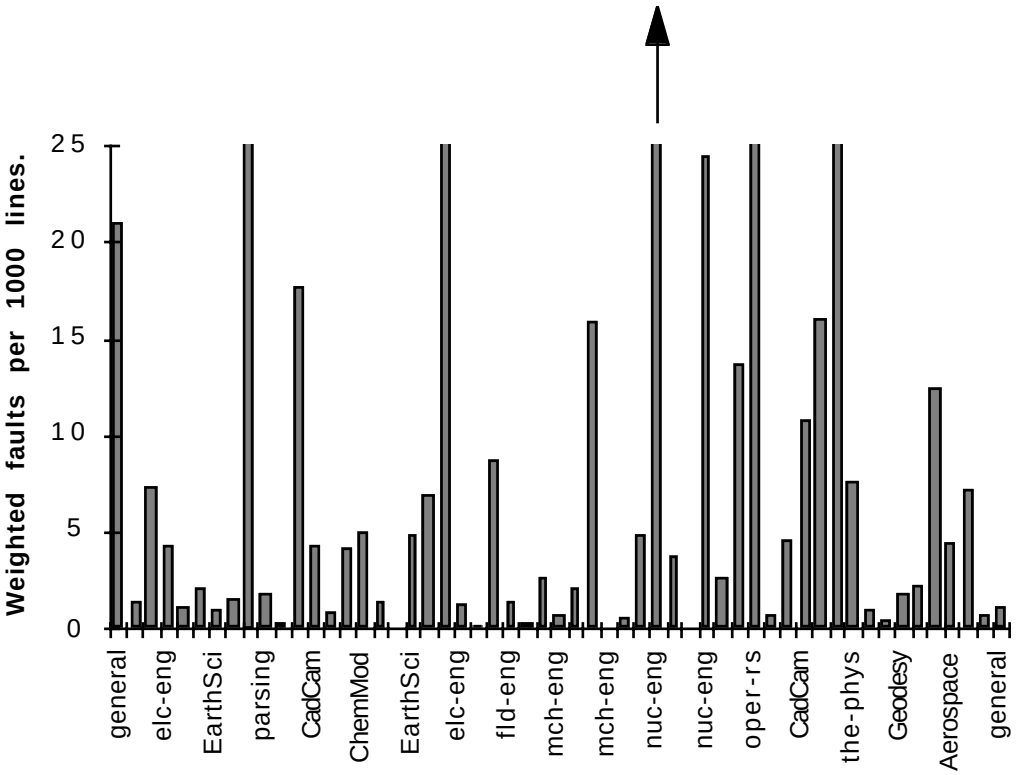

Figure 1.12. A study similar to that shown in Figure 1.11 but this time for the language Fortran 77. A very similar pattern of dependence on fundamentally undefined behaviour can be seen.

The reason that all these potential faults sit inside commercially released code ticking way like the time-bombs they are is that there is nothing in standard commercial software development processes to prevent them appearing, and yet we know how to prevent them and have known for a long time. Statistically, we also know that around 40% of these will develop into failures in typical software systems during the lifetime of the software products. This is simply collective culpable stupidity and is impossible to defend on any reasonable grounds.

Quantifying fault and failure

The reader should note the following, [20], [13], [21]:-

> If all faults, major or minor, are counted, software which exhibits less than one fault per 1000 lines of source code[7] during its life-cycle is just about as good as has ever been delivered and very few software development groups achieve this systematically. Note that a software life-cycle could be anything up to tens of years. Software which exhibits between 3 and 6 faults per 1000 lines of code during its life-cycle would be considered reasonable quality commercial code. Furthermore, it is not uncommon for software systems to exhibit >10 faults per 1000 lines of code during their life-cycle and some studies show up to 30 faults per 1000 lines of code.

Put simply, this means that if a software development group delivers a million line system for some critical application, (a not uncommon scenario), then if they have done a really good job, the software will only contain around a 1000 faults. This is not a particularly happy prospect to contemplate. Now, it is also known, (e.g. [22]), that only perhaps 5-10% of all faults in a software system have a major effect on the system behaviour,

[7] It is common for software engineers to measure overall reliability using *defect density*, i.e. the number of defects exhibited per 1000 lines of source code during some defined part of the life-cycle, rather than the more conventional *mean time between failures*, or *probability of failure on demand* used by hardware engineers.

which means in plain English that such a system will have around 50-100 faults which are likely to appear as serious failures during the life-cycle of the product. In other words, a million line state of the art system will exhibit 50-100 serious failures in its life-cycle. !

From a legal point of view, there is also a significant difficulty in ascribing failures unequivocally to faults in software. For example, it is fairly typical to find that around 1/3 of all failures could not successfully be associated with any fault as witnessed by Figure 1.13, taken from [22]. These appear as the category ‘unassigned’, in the sense that although the system failed, the failure could not be ascribed to any particular fault. This problem is usually caused by a combination of complex systems and inadequate diagnostics. A wonderful example of this is quoted by [7] whereby an error originating in a trimmer on an Airbus rattled round the computer network and eventually manifested itself as a completely spurious ‘Smoke in the Lavatory’ message. Given this tenuous relationship between cause and effect in such computer systems, we should not be surprised that some categories of failure are simply ‘unassigned’ ! One further comment is worth making about Fig 1.13. There are no standard categories for describing failure. In this case, documentation has been split up as a separate category and the words ‘major’, ‘medium’ and ‘minor’ reserved for actual system behaviour discrepancies.

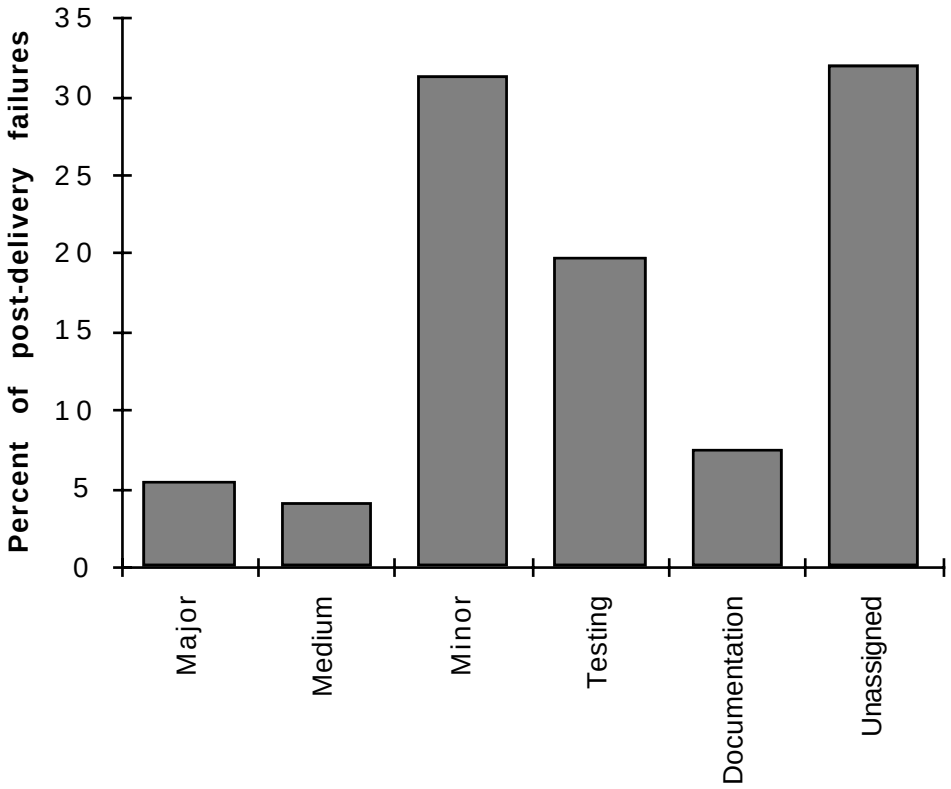

Figure 1.13. This shows typical categories of failures in an air-traffic control system. Note that around a 1/3 of these could not be assigned to any particular category.

In other words, in a court, if an expert witness were asked to state categorically whether a particular failure was a software failure or a hardware failure or perhaps some other system interaction, there is a significant chance that they could not. This is made worse by the observation that many programming languages contain features which are not portable between different compilers for the same language, [23]. In other words, precisely the same source code could be moved between different compilers and the results could be legitimately different because programming languages are the product of committees, and as a result, generally contain significant amounts of behaviour the committee could not agree upon[8]. This is known as the problem of *software portability* and

[8] There are 197 such features in the programming language C, a language widely used in embedded systems in consumer products. This is quite normal. In addition, in this language, in the period 1990-1997, a further 119 defects were found in the programming language standards document itself, as issued by the ISO, (the International Standards Organisation). This also is quite normal.

achieving such portability, whereby the same source code has precisely the same behaviour on different machines can in practice be formidably difficult, particularly when modern and highly complex languages such as C++ are used. Only one language has ever achieved exceptional portability relatively easily and that is Java, and even this is threatened by the current U.S. law suite between its designers, Sun Microsystems and Microsoft who are allegedly changing the specification such that Java programs developed on Microsoft systems will only run on Microsoft systems. This is evidenced in the Microsoft product *Visual J++* which has apparently built-in access to MFC, the Microsoft Foundation Classes on which Microsoft systems are built. The problem of course is that if you use such access, the resulting software can only run on such systems. This is completely against the entire spirit of Java.

Inadequate process control

Enough is already known about the process of producing software to identify certain technologies which if absent, greatly increase the risk of a software project failing. This should help cast light on what would be considered reasonable care in software development, a matter to be discussed later. The four primary required technologies are:-

Change and Configuration Control

Change and configuration control is very simple. In essence, it describes the ability to trace every requirement to the components which had to be modified or added whilst disallowing the notion of any change without requirement. In addition, it allows the developer to re-create the system at any point in its history, especially at key release points and provides a complete inventory of the necessary components and how they must be assembled together to create the full system. Although this can be carried out manually, all but the most trivial systems would overwhelm the developer and modern systems can consist of many thousands of components and millions of lines of code so automation is essential.

Fortunately, such automation is widely available and of high quality. Less fortunately, only the minority of companies employ it.

Project Estimation and Planning

This would be immediately familiar to a project engineer from another engineering discipline. It can take many forms such as PERT or GANT charts but in essence is a complete list of all tasks which need to be carried out with their appropriate resources, time deadlines and most importantly, their interactions and dependencies. Many excellent software tools allow this to be fully supported with automation and it is known to be a very significant factor in a successful project. Again, its use is relatively rare in the "Ready, Shoot, Aim" software development industry.

Project Management and Tracking

A project plan is of course of no use unless progress against it is regularly monitored. Experience suggests that if a project is broken down into tasks taking no longer than about 5 days and if progress is monitored weekly, it is very common for software projects to complete successfully within about 10% of their schedule. In addition, risk is greatly reduced because if problems arise, they are identified early, allowing backup procedures to be used. In contrast the absence of project management and tracking is responsible for some of the enormous project over-runs for which software engineering is justifiably infamous[9].

A very useful and very simple piece of data often missing from software development projects is the difference between the planned delivery date of the next milestone and the current date plotted against the current date. A simple graph of this can tell even the most IT resistant senior manager exactly how a project is progressing. Some examples are shown in Figure 1.14.

[9] Even when project management is present and in use in mature disciplines such as civil engineering, over-runs still occur as witnessed by Concorde, the Channel Tunnel and now the Millennium Dome as well as countless others.

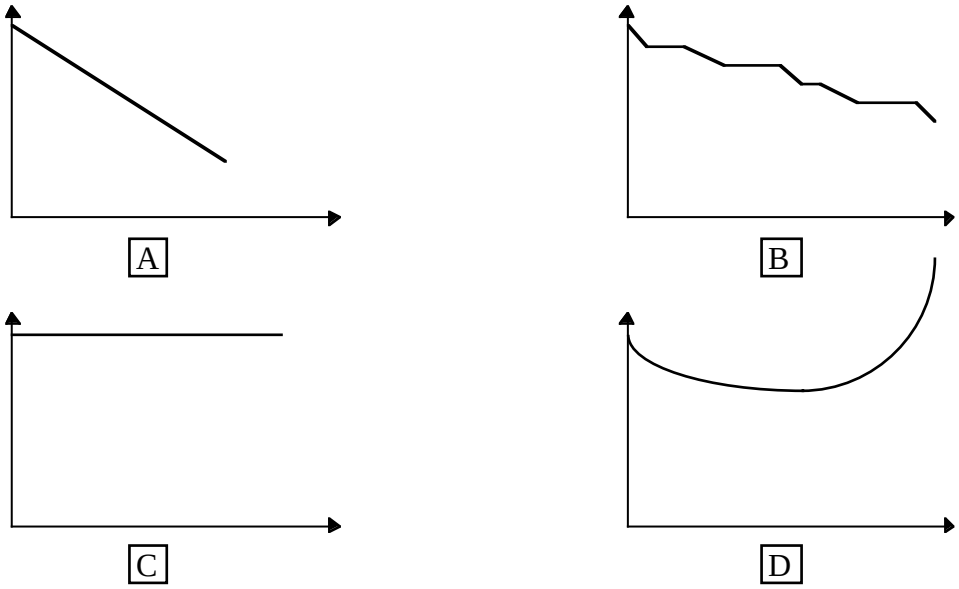

Figure 1.14. Examples of plotting the difference between planned delivery date of the next milestone and current date (vertical axis), against the current date, (horizontal axis). When the line hits the horizontal axis the milestone is complete.

(A) shows the ideal case of a project progressing steadily and on time without problems. This is the ideal case and is a result of careful requirements capture, and regular, fastidious project tracking against a well-defined plan. It happens but requires great experience and pragmatism. (B) is a project experiencing some difficulties but they are minor problems in general and the overall trend is downward. The project will be late but should be delivered essentially intact. (C) is the classic pattern of a project whose requirements are never initially specified. The project is simply going nowhere. These are surprisingly common, (c.f. for example [14]) and tend to be characterised by an unwillingness by anybody involved to recognise this simple fact. A graph like this makes it blindingly obvious from an early stage so that corrective action can be taken. (D) is an example of creeping 'feature-itis'. The users are presented with a system and then begin increasingly to request new features. The project begins to spiral out of control. These too are unpleasantly frequent and are a symptom of the popular misconception that software is easy to change. Again, they are blindingly obvious and corrective action can be taken early

rather than late into the project when huge amounts of money will have been wasted.

Software Quality Assurance

Most readers would think it unthinkable in this day and age that software could be released to the general public with little or nothing in the way of quality assurance given the massive strides in this area made by the various manufacturing industries in the last few years. However, again, many software companies are deficient in this area with software quality assurance functions having little power and very little agreement about what actually constituted software quality assurance.

The absence of any these technologies leads to a process state formally known as chaotic, (the 1st level of the CMM model discussed in more detail in Chapter 3). According to early studies done in the United States, [24], 85% of all companies were in this state. A disturbingly large number of companies operate with two or more missing. From a legal point of view this is important, as it is now relatively easy to identify poor software development processes in a highly objective manner in a way in which expert witnesses would find considerable grounds for agreement.

The deliverables

This problem has become much more acute in recent years. It is convenient to think in terms of a software producer producing a piece of software, compiling it into an executable and delivering it to the customer. This has always been slightly problematic in the sense that the executable will contain not only software written by the producer but also system software written by other parties. This is indicated graphically by Figure 1.15

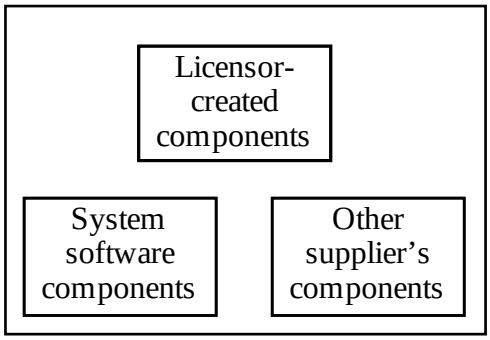

Figure 1.15: The various components which can make up the delivered software executable.

In other words, all the issues of contract, liability and so on considered in this thesis are complicated by the fact that determining where a failure has taken place and more to the point, who is responsible can be very difficult indeed. Of course, to the end-user, the result is a single executable but there may be all kinds of collateral contracts in existence between the licensor, the producers of the system software and the suppliers of other components. Just to flesh this out a little, if a user purchases a piece of contact management software for example, this will contain all the code pertaining to contact management, but will also contain functionality supplied in the system software, for example, functions returning the system clock time, and is very likely to contain yet another party's database components as contact management software relies heavily on databases and these are usually supplied by specialist manufacturers of database management software. It may also depend on yet another producer's graphical user interface (GUI) components.

In recent times, this has become even more complex with the growth of the **shared library**. Historically, executable software as shown in Figure 1.15 was complete, self-contained and self-consistent. In theory, it could be put on any machine compatible with the way it was made and run on its own. The price for this flexibility was that the executable could be

very large in machine terms. For example, it might have taken up a significant percentage of the available random-access memory (RAM) in the machine, into which all executables must be loaded (i.e. copied) before they can be run. To circumvent this problem, the practice of shared libraries has arisen. Here, parts of the executable for example the system components or another supplier's components, can be split off so that they can be shared between different executables, so that only the licensor's components are present in the delivered executable. The resulting executable program might take up only 200,000 bytes rather than 2,000,000 bytes, a space saving of some 90%. The other 1,800,000 are still necessary but are only accessed when needed and are shared by any other executable that might be running at the same time[10].

The executable will only run if the split off components are separately present on the licensee's machine. In practice, this should be identical from the licensee's perspective who is unaware that all this is happening. However, it must be remembered that shared libraries also evolve with time, so it is perfectly possible for a licensor to test their own software using version A of the shared libraries and deliver it to the licensee who has version B of the shared libraries. The result can be chaos with responsibility very difficult to apportion. The situation is shown graphically in Figure 1.16

[10] On a multi-tasking machine, many such executables might be running effectively simultaneously. Originally PCs were single-tasking machines with only one executable runnable at any one time. However, even PCs have some elements of multi-tasking present, for example, one document can be printed whilst another is still being worked on from within a word-processor.

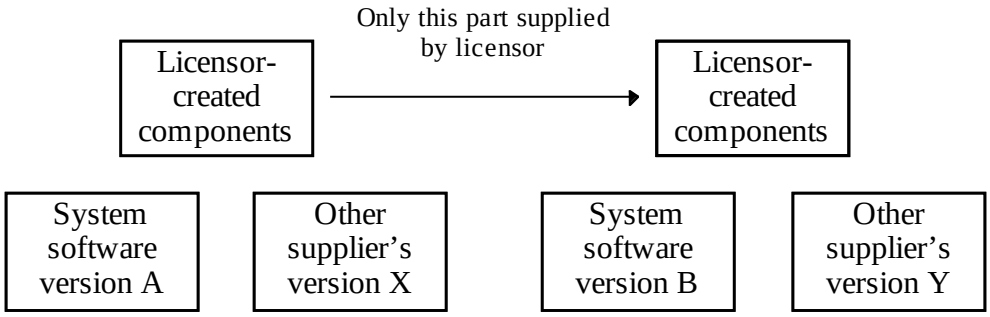

Figure 1.16: Illustrating the concept of shared libraries whereby the licensee's running executable may not be the same as the version the licensor tested on their own system.

The author has had so many problems with shared libraries over the years that he now tends to avoid them even at the expense of producing a much bigger executable than would be the case with shared libraries.

Common misconceptions about software

There are many misconceptions about software, a large percentage of which are regrettably held by senior management in companies developing software. They should know better and this continues to exacerbate an already difficult technical problem by surrounding it in managerial problems. Some of the worst misconceptions follow:-

a) Software is easy to change.

This is correct on a trivial level, however, it is NOT easy to change *correctly*. Enhancements to a system suggested after the basic design has taken place are frequently difficult if not impossible to incorporate. There is simply too much evidence against this, for example as provided by [18], who showed that in effecting corrective change (i.e. fixing defects) in big systems, *there was about a 15% change of introducing a defect at least as large as the one which was being corrected*. This occurs because it can be very difficult indeed

for a programmer to predict all the effects a change might have on a big software system. These are known as *side-effects* or *spoilage*.

b) External deadlines can be imposed successfully.

Most software projects are late and some are very late, [25]. Analysis of this shows that it tends to occur because deadlines are set by management without any attempt to estimate or plan a project and this is one of the most basic failures in the well-known Carnegie-Mellon CMM software process model, [24]. A recent fascinating example of this management idiosyncrasy at work could be seen in an article in the Times[11] in December 1997 whereby a large number of companies were interviewed and asked when they would finish their Year 2000 conversions. Every company named the same deadline of November 1998, irrespective of what stage they were at, (some had not started). This was obviously the politically acceptable deadline for management to set. As many of these companies will find out, it has little if any relationship with reality. The effect is clearly visible in *Salvage Association v. CAP Financial Services Ltd. (1993)* for example.

c) Adding people to a project can be used to help bring in a late project on time.

This was profoundly dismissed long ago by [26], who showed that adding people to a late project simply made it *later*. In addition other software process models such as the COCOMO model [3], which resulted from real life experiences of defence software projects show a *minimum* in the duration of a software project. As the number of engineers increases, the delivery time becomes closer up to a certain relatively modest number, after which the delivery time begins to recede again.

[11] This was one in a series on the Millennium produced in the middle of December.

d) Defect-free software of any significant size can be delivered.

From the figures given earlier, this is clearly an absurd notion. The very best software will still exhibit around 1 defect per 1000 lines of source code during its life-cycle. Less robust systems could be 30-50 times worse than this.

In preparation for the next chapter, the key points of this chapter are here summarised.

- Software fails frequently. When it does fail, it frequently proves impossible to fix
- Many software failures are entirely avoidable, but a significant number remain unavoidable in spite of our best efforts
- Software failure is entirely unpredictable. A given fault can have no effect or conversely an absolutely catastrophic effect on the system behaviour. In this respect, it is truly chaotic
- Software development is immature and little progress has been made in making it more reliable in the last 20-30 years
- New bespoke projects have a very low success rate
- It is debatable whether software is tangible or not
- Expert opinion is likely to differ very considerably.

Clearly, software presents some interesting legal challenges, but they must be addressed to avoid the possibility of throwing out the baby with the bathwater as outlined at the beginning of this chapter.

In the next chapter, various legal issues will be discussed using a standard legal approach but with commentary on those aspects relevant to software engineering given the nature of the beast as summarised above

Chapter 2: The nature of legal liability for software

In the previous chapter, a general introduction to software, its nature, its development and most importantly its problems has been given from a computer scientist's point of view. In this chapter, software engineering will be described again but this time from a typical modern legal point of view. This process should be considered as essentially territorial. When trying to bridge the chasm between two disparate subjects, there is usually no natural way to proceed. In the absence of one, all which will be done here is to cover the ground covered in standard legal texts and add interpretations, possible sources of confusion and other commentary from a computer scientist's point of view. For a very detailed exposition of information technology law, the reader is referred to [27] and [28].

The interpretations relevant to software engineering will appear with a left border thus:-

A software engineering interpretation.

In a later chapter, particular aspects will be developed further crossing the divide between the computer science aspect and the legal aspect. As will be seen from these observations, natural links of consistent interpretation emerge.

We have already seen in Chapter 1 that software is very likely to fail in its life-time, that such failure is likely to lead to loss and that at least some of this failure is unavoidable. We must therefore ask what are the legal consequences of such failure ?

Important statutes covering software delivery and production

English law and indeed much of US law is based around common law unlike the rest of Europe which is essentially based around civil law. Although there is considerable convergence by means of statutes, which are essentially tried and trusty recipes for dealing with particular legal situations, these recipes have considerable shortcomings when attempting

to deal with actions arising as a result of software. One particular issue is for example, whether software should be classified as goods or as a service. So far the courts have essentially (and very successfully) ducked this issue and have resolved cases effectively on contractual issues according to basic statutes and the application of common law principles. However, this somewhat uncomfortable relationship will need to be clarified in the future as litigation increases, (perhaps fuelled by the coming so-called Year 2000 problem), and the scope and range of cases which reach the courts increases.

The following statutes are of direct relevance and will be referred to frequently:-

- Sale of Goods Act (1979), **SGA79**
- Supply of Goods and Services Act (1982), **SGSA82**
- Sale and Supply of Goods Act (1994), **SSGA94**
- Unfair Contract Terms Act (1977), **UCTA77**
- Consumer Protection Act (1987), **CPA87**

It should be noted that to a certain extent, SGA79 is largely replaced by SSGA94, for example in the area of implied terms except where defects in software before 1994 take a long time to materialise, a not unlikely scenario given the discussion in Chapter 1. SGA79 is also still relevant with respect to fitness for purpose.

The nature of liability

In English law, software-related liability can arise from three sources as shown in Figure 2.1. In general, only civil liability will be considered here, although case law from criminal proceedings will be used later to help clarify the nature of software in the eyes of the law.

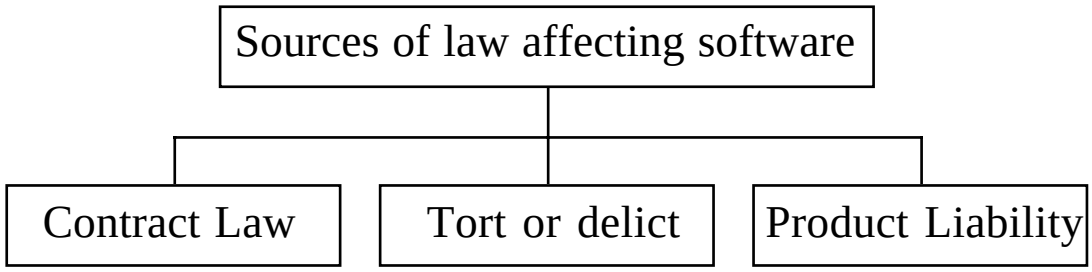

Figure 2.1: The three principle sources of law affecting the production and distribution of software.

In essence then, civil liability arises either from contractual arrangements or from a non-contract basis. We can divide this latter category into liability arising under the general law of Tort, and liability arising under the Consumer Protection Act or Product Liability. These sources will be considered in order

Liability in contract

To date, this is the only significant area of liability which has been tested in the courts as illustrated by the two influential cases discussed in the Appendix. The nature of contractual liability in the case of software depends on whether a contract is interpreted under SGA79/ SSGA94 or SGSA82, the goods versus services issue discussed later in this thesis in some considerable detail, where it will be concluded that the situation is sufficiently poorly explored at the moment that to avoid problems it is in the best interests of both supplier and buyer to ensure a suitably robust contract using the techniques described in the Chapter 4 below.

The approach taken is represented in the form of the following graph[12], Figure 2.2:-

[12] A suggestion I owe to Professor Ian Lloyd with thanks.

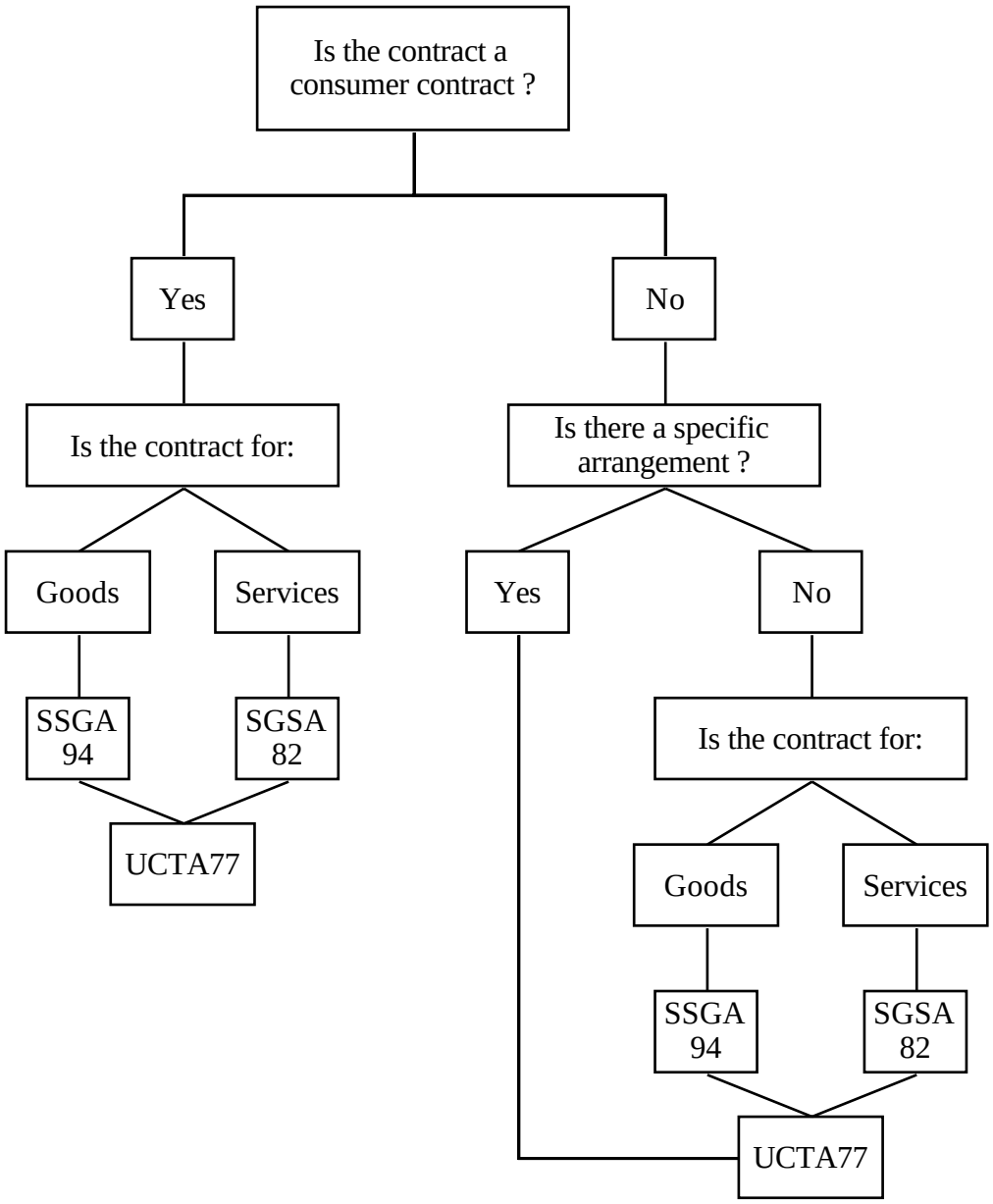

Figure 2.2 A structure for the discussion of contractual liability in the text.

We can transform this structure into the following:-

Non-consumer contract with a specific arrangement

In this case, the contract is subject only to the provisions of UCTA77, although s.4 of SSGA94 added further provisions for the supply of goods by inserting the following after s. 15 of SGA79:

"(15A) (1) Where in the case of a contract of sale:-

(a) the buyer would, apart from this subsection, have the right to reject goods by reason of a breach on the part of the seller of a term implied by s. 13, 14 or 15 above, but

(b) the breach is so slight that it would be unreasonable for him to reject them,

then, if the buyer does not deal as a consumer, the breach is not to be treated as a breach of condition but may be treated as a breach of warranty."

In general, where one deals not as a consumer, the law takes the view that whatever contract one signs is up to the two parties although via UCTA77, it will take into account various factors including the relative bargaining strengths and availability of insurance. In addition, certain implied terms such as title and description cannot be excluded.

An additional word on insurance cover seems appropriate. This area was discussed in detail in *Salvage Association v. CAP Financial Services Ltd. (1990)* and was indeed "a major point of contention throughout the trial". As ever, hindsight seems so much clearer. The defendants, argued that insurance cover was available to the plaintiff at a relatively reasonable rate of 3-4% of the cover required. However, as became clear, this only covered against possible delay, rather than complete failure to produce a workable system. When pressed on the possibility of getting cover against this latter eventuality , one of the witnesses said:-

"Well, you can insure anything and you can get cover from Lloyds to ensure any risk, provided you are willing to pay a high enough premium" (or words to that effect).

Regrettably, complete failure is *the norm* in software systems as evidenced by the data presented in Chapter 1, so in effect, the kind of insurance cover contemplated by the courts is effectively precluded.

Any other contract - the consumer standpoint

For all other cases, the viewpoint is as if from that of the consumer. The involvement of the consumer is a relatively recent phenomenon and this exposes producers and suppliers to a more extensive legal regime. In this case, we must first ask if the contract is one for goods or services. If it is a contract for goods, it falls under the auspices of SGA79/SSGA94. On the other hand if the contract is one for services, it falls under the auspices of SGSA82.

If it is one for goods, the essential difference between SGA79 and SSGA94 from the point of view of software is the definition of what is acceptable and when it is accepted. SGA79 contained an implicit term that goods would be of 'merchantable quality'. This definition changed in SSGA94, (s.1) to the following:-

> "(2) Where the seller sells goods in the course of a business, there is an implied term that the goods supplied under the contract are of satisfactory quality.
>
> (2A) For the purposes of this Act, goods are satisfactory quality if they meet the standard that a reasonable person would regard as satisfactory, taking account of any description of the goods, the price (if relevant) and all the other relevant circumstances.
>
> (2B) For the purposes of this Act, the quality of goods includes their state and condition and the following (among others) are in appropriate cases aspects of the quality of goods-\
>
> (a) fitness for all the purposes for which goods of the kind in question are commonly supplied,
>
> (b) appearance and finish,
>
> (c) freedom from minor defects,
>
> (d) safety, and

(e) durability.

(2C) The term implied by subsection (2) above does not extend to any matter making the quality of goods unsatisfactory -

(a) which is specifically drawn to the buyer's attention before the contract is made,

(b) where the buyer examines the goods before the contract is made, which that examination ought to reveal, or

(c) in the case of a contract for sale by sample, which would have been apparent on a reasonable examination of the sample."

SSGA94 s.2 also has this to say about acceptance:

"(4) The buyer is also deemed to have accepted the goods when after the lapse of a reasonable time he retains the goods without intimating to the seller that he has rejected them.

(5) The questions that are material in determining for the purposes of subsection (4) above whether a reasonable time has elapsed include whether the buyer has had a reasonable opportunity of examining the goods for the purpose mentioned in subsection (2.2) above."

In general, SGA79 will be used in the discussion that follows except where the above differences are relevant. (It can be noted parenthetically that so far most of the significant cases have arisen under the regime of SGA79 and in so far as any statutory guidance was used, that contained in SGA79 would have been relevant).

As was discussed in Chapter 1, the appearance of large amounts of software in basic consumer products such as washing machines, answering machines, televisions, cars and so on, mean that the consumer will be directly in the firing line of dubious software quality. Here various

relevant aspects of consumer law will be covered in the same order as discussed by a typical widely-used legal text, in this case, [29].

Classification of consumer contracts

Contracts for the Sale of Goods

SGA79 regulates only a contract for the sale (but not hire) of goods (and not services). Such a contract is defined as 'one by which the seller transfers or agrees to transfer the property in goods to the buyer for a money consideration, called the price'. The price must contain a money element to fall within the scope of SGA79. If the price is not fixed by previous dealings or the contract, it must be reasonable.

Whether or not software is considered as goods is a very simple question leading to a very complex answer which will be addressed later.

Contracts for the Supply of Services

Contracts for the supply of services (work and materials) are regulated by Part II of SGSA82, (Part I considers transactions for the hire of goods which will not be discussed further here, as software is not normally hired[13]). Section 12 defines a 'contract for the supply of a service' as 'a contract under which a person (the supplier) agrees to carry out a service'. Classification becomes difficult when a contract supplies a mixture of goods and services. The test used for classifying a transaction is: What is the essence of the transaction, the production of something to be sold (sale of goods) or the expenditure of skill and labour (work and materials) ?

The nature of the difficulties of classification from a software point of view is well-illustrated by the fact that painting a portrait, repairing a car and roofing a house have all been held to be supplies of services, whereas a meal in a restaurant is held to be a sale of goods. The production of software involves an intense intellectual activity such that the cost of the materials

[13] Actually, this is a non-trivial point. Although on the face of it, software is 'returned' if a customer ceases to comply with the terms, in practice, contracts generally require all copies of the software including

supplied with the software is *absolutely negligible* compared with the intellectual cost of developing the software. Furthermore, software is unique in that once written, it can be copied for a negligible cost - there is no further expenditure of work. In fact, in the increasingly common case of software downloaded from the Internet, there are typically *no* materials supplied either. The software is downloaded onto materials which are generally already the property of the person downloading the software. The downloaded material will normally contain the documentation also, which as often as not is intended to be read not by printing the material but is in HTML format (Hypertext Markup Language) in order to be read by a HTML browser, (a piece of software which can read HTML and convert it into something which can be read by a human, although HTML can actually be read unaided with some difficulty).

Analogies with the distribution of music will be analysed later but it would appear that *downloaded software contains neither work nor materials* which would seem to lead to distinct difficulties in so far as SGSA82 is concerned. On the face of it, it should be related to work, but any number of copies can be downloaded without any further expenditure of effort on behalf of the developer, so that each individual copy does not contain any work.

Ownership and risk

Ownership and risk is directly associated with the sale of goods as defined and governed by SGA79. In this sense, risk is the risk of loss. In English law, the transfer of the risk of loss from trader to consumer is intimately associated with the transfer of ownership. The time at which this takes place is therefore of considerable importance.

Although the rules for transfer of ownership are typically very complex, one of the principle requirements is that goods must be *ascertained*. In essence this means that goods must be identified in order to transfer ownership. For example, 'we will sell you one of the televisions

---

backups to be deleted in this eventuality. In other words, the copies are *destroyed* even though the

in this shop' does not ascertain goods. Section 18, Rule 1 of SGA79 states that in an unconditional contract for the sale of specific (i.e. ascertained) goods in a deliverable state, property passes when the contract is made even though payment and/or delivery may be delayed. In modern times, it is takes little for courts to construe that property passes only on delivery or payment.

The point at which software is delivered is an important issue which will be returned to later. However, it can be noted that software is never really ascertained. There is no specific copy which is 'sold' as it could be trivially copied and generally has no identifying marks, (even the licence number of licensed software is not attached to the copy but is usually part of a separate licensing mechanism). Bit-wise copies, (i.e. a faithful copy of every binary 1 or 0), are absolutely indistinguishable and could be produced many times a second with no user input for a typical piece of software.

To expand a little on licensing mechanisms, their distinction from the licensed software is emphasized by the fact that such mechanisms can be bought from third-party manufacturers. They essentially work as part of a client/server architecture. The licensed product starts up and communicates with a completely separate stand-alone server embodying the licensing mechanism. The licensing mechanism determines from some kind of database whether the request should be honoured and either denies or accedes to the request. The licensed product then acts on this command by either continuing or stopping. Some licensing mechanisms service several different licensed products simultaneously. Licensing mechanisms are generally as distinct from the products they licence as is the operating system which hosts both of them.

Pre-contractual statements and contractual terms

documentation might be returned. This is unprecedented in the hiring of goods.

Pre-contractual statements are statements made during negotiation prior to a contract. They are of particular importance in software because in general, nobody including its designers can describe the actual behaviour in precise detail. In such circumstances, misrepresentation of some form or another seems almost inevitable.

Pre-contractual statements take four forms:-

Trader's puff

Certain statements made by a trader are regarded as mere' puffs' as in 'whiter than white' or 'the greatest product in the universe' for example. However, if there is any intention to create legal consequences, the courts will seek to give legal effect to such statements.

Representations

A misrepresentation is a representation which is false. In effect. it is a false statement of fact which induces the other party to enter into a contract. A half-truth can be a misrepresentation and indeed silence, where there is a duty to disclose such as in a contract for insurance. Note that there is no duty on the consumer to verify the truth of a representation - it is enough that the statement was believed even if its falsehood could have been easily verified, (*Redgrave v. Hurd (1881) CA*).

There are three kinds of representation:- fraudulent, negligent and innocent. Fraudulent misrepresentation whereby statements are deliberately made in the knowledge that they are false or without caring whether they are true or false are likely to be infrequent in the sale of software. Fraud is also very difficult to prove in practice. However, negligent misrepresentation whereby false statements of fact are made negligently seem to be much more likely as demonstrated by *Mackenzie Patten v. British Olivetti (1984)* where the defendant supplied a system based on a salesman's promises which turned out to be false and the system was completely unusuable as a result. The court inferred that a collateral contract was in place and that the defendant was in breach and liable in damages. Given that the average

software developer cannot even predict how his or her piece of software will behave in a wide set of circumstances, it is to be expected that the supplier of piece of a COTS software will be particularly disadvantaged. The state of software engineering is sufficiently bad that almost any statement that they might make could be found wanting and each is generally made with the specific intention of enticing the consumer to buy a software product.

Collateral warranties

It is often difficult in practice to detect when a representation crosses the line into a binding promise. If it does however, it becomes a collateral warranty

It is worthwhile to ask when a pre-contractual statement amounts to more than a representation and be considered as a binding contractual stipulation. The courts will apply the test that it is not simply what the parties actually intended, but is what the intelligent bystander would infer as to the parties' intention from their words or contact. The court will readily infer such an intention where the trader states a fact which is or should be within their own sphere of knowledge and of which the consumer is ignorant, *Oscar Chess Ltd. v Williams (1957)*, or where the trader makes a promise about something which is or should be within their own control, *Dick Bentley Productions Ltd. v. Harold Smith (Motors) Ltd. (1965).* If it is a representation, then if the supplier can prove that he believed on reasonable grounds that his statement was true, he will not be liable in damages under s. 2(1) of he 1967 Misrepresentation Act. However, if his statement amounts to a collateral warranty, the supplier's honest and reasonably held belief will be no defence against an action for damages for breach of warranty.

This is deep water indeed. The author has personally returned products (for example, the route-planner Autoroute) which the supplier's literature and the supplier claimed would solve his route-planning problems. He subsequently discovered on using the software that there were so many defects in the software that it was positively misleading. The original

software producer was unable to correct these defects in a reasonable time in spite of issuing numerous releases and the author simply accepted his money back. A more assiduous user might well have got lost on several occasions. There is of course an obvious danger here that a broadly beneficial product will inevitably contain defects. The consumer must as always weigh the advantages against the disadvantages given that COTS software is often of rather poorer quality than the reasonable user might expect after experience with other consumer items.

Contractual terms

In contrast to the above, contractual terms take the forms:-

Conditions

A condition is a vital or major term in a contract, a breach of which allows the innocent party to consider the contract at an end. Note however that in Sale of Goods cases, the consumer's right to reject the goods and reclaim his money is subject to s. 11(4) of SGA79, which states that 'where a contract of sale is not severable and the buyer has accepted the goods or part of them', the buyer can only claim breach of condition by the seller and he cannot reject the goods. Note also that SSGA94, (s.2) clarified the notion of acceptance in the case of contracts dealing as a consumer whereby the right to examine goods could not be waived or otherwise lost.

The key word here is of course *acceptance*. Many COTS software suppliers invite the user to fill in a registration form, nominally to 'receive notification of future updates', although its main apparent intention seems to be to secure acceptance by the back door. Acceptance will be discussed in detail later.

Warranties

According to SGA79, a warranty is 'collateral to the main purpose' of a contract and is of minor importance. Breach of a warranty entitles the innocent party to damages only.

Intermediate stipulations

An intermediate stipulation is neither a warranty nor a condition but allows an innocent party to claim damages on breach and also to terminate a contract if the breach were sufficiently serious.

Implied terms

Statutory terms within SGA79

A key step forward for consumer rights was the 'statutory' or 'inalienable' rights enshrined in s. 12-15 of SGA79. In essence, these are:-

Title

The consumer is entitled to freedom from undisclosed encumbrances (i.e. a lien) and quiet possession of the goods.

Description

The goods including any samples must correspond with their description. In any 'sale by description', the description does not have to be a complete description of the product, but must materially fit the goods concerned. This would apply to any claim to compatibility. To describe a piece of software as compatible is to make an easily testable statement as to its behaviour on a particular platform. Any unexpected and undesirable behaviour would be considered to render it incompatible. An action based on misdescription is likely to be more successful for the consumer than one based on misrepresentation as it falls under SGA79 and the supplier cannot rely on a 'honest and reasonably held belief', as this is essentially a contractual term.

Merchantable / Satisfactory quality

This is a fundamental principle in English civil consumer law. It was originally enshrined in SGA79 s. 14(2) as:

> "Where the seller sells goods in the course of a business, there is an implied condition that the goods supplied under the contract are of merchantable quality, except that there is no such condition:

a) regarding defects drawn specifically to the buyer's attention before the contract is made, or

b) if the buyer examines the goods before the contract is made, as regards 'defects which that examination ought to reveal'.

This was subsequently modified by SSGA94 as described earlier in this section. In the above, the use of the word 'defect' was modified to 'any matter making the quality of goods unsatisfactory' and 'merchantable' was changed to 'satisfactory'.

a) above appears to offer a blanket way of escaping responsibility for the seller of software. For example, an unusually honest seller might say, "oh by the way, this product has numerous bugs and crashes frequently, although generally performs the function expected of it provided you save everything every 5 minutes." However, the use of the word 'specifically' should be noted. It is generally agreed that generic phrases such as 'damaged' or presumably, 'full of bugs' would not suffice. Rather specific defects should be pointed out. This would clearly be impossible for most software products even if the seller knew they were there as re-creating the conditions under which many software faults fail can be very difficult in practice. Note that for popular products, it is possible to get some idea of the most serious defects by consulting user groups and sometimes warnings issued by the manufacturer[14]. A seller of software is strongly recommended to be familiar with such sources of information.

As far as examination of the goods goes, there is no obligation on the buyer to examine anything. Moreover, the examination is subjective in the sense that it relates to what the actual examination might have revealed than what a reasonable examination would reveal. Clearly, the average seller would

[14] As I write this, I have in front of me a note published in the 27 August 1998 edition of *Computer Weekly* which describes a nasty defect in Word 97 which only showed up when users moved the underlying operating system from Windows 95 to Windows 98. In essence, the "AutoCorrect" option then causes Word 97 to crash losing all the user's changes. This was reported by users and Microsoft only then admitted to knowing of the problem but as yet have no fix. The work-around is not to use the option. An even better work-around is not to use Word.

be at a significant disadvantage in any examination otherwise, because most software products are so complex that it may be months or even years before a user becomes sufficiently fluent in their use to appraise the entire product as being of merchantable quality. For example, if the buyer was buying a compiler for a programming language. It could be a very long time before the buyer finally discovered a fatal flaw in the optimising part of the compiler, (the part that produces the most executably efficient form of the user's source code), rendering it effectively useless. It is very likely the seller would not have known this and the buyer may have made this requirement clear at the time of the contract. The law needs to accommodate possibly very long delays between delivery and a decision on merchantable quality and this will be discussed in detail later. The reader may recall from Chapter 1 that fully a third of faults take at least 5000 execution years to manifest themselves as failures.

Note also that that s.14(2) of SGA79 as modified by SSGA94 requires that goods 'are' of satisfactory quality, not 'can be made' of satisfactory quality. So SGA79/SSGA94 implies that if goods are unsatisfactory, the fact that they can easily be repaired is no defence. This can be compared with the decision in *Computing Ltd. v. Allied Collection Agencies Ltd (1989) (Court of Appeal)*, where Staughton LJ specifically made the comment

> "... software is not a commodity which is delivered once, only once, and once and for all, but one which will necessarily be accompanied by a degree of testing and modification. Naturally it could be expected that the supplier will carry out those tasks. He should have both the right and the duty to do so ..."

This appears to lead to the conundrum that on legal grounds, defects in packaged software should be treated differently to software with an element of service. From the point of view of a computer scientist however, they are the same. This point will be discussed later as will the issue of minor defects, which are supposed to be absent also according to SSGA94, but which will be present in just about any software, packaged or otherwise.

It is clear that just as in other aspects of law, what constitutes merchantable quality is likely to be very difficult to define satisfactorily for software.

Fitness for purpose

If the seller sells goods in the course of a business and the buyer, expressly or by implication, makes known to the seller (inter alia), any particular purpose for which the goods are being bought, there is an implied condition that the goods supplied under the contract are reasonably fit for that purpose. This is true whether or not that is a purpose for which such goods are commonly supplied except where the circumstances show that they buyer did not rely on the seller or that it would have been unreasonable for him to do so.

There is considerable overlap in practice between fitness for purpose and merchantable quality. Given that merchantable quality may be a little difficult to pin down for software, it may be particularly important for a consumer to make known precise requirements for his software before the contract is made. (The onus of proof is on the seller to show that the buyer did not rely, or that is was unreasonable for him to rely on the seller's skill or judgement. On the other hand, reasonably fit for purpose is enough, (for example (*Heil v. Hedges, (1951)*), wherein pork chops containing a parasitic worm were considered reasonably fit for eating because proper cooking would have killed them. It is not revealed whether their Lordships would have been keen on eating the result however.)

Durability

This is of considerable interest to purchasers of software. SGA79/SSGA94 contains no provisions as to durability, although it is recognised in the courts, (e.g. *Crowther v. Shannon Motor Co. (1975)*), that if something fails very soon after purchase, it is likely that the offending defect was present at the time of purchase.

Although there has been some progress in this area, an essential feature of software is that any defects *have always been present*. It is unprecedented

in English law for goods to have an effectively infinite lifetime in precisely the same condition and this will have to be considered carefully along with the issue mentioned earlier that a significant class of defects may take a very long time to manifest themselves. This is discussed in much more detail in Chapter 4.

Statutory terms within SGSA82

This provides a number of implied terms however, only the following will be discussed:-

Care and skill

There is an implied term that for any services component, they will be supplied with 'reasonable skill and care' in contrast to the requirement for satisfactory quality and fitness for purpose of a goods component.

This is clearly not very stringent. It is certainly true that the professional or skilled person, and this includes software engineers, (although not in the United States !), has to exercise the ordinary skill of an ordinary competent person exercising that particular profession or art, (e.g. *Bolam v. Friern Hospital Management Committee (1957)*). There is no implied warranty that the professional person will achieve the desired result however. 'The surgeon does not warrant that he will save the patient. Nor does the solicitor warrant that he will win the case', as Lord Denning MR stated in *Greaves & Co. (Contractors) Ltd. v. Baynham Meickle & Partners (1975)*).

Particularly in view of the fact that a professional person does not have to warrant success, it would be very desirable for software suppliers to have their wares treated under SGSA82 rather than SGA79/SSGA94. However, this does not look promising given the ruling in St. Albans v. ICL discussed in Chapter 3, whereby a term relating to merchantable quality was implied in what was essentially a contract for services. This is analogous to the legally acceptable doctrine described above that a surgeon does not warrant to save the patient but if the surgeon makes a mistake, that is a different matter.

It should also be noted that in a discipline as poorly defined as software engineering, very wide variations in 'expert opinion' are to be expected. As a case in point, the reader should note the differences in *Saphena Computing v. Allied Collection Agencies*, where the court specifically commented on the relative quality of the testimony of the two expert witnesses.

Legal control of exemption clauses

The concept of *consensus ad idem* historically forms the basis of contract law. In essence, if somebody is prepared to accept contractual obligations, the law will not intrude. Since 1977, with the advent of UCTA77, the law seeks to protect weaker parties against unreasonable contract terms, however, certain basic techniques still apply. These are:-

Incorporation

Generally, when a consumer signs a contract, he will be bound by its terms, and he cannot argue that he did not read the contract or understand it, *(L'Estrange v. Graucob (1934) CA).* However, when printed conditions (i.e. exclusions) are not regarded by the courts as constituting a contractual document, they are not incorporated into the contract and so are ineffective.

Note particularly that even if an exclusion clause is incorporated into a contract, it cannot by reason of the doctrine of privity of contract, benefit a person not a party to the contract, (s. 9(2)). Thus a negligent employee or agent will not normally be protected by an exclusion clause incorporated into a contract between his employer and the consumer, even though the clause purports to cover employees, *(Adler v. Dickson (1955) CA).* This would seem to have interesting ramifications of liability for software engineers employed to produce a product but who do a demonstrably inadequate job. Of course, ascribing any particular part of a software project to a particular employee presupposes the existence of a reasonable change, configuration and revision control system as discussed in Chapter 1. This is traceability with a vengeance.

Misrepresentation

If the effect of an exemption clause is misrepresented to the consumer, then the clause will be ineffective. Likewise, an oral warranty or assurance made to the consumer may bind the trader and overrule any exemption clause contained in the written contract, (*Curtis v, Chemical Cleaning and Dyeing Co. (1951)), J. Evans & son (Portsmouth) Ltd. v. Andrea Merzario Ltd.*).

Many software contracts still attempt to exclude all liability howsoever generated. It is quite conceivable that a trader could make statements which would greatly weaken this position even supposing the courts found it did not constitute unreasonable terms.

Contra Preferentum rule

The courts do not like exemption clauses and any exclusion clause will normally be construed against the person putting it forward. However, clauses seeking to limit liability are more generously received than claims which attempt to exclude all liability.

(The reader may note however that a clause seeking to limit liability in *St. Albans v. ICL (1996)*, c.f. Appendix, was still rejected as being insufficient). Furthermore, *Salvage Association v. CAP Financial Services Ltd (1990)* gives a detailed description of how not to do it. In this case, confusion within the defendant itself caused an out-of-date and very inadequate limit of £25,000 to appear rather than its updated amount of £1,000,000, a factor of 40 greater. This did not impress the court.

An interesting form of exclusion clause appears in the current Microsoft contract for its product Windows NT 4.0, in which it specifically excludes any liability for the embedded Java parser. Java is a programming language widely used to add a movie and sound capability to Web pages, *inter alia*. Although Java has a publicly defined definition, the Java parser is designed and implemented by Microsoft. They appear to be seeking to avoid any liability for this implementation. This is akin to excluding any liability for an

implementation of any external standard, for example, the CCITT telecommunications standards.

Fundamental breach

The doctrine of fundamental breach asserts that any contractual breach so fundamental as to defeat the main purpose of the contract cannot be excluded. In practice, the advent of UCTA77 effectively subverts the doctrine entirely.

It should perhaps be noted that where UCTA77 makes an exemption clause totally void, the Act is all that the consumer need rely on. However, where the Act subjects an exemption clause to the reasonableness test, the consumer might be better placed under the common law where the common law strikes down the clause altogether, than to find the clause incorporated whereby the trader might succeed in satisfying the reasonableness test.

With regard to UCTA77, it should be pointed out that the burden of proving the reasonableness of an exemption clause falls on the party seeking to rely on that clause. It should also be pointed out that the consumer in UCTA77 could be a business in a number of significant circumstances, (rather simplistically a consumer contributes nothing to the contract).

A further layer of statutory controls over contracts specifically between a supplier or seller and a consumer appeared with the introduction of the EC Directive on Unfair Terms in Consumer Contracts (Directive 93/13/EEC, OJ L95, 21 April 1993), which was implemented in the UK as the Unfair Terms in Consumer Contracts Regulations 1994 (SI 1994/3159). This Directive provides that in a contract between supplier or seller and a consumer, unfair terms shall not be enforceable against the consumer. In this Directive, a term is defined as being unfair if

(a) it has not been individually negotiated,

(b) it is 'contrary to the requirement of good faith, it causes a significant imbalance in the parties' rights and obligations arising under the contract, to the detriment of the consumer', (art. 3(1)).

In addition, the annex to the Directive contains a non-exhaustive list of terms which 'may be regarded as unfair'. This does not mean to say that they will but as a general note to suppliers or sellers, the presence of any of these is likely to invite close scrutiny.

Unlike UCTA77, the Directive is not relevant to contracts between a supplier or seller and a business.

Performance of consumer contracts for the Sale of Goods

This section is of considerable interest in that it casts some light on a notoriously difficult area in software trading, viz. when is software actually delivered. Before doing this the position of a consumer in UCTA77 will be briefly reviewed. In essence, a buyer deals as a consumer under s. 12 of UCTA77 if:-

(a) He does not buy in the course of a business, *and*

(b) The seller sells in the course of a business, *and*

(c) The goods are of a type normally supplied for private use or consumption.

As Reed points out in Chapter 1 of [28], (c) might have been problematic since certainly hardware and perhaps software also might have been considered as not belonging to this category. *He goes onto say that any statement on this would quickly go out of date, however today, the sophisticated nature of the graphics and CPU power requirements of the average interactive game mean that both computer hardware and software for consumer systems are substantially identical in both performance and complexity to typical business products and are very likely to be running the same operating system, so there would be no question of a supplier*

escaping responsibility to a consumer by virtue of disclaiming their status under (c) above.

Delivery

Delivery is defined as 'the voluntary transfer of possession from one person to another' in s. 61, SGA79. It is usually affected by a simple physical transfer of goods to the consumer.

It is attractive to think of the delivery of software as taking place when the disc or CD containing it is handed over, however this is far too limited to take into account software, which could be physically sent by e-mail on the seller's instigation or downloaded on the buyer's instigation. In either circumstance, **any** computer scientist would argue that the software had been delivered **if** accompanied by the appropriate documentation whereby the software could be invoked, **and**, if the software was accompanied by a licensing system, the appropriate key such that the licensing system allowed the software to be run. This is analogous to the transfer of the means of control as for example, when the ignition key is handed over with a car, an equally acceptable alternative in the eyes of the law. It is therefore misleading to associate delivery of software only with the transfer of a tangible commodity such as floppy disc or a CD. The reader should also note at this point, the relationship with the ascertainment of goods as described earlier where it was stated that software is never really ascertained as each copy is absolutely identical with no means of being distinguished from any other copy.

Time of delivery

The time of delivery of goods is not generally 'of the essence', (i.e. incorporated into the contract). In this case, if delivery is delayed unexpectedly, the consumer can with reasonable extended notice, make it of the essence such that if delivery still does not take place, this is a breach of a condition of the contract and the buyer is entitled to repudiate the

contract. A similar rule applies to contracts for the supply of services, c.f. *Charles Rickards Ltd. v. Oppenheim (1950)*.

This is particularly relevant for software developers. Software systems which contain a bespoke element are late more often than they are on time as can be seen from the statistics quoted in the introduction. If the buyer gives appropriate extended notice requiring delivery, this will be incorporated into the contract, (i.e. it will act as if 'of the essence'). This may of course may not be the most attractive solution for the buyer, but given that a very high percentage of bespoke systems which are late never get delivered at all, it may well be the lowest risk avenue the buyer can take. The seller loses out of course by losing whatever resources have been expended so far, but that's show biz. as they say.

Delivery by instalments

The buyer was under no obligation to accept only part of the goods under s. 30(1) of SGA79 and this has been further clarified in SSGA94. Further more, unless otherwise agreed, the buyer is under no obligation to accept delivery by instalments, s. 31(1), SGA79.

This is peculiarly relevant for software. The reader may recall from the discussion in Chapter 1 that software has a somewhat artificial separation of its life-cycle into Development and Maintenance. The idea is that software is developed, (i.e. finished) and then supplied, (i.e. maintained). Rarely has the use of English been stretched to such breaking point. In point of fact, bespoke and modified software systems are almost invariably supplied in instalments as the system gradually (and hopefully) converges to something the buyer requires. What is initially the product of the so-called development phase is very rarely what the buyer actually wanted or indeed finishes up with. Buyers are used to this particularly as they are frequently at least partially culpable as a result of being unable to define precisely what they want in the first place. It is interesting to consider this in the light of the comments made by the court in *Saphena Computing v.*

Allied Collection Agencies as discussed in Chapter 3, whereby software can be expected to be delivered with faults in which the buyer is under obligation to give the seller time to correct. This surely is delivery by instalments by any other name. The formal name for delivery by instalments is *incremental delivery* and it is considered by many to be a key feature of a successful product given the lamentable performance of its opposite, the so-called *big-bang approach*. Here we have an example whereby a delivery strategy which greatly facilitates the delivery of a successful product conflicts with the legal position.

Acceptance

Acceptance and delivery are of course related, but delivery is a necessary but not sufficient condition for acceptance. Acceptance takes place when:-

a) The consumer tells the seller that he has accepted the goods or

b) The goods are delivered and the consumer either does an act inconsistent with the seller's ownership, or he retains the goods beyond a reasonable time without telling the seller that he rejects them, (s. 35, SGA79 and some clarifications in SSGA94).

Note of course that if the consumer signs some form of acceptance note on delivery, this would be deemed acceptance and would very likely put the consumer beyond help of UCTA77 for example as such an acceptance note would not be considered a 'contract term' within the scope of the Act.

This is particularly murky water for software because the seller only sells a licence. Even with packaged software for which there would be a temptation to treat as goods, the media content of the package is generally irrelevant. Furthermore, if the user is given a demonstration copy, his pattern of use would be indistinguishable from the full copy unless some particular function was temporarily disabled. Software of course shows no wear or outward signs of being used further complicating the issue.

Software suppliers frequently exhort users to sign a registration document. The claimed rationale behind this is to inform the consumer of new products, updates and so on. However, there is a risk that it might be construed as a form of acceptance.

Examination of the goods

The consumer is not deemed to have accepted goods unless and until he has examined them or has had a reasonable opportunity of examining them, s. 34, SGA79 with clarifications in SSGA94

This is also murky water. We must ask the question what constitutes a reasonable time to examine a piece of software for satisfactory performance. In *Bernstein v. Pamson Motors (Golders Green) Ltd.,* the emergence of defects three weeks after delivery of a car was not deemed sufficient cause for the rejection of the product as the car had then been accepted. Software is now so complicated that such examination is likely to take a very considerable time and expend significant resources on behalf of the consumer[15]. Indeed, if test coverage, (the percentage of source code exercised by a particular set of tests) is a reasonable criteria for examining the functionality of a piece of software, the reader will recall that even the developers do not examine the software in much detail, (only around 40% of all source code statements of a program are exercised by any of the tests carried out prior to release, [9]).

The most reasonable solution from the point of view of the consumer is that given that software does not wear out, that the consumer should be entitled to return a piece of errant software at any time after purchase if it fails to perform a critical function. To restore a reasonable parity, the courts could perhaps consider the trade-off with any benefits which accrued during the same period. There could be none or many, depending on the software. For example, supposing the user purchased a word-processor and used it

[15] We have gone full circle here. A modern car contains several hundreds of thousands of lines of software so that the judgement in Bernstein may become irrelevant in the near future simply because of technological advance.

extensively producing many files before discovering eventually that it failed to carry out a critical function which he understood it to perform. If the user simply returns the software, he would not be able to read any of the files from then on, so there may be no actual benefit.

Effects of acceptance

Once goods have been accepted in a non-severable contract, even grounds of rejection based on the statutory rights enshrined in SGA79 are insufficient. The consumer may be left with a claim for damages only.

Acceptance of goods having latent defect

As has been discussed above, by the time a latent defect comes to light, the consumer may be held to have accepted goods on the basis of having had them a reasonable time without notifying the seller that he has rejected them. Once again the consumer could only claim for damages, however, it could be argued:-

a) that use of the product is the only method of giving it a reasonable examination.

b) that use of the product would not be inconsistent with the seller's ownership unless it had the effect that the consumer could no longer return the goods in substantially the same condition as when purchased.

Software clearly falls into both categories. First of all, given the widespread differences between written claims as to the functionality of software and its actual behaviour, a) is unquestionably true. Similarly, use of the software in no way affects the condition of the software itself, as it will be in precisely the same condition as when purchased, so even substantial use of the product would not prejudice the seller's ownership with one proviso:- software is upgraded by its original suppliers on a fairly regular basis to include new features, correct defects and so on, so the market value of a

prior release will be diminished at this point often to the point of being valueless.

Repairs

We have already seen from the ruling in *Saphena* that a supplier should be given the right to correct any defects notified in a product in a reasonable time. There is a danger that the courts could argue that repeatedly doing this could lead to the goods being deemed to have been accepted.

This is relevant to software in that it is likely to contain numerous defects which the consumer would like to be fixed. In which case, the consumer should make it clear that if repairs are agreed upon, that failure to carry them out effectively, (a significant risk for software as pointed out in Chapter 1), would be without prejudice to the right to reject.

Note finally that the common act of replacing a defective item is completely useless in the case of software. All copies are similarly flawed.

Remedies

In this section, only the position from the point of view of the trader being at fault will be considered.

Damages

The basic remedy for breach of contract is damages. The object of damages is to compensate the innocent party in monetary terms in so far as money can do this, and thereby to put him in the same position he would have enjoyed had the contract been duly performed. It used to be thought from *Addis v. Gramophone Co. Ltd. (1909)* that damages for breach of contract were confined to compensation for financial loss, that is loss quantifiable in monetary terms. However a series of cases beginning with *Jarvis v. Swans Tours Ltd. (1973) CA* has allowed moderate damages in compensation for disappointment and distress suffered by the innocent party.

Given the not unknown complete corruption of file systems and subsequent loss of everything, given frequently inadequate back-up procedures, which can occur with off the shelf products such as Windows ‘95, it would not take much to convince a court of ‘disappointment and distress’.

As a final point, however, it is worth noting that where there is contractual liability for defective software, the position will be very similar to that in negligence as discussed next, except where this liability arises from the express terms of the contract or the terms implied into contracts for the supply of goods. Note however that liability is strict (i.e. independent of fault) in contracts for the sale of goods and also for product liability in comparison with tort where it is not strict.

Delictual Liability

This area will also be considered later when a model of delictual liability and its relationship to duty of care is discussed. The whole area is somewhat theoretical as there have to date been no cases in which liability for software defect has been decided in delict. For now, it will simply be noted that delictual liability arises by definition from negligence and the discussion will follow a typical up-to-date legal viewpoint as advocated by Antony Garrod in [23].

Delictual liability can be usefully subdivided into:-

- Physical injury or property damage
- Negligence claims for financial loss
  - Consequential losses because software is unusable
  - Loss caused by reliance on unreliable information, for example a financial loss which might result from a defect in a piece of financial modelling software.

Negligence is "conduct falling below the requisite standard to protect others against unreasonable risks of harm". Everybody has obligations to third parties and delictual liability deals with those relationships which fall

outside the law of contract, the area of delict most relevant to software being negligence. This does not mean to say that in either business or private lives, people are strictly liable i.e. liable without fault, for the consequences of all of their actions. This is covered by the area of statutory liability to be discussed shortly. On the other hand, it would be reasonable to expect that manufacturers of defective products including defective software, have some liability for damage caused to consumers.

Tests have been developed by the courts to achieve a sensible balance leading to the following questions which can be:

- is there a duty to take care ?
- what is the standard of care ?
- did the defect cause the damage ?
- could the ordinary man have foreseen that this damage might result from this cause ?

Duty of Care

First and foremost, a duty of care is owed to the subject's *neighbour in law*. The "neighbour in law" is:

"Those persons who are so closely and directly affected by the subject's act that the subject ought reasonably to have them in contemplation as being so affected when directing his or her mind to the acts or omissions which are called into question".

In essence, this means those people who the subject could reasonably have foreseen being adversely affected. In 1932 the House of Lords first held a manufacturer owed a duty of care to the ultimate consumer and was therefore liable in negligence for a defective *product*. In the case of *Donoghue v. Stephenson (1932), AC 562* a woman allegedly drank ginger beer from an opaque bottle which she subsequently discovered contained a decomposing snail. She was later ill and since the court found that

decomposing snails had no place in ginger beer bottles, and that the manufacturer had a duty of care to somebody who might drink its ginger beer, the unfortunate lady was awarded damages.

Standard of Care

In delict, the standard of care would be that expected of a reasonably competent person. In areas of greater risk, the standard imposed by the court is likely to be higher. However, given that standards are currently not very high in software development as described in Chapter 1 to the point where it is hard even to define the concept of a reasonably competent programmer[16], it is highly desirable that there be a development in the law requiring higher, more professional standards to be exercised by producers. This cannot be addressed by delict however, and should be addressed by contract. If a higher standard is desired, it should be contractually required.

To see that standards change, for example, would the reader expect the same standard of care from a doctor at the beginning of the century as now ?

In the U.K., the British Computer Society has in the last few years become a member of the Engineering Council, which governs standards in various engineering disciplines, and suitably qualified software developers can as a result, be granted the Chartered Engineer[17] qualification (C.Eng.). Amongst other things, this has recently enabled developers to get professional indemnity insurance for probably the first time, so evidently the insurance companies believe in the screening process[18] !

---

[16] This can be illustrated by the somewhat 'tongue-in-cheek' but nevertheless accurate duality in which a person with 10 years experience is frequently described in the industry - 10 years experience or 1 year 10 times ... The distinction is quite clear to an engineer but is likely to be lost in a courtroom.

[17] The British Computer Society now recommend that the quality of every safety-related computer system be the responsibility of a named engineer holding this qualification or its equivalent, as was discussed in Chapter 1. The equivalent European qualification is the Eur.Ing. which is available to those British holders of the C.Eng. who have spurned a centuries old tradition of not being able to speak any language other than English.

[18] Although not for engineers involved in Year 2000 work which is specifically excluded !

This is in stark contrast to the U.S., where the principle of professional malpractice requires the observance of a higher duty of care from members of recognised professions, and to which software engineering does not yet belong. As a result, a number of cases have been dismissed by the courts on the basis of a refusal to accept the concept of computer malpractice. No doubt this will be reviewed as similar initiatives to the C.Eng. qualification spread to the U.S. In fact, a draft international standard for the use of computer-based products in safety-related systems, IEC 61508, specifically provides for the notion that as the potential for hazardous effects of a system on its end-user increase, so there is an increasing requirement for more sophisticated development techniques to be used for its implementation. In other words, developers are encouraged to recognise the duty of care from the beginning in terms of five levels of increasing integrity, (known as SIL levels 0-4 or System Integrity Levels) and IEC 61508 then attempts to define (with varying success it has to be said), the appropriate standard of care. Whilst not perfect, it is an important step in the right direction and has a close relationship with legal models.

Another example illustrating by way of analogy as to how the law develops with respect to changing views of acceptable practice, concerns the attitude to the wearing of safety belts. If drivers or passengers were involved in an accident and suffered damage as a result of another road user's negligence, then they would be awarded damages. However, Lord Denning in the English Court of Appeal found that an occupant of a car involved in an accident caused by somebody else's negligence *contributed towards their own injuries by failing to wear a safety belt and the damages awarded were reduced by 20*% from what they would otherwise have been. Now it is a criminal offence not to wear a safety belt in the U.K. The same holds true in many other countries.

It is enlightening to compare the concept of standard of care as it has arisen in well-known cases and apparently highly relevant cases in maritime law. Three important cases have occurred, the U.S. case of the

T.J. Hooper, and the cases of the *Lady Gwendolen* and the *Marion*, heard in the English courts. Each case concerns the failure of process in the sense of that defined in Chapter 1, and the issues are summarised in the following table 3 to avoid going into too much detail.

Case	Result	Software Analogy
T.J. Hooper (U.S.)	Failure to provide radio led to a storm sinking two barges which they could have avoided.	Failure to supply tools or process preventing a known class of defect.
Lady Gwendolen (U.K.)	Radar was provided but no steps were taken by the employer to ensure its proper use even though the employer was aware of regular transgressions from the evidence of logs which showed the ships were charging about in fog, leading to a collision.	Tools are provided to avoid well-known problems, but their use is not mandated even though evidence exists to show that they are not being used. The developers are charging round in a software fog.
Marion (U.K.)	Ship's anchor hit pipeline unmarked on out of date charts. The court ruled that it is the owner's duty to ensure that up to date charts are available and used for navigation.	Senior Management's duty to make sure standards and other guiding documentation is up to date before the project starts and that they are used throughout the project.

Table 3. Some cases from maritime law which may provide relevant legal guidance.

To conclude, in software development, if a regular agreed set of checks are verifiably used, the developer would appear to be in a far better position to demonstrate it had discharged its duty of care than if not. If this is the case, it would certainly help justify the presence of formal quality systems such as those based around ISO 9001, but, and this is a very big but, this is not sufficient to avoid responsibility, as evidenced in *St. Albans v. ICL (1996),* (discussed in Appendix A), whereby ICL's ISO accreditation was not to the

author's knowledge mentioned at any stage of the proceedings, even by the defence.

Defect and damage caused

It is normally a question of fact whether or not the defect caused damage, although development of artificial intelligence which either directly or indirectly causes damage, will further complicate the issue. If there is no factual link between the defect and the damage, there will be no liability in delict. It is also necessary to recognise that the question of liability may well be affected by the type of damage caused. If there is physical injury or property damage, this is usually straightforward and the test for an existence of duty of care will be that used in *Donoghue v. Stevenson (1932).* If, as is likely, damage is restricted to economic loss, the type of damage is an important factor.

Foreseeability

This revolves around whether the reasonable man or woman could have foreseen that the particular fault could have caused the damage. There is a famous case which helps delineate this. Some years ago, there was an oil spillage in Sydney Harbour and some workmen were operating an oxyacetylene torch on a pier. The foreman on hearing of the spillage first instructed them to stop work but after a brief inspection told them to continue. Molten metal from the oxyacetylene torch then dripped from the pier and instead of falling into the water, landed on a piece of driftwood covered in cotton waste which ignited and then set fire to the oil. This fire in turn destroyed the pier and ships. Not surprisingly, the Privy Council held that the ordinary man or woman would not have foreseen that particular damage could have been caused, *The Wagon Mound (No. 1) (1961)* ! This can be distinguished from the so-called 'egg-shell skull' cases where damage is foreseen but not its full extent.

This may well turn out to be an important defence in any software case as the connection between a software fault and the failure it causes is

frequently very tenuous and would have been very difficult indeed to foresee or even if foreseen, the full extent of the failure would not be apparent[19]. In fact, as described in [22], a significant percentage of failures, 35% in their case could not be ascribed to any particular fault. Indeed computer science has no satisfactory model whereby a particular fault can be predicted to lead to a certain kind of failure, without actually running the software to see what happens. The reverse is also true that there is no model whereby a failure can be related to one or more responsible faults, although this is compounded by a laissez-faire attitude to the inclusion of standard diagnostic procedures which amounts to negligence in many software systems. The reader may recall the exceedingly complex relationship between fault and failure investigated by [18] and described in Chapter 1.

Physical loss v. Economic loss

The examples of negligence which have been discussed so far relate to physical loss or damage. However, it is likely that loss caused by defective software will be economic loss. Where economic loss such as loss of profit is consequent upon physical loss or damage, then the courts may award this. The case of *Martin v. Spartan Steel Alloys* illustrates this point well. Workmen hit an electricity cable whilst digging the road. The local electricity board disconnected the cable as a result of which Spartan Steel had to close down their electrical furnace in which they were smelting stainless steel, around the clock. To save the furnace they injected oxygen into the metal to enable it to be poured out and were successful in claiming damages for the metal, the loss of profit on that smelt but were not allowed loss of profit on four other smelts which they could have carried out before the electricity was turned back on again. Lord Denning and Lord Wilberforce both said that the loss of profit on the lost smelts were equally

[19] Consider for example the AT&T failure in 1990 discussed in Chapter 1. Any software engineer would have foreseen a failure of some kind given the nature of the fault, but very few if any software engineers would have ventured the opinion that the fault could collapse the whole of the US long-distance telephone network within a few minutes which is precisely what happened.

as foreseeable as the loss of profit on the damaged metal *but, as a matter of policy, they drew the line on economic loss consequent upon physical damage*. This is in some contrast to the case of *Junior Books Ltd. v. Veitchi Ltd. (1983)* where the House of Lords found for the plaintiff ruling that when there was sufficiently close relationship between the plaintiff and the defendant, that there could be a duty of care to avoid financial losses. In this case, a floor laid by the defendant proved defective and the plaintiff was able to recover both the costs of replacing the floor and lost profits while it was being relaid.

The courts have awarded damages for pure economic loss caused by negligence where there has been a special relationship between the person causing the loss and the person suffering it, or where it can be demonstrated that the person suffering loss relied upon the person causing it exercising a particular skill.

There were a number of cases in England in the 60's and 70's which indicated that where a professional man or woman gives advice which affects the safety of buildings, machines or materials, his or her duty is to all who may suffer loss. However, in a recent case in the House of Lords, *Murphy v. Brentwood District Council*, these cases have been largely overturned. Lord Harwich made the following points in relation to the difference between dangerous defects and defects of quality:

- When a manufacturer negligently puts into circulation a product containing a latent defect which renders it dangerous to persons or property, he will be liable in tort for injury to persons or damage to property which that product causes.

  This is particularly relevant to safety-related development where, by definition, failures in product may be dangerous to persons or property.

- If a manufacturer produces and sells a product which is merely defective in quality i.e. it does not cause loss or damage - even if it is valueless for the purpose for which it is intended - the manufacturer is only liable at common law under contract - the common law does not impose any such liability in tort except where there is a special relationship or proximity (as in *Hedley Byrne v. Heller*) imposing on the manufacturer a duty of care to safeguard the user from economic loss.

- No such special relationship exists between a manufacturer and a remote user.

The above would seem to suggest that, if you discover a defect in COTS software which would render it unusable, e.g. because it could cause personal injury or damage, this is a defect merely in quality and no liability lies in delict because of the absence of a special relationship.

So far, both physical loss and damage and consequential economic loss caused by software have been considered. However, the situation when software corrupts data which is used or relied upon by a third party who as a result, suffers loss, has not yet been considered.

Consider for example, the position of accountants or auditors who negligently produce accounts. They would clearly have liability both in contract and in negligence to their client if they suffer loss. It is reasonable to ask if they are also liable to shareholders or third parties who rely upon the accounts for valuing shares in the company which they purchased. This of course is an example of economic loss.

In *Caparo Industries v. Dickman* the court added to the list of tests for negligence that of the reasonableness or otherwise of imposing a duty of care.

In this particular case, the financial accounts were in more or less general circulation and may foreseeably have been relied upon by

strangers for one of a number of different purposes which the auditors had no specific reason to anticipate. There was not sufficient proximity in the relationship between them and the person relying on the statement unless it could be shown that the auditors knew that the accounts would be communicated to the person relying on it, either as an individual or a member of an identifiable class, specifically in connection with the particular transaction, or transactions of a particular kind and that the person would be very likely to rely on the accounts for the purposes of deciding whether to enter into a transaction. In this case, it was ruled that the auditors had no duty of care to the public at large who relied on the accounts to buy shares in the company.

The above seems to suggest that apart from the immediate client, if software is written with the knowledge that it is going to be used by another person or a group of people, e.g. a trade association and it is known that they are going to rely upon the software for the particular purpose for which it has been developed, then there may well be liability to those third parties for economic loss. On the other hand, if software is written to be issued generally to the public at large then, outside of the contractual liability, unless it can be shown that there is a special relationship or close proximity with the customer, the software developer is unlikely to be liable under delict for economic loss caused by the software. This does leave at least one area where the water is very muddy indeed. Supposing a company issues an accountancy package with amongst other things a claim that it is suitable for small businesses, but the software is issued as a COTS package. Then the package is being issued to the public at large although it is clear that a section of its intended users will depend on its correct performance. What then is the position ? In this case, it would seem that, although issued to the public the mere functionality of the program suggests a close link with the intended user even though anonymous. *This would presumably be amplified if the company solicited customers for a software support service for a fee, thus inviting them into a close relationship*.

Statutory Liability

This is liability under Act of Parliament and follows directly from the European Product Liability Act of (1985) as embodied in the Consumer Protection Act of 1987, (CPA87). It covers only damage to person or property and it is far from clear whether software actually falls within its area of influence. The Act provides that subject to the remaining provisions of the Act:-

> "... where any damage is caused wholly or partly by a defect in a product, every person to whom subsection (2) below applies shall be liable for the damage."

The first and most obvious question to ask is does this Act apply to software. This is a far from trivial question. Is software a product ? This is not related to the "Software as goods or services" issue described later because the Act defines a product as 'any goods or electricity' including components. As a former physicist, this immediately baffles me as electricity is certainly not a product in any tangible sense. Indeed, it is undetectable apart from its effects. Furthermore, it has been known since the 19th century through pioneering work by James Clerk Maxwell and numerous distinguished predecessors such as Oersted and Edison that electricity in one of its manifestations is magnetism. Now software is stored on a floppy disc as a modulation of magnetic fields. In order for a computer to read the software on a floppy disc, the floppy disc spins. In other words, the software then appears to the reading heads of the floppy disc drive as a variation in time of a magnetic field. According to Maxwell's Laws of electricity and magnetism, a time-variant magnetic field *is* also an electrical field. Similarly when software is transmitted down a communications line it also appears as a modulation of electric and magnetic fields. Clearly then software is 'electricity' in both cases and any attempt to distinguish between even these is doomed by the laws of physics.

The Act goes on to define ‘goods’ as including inter alia, ‘substances’ which are then defined in a magnificent piece of circularity as ‘natural or artificial substances’. Some legal authorities such as Reed in [28] in a very detailed review have taken this to infer that software on a floppy disc or tangible medium is ‘goods’ as ownership of this tangible manifestation is transferred, whereas software installed by copying from some other source would lack the necessary tangibility. The author must admit to disagreeing with this viewpoint. Given the choice, he would argue that software is part of this Act by virtue of the admissibility of electricity independent of the method in which it is supplied. Having said this, he would also argue that the wording in its present form is effectively useless and needs clarifying considerably for the needs of the digital age. Perhaps a wording to the effect that ‘any goods or electricity or digital information howsoever stored or transmitted’ would be more appropriate. Given that software manifestly can harm people, (c.f. for example the Therac-25 incident described in Chapter 1 whereby 6 people were massively overdosed by an active radiological scanner between 1985-1987 leading to 3 of them dying of their injuries), this would be more appropriate than excluding it.

Other authors such as [30] have been rather more evasive about the relationship with software and have stated simply that software ‘does not lie outside the scope of the Directive’. It is the author’s opinion and not to put too fine a point on it, that this kind of legal equivocation needs to stop and suitable wording drafted to protect society soon as it becomes increasingly reliant on and exposed to software controlled systems so that when the inevitable disasters occur, innocent parties are appropriately protected and the attention of software suppliers is more forcefully directed to the need for reliable and safe software systems.

Assuming that the Act is indeed relevant, it contains two elements of particular interest to software developers.

First of all, it provides for *strict liability*, i.e. liability without fault, in contrast to delictual liability where negligence has to be proved by the

plaintiff. In essence, the Act implies a statutory partnership of producer / supplier / retailer, all of whom are jointly and severally liable for damage (which means death, personal injury or damage to private property caused by a defect, i.e. a fault rendering a product less safe than it could reasonably be expected to be).

Second, it provides for a number of defences under the Consumer Protection Act 1987. One of particular interest to software developers is that relating to state of the art knowledge, also known as the *development risks defence*, [31]. In essence, this states:

"... the state of scientific and technical knowledge at the relevant time was not such that a producer of products of the same description as the product in question might be expected to have discovered the defect if it had existed in his products while they were under his (or her) control."

In essence, this means that the onus is on the producer to prove that no producer of the same products could have avoided a particular defect. So as [31] points out, such a defence would *not* cover failures in a quality system which led to a software producer releasing software containing say some known frequency of defects. In other words, the producer is knowingly releasing a defective product, but without negligence. It appears that this defence is applicable only if:

- the defect arises from something unforeseen at the time of production; or
- the defects are foreseeable, but that current technology does not allow the elimination of risk, for example, on economic grounds.

Given the known incidence of a class of entirely avoidable failures as described in Chapter 1 for example, it would be unlikely for a producer of a safety-related system to succeed in a claim that faults, statically detectable by a number of tools described in [23] for example, could not economically be removed. It is equally unlikely that the producer could argue that a non-

zero occurrence rate of faults during unit testing, dynamically detectable by the tools also described in [23], was reasonable. *The state of the art is quite clear, and it seems therefore that the presence of such faults could not reasonably be argued to fall within the development risks defence.*

The clear nature of such avoidable failures was emphasised in the BSI debate described in [23] which moved that ISO 9001 certified companies should not release product with statically detectable faults. Even if a particular software package had a much lower than average, but nevertheless non-zero frequency of such a fault, this would not be a defence as was proven in the *Smedleys v Breed* case, which involved the presence of a caterpillar in a can of peas. A defence to the extent that only four complaints had arisen in an annual production of 3.5 million cans was rejected on the grounds that an inspection, *could have found even those, if it had looked in the right place*. A further nail in the coffin of invoking the development risks defence for software, would seem to be that *any defects in software are man made*. It is not subject to the laws of the natural world as has been discussed at length in Chapter 1.

One final point about the Act is worth making. Since much software is produced outside the EU and is distributed within the EU, it should be noted that *the first importer into the EU may end up with residual liability for defective products which it distributes* and should therefore in its agreements with principals outside the EU, have appropriate indemnity provision in its contract.

To the author's knowledge, no cases in Europe have yet tested this Act, (it is only enacted in England and Greece currently in spite of coming into force over 10 years ago and even in England there was an alteration to the wording shifting the balance in favour of software suppliers which was not greeted kindly by the EU). So, although there has been much talk of potential liability, it currently has the aspect of a toothless tiger although things may change, particularly at the Year 2000. It is to be hoped so.

Summary

In contractual relationships, many lawyers now believe that much of the effort currently expended in drafting exclusion clauses might more profitably be spent on attempting to allocate risks reasonably, as a clause which acknowledges liability but seeks to limit it in some reasonable fashion, is much more likely to be acceptable to a court than one which seeks totally to exclude liability, or to restrict it to derisive levels. This is discussed in considerable detail shortly.

In considering negligence, legal precedents, notably from marine law, clearly imply that *an employer must supply proper tools, lay down proper procedures for their use, and ensure that these procedures are followed.* In this regard, many software developers today would be decisively inadequate.

For statutory liability, at least in Europe, the development risks defence is not likely to provide software developers with much protection. There remains therefore a strong incentive, especially in safety-related systems to do the very best that the budget will allow, recognise risk explicitly in the system, and keep careful records as to exactly what was done and why. In this regard, the law used wisely, can lead developers to much higher levels of quality in safety-related systems, to everybody's benefit.

Chapter 3: Influential cases before the courts

Since UK law is fundamentally based on common law and precedent and software engineering is a comparatively recent phenomenon, it is particularly important to seek out those few relevant cases looking for important clues as to how courts can be expected to treat disputes, given that software engineering itself is so poorly understood. Until very recently, there were no cases which provided such insights. Latterly however, two important cases have been tried and both cast significant, albeit somewhat different light on the subject. These two cases are notable for a number of reasons:-

- Both cases went to the Court of Appeal and the judgements therefore carry equal weight
- Both are recent
- The cases are very typical of software development failures as detailed in Chapter 1
- Both contain important material on the "Goods v. Services" issue. Recall that this relates as to whether delivered software is *goods*, and therefore covered by the Sale of Goods Act, 1979, SGA79[20], with the attendant implied terms of title, description and fitness for purpose, or *services*, in which case it is covered by the Goods and Services Act, 1982, SGSA82, and is subject only to the requirements of reasonable skill and care

In this discussion, the judgements will be compared carefully against the engineering background. The discussion will be heavily footnoted so that the main discourse is in the text, and engineering comments with references will be placed in footnotes. The footnotes will therefore provide

[20] Future cases will of course be covered by SSGA94.

a relatively painless engineering viewpoint hopefully without disturbing the flow of the narrative too much.

Both cases will be discussed individually, a compliance matrix of the two cases will be assembled, and then they will be discussed collectively to compare the agreement and consistency of the two judgements. Following this, a third case

Saphena Computing Ltd. v. Allied Collection Agencies Ltd. (1985)

In this case, Saphena Computing, the plaintiff, agreed to supply a mixed system of hardware and software to Allied Collection Agencies, the defendant under two contracts. The first was for the supply of software ordered in January, 1985 with installation between February and April[21]. There were teething problems but satisfactory performance was achieved by April/May[22]. In August 1985, a second agreement was concluded involving delivery of further software and modifications to the existing software to ensure compatibility. This was carried out against a backdrop of varying defendant specifications[23] . This ran into trouble with unsatisfactory attempts to correct defects and the parties mutually agreed to terminate their contractual relationship on February 11th, 1986.

Subsequent to this, the defendant contracted another programmer to attempt to finish the job off satisfactorily. During the process of this work, the programmer copied some of the original source code and the plaintiff

---

[21] This is a relatively small project in software terms. It is not known how many programmers were involved but it is unlikely to be more than a handful. Even with small projects, project planning in usually very poor and there is a considerable uncertainty in the project time of delivery as was noted in Chapter 1.

[22] This would be considered quite successful. As was seen in Chapter 1, there is a very high risk of software projects failing. Finding a software project without teething problems is rare to the point of non-existence.

[23] This is the key area to pursue from an engineering point of view. One of the most disruptive factors which can occur in software developments is when the end-users change their mind about the desired functionality part way through the development. It is a common misconception that because software is easy to change, it is easy to modify 'on-the-fly' to new requirements. Nothing could be further from the truth as was described in Chapter 1. It is certainly easy to change, but the effects of even simple changes on all but the most trivial of software systems are almost impossible to predict. The reader might like to reflect on the 3-line change which AT&T did to one of their network management systems in January 1990 as described in detail in Chapter 1. The change was incorrect and the result was that the entire eastern seaboard

commenced an action against the defendant on grounds of copyright as well as wrongful termination of the second agreement. The defendant responded with a counter-claim on the grounds that the original software was not fit for purpose.

The judgement

The plaintiff, Saphena Computing, succeeded in all significant aspects. Particularly relevant to the comparison here is that the court implied a fitness for purpose term as witnessed by the following words of Mr. Recorder Havery QC:-

> "In my judgement, it was an implied term of each contract for the supply of software that the software would be reasonably fit for any purpose which had been communicated to the plaintiffs before the contract was made and for any further purpose subsequently communicated, provided in the latter case that the plaintiffs accepted the defendants' instructions to make the relevant modification."

However, the defendants counter-claim that the software was not fit for purpose was quashed as the software had been deemed to be accepted at termination and the fitness for purpose term was therefore deleted.

The plaintiff got a reasonable sum and was freed from the obligation to complete the work. It should also be noted that the court upheld the plaintiff's argument that the requirements had indeed been significantly changed by the defendant.

Discussion

Given that the supplier had originally supplied something apparently deemed satisfactory, the likely primary cause of failure of this software project was the changing requirements, so the judgement seems just. In particular the court noted the following comments from an expert witness:

---

of the US lost its telephones for many hours. The direct cost of this was estimated at $1.1 billion. So much for easy to change !

> "Just as no software developer can reasonably expect a buyer to tell him what is required without a process of feedback and reassessment, so no buyer should expect a supplier to get his programs right first time".

In other words, it is not of itself grounds for breach of contract to deliver software with faults in it and the supplier should be given the opportunity to make good. Mr. Recorder Harvey QC went on to make the remark:-

> "Further, even programs that are reasonably fit for their purpose may contain bugs."

It should be made very clear however that this judgement was made on a case for which specifications were a particular problem and indeed were changing, an issue which will be returned to later.

Criticisms

The case contained two specific areas which can be criticised:-

- In the judgement, it was stated that it was unnecessary to decide whether the software was goods or services as the same requirements would be imposed under both headings. At first sight, this appears to be manifestly odd as the first falls under SGA79, with the attendant implied terms of title, description and fitness for purpose, whilst the second falls under the SGSA82, requiring only reasonable skill and care[24]. However, closer analysis suggests that the judgement was based on the nature of the contract between the two parties rather than with reference to any particular statute.

[24] The state of the art in software engineering is so bad that reasonable skill and care would be relatively easy to demonstrate even if the software didn't work at all. In fact, many well-meaning and skilfully implemented projects are never delivered as evidenced by the US Department of Defense statistics quoted in Chapter 1 for example !

- The court decided that although the defendant had the right to the source code to correct the system[25], they did not have the right to copy it. This is a naive judgement and seems a little harsh against the defendant. There are occasions where software correction necessarily requires copying. Indeed in the now ubiquitous branch of software engineering known as Object-Oriented design and development, there is a specific concept known as the copy constructor. Also, at what point is software copied ? If the defendant had used interfaces to libraries supplied by the plaintiff, is this copying ? Any reasonable engineering decision would argue that this was not copying. A final point which should be made relates to the general discussion in Chapter 1. In practice the boundaries between corrective, adaptive and perfective maintenance are frequently ill-defined. In other words courts may in practice find it very difficult to distinguish between legal copying for corrective purposes and illegal copying for adaptive or perfective purposes. This problem is not likely to go away in the near future.

  In fact, copyright law now specifically allows for copying for the purpose of error correction as evidenced by Section 50C for example:-

  "It is not an infringement of copyright for a lawful user of a copy of a computer program to copy or adapt it provided that the copying or adapting is necessary for his lawful use ... It may, in particular, be necessary for the lawful user of a computer program to copy or adapt it for the purpose of correcting errors in it."

  On the face of it, this considerably assists the user of flawed software. However, in practice it is nearly useless, because it is rare for the source code to be supplied with any system, so the user is critically dependent not just on the supplier, who may well not have the

[25] The plaintiff had apparently inadvertently left the source code on the defendant's system.

source code either, but on the original author who presumably does[26]. In practice, source code access is supplied only through frequently complex escrow agreements as discussed in Chapter 3, and then usually only for the purposes of continuation of benefit should the supplier cease trading.

There are further complications concerning what precisely constitutes necessary copying which even experts might disagree upon as have already been hinted at.

The central problem here is that the contract neither allowed for appropriate acceptance criteria based on an agreed specification, nor for a constructive recovery of the situation as is discussed in much more detail in Chapter 4. Such occurrences are relatively common in software engineering as evidenced by [14] for example. In fact, the defendants in this case were particularly IT illiterate and failed to understand even the distinction between on-line and batch software.

The problems with requirements generally here can be seen from the words of Mr. Recorder Havery QC:-

> "... The contract document does not describe the software, but is common ground that the software in question is described in the third and final draft of a system proposal ... Unfortunately, for present purposes, the proposal was not designed as a legal document; indeed it is not even a technical specification as that expression is understood in the trade ... It became clear to me during the course of the evidence ... that the document is ambiguous in its description of the software on page ..."

[26] This presumes that the author developed the software under the aegis of a formalised Change and Configuration Control System. However, as commented in Chapter 1, such an author is still likely to be in the minority.

St Albans City and District Council v. International Computers Ltd. (1996)

Although this case appears rather similar in principle to the Saphena case, (and Saphena was indeed quoted by the defence), closer analysis suggests that they are quite different certainly on software engineering grounds.

In essence, the defendant was invited to supply a system for the computerisation of a number of areas of Local Government work to cope with the requirements of the Local Government Finance Bill then proceeding through Parliament. Included in this was a strict contractual requirement:-

> "to provide a firm commitment to supply a system to cope with all the Statutory Requirements for registration, billing, collection and recovery and financial management of the Community Charge and Non-Domestic Rates; including Community Charge Rebates".

It was further pointed out that some 16 data items were:-

> "subject to addition/amendment as a result of the continuing Parliamentary process"[27].

In response to this tender, the defendant used a number of statements such as:-

> "To develop a system using a 70 strong development team, which meets fully the legislative requirements, and which is easy to use and operate".
>
> (The plaintiffs would have the opportunity) "to input into the development process in order to be sure that this product meets your specific requirements."

[27] This would be enough to set alarm bells ringing in any professional developer. It is difficult enough writing software even when the requirements are not changing as a result of a party not privy to the contract. In essence, change was uncontrolled. This was indeed tempting fate.

In response to the plaintiff's statement of user requirements, the defendant stated:-

> "The register will contain the data items necessary to meet at the very least the legal requirements plus any other fields the User Design Group deem advantageous. The system is planned to handle all debits.
>
> All other requirements will be met"[28].

In essence, what happened was that a software fault caused the relevant population to be miscounted, giving a value 2966 too high[29]. St. Alban's District Council therefore set the Community Charge based on this artificially inflated population figure and received too little income as a result, leading to a series of directly quantifiable losses.

The judgement

The court ruled unanimously for the plaintiff, although it reduced the amount of damages by a sum which the plaintiff could recover anyway in a later year in the normal course of local authority affairs. A term in the contract limiting liability was deemed inadmissible as it was judged to form part of the standard terms and conditions therefore falling under the auspices of UCTA77. The Court of Appeal upheld the previous court's analysis that this clause was deemed not to be reasonable.

The details under which the appeal was in essence rejected are worth repeating. Mr. Dehn QC acting for the defendants argued to the Court of Appeal that the system would essentially be in a state of development until its fully operational date of the end of February 1990. Thus, unless the defendants had acted negligently, the plaintiffs had implicitly agreed to

---

[28] In engineering terms, this is specifically not an advertising puff, but a clear commitment to satisfy requirements which were not particularly onerous and would be precisely defined by an external statutory agency.

[29] The defendant's software gave a population figure as 97,384**.7**. I must confess that any engineer seeing a population estimate with a decimal place would begin to have serious concerns about the software responsible. It is also worth noting that this is not a requirements fault at all. It appears to be a simple logic fault, (c.f. [32] for more information on these categories).

accept whatever software was supplied, bugs and all. Mr. Dehn relied on observations of the Saphena case discussed above and specifically argued that the defendant was not contractually bound to provide software which could count properly on 4 December 1989 when a necessary return was carried out by the plaintiffs.

This argument was rejected by the judges who ruled that the defendant was under an *express* contractual obligation to supply the plaintiffs with software that could enable them accurately to complete the return by the required date of 8th December 1989. However, the most interesting and perhaps portentous part of the judgement then followed given by Sir Iain Glidewell. He was addressing the following issue. If there had not been an express term breached, then was the contract between the parties subject to any *implied* term as to quality or fitness for purpose, and if so what was the nature of that term ? To answer this, he had to plunge into the issue of whether software should be considered goods or services. This point is taken up again in more detail in Chapter 4. However, Sir Iain Glidewell's arguments ran thus: If software is goods, it is subject to the implied terms of SGA79. If it is services, it is not thus constrained. He concluded that although a program of itself was not goods according to the definition contained in SGA79 and SGSA82, he did not see how it could be reasonably separated from the medium on which it was supplied, just as erroneous instructions in a car maintenance manual could not be separated from the physical medium of the manual itself. He went on to quote other sources supporting this view.

To resolve this unsatisfactory position, he first concluded that there was no statutory implication of terms as to quality or fitness for purpose but went on to ask if the contract itself should contain an implied term as if it were a contract for the supply of goods. For this he looked back in time to the Common Law roots of SGA79 and asked the question, under what basis is a court justified in implying a term into a contract in which it has not been expressed. This basis is strict and was summarised by Lord Pearson

in *Trollope & Colls Ltd. v. North West Metropolitan Regional Hospital Board (1973)* as follows:-

> "An unexpressed term can be implied if and only if the court finds that the parties must have intended that term to form part of their contract; it is not enough for the court to find that such a term would have been adopted by the parties as reasonable men if it had been suggested to them; it must have been a term that went without saying, a term which, though tacit, formed part of the contract which the parties made for themselves".

Sir Iain Glidewell then continued by holding that a contract for the supply of software transferred howsoever when intended to fulfil specified functions fell into the category addressed by Lord Pearson's words above. In other words

> "... in the absence of any express term as to quality or fitness for purpose, or of any term to the contrary, such a contract is subject to an implied term that the program will be reasonably fit for, i.e. reasonably capable of achieving the intended purpose".

Sir Iain concluded by saying that were the matter not resolved by the express term, he would still hold that ICL were in breach of this implied term. Tellingly, Nourse LJ had stated in his initial summing up:-

> "... it becomes strictly unnecessary to consider whether the contract was subject to an implied term to the same effect. However, having had the advantage of reading in draft the judgement to be delivered by Sir Iain Glidewell, I would, like him and for the reasons he gives, have answered the question in the affirmative".

In his judgement, Hirst LJ agreed with the judgement by Nourse LJ. In other words, *the existence of an implied term for quality and fitness for purpose in a contract for the supply of software for specified functions howsoever transmitted was unanimously agreed* by this appelate court. This is likely to be very influential indeed in cases where the software

requirements are well-defined, particularly as the same thing happened in the Saphena case discussed earlier although the term was not used in the judgement.

Discussion

For an engineer, the court's ruling seems entirely reasonable. the defendant delivered a system with a software fault in it known as a "show-stopper" in the trade. The requirements although not complete at the time the contract was signed were well-defined and essentially specified by a third party, in this case a statutory source. The fault caused a failure, the nature of which led to a large quantifiable loss. This could have been mitigated to some extent if the defendant had taken more care to inform the plaintiff about the nature of the many software updates which were taking place at the time. The degree to which testing was performed on this intermediate release is unknown but such a gross error should have yielded even to a relatively cursory series of tests. An attempt to limit their loss was deemed part of the standard terms and conditions and was ruled inadmissible on the grounds of unreasonableness.

Criticism

This section is only included for completeness. The ruling seems accurate and reasonable. It was in every sense, 'a fair cop'.

Compliance matrix

In this section, the key facts of the two cases will be compared side by side in the form of a compliance matrix. A compliance matrix in essence compares two or more objects, (columns 2 and 3 here) according to a set of categories, (column 1). The degree of compliance, (column 4), is measured subjectively using three independent categories as follows:-

POOR | AVERAGE | GOOD: The similarity between the two cases for this item.

CONSISTENT | INCONSISTENT: Whether the judgements were consistent.

CRUCIAL: If present, this indicates that this item of compliance has a fundamental part to play in understanding the differences between the two cases.

Thus for example, the first item compares the nature of the plaintiff. In Saphena, the plaintiff was the supplier and in the St. Albans ruling, it was the user. Thus the nature of the plaintiff is quite different leading to a POOR similarity, but the judgements in each case were not inconsistent with each other, thus attracting the category CONSISTENT. The CRUCIAL key word does not appear as the nature of the plaintiff is not of fundamental importance in understanding the differences between the two cases.

We are looking for any AVERAGE or GOOD agreement between the cases whereby the ruling was INCONSISTENT. These would be damaging in understanding the cases. If a CRUCIAL item turns out to be INCONSISTENT, this would be very damaging.

Item	Saphena	St. Albans	Degree of Compliance of Saphena with St. Albans
Plaintiff	Software supplier with case based on unfair termination and copyright transgression	Software user with case based on catastrophic failure.	POOR CONSISTENT
Type of problem	After apparently successful initial installation, couldn't get it working - bugs, delays, etc.	Single 'show-stopper'	POOR CONSISTENT
Nature of loss of plaintiff	Lost opportunity + copyright transgression	Direct quantifiable loss	POOR CONSISTENT
Modification to existing system	Yes. General modifications	Yes. New module.	GOOD CONSISTENT
Quality of specification	Apparently poor and variable.	Pending completion of statutory legislation, but then good.	AVERAGE CONSISTENT CRUCIAL
Attitude to software failure by court	*Court ruled that failures are inevitable and the supplier should be allowed to make good if possible. Court also ruled that there was an implied term that the software should be reasonably fit for its purpose, but the term was deleted by the defendant's acceptance..*	*Defendant quoted Saphena, but court ruled "Parties who respectively agree to supply and acquire a system in development, cannot be taken merely by recognition, to intend that supplier can supply software which cannot perform its function. The court referenced the* ***requirements*** *in making this point".*	AVERAGE CONSISTENT *Quality of requirements emerges as a key issue.*
Implied fitness for purpose	Termination by defendant deemed as acceptance, therefore fitness clause deleted.	Central in ruling	POOR CONSISTENT
Implied terms	Excluded by termination but deemed in force prior to that	Standard terms and conditions deemed in force, therefore cannot exclude.	AVERAGE CONSISTENT
Influence of ruling	Court of Appeal	Court of Appeal	-
Goods v. Services	Deemed irrelevant in a strange decision, although fitness for purpose invoked.	Essentially irrelevant to ruling but in a long analysis, Sir Iain Glidewell argued it to be supply of goods.	POOR IN-CONSISTENT

As can be seen, there are no important inconsistencies between the two cases, in fact on deeper analysis, in spite of the significant engineering

differences between the two cases, there are important similarities in the rulings. The most important of these in the author's opinion is the willingness of both courts to infer an implied contractual term that software should be reasonably fit for its purpose even though the quality of the software specifications in the two cases differed markedly. The fact that the implied term was not applied in Saphena as it was deemed to have been deleted by the nature of the termination does not materially affect this view.

Discussion

The poorly misunderstood state of software engineering in legal terms is well illustrated by the attempts of the defence counsel in the St Albans case to use the Saphena ruling to justify the presence of faults in his client's software. As we have seen, the two cases are very different in engineering terms and in the Saphena ruling, the court's acceptance that software can be expected to be delivered with bugs in was strongly influenced by the nature of the changing requirements. This was not a feature of the St. Alban's ruling.

As discussed in Chapter 1 and also Figures 1.10, 4.1 and 4.2, the following kinds of software development are distinguished:-

Bespoke software development

This category is usually characterised by continually varying requirements due to misunderstanding and unforeseen issues and, if successful, a slow convergence to a system which satisfies both the supplier and the end user. It may be developed from scratch or it may perhaps be a substantial modification to an existing piece of software. The software described in the Saphena case falls into this category. The court's acceptance of the expert witness opinion that:-

> "Just as no software developer can reasonably expect a buyer to tell him what is required without a process of feedback and reassessment, so no buyer should expect a supplier to get his programs right first time".

is entirely reasonable for such a system. (On the other hand, the court still implied a term that the software should be reasonably fit for its purpose even though this term was not used).

Modified software

In such a system, the requirements for the software are already specified by a third party and some tailoring work of a related package is necessary. Such requirements are usually specified rather more accurately than in true bespoke software. The St. Alban's case fell into this category even though the legislation was not complete in the first instance, which immediately separates it in engineering terms from the Saphena case. In the St. Alban's case this led *inter alia* to the ruling:-

> "Parties who respectively agree to supply and acquire a system recognising that it is still in course of development cannot be taken, merely by virtue of that recognition, to intend that the supplier shall be at liberty to supply software which cannot perform the function expected of it at the stage of the development at which it is supplied. Moreover, and this is really an anterior point, the argument is concluded against the defendant by clause 1.1 of the plaintiff's statement of user requirements which, having referred to the Bill that later became the Local Government Finance Act 1988, ..."

On the basis of this, the defendant's reference to Saphena was rebuffed. The author would disagree only in the sense that the reference to requirements is not anterior. Indeed it is fundamental to the relevance of the rulings. Once again, it can be seen that the court found an implied term that the software should be reasonably fit for its purpose.

COTS (Commercial Off The Shelf)

COTS software is something which the supplier themselves has specified, often without consulting any potential users at all, implements and then markets in the hope that the product will be sufficiently appealing that it will

sell. In engineering terms it is rather different from the previous two categories and legally it may be very different. Trying to interpret the Saphena and St. Albans rulings for this category is by no means clear, although if we follow the requirements principle, neither of the rulings are applicable. It seems likely that this category has the greatest claim to be considered as goods although this will be discussed further below. If it is considered as goods, then once again, although this time for statutory reasons, the software would be required to be reasonably fit for its purpose.

In fact, this is a good reason to treat COTS software as goods as the three different kinds of software development would then be treated similarly even though the rulings would be arrived at in different ways.

The key to understanding these two cases from an engineering as well as a legal point is the nature of the requirements placed upon the computer supplier. In the Saphena case, they were ill-defined and variable, although this was not tested as the defendant in that case terminated early. It is reasonable to view the court's ruling as to whether one should expect a product to be delivered with fault, as being based on the quality of the input requirements. Even then, the court ruled that fitness for purpose was implied. In the ICL case, they were much more clearly defined and subject to a third-party, and in this case statutory agency, and the court ruled that there could have been no agreed intention to deliver something which did not function as required. This was expressed as a breach of an *express* contractual term, but the court again made the point that there was an *implied* term for quality and fitness for purpose in such a contract. In other words, there was a clear criterion for "did not function" here, notably, that the software failed very expensively with an obvious quantifiable loss to the plaintiff. That the failure was nothing to do with requirements in engineering terms is irrelevant.

These rulings however, reasonable as both seem, greatly increase the risk to software developers in the author's view, if the software

requirements are well defined. Only if the requirements are not so well defined does the Saphena ruling seem appropriate. It is interesting to speculate what would have happened in the Saphena case had the defendant not terminated the contract, thereby deeming to have accepted the software, and leading to the implied term of fitness for purpose being deleted.

The general point in this detailed comparison of existing influential cases along with the essential but as yet untested case for COTS software is this however:-

Courts consider that reasonable fitness for purpose for software is appropriate whatever its origin and are very happy to imply it where necessary under the two accepted methods, i.e. to give a contract business efficacy or when it must have been unconscionable to parties privy to the contract that software could reasonably be delivered when not fit for its purpose.

Chapter 4: Bridge building:- issues worthy of further discussion

In any appraisal of software from a legal point of view, it becomes immediately obvious that there are a number of particularly thorny issues. Software has been specifically addressed in legal affairs in a number of ways, for example, through the medium of case law on civil matters, through copyright, through criminal acts and delictual responsibility, using analogies with issues *inter alia*, in maritime law. The appearance of references to software in these very different areas invites and ultimately will demand a holistic view with the intention of finding a consistent and coherent viewpoint.

The first thorny issue to discuss follows on appropriately from the words of Sir Iain Glidewell in *St. Albans v. International Computers Ltd*. as described in the previous chapter.

Software as Goods or Service

As has been seen earlier, a recurrent theme in much legal discourse on software is the 'goods v. services' issue. To recap, the reason for this discussion is that goods and services are treated differently in statute and therefore it would seem important to determine which of these is appropriate for the essentially intangible nature of software. This point will be discussed according to a number of legal perspectives.

To recap, the sale of goods is covered by the 1979 Sale of Goods Act (SGA79) and the modifications inherent in the 1994 Sale and Supply of Goods Act (SSGA94). Under s. 14 of SGA79, the supplier is required to supply software that is of satisfactory quality and reasonably fit for the purpose the buyer has made known to him. If the buyer has not made known any particular purpose, then the fitness for purpose will be assessed by the courts in relation to the common purposes for which software of that type is generally purchased. In addition, the 1979 Sale of Goods Act provides for certain implied terms. For example, that the supplier has the

right to sell which can be excluded, and that the goods are of reasonable quality and comply with their description which either cannot be excluded at all if the buyer is a consumer, or will be subjected to a test of reasonableness according to the 1977 Unfair Contract Terms Act, if not. However, we should note that goods are defined as personal chattels and are tangible. For software, at most the medium of transfer is tangible, and in the case of transfer by electronic means such as an Internet download, not even that. By this definition, it is unrealistic to classify software as goods even though COTS software bears all the hallmarks of a sale of goods and this has led to an unnatural and essentially irrelevant focus on the quality of the medium on which the software is supplied, for example, the floppy disc or CD.

In contrast, services are supplied under the auspices of the 1982 Sale of Goods and Services Act, (SGSA82). for which there is only a requirement for reasonable quality, which can only be excluded subject to a test for reasonableness. This would seem a far less onerous requirement on the supplier and leads to the legally acceptable notion that 'a surgeon does not warrant to save the life of his or her patient'. For bespoke software where nothing tangible is delivered and the user is instrumental in defining what it is to be delivered, it seems likely that this is a provision of a service. Some authors (for example, Smith in [28], p 77) have commented that the application of the 'substance of the contract' test used by the Court of Appeal in *Robinson v. Graves (1935) 1 KB 579* could result in the conclusion that a bespoke software development contract is a contract for services alone if no materials are transferred. Alternatively it is one for work and materials, the major component of which is work on the grounds that the work involved in developing a piece of software far outweighs the cost of the medium on which it is supplied. It seems to me that this view, based on a decision made at a time when the idea of doing huge amounts of work and supplying nothing material was novel to say the least, is simply too remote to be useful. Amongst other things it neglects the fact that true

bespoke development whereby a piece of software is crafted from nothing and handed over such that none of the experience or source code is ever used again is rare to the point of practical non-existence. The author has never heard of such a development in 25 years in the computing industry.

Note that the author used the word 'experience' here. It should never be under-estimated just how much value is contained within the experience of developing a software system. *It is probably true to say that the experience of developing such a system is at least as valuable as the source code which was actually delivered*. If this seems an outrageous statement, the reader might like to consider the following factors. First of all, as illustrated in Figures 1.3 and 1.4 in Chapter 1, software engineers typically spend far more effort on a system after it has been delivered than before, (about a factor of 4). Much of this effort is because the development of an essentially new system is a prodigious learning experience characterised by many mistakes. On the contrary, re-creating a system a second or subsequent time is dramatically easier[30]. Software engineering is sufficiently immature that most of the effective training that software engineers get arises from practical experience and not from any formal education.

Finally, it is almost certainly true to say that the time taken by engineers to re-create source code for an application in which they are experienced is comparable to the time taken for an engineer inexperienced in that application to understand an *existing* piece of source code for that application.

Since there is a software continuum between COTS and bespoke development in terms of both requirement specification and user interaction with development as illustrated in Figure 1.10, it is inevitable with the current legal interpretation of COTS software as goods and bespoke

[30] Providing the ever-present trap of adding a wealth of new features is avoided of course. It has long been known, ([26]), that in this case the second system is often worse than the first because ambition generally overreaches capability. The third system is usually the best and this is known as the *third-system effect*. If the lure of enhancement is ignored however, all subsequent systems benefit immensely from the mistakes encountered building the first.

software as services, that there is a corresponding legal continuum between goods and services when dealing with software, as illustrated by Figure 4.1.

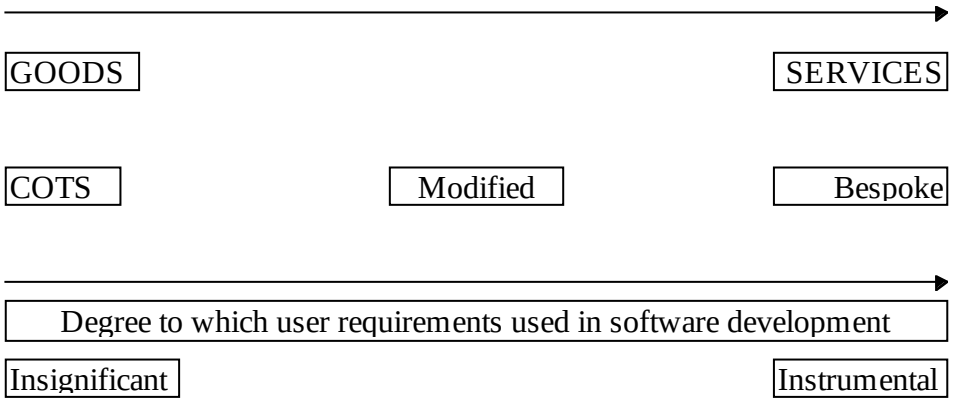

Figure 4.1: The continuum of types of software tentatively associated with the legal concepts of goods and services.

Given the rather different nature of implied quality between the two, this seems undesirable to say the least and is likely to lead to substantial problems of interpretation in the future which expert witnesses are unlikely to be able to resolve owing to the immature nature of software engineering. There is already evidence that this is occurring in that in all cases so far considered, the courts have assiduously avoided the subject and have effectively said that they are interpreting the provisions of a particular contract between the parties rather than the meaning of a statute. The situation is not unlike the wave v. particle duality of light argument which has persisted for most of the last 300 years. Is light a wave or is it a particle ? The answer is that it is both depending on how you look at it. This is clearly unacceptable for a legal position given the significantly different requirements laid down for the two extremes.

To give some idea of the interpretations which could arise in this grey zone between services and goods, the words of Hilbery J in *Marcel*

(Furriers) Ltd. v. Tapper (1953) 1 WLR 49, in a case concerned with the supply of a fur coat will be quoted:

> "I cannot discover anything to distinguish it from the case of an ordinary article which it is part of someone's business to supply and which the person has to make to special measurements for the customer. It requires skill, labour and materials of course, but the purpose of the transaction is the supply of the complete article for the price."

These are powerful words and particularly relevant to the provision of software which is to a certain extent tailored for the customer. This kind of tailoring using existing components in great part, a software equivalent of Lego™, is now overwhelmingly common and is viewed as a significant aim in bespoke systems partly because it allows a product to be delivered earlier and with less risk, (and at greater profit it is fair to say). This quotation is really leaning towards the fact that software which is predominantly component based with an element of bespoke modification, is really goods.

Given these words and other comments made above, there seems to be two ways out of the dilemma of categorising software as services or goods, both of them relatively radical. Either software can be categorised *sui generis*, i.e. as something new being neither goods nor services or it can be categorised as either goods or services irrespective of the nature of its development. There is some legal precedent for this latter view in both English and Scottish Law. Sir Iain Glidewell in the English case *St. Alban's v. ICL, 1996* as discussed in detail in Chapter 3, devotes some time to discussing this issue arguing cogently that software be considered as goods, although he readily admits the difficulties. The unnatural obsession with the nature of the physical medium rather than its content could lead a supplier to argue that the goods provided are *only* the disc and that only its quality is therefore relevant. However, the buyer pays substantially more for the disc containing the software than he would for a blank disc alone, so it

must contain *something* of value. Worse, this argument would be inconsistent with other influential rulings in different areas of the law. For example, in *Cox v, Riley (1986),* the defendant had erased programs from a magnetic medium and claimed in his defence that he had not damaged the medium itself. This was rejected by the courts which found that he had damaged the 'card as programmed' and that this was property for the purposes of the *1971 Criminal Damages Act.* There is no doubt whatsoever here that the courts considered the damage of an intangible object as the damage of property. Any other decision would have invited very adverse comment and created a very unhealthy precedent. As it is, it is hoped that this case creates an important and healthy precedent.

The notion of software being considered *sui generis* also emerged in the well-known Scottish 'shrink-wrap' case of *Beta Computers (Europe) v. Adobe Systems (Europe) Ltd. SLT 604 (1996).* Here the judge albeit invoking a legal principle not relevant to English Law[31] indicated that a contract for the supply of software was *sui generis* (i.e. a contract of unusual and unique nature) rather than one for goods or services.

Separating the content from the medium by claiming that the goods only referred to the disc and not the software would also lead to problems with the established understanding of copyright. Software is unequivocally associated with copyright through the auspices of the *1988 Copyright, Patents and Designs Act*. If the supplier claims that goods only applies to the medium and this is clearly the intent of many contracts originating in the U.S[32], then the supplier should not object to the disc being copied without

---

[31] The rules to be implied in contracts (i.e. in the absence of express terms) are broadly similar in Scotland and England and are described in more detail in Chapter 3. However the sections on implied terms to use reasonable skill and care and provide services on time under SGSA82 do **not** apply in Scotland (it is a common misconception that they do), although a similar term will be implied by Scottish courts at common law leading to the same result. It should also be noted that it is not unknown for defects in software to take a long time to manifest themselves. A tragic example of this occurred in the Therac-25 incident discussed in Chapter 1 whereby it took the presence of a new defect to cause an old defect previously hidden for years to fail.

[32] Phrases such as the following are common:- a) The software house warrants only that the medium is free from defect., b) that this warranty is in lieu of all liabilities, express or implied, whether by statute or

restriction. However, the very same contracts profoundly object to this. Another example can be found in the treatment of the European Patent Office of an 'image', [28]. Although the Board talks of an image as a 'physical entity', and as a 'real-world object', *it is made clear that an image stored in any form, hard-copy or electronic, will be regarded as a physical entity*. Given the comments made in an earlier section on digital convergence and the very close association of digital information all kinds, once again consistent interpretation suggests that the medium be considered irrelevant.

Another example wherein there has been recognition of the independent existence of software can be seen in the words of Aldous J in a patent application case, *Wang Laboratories Inc.'s Application (1992) RPC 463* when addressing a claim concerning an expert system.

> "The machine, the computer, remains the same even when programmed. The computer and the program do not combine together to produce a new computer. They remain separate and amount to a collocation rather than a combination making a different whole. The contribution is, to my mind, made by the program and nothing more."

The last sentence should particularly be noted. The Judge is obviously in no doubt as to the existence of the program as a separate entity.

Before leaving this point, an argument which further strengthens the notion that software has an existence quite distinguishable from the medium on which it is transferred will be developed. Software is frequently sold in a licensed form with a licence key which allows the software to work correctly. Some licence keys are physical and are called 'dongles' (for some unknown reason). Without the licence key, the software is useless. Supposing someone purchases a copy of a piece of 'dongled' software.

---

otherwise, c) that liability is limited to replacement of defective media and d) no liability is accepted for

Supposing further that the purchaser then copies this on several different pieces of media. On some of these media, for example a programmable read-only memory or a 'write-once' CD-ROM, there are irreversible physical changes to the substrate. However, there is still only one licence key and therefore only one copy. If software is not protected, each copy is a valid copy. In other words, whether a software copy is a copy or not is completely unrelated to the media on which it is stored. There is absolutely no doubt whatsoever that a computer program is a separate entity and this should be universally recognised.

To make one final point, it has already been seen in Chapter 3 that courts are very willing to infer reasonable fitness for purpose in existing rulings whatever the source of software. Since, from the point of view of software anyway, this is one of the central differentiating factors between treating software as goods or services, it can be taken as a willingness to treat software in a currently similar way to goods.

To summarise the above comments, the nature of software is such that to be consistent with copyright, criminal damages and other legislation, it appears necessary to define it as goods whatever the nature of its development. This may be a controversial statement but it is entirely consistent with Sir Iain Glidewell's view and there is *absolutely* no question that computer scientists consider software a tangible asset, as evidenced by the importance attached to the mechanism of **re-use**, whereby previously developed components are used again and again as the prescient comments of Hilbery J. quoted above foresee. It is much harder to understand how you would re-use a service, without repeating the work effort. This is manifestly NOT true with software.

In spite of the examples quoted above, simply including software within the remit of goods by some future statutory instrument is not without difficulties however as can be noted from the wording of the 1994 Sale and

---

consequential loss.

Supply of Goods Act, which states in s. 2b) that the quality of goods includes such factors as:-

a) fitness for all purposes for which goods of the kind in question are commonly supplied

b) appearance and finish

c) freedom from minor defects

d) safety, and

e) durability

It is very unlikely in the entire history of computer science that any significant piece of software has been delivered which satisfies c) above. This may support the argument that software should form part of the Sale of Goods Act but perhaps *sui generis*. At least two 'dark corner's however, will have to be addressed. First, the identical nature of software copies means that software copies can never be ascertained as was discussed earlier in the section on ownership and risk. Second, the problems associated with the engineering benefits of incremental delivery against the fact that the customer does not have to accept goods delivered by instalment, s. 31(1), SGA79, will have to be resolved.

From the buyer's point of view, until a clearer policy emerges perhaps by considering software uniformly as goods, for any other than pure COTS software there is therefore a clear inducement to specify the product carefully in the contract, including detailed acceptance criteria and also to be satisfied that the supplier is capable of producing such a product, which is by no means a foregone conclusion as can be seen by the discussion in Chapter 1. From the supplier's point of view, there needs to

be a satisfactory way of limiting liability for an unavoidable[33] failure of a clearly-defined system, recognising in the aftermath of the *St. Albans v. ICL (1996)* ruling that if a bad defect leads to substantial quantifiable loss, then it may be very difficult to avoid liability. On the other hand, if this is to become the standard ruling, then software is going to become a lot more expensive, particularly in the year 2000 when software failure is likely to become a very serious issue for society in general.

Finally, it should be noted that for pure COTS software, there is simply no legal precedent in UK law at present and therefore no clear guidance, although of the categories discussed here, this seems most likely covered by existing legislation under SGA79 and SSGA94.

Following on from the above discussion, it is tempting to suggest that attempts to resolve legal cases based on statutes involving goods and services is doomed to failure at present and it is not surprising that the courts have shied away from doing so. To conclude, this is because of the difficulty of accepting precisely where on this continuum a particular piece of software lies, other than perhaps at the extrema, a situation exacerbated by wide variations in expert opinion as discussed in Chapters 1 and 3 for example. This simply emphasises the continuing importance of contract in resolving such issues as has already been seen in *St. Albans v. ICL (1996)* and *Saphena Computing v. Allied Collection Agencies (1985).*

Adding Delictual Liability to the Spectrum

At the time of writing, no software related case in English Law had been decided in the courts. This has led to a large number of so far hypothetical arguments as to the precise nature of this liability and a comprehensive summary of these can be found in chapter 3 of [28]. The basic legal principles of delictual liability have already been discussed in Chapter 2. However, the discussion on p. 96 of chapter 3 of [28] clearly implies that there is a progressive duty of care between the two opposite ends of the

[33] A wide class of failures in software are unavoidable, [33].

software spectrum. This progressive duty of care reflects the closer relationship present in Bespoke systems whereby the customer usually makes a substantial contribution to the definition of the requirements. The higher duty of care owed in a Bespoke development reflects the decision in *Hedley Byrne & Co. Ltd. v. Heller & Partners Ltd (1964).* A medium duty of care owed to a modified system is reflected in the decision in *JEB Fasteners Ltd. v. Marks, Bloom and Co. Ltd. (1983).* A low duty of care at the COTS end would correspond to cases where the proximity of the relationship failed the tests described in these two cases. An important element in deciding the proximity question appears to be the purpose for which the advice was produced. For example, in *Caparo Industries plc v. Dickman (1990),* the House of Lords held that the company's auditors owed no duty of care to the shareholders in respect of the accounts because investment was not the purpose for which the accounts were produced, even though it was foreseeable that they would be used for such a purpose.

In contrast, the strength of statutory requirements goes in the opposite direction with those pertaining to Goods being significantly stronger than those for Services as described earlier in this Chapter. This invites us to extend Figure 4.1 as shown below in Figure 4.2

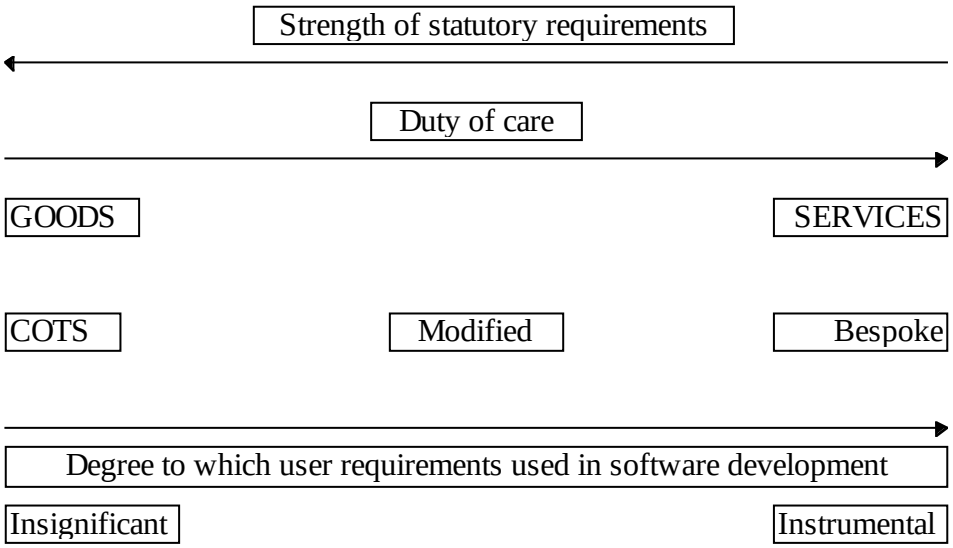

Figure 4.2: The continuum of types of software tentatively associated with the legal concepts of goods and services with the delictual concept of "duty of care" added and the respective strengths of statutory requirements.

Here "duty of care" and "strength of statutory requirements" axes have simply been added to Figure 4.1. There is therefore a clear relationship between the computer science issue of the degree to which specification of user requirements has been used and the legal issue of deciding the duty of care. *It is also interesting to note that the law has evolved to be self-balancing. As the statutory requirements weaken, there is a correspondingly higher duty of care under delictual liability*.

Given the general and continuing difficulties which computer scientists experience with the specification of requirements, it is tempting to speculate that the appropriate duty of care will be also very difficult to decide if a case involving delictual liability reached the courts. The corresponding legal simplification is another argument in favour of deciding that software is always of a particular category.

At what time is software deemed to be of satisfactory quality ?

This too seems on the face of it a particularly vexed issue with software. Historically in the UK, goods are deemed to be of satisfactory quality at the

point of delivery, although under SGA79, there is an implied term of merchantability which was replaced by a requirement for satisfactory quality in SSGA94, and the fact that software does not deteriorate could sustain a purchaser's argument that a defect occurring some time later must have been in the product at the point of delivery, potentially breaching this implied term. This is assuming that the purchaser does not 'amend' the code. Even so, if the time elapsed before reporting the defect was sufficiently long, it might be argued by the supplier that the purchaser had had *full value.*

The implication of deeming goods to be of satisfactory quality at the point of delivery of course is that even with complex consumer items like cars, minor defects discovered a couple of weeks later do not constitute grounds for rejection, although of course, there is an obligation on the supplier to correct them. The words of Mr. Recorder Havery QC in *Saphena Computing Ltd. v. Allied Collection Agencies Ltd*. may be noted:-

> "In the present case, on the other hand, once the software is fit for its purpose it stays fit for its purpose. If by any chance, a flaw is discovered showing that it is unfit for its purpose (which is hardly likely after prolonged use) there is a remedy in damages against the supplier, if solvent, until the expiry of the period of limitation."[34]

As consumer products become more and more complex, this will become a correspondingly more difficult issue to resolve, indeed most delivered software is now of such complexity that for any other than COTS software, it is wholly unreasonable to expect a user to accept it unconditionally on delivery, when it is most unlikely that not only the supplier will understand in full the actual functionality but the developers also.

---

[34] In the light of Chapter 1, we can however criticise this in a number of ways. First, software can easily be deleteriously affected by future updates and second, flaws can indeed arise after a very substantial time.

What is a reasonable time ?

How do products fail and what is a reasonable time between failures from the point of view of the end-user ? There are a spectrum of answers to this question with different end-users rights as exemplified in Diagram 4.3

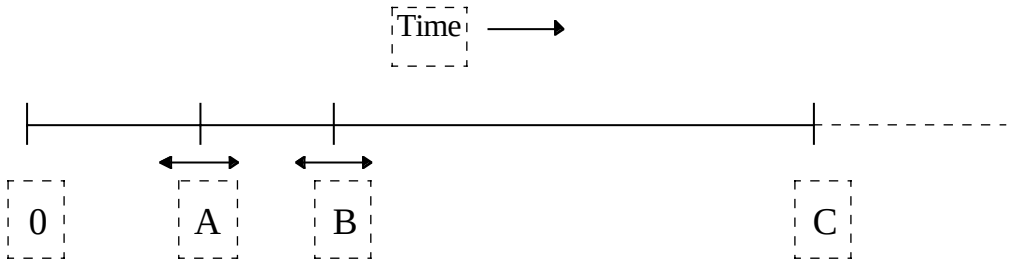

Diagram 4.3 Times of interest to the purchaser of a product. At time 0, the product is delivered to the end-user. Time A is not well-defined and is the end of the period in which it is legally correct to return a faulty product. Time B is the time at which a product is accepted either explicitly or implicitly by the end-user. If it is implicit through some act which is considered inconsistent with the supplier retaining ownership, it may not be well-defined either. Time C is the end of the warranty period and is the maximum of the statutory and supplied warranty periods. These times obey the relationship $0 < A < B < C$. After C, it would be assumed that the purchaser had had full value even though the product would typically continue in use for some considerable period afterwards.

In product failure, there are two perspectives, the supplier's and the purchaser's and they are very different.

The supplier's perspective of product failure

Consider the manufacturer of a mass-produced consumer item such as a car or a video-recorder. Over the past fifty years, manufacturing process improvements based on the notions of Statistical Process Control have inexorably improved the average quality of many consumer items. Cars for example are dramatically more reliable than they were only 20 years ago. Hard disc drives are another example of the dramatic progress which has been made. Only 15 years ago, such a drive had a mean-time between failures of around 100-1000 hours. Today, for £100, it is possible to buy a disc drive with approximately 100 times the capacity with a mean time between failures of 1,000,000 hours, or 1000 times more reliable. One of the measures of manufacturing excellence today is known as *six sigma*

quality, which corresponds to a manufacturing process which produces only around 3 defects per million items. Suffice it to say, that it is difficult to fault many manufacturers of consumer items on the grounds of reliability. For a very reasonable price, the public today has access to a level of manufacturing quality which engineers of only 20 years ago would not have dreamed.

The purchaser's perspective of product failure

Of course the purchaser has an entirely different perspective. When their product fails, it is the worst thing in the world. It is of no comfort to know that in the case of many manufactured items today, they are literally one in a million. The fact of course that so few manufactured items fail makes it very easy for a supplier to be generous and exceed statutory requirements by replacing the offending item without complaint throughout the warranty period, (up to time C in Figure 4.3) and in some cases long afterwards. The goodwill this buys comes cheap because it doesn't happen very often. Of course with an expensive product such as a car, a manufacturer is much more likely to try and mend it first but even replacement is not really a problem if such failure does not occur very often. The commercial world is so competitive that manufacturers of consumer items cannot afford to alienate customers.

Does it matter that software faults have always been present ?

We now turn to software. Here we have a product which has not yet been subjected to fifty years of Statistical Process Control and which is consequently much less reliable. However, cars breaking down fifty years ago were not greeted by a spate of litigation and nor should software today. Its immature status in spite of the huffing and puffing of software producers and the impenetrable jargon should be recognised. What is fundamentally different of course is that it does not wear out - any defects discovered have always been present. This point is often considered significant, but it is not really as the following argument reveals.

Just like any population of consumer products, a certain number of copies of software will fail within a given period. In one case, *Bernstein v. Pamson Motors (Golders Green) Ltd.,* the emergence of defects three weeks after delivery of a car was not deemed sufficient cause for the rejection of the product as the car had then been accepted. The discovery had come too late. Supposing now we have a population of 125,000 copies of software in continuous use. Now consider a rare class of defect which occurs only about once every 5000 years of use. Such defects are exceptionally difficult to remove during software testing so most if not all software products will contain them. However, in a population of 125,000 copies in continuous use, one such defect will occur about every two weeks. From the point of view of the purchaser, this is extremely serious of course but it represents an exceptionally unlikely scenario in general terms and it is probably not a defect which a producer could reasonably be expected to fix. The fundamental difference is that *all* copies of software are identical so *all* 125,000 would typically have failed the same way given the same external set of circumstances. *In other words, software failures tend to be far more systematic than normal product failures*. Is this sufficient to fuel a legal distinction ? In the author's opinion it is not, simply because conventionally engineered items also suffer from this problem. For example, marching troops are invariably required to 'break step' when crossing a bridge because even after several thousand years of bridge building, bridges are still susceptible to certain kinds of harmonically induced failure, (the Tacoma Bay Narrows suspension bridge collapse in the U.S. some years ago was a graphic reminder).

In other words, the fact that defects have always been present and that all copies fail the same way is not sufficient grounds for a legal distinction between software and other products. Bridges amongst other things share some of the same properties.

It is important to recognise this formally for this reason. The trend to increasingly more reliable consumer items which has held for the last fifty

years is now being challenged by the increasing amounts of fundamentally less reliable software now included in most consumer items as evidenced by the telephone answering machine described in Chapter 1. This should not lead to a different legal regime, unless it can be shown that the software developer concerned was fundamentally negligent in their design and production of the software and this point is discussed elsewhere in this thesis. This issue is likely to prove the most contentious. If software like a word-processor fails, returning the product is trivial and costs the supplier nothing except loss of revenue of the product if the purchaser rejects it. If software embedded in the braking system of a car fails, this is altogether a more expensive issue as evidenced by the author's own experiences as described in Chapter 1.

It would seem therefore that the issue of when a software product is delivered and more importantly when it can be rejected is not so different from normal consumer products. With normal goods however, the law and the purchaser have generally got used to the idea that things can be fixed given time and money.

However with the best will in the world, this is manifestly *not* true with certain classes of software defect so in the author's opinion there should be some kind of legal proviso based on a *sui generis* view of software that if serious software-related defects occur embedded within a consumer item and they cannot reasonably be corrected by the manufacturer, that rejection of goods should be allowed on the grounds that the defect has always been there. Note that this is not a matter for tort and the requirement to show that the supplier was negligent. Such defects can escape the best suppliers in the world.

This argument of course represents the COTS end of the spectrum. What about the bespoke end of the spectrum ? Here we are faced with a number of daunting statistics highlighted in Chapter 1. First of all, somewhere between 75 and 90% of all software projects fail to deliver their

intended functionality. Those that do succeed are accompanied by an inordinate amount of additional work after delivery known euphemistically as maintenance as indicated in Figures 1.3 and 1.4 in Chapter 1. How long after delivery should rejection be allowed in this case ? This may be a very difficult question to answer although it seems reasonable that it should be rather longer than the time allowed for products. This is precisely one of those areas which the author is sure the courts would be reluctant to comment upon and on which software experts are most unlikely to come to any form of agreement. The supplier and purchaser of bespoke software should therefore concentrate most carefully on contractual solutions as described later.

Assessing best practice in software engineering

As a general guide to the legal reader, this section attempts to define aspects which would be considered best practice in a software development environment. Some reference will be made to these in desirable contractual requirements discussed in the next section.

Perhaps the most significant step forward towards this goal in that it was underpinned by measurement unlike so much of what we do in software engineering, is the Capability Maturity Model (CMM) of Carnegie-Mellon University in the U.S., [24]. This model emerged as a result of the very high rate of failure reported in data such as that shown in Figure 1.8 in Chapter 1. The CMM effectively categories software engineering environments in 5 levels of decreasing quality or maturity as follows:-

Level	Characteristic	Key Challenges
5 Optimising	Improvement fed back into the process	• Still human intensive • Maintenance of optimisation
4 Managed	(Quantitative) Measured Process	• Changing technology • Problem analysis • Problem prevention
3 Defined	(Qualitative) Process defined and institutionalised	• Process measurement • Process analysis • Quantitative quality plans
2 Repeatable	(Intuitive) Process dependent on individuals	• Training • Technical practices • Process Focus
1 Initial	(Ad hoc / chaotic)	• Project management • Project planning • Configuration management • Software quality assurance

Table 4 The basic structure of the Carnegie-Mellon Capability Maturity Model for software.

The model is incremental, so that in order to move from one level to the next higher, the deficiencies at that level *must* be removed. For example, the priorities for a level 1 company to address before it can move to level 2 are project management, project planning, configuration management and software quality assurance. These are expanded upon and re-ordered in Table 5:

Standards and Procedures	Formal Project Management System to cover planning, estimation, scheduling and tracking. Formal Information Strategy Planning Formal Data, Process and Interaction Modelling Data Administration Prototyping Regression Testing
Organisation	Software Quality Assurance Function Staff Training Programme to support standards and procedures.
Tools and Technology	Systems planning, analysis and design support Project Management support Source code configuration management system Data dictionary if applicable Regression Testing support
Process Metrics	Collection and analysis of code error and test efficiency measurements

Table 5 The priorities which must be addressed in a level 1 company wishing to get to level 2 of the CMM.

These can be compared with the equivalent challenges facing the level 2 company wishing to move level 3 shown in Table 6:

Standards and Procedures	Defined Software Development Process Risk Management Inspections and Walkthroughs Testing Standards Design Level Maintenance Quality Management
Organisation	Software Engineering Process Group.
Tools and Technology	CASE strategy Requirements traceability
Process Metrics	Collection and analysis of life-cycle metrics.

Table 6 The priorities which must be addressed in a level 2 company wishing to get to level 3 of the CMM.

The CMM initially takes the form of a questionnaire. This should be filled out independently by both programming staff *and* management because in the author's experience, there are frequently dramatic

variations[35]. These are due essentially to over-optimism as to what can be achieved by programming management, and undue pessimism by programming staff who are frequently ignorant or dismissive of what has been achieved within the organisation. The results must therefore be analysed carefully. When inconsistencies have been resolved, the results give a good idea of the level and key deficiencies remaining and perhaps most importantly, different trained assessors will produce consistent evaluations. The author has carried out this exercise frequently although mostly on level 1 companies which are generally easy to improve.

The CMM has many intuitively attractive features which is perhaps not surprising given its strong empirical lineage. Perhaps its most intuitively appealing aspect is its incremental nature. This is in stark contrast to ISO 9001 which an organisation satisfies or not, and after which, there is no real guidance as to what the next quality steps should be. In the CMM, there is a clear set of objectives with well-defined problem areas to be solved at each stage. In particular, it illustrates that *balance* is crucial to an organisation's performance. There is no point trying to optimise a deficient process. Even quite recently, of the safety-related companies with which the author has dealt, configuration management was often a problem. This has led to extreme situations such as were described earlier when an automated code audit against standards had to be carried out on site because the company concerned were unable to deliver all the required source code components after several attempts. Furthermore, more than half of the safety-related companies with which the author has dealt were deficient in project management. This is a very dangerous deficiency as it invariably leads to a rush at the end of a development project in order to meet a deadline which has always been unrealistic. Quality cannot help but suffer, usually terminally, in such circumstances.

[35]In one company which I encountered, 80% of the questions received a *yes* from a software manager, in contrast to about 20% from a programmer working in the same group, when filling out the questionnaire independently.

[24] found that perhaps 81% of all companies audited were at level 1 with around 12% at level 2 and around 7% at level 3. At the time of writing, there were no companies known at level 5 although one or two software groups are believed to operate at this level. This highlights a further deficiency of the CMM, viz. there is so little data at levels 4 and 5 that it is difficult to define them adequately. It is important to note that Humphrey, after extensive research, believes that at least 1-2 years is necessary to graduate between each level, so climbing the CMM ladder represents a major commitment in resources and could be expected to take 10 years. The problem of so many companies being at level 1 has been resolved in a somewhat tongue-in-cheek manner by [34], who proposed a software process *immaturity* extension of the CMM model to include level 0 (foolish), level -1 (stupid) and level -2 (lunatic). Although the descriptions are highly amusing, the author has seen examples of each level in real life.

In spite of the immature state of process assessment, a great deal of progress has been made in this area in the last 10 years and assessing best practice using such models as the CMM is a well-defined activity for which formal training exists.

The CMM is not the only software process quality assessment that can be done. Sufficient is now known about the defect injection rate of software processes that good comparisons can be made and reasonably objective statements made about the standard of care used in a project. This is clearly relevant to any assessment of delictual responsibility.

Implications for software contracts

Perhaps the most important point to make is this:-

As has been seen above, the uncertain nature of software from a legal point of view is a very significant contributing factor to the current state of affairs that all cases so far decided under English Law have been decided under the Law of Contract. Furthermore, as we have seen in Chapter 1, *it is the rule rather than the exception that software projects fail to deliver what their*

intended users require. In normal life, contracts are essentially written not to be used and the overwhelming number are not. In software projects, the opposite holds. *It is very likely that they will be used and so as much care on them should be lavished as possible, particularly from the customer's side*. Another possibility is to sign the standard terms and conditions of the supplier and rely on a judge subsequently finding that some of the terms were unreasonable under UCTA77, but this seems a little more haphazard to say the least.

Given that this situation is likely to continue for some time at least, it is important to consider those factors which are of the essence in determining a successful contract for both parties because it is generally in both parties interests for a contract to be successfully concluded.

Relevant factors from the point of view of a potential buyer are these:-

a) A delivered piece of software is overwhelmingly likely to contain defects.

b) Some of these defects may prove impossible to fix.

c) Unless the software is COTS software, the software is very likely to be delivered late.

d) If the software is COTS software, its function may not be described well enough for the purchaser to decide if it satisfies the function required of it.

e) The retailer of a piece of COTS software is very unlikely to understand its function in any depth.

f) The software may only run on a very limited set of target computers and moving it to other different types of computer may be impossible.

g) If the software is in any sense bespoke, it is very unlikely that the software will do precisely what the user required when delivered.

h) It may take some considerable process of interaction between the supplier of bespoke software and the user to converge the desired and actual behaviour to a satisfactory extent.

i) It is quite possible that some of the user's desired functionality will never be successfully incorporated.

Against the backdrop of this impressive list of negatives must be balanced the fact that software generally performs some functions very satisfactorily and the prospective user must take this into consideration given that each of the above list diminishes the value of the software to the user in some way[36]. The prospective supplier is simply trying to make a profitable business out of selling licences for the supply of either its own or somebody else's software, (perhaps together with hardware), and its maintenance. When a contract fails, the basic remedy is *rescission*, ('a giving and a taking back on both sides'), which is to return the parties to the position they would have been in if the contract had never been formed. It is clearly in the interests of both parties to a contract that the contract be completed successfully, (otherwise they would not have been signatories in the first place), and certain issues must be addressed to maximise the probability of successful conclusion, over and above those normally arising in contract.

Contractual clauses for the customer's benefit

Basic process issues

Broadly speaking, the customer would like to be satisfied that the supplier had certain minimum process standards for the production of software and that as a consequence, there is an acceptably low risk that the supplier will fail to supply something satisfactory. In general, a good starting point would

[36] In my experience, user's very often suffer from inflated ideas as to what software is actually capable of. In the past, when demonstrating a particular piece of software satisfied the user's requirements closely, on occasions a user has replied, "Very good, but does it do this ...". Frequently, *this* can be so remotely related to what the software is actually supposed to do that I have been lost for words. The situation is not dissimilar to the seller of a refridgerator being asked if it can also function as a knitting machine, and if not, why not.

be a guarantee that a company operated at level 2 or above of the Carnegie-Mellon model as delineated in Table 5 above. If a company was deficient in any of these areas, it is likely that the customer would undertake an unacceptable risk. In general, most companies are not ratified to this level so the following clauses would help to reduce the risk.

- The supplier undertakes to keep the entire development including design documentation, project planning information and all source code under an acceptable Change and Configuration Control system from the beginning of the project. This system must be capable of re-creating a software build at any point in time and must be open to external inspection.

- The supplier undertakes to maintain a formal project plan for the development including at least the ability to produce PERT and GANT charts. The project plan shall contain agreed milestones no further apart than 12 weeks and individual tasks will be limited to a maximum of no more than 5 days duration if possible and should never exceed 10 days for any reason. The project plan will be tracked and updated on a weekly basis and the records will be open to external inspection. In particular, the difference between the planned date of the next milestone and the actual date must be plotted and supplied to the customer on a weekly basis, (see also Figure 1.14 in Chapter 1).

- The project must be designed in such a way as to provide incremental delivery of intermediate versions of the contracted product at regular periods. The contract must provide for fallback procedures to handle delays or significant departures from the expected functionality. Each time a project milestone is missed, negotiations must be entered either to drop functionality or allow delays, with appropriate financial balancing. Avoiding these simply puts off the inevitable.

- The supplier undertakes to maintain a defect tracking database which lists at least where a defect is found, the nature of the defect, at what stage of the life-cycle it is found, how it was resolved, and how serious it was. Every defect must be entered in the database before and after milestone deliveries. This must be open to external inspection.

- The supplier undertakes to use code inspections and to keep records of such inspection detailing the lines per hour and the number of defects found by the inspector. The records should be open to external inspection. Inspection rates must not exceed 120 lines per hour.

- The supplier undertakes to ensure that every executable statement is either executed by at least one test or an explanation provided why the statement was not executed, (for example if the statement represents a hopefully unreachable state such as a failure of internal correctness, (known as an *assertion failure*)). In other words, 100% of all effective statements will be executed by at least one test before delivery. The records should be open to external inspection.

- The supplier undertakes to correct defects in a reasonable time in return for a maintenance fee. The contract should provide for patch releases if a defect is particularly serious or regular updates for less serious defects so as not to overload the supplier too onerously. (Most products are updated twice a year).

- The supplier must set up an escrow agreement and demonstrate annually that the system can be built from the source code kept in escrow, (see the earlier section on escrow agreements and their difficulties).

- If there is any intention to future-proof a development to give it a long-lifetime, the supplier should be urged to undertake to demonstrate

that the software runs on more than one different platform without essential change. This makes sense because history teaches again and again that software generally has a much longer life-span than hardware and the customer will wish to take advantage of regular changes in hardware technology without changing the software. As a classic example, a modern 400 MHz. Pentium II PC costing around £1200 at the time of writing will comfortably out-perform a vintage 1989 Cray X-MP single processor super-computer costing around £3 million - a mere 9 years and a factor of 3000 in price-performance ! Against this backdrop, software systems very often have lifetimes of 20-30 years trapped on the same ancient hardware as witnessed by the Year 2000 fiasco discussed earlier.

- Of course, Year 2000 compliance must be guaranteed.

- Whatever clauses for liquidated damages (i.e. penalty clauses) can be introduced.

- The customer should make sure that they own the rights to any delivered hardware or software. Though this might sound a trivial point, supplier's contracts frequently contain clauses such that title to these reverts to the supplier in any dispute. Furthermore, it is generally true for software that copyright subsists in the author unless otherwise agreed, (a point discussed in detail in *Saphena Computing v. Allied Collection Agencies (1985).*)

From the customer's point of view, if a supplier is unwilling to agree to any of these, it is probably best not to enter into any contract with them. A high-quality supplier would use these techniques anyway. Of course the customer must also realise that the piece of mind bought by the above requirements only comes at a price. *Reducing risk costs money.* Doing business on price alone is manifestly stupid when the costs of failure are so high, and yet Western I.T. managers seem incapable of conducting themselves any other way and simply do not understand risk or its

management. Of course, if no supplier is prepared to enter into a contract on these terms, it almost certainly means that the project is infeasible anyway. There is a natural and frequently misguided imperative to computerise, [14].

Contractual clauses for the supplier's benefit

The essence of a contract for the supplier's benefit is to get paid and avoid liability for everything possible. This ideal (and entirely unreasonable) position of course will only be achieved with the most naive of customers, although there are regrettably significant numbers of these as documented by [14]. Legal precedent so far dictates that reliance on exclusion clauses is a high-risk venture and a supplier should be much more cogniscent of the possibility of their failure. As a result, suppliers should seek to limit their liability to reasonable amounts given that a court will take such factors as relative bargaining power and the possibility of insurance cover into account. The following clauses might appear.

- A key factor to protect against is the influence of changes in customer requirements. A supplier should attempt to secure a contractual position covering each change to requirements instigated by a customer recognising not only that each one costs money but that late changes can and frequently do prejudice the success of the project as a whole, (this effect is strongly visible in the Saphena ruling described in detail in Chapter 3).

- The supplier should seek to enter a preliminary contract for the capture of requirements allowing for any necessary prototyping to evolve those requirements into a design before any contract for the work itself arises. There should be no requirement to continue if the preliminary contract proves that a project is high-risk.

- The supplier should seek to limit liability to a reasonable amount. The lessons of *St. Alban's v. ICL* as discussed in Chapter 3 and also in *Salvage Association v. Cap Financial Services Ltd. (1993)* should

not be ignored here. A derisory amount will not help and neither will inconsistency. Any clauses forming part of a Standard Terms and Conditions will be subjected to a reasonableness test based on UCTA77. There is regrettably little case law on what is reasonable. In some cases consequential damages have not been awarded as a matter of policy, however for a direct quantifiable loss as in St. Alban's, there seems no escape.

- It is in the interest of the supplier to get the customer to accept delivery by instalments. This has a two-fold benefit. First, it follows successful software engineering principles whereby incrementally delivered software presents a much lower risk than the so-called 'big-bang' approach. Secondly, if software ever does come under the aegis of SGA79 or SSGA94, delivery by instalments would otherwise be precluded, (unless otherwise agreed, the buyer is under no obligation to accept delivery by instalments, s. 31(1), SGA79).

- The supplier will normally like to hang on to the title to any hardware and the IPR of any software if possible for ever and if not certainly until it is paid in full for the contracted work.

Contractual clauses for both parties benefit

To re-iterate, it is strongly in both parties interests to agree on a suitable set of detailed requirements, a suitable set of acceptance tests to demonstrate that these requirements have been achieved and also a documented fallback procedure perhaps in the form of penalties if some requirements are not met or extra requirements added.

- Both parties must agree to a written set of system specifications known as the requirements. The requirements must be used to write

the acceptance tests. The system specifications must be attached to the document[37].

- Both parties must agree to a detailed set of acceptance tests to ensure that the expected functionality and performance is attained. These must be attached to the contract. The acceptance tests should contain unambiguous objectively measurable targets to be agreed before the contract is signed. Disagreement on what constitutes acceptance is a common feature in the failure to deliver complex systems, [14]. The definition of suitable acceptance tests is a complex area and takes considerable time and *they are very likely to be tested in practice*.

- Both parties must agree to a suitable fallback procedure if the acceptance tests are not met. This must be attached to the contract. Given that a substantial percentage of projects overrun in cost, such procedures will usually involve the delivery of only part of the system. There is therefore a strong incentive to design a system so that it can be incrementally delivered[38]. However, the comments made in Chapter 2 suggest that if software is considered as goods, there is an inconsistency that would have to be resolved as the buyer is under no obligation to accept only part of the goods under s. 30(1) of SGA79.

  The dangers of not producing suitable fallback procedures are well-illustrated by the words of Mr. Recorder Havery QC in *Saphena Computing Ltd. v. Allied Collection Agencies Ltd.*:-

  "I do not regard this document, setting out terms which were designed to settle the dispute between the parties, as being of the

---

[37] See for example, *Saphena Computing Ltd. v. Allied Collection Agencies Ltd.* where the court ruled that they were not. This had a significant bearing on the case as was discussed in Chapter 3.

[38] This means that the system functionality is delivered piece by piece in a gradual controlled way. This is possible in most designs. If the supplier cannot offer this, it is probably symptomatic of an ill-considered and inflexible design.

slightest assistance in determining the questions I have to decide in this case, including the reliability of the witnesses."

- If the acceptance tests are passed, the customer will pay promptly and in full.

The copyright nature of software

This is an excessively complex area as can be divined by reading the comprehensive review by Millard in [28]. The relevant statutory legislation is the Copyright, Designs and Patent Act of 1988, (CDP88). Section 1 of this Act states *inter alia*:-

(1) Copyright is a property right which subsists in accordance with this Part in the following descriptions of work:-

  (a) Original literary, dramatic, musical or artistic works

  (b) Sound recordings, films, broadcasts, or cable programmes, and

  (c) the typographical arrangement of published editions

(2) In this Part, 'copyright work' means a work of any of these descriptions in which copyright subsists.

Software is included through section 3(1) of this Act under the auspices of a literary work, (a 'literary work; is defined to include 'a computer program'). The situation has already become very confused by virtue of the fact that some software systems such as those associated with multi-media are by definition 'multi-copyright', in that copyright subsists in different ways in different parts of the system which for a multi-media program would typically contain film clips, pictures, sound recordings and so on as well as the program code. It might also contain databases which themselves are covered under the guise of a 'table or compilation'.

There are significant differences in the treatment of each of these categories. For example, unauthorised adaptation of a program code constitutes infringement whereas there would be no copyright restriction on

the adaptation of the various artistic works provided it did not amount to copying or some other restricted act. To fling a final spanner in the works, the reader need only consider the above variations in the light of the section on digital convergence above. Supposing someone wished to (unlawfully) adapt a program which will be called for the sake of argument Program version 1. The nature of the adaptation could be determined and this Program called version 2 say. Consider now the situation depicted in Figure 4.4

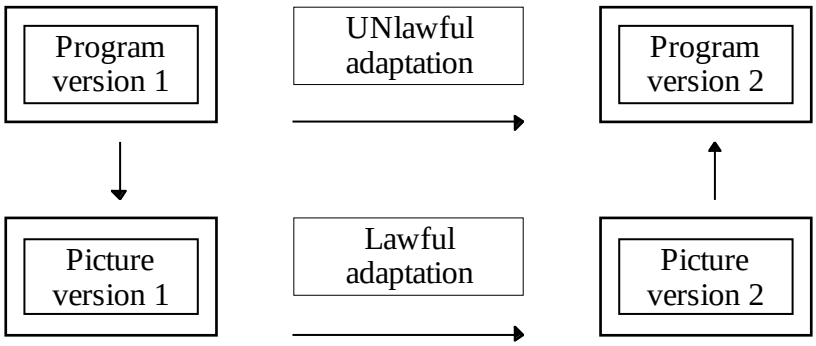

Figure 4.4 This diagram illustrates potential inconsistencies in the treatment of software in CDP88.

If the points made in the section on digital convergence are used, the program is simply a picture. If this is edited (lawfully) *as a picture* into an adapted picture which happens to have a compatible digital representation with Program version 2, then a computer program has apparently been adapted lawfully. If this action was carried out in the source code text of the original Program version 1, this would be unlawful adaptation. This could equally well have been treated as a piece of music. There is some weighty support for this point of view as evidenced by the words of Mr. Recorder Havery QC in *Saphena Computing v. Allied Collection Agencies (1988),*

> "Copies of object codes produced by Mr. Hughes by way of improvement to the software are infringing copies in so far only as they have been produced by use, i.e. copying or adaptation, of the source programs."

The clear implication of these words is that modification of the object code without modifying the source code is not an infringing act. This is clearly a difficult semantic area and simply emphasises the need for the legal system to understand the true implications of digital convergence.

Making escrow agreements work

In essence, an escrow agreement involves the licensor placing the source code of an application with a reliable third party (an escrow agent) with no vested interest in the intellectual property content of the source code. The third party is required under a tripartite agreement between licensor, licensee and escrow agent, to divulge the source code to the licensee only in a certain restricted set of circumstances, such as the liquidation of the licensor. The intention is to protect the licensee from catastrophic loss of benefit of the software by providing them with ability to maintain the software, although the scale of what is considered reasonable maintenance was considered for example in *Saphena Computing v. Allied Collection Agencies Ltd. (1985).* The court's clear verdict here was that such maintenance should be restricted to corrective maintenance only[39].

There are a number of potential problems with escrow.

Legal problems with escrow agreements

In the case of liquidation or receivership, the official agent may have considerable problems with an escrow agreement as noted in Chapter 2 of [28], and could conceivably attack it in the following ways:-

- an escrow represented a preference under s. 239 of the Insolvency Act (1986), (s. 340 for personal bankruptcy). This section would only apply where the escrow was set up within 6 months of the insolvency AND where the licensee was creditor of the licensor, which may very well be the case for software,

[39] Recall from Chapter 1 that there are three forms of maintenance, Corrective, Adaptive and Perfective. The court ruled in the case of Saphena that only corrective maintenance was a permissible activity and any other act effectively breached copyright. The distinction between corrective and adaptive maintenance was pointed out by the plaintiff's expert witness although not in these words.

- that the arrangement breached the *pari passu* rule embodied in s. 107,
- that the arrangement was an unprofitable contract under s. 178,
- that the arrangement could be repudiated as a matter of general law by the official agent.

A further legal difficulty is that the release of the escrow would require a court order under s. 127, which would only be granted if the court was satisfied with the legality of the escrow arrangement.

Technical problems with escrow agreements

In spite of the above potential legal problems, escrow arrangements appear to have been set up quite satisfactorily in the past. Regrettably, the technical problems can even surmount the legal problems and undermine the most legally satisfactory of escrow arrangements.

The problem is a simple but profound and ubiquitous one. Recall that an escrow arrangement exists to allow the licensee to reconstruct (perhaps on a more up to date computer) or modify (for the purposes of corrective maintenance) an executable form of the licensed software. In order to do this, all the necessary software components must be present in source form along with the precise recipe for building the required executable from these components and herein lies the problem. The area of computer science which covers this activity is known as *change, configuration and build control*. According to studies carried out by the Software Engineering Institute at Carnegie-Mellon University, [24], a significant percentage of companies, (certainly over 50%) had no satisfactory means of rebuilding executables themselves. The acid test is whether the software package can be built correctly on a 'clean machine', i.e. one which is physically disconnected from any of the other machines on a licensor's network and on which only the basic system software is installed.

To carry out this test, the licensor must produce on some standard medium, (floppy disc, cartridge tape or some similar form), a complete distribution of all the components of a particular application in source code form. The clean machine must contain the appropriate system software components such as a compiler to build the executable. This is precisely the circumstances in which a source code audit is carried out, [23]. The author's experience is that most companies are unable to supply a complete distribution when asked the first time and a significant dialogue then follows whilst the remaining pieces are supplied bit by bit. The process often takes two or three iterations and in some cases, fails completely so that the source code audit has to take place on one of the licensor's networked machines where all of the code can be found 'somewhere'. It really is that bad[40]. As a result, it is the author's view that although an escrow arrangement may be legally satisfactory, it is very unlikely to be able to produce the end product of a satisfactory working executable which is of course the whole point of the exercise. The following piece of advice is therefore offered to the would-be user of an escrow arrangement.

It is highly recommended that an escrow arrangement should include a clause which requires the licensor to demonstrate annually that the source code in escrow along with build recipes can indeed on their own be used to produce the desired executable by any reasonably qualified person following the escrowed instructions, and that acceptance tests sufficient to demonstrate that the executable has the desired behaviour be run.

This simple expedient should remove any technical difficulties. The licensor should not feel put out by this. First of all, it is a useful test of their own change, configuration and build control, and second, they should not be expected to do it for nothing. The licensee also benefits in the

[40] I am not exaggerating here. This actually happened to me when auditing the source code for a safety-related software system under development for a water company.

knowledge that the escrow arrangement will function as desired in the unpleasant but hopefully unlikely eventuality of it being necessary.

The Year 2000 problem

A huge amount of legal opinion has already been written on this subject but of course there is as yet a paucity of case law. It will not be long in coming. Here only the nub of the issue will be addressed from the point of view of a computer scientist. In essence, the central points are these:-

(a) Were the software engineers who were responsible for systems which are not capable of handling the millennium change negligent ? A related question is what cut-off date should be used to judge this or is it one of simple intent. In other words, can we say that before 1995 say, engineers had compelling reasons for not allowing for the millennium change.

(b) What is the position of engineers now working on fixing this problem and who fail ?

For the reader's information, the Year 2000 problem, also known as the Millennium problem or bomb, Y2K and other contractions, comes about because programmers historically have not kept century information with their dates. So the year 1985 is internally stored as 85 and so on. Of course as soon as 2000 comes, this form of year counting resets itself to 00. Why was the century information discarded ? The normal reason given is that the limited storage of computers in the 1960s and 1970s meant that storage was at a premium and many programmers believed that their software would be replaced long before 2000 arrived. There is a long history throughout the 1960s and 1970s of large software systems lasting 20 years or more. This is the rule rather than the exception. In other words, from 1980 onwards, it is increasingly more likely that software would still be in operation in 2000. In parallel, the explosive improvement in computer resources such as volatile memory (i.e. RAM), non-volatile memory such as discs meant that the extra space problem rapidly became insignificant.

Furthermore from the mid 80s onwards, the problem began to arise as applications such as annuities and investment and retirement plans began to creep into the next century.

We can summarise that from several points of view, it is very difficult to argue in favour of discarding the century part of a date from around 1985 onwards. However, as recently as early 1997 according to various surveys done in England and indeed in other parts of the world, some 80% of all companies had done precisely nothing for the simple reason that the conversion costs a lot of money. Some so-called industry pundits were even claiming that it was a huge confidence trick brought about by software consultants to drum up business. It is certainly the author's opinion that companies have been extremely lethargic in doing anything, however, it seems fairly clear that unless a system was actually designed since about 1985 with this problem that the bulk of the blame lies on the senior management of companies for continuing to do nothing. The bottom line of all this taking into account the status of those who started on the problem earlier than others is that if a company with any reasonable amount of software started on this problem later than about 1995, they are unlikely to finish in time. For some, 2000 will come even quicker than others because the code '99' is often used as an end of record marker in COBOL, (a language very commonly used for business systems). They will have to finish by the end of 1998.

Perhaps one last point will underline the hapless state with which we have prepared ourselves. Those very few conversions which have already been done successfully, ([15]), suggest that somewhere between 50 and 60% of the entire exercise is in testing, an area of computer science which has been massively neglected over the years. Test expertise is very hard to find. On top of this, the results of a famous study done at IBM in the 1980s were mentioned in Chapter 1 which indicated that every corrective change to a piece of software had about a 15% chance of introducing a defect at least as bad - 'fix 7 introduce 1'. It is widely believed that Year 2000 work is

so boring, (the author can personally vouch for this), that the spoilage ratio is twice as bad as this. Needless to say, this factor has been left out of all the calculations. If this weren't enough, of the three standard ways of correcting this problem, (they are known as windowing, compression and expansion), most companies are choosing windowing which simply puts the problem off a little while and worse, is the most likely of the three techniques to introduce new defects through the mechanism of unexpected *side-effect*, a common problem faced when modifying a system whose behaviour is not well-understood.

Perhaps as a computer scientist, the author should defend his own industry but it is difficult to defend the woefully ignorant and sluggish approach to this problem exhibited particularly by I.T. management and they richly deserve what they are about to get. Unfortunately, a lot of innocent users will suffer too. Let the litigation roll ...

Scene: It is the Year 2000. Yet another case has appeared before the courts concerning the complete and traumatic failure of a financial system.

Prosecutor (P): "When were you first aware of the need to handle the end of the century correctly in your computer programs ?"

Defendant (D): "Er, 1998"

P: "Isn't that a little late don't you think ?"

D: "Oh no, we felt that 2 years was plenty of time - at least that was what all the magazines said - 'act now before its too late'".

P: "But it already was too late wasn't it ?"

D: "Er, yes, we under-estimated how long it would take to test the new systems".

P: "By how much ?"

D: "We don't actually know yet".

P: "What approach did you take to handling the Y2000 problem ?"

D: “We decided to re-write the entire system using a Client/Server Object-Oriented architecture using C++ to promote maintainability, portability, reliability and easily handle the coming problems.”

P (who has heard this sort of thing before): “On what grounds ?”

D: “We read about it on an advertising board in Waterloo station and it was confirmed at a major computing exhibition in a talk entitled “Re-write your system using C/S O-O architectures using C++ to promote etc. etc.” Also a friend in the Gromet and Flywheel, my local pub, told me it would be perfect.“

P: “In other words, you embarked on this without any evidence that it would help ?”

D: “Er, I suppose so.”

P: “Did you tell your investors of your intentions ?”

D: “No, because as system administrators we knew what was best for our investors.”

P: “I take it these are the very same investors who can no longer find out what has happened to their investments which seemed to have disappeared in the post 2000 black hole ?”

D: “Er, yes”.

P: “Before you started writing did you actually track defects and defect costs so that you would know if your proposed new system would be better or worse ?”

D: “Er, no”

P: “Did you design regression testing suites so that you could incrementally test the new system’s behaviour against that of the old ?”

D: “Er, no, not exactly, as we knew we would get it right and we didn’t really have time”.

P: “Did you have prior experience of Client-Server Object-Oriented Systems before you embarked on this rather ambitious scheme ?”

D: “Er, no, but we decided to use the best sub-contractors”

P: “Did you employ these by means of competitive tender ?”

D: “Yes”

P: “So you mean the cheapest ?”

D: “Yes, I suppose so”.

P: “Did they have any such experience ?”

D: “Er, no, but they insisted on doing it this way.”

P: “Why did they insist on doing it this way ?”

D: “Er, because they wanted it to put it on their CVs.”

P: “So, you were prepared to invest a significant amount of your investors money and risk the rest to develop a system about which you understood nothing using sub-contractors who did not understand anything either so that they could put their experience on their CVs ?”

D: “I suppose so.”

P: “And how did you control these sub-contractors ?”

D: “Well they were all rocket scientists”

P: “I beg your pardon !”

D: “They had Ph.Ds in mathematics and physics and things like that.”

P: “Did they have any qualifications in software engineering and software reliability ?”

D: “No, I don’t think so.”

P: “How did you control basic issues such as change and revision, project planning, estimation, management and risk, and the ultimate quality of the software”

D: “We left it up to the sub-contractors.”

P: “Did you pay them to do this as part of the project ?”

D: “Er, no, it would have added too much to the expense and we have a duty to look after our investor’s money.”

P: “These are the same investors who have now lost everything I take it ?”

D: “Er, yes”

P: “Did you in fact have any change, revision, planning, test and quality records for the delivered software ?”

D: “Er, I don’t think so”.

P: “Would you say that this was something to do with the fact that the delivered source code does not compile properly, does not correspond to the delivered object code or system documentation and that the sub-contractor has now gone on to his next job.”

D: “Yes, I suppose so.”

P: “I rest my case.”

The People: “Guilty, guilty, guilty !”

Perhaps the reader might find this a little alarmist, but it seems in many ways inevitable after the practices the author has witnessed in the good name of software engineering in the last few years and reflects the almost unbelievable nonchalance with which the software industry in general and the financial sector in particular views both software reliability and the end of the century. *As a matter of interest, although the whole of the above confrontation is a collage, each of the responses by the “Defendant” has been taken from a real-life response by one of a large number of companies the author has talked to in the last few years.* ***None*** *of it was made up, so if the “Defendant” is taken as the software engineering industry as practised in general, the dialogue is real, (and so should be the verdict).*

This section will end with a brief list of application areas which could be affected by Year 2000 problems:-

- Fire and burglar alarms
- Bar-coders
- Embedded control systems in manufacturing
- Air-traffic control systems (possibly very badly)
- Global navigation systems
- Anything with expiry dates - canning factories and so on
- Time and date logs in FAX machines and similar
- Employment clock-in systems

- ATM machines and indeed any card-access systems which expire the cards
- lifts
- Central heating, air conditioning, light and heating systems
- Building management systems
- Medical system
- Safes and time locks
- Security and access control systems
- Telecommunication systems
- Utilities such as water, gas and electricity distribution
- Banking systems generally as they have some of the most complex, out of date and poorly designed software the author has yet seen.

This list is by no means complete. The author did a project on this topic in 1997. At the project start, he felt that the whole area was somewhat hyped up. After finishing the project, he found that if anything, it may have been understated. There are clearly problems ahead and litigation may be the least of our worries.

Benefit versus inconvenience

In the author's view, any discussion of liability for defects in software should be mitigated with the benefits which accrue for the price paid. The unique property that software can be copied limitlessly and precisely at very low cost means that purchasers of COTS software generally get a very good deal indeed. Chapter 1 has highlighted the difficulties faced by computer scientists as they attempt to produce reliable software, COTS, bespoke or hybrid. Yes it is certainly true that we have so far failed to achieve standards even approaching the rest of engineering, but software

engineering is relatively immature as yet. Any legal witch-hunt against software suppliers such as might arise in the wake of the Year 2000 problem is in grave danger of throwing the baby out with the bath-water and it must be recalled, society now depends completely on software and has for some years now. There is a certainly a lot of substandard practice but given that even the most qualified of computer scientists is quite likely to be unable to deliver a complex system in any but a general way resembling what its end-user expected, any liability must be traded off against the benefits and the cost of such benefits.

For example, most of the author's PC software is substandard in terms of general reliability. Such well-known products as Windows in its various forms collapse at very regular intervals including at the official opening of Windows '98 by none other than the CEO of Microsoft, Bill Gates[41]. However, it is also very inexpensive on a single user basis costing a few 10s of pounds. The author's own annoyance with these products stems from the fact that they could be considerably better for a similar price, a personal view based on his experience as a computer scientist. Where there seems real potential for action is when a supplier builds a generally high-integrity system such as a medical imaging system around a COTS product of known relative unreliability such as Windows simply because it is widely available and therefore 'acceptable' in some sense[42]. This is not dissimilar to building a bridge from cardboard because there was a good supply at the local supermarket. Ready availability at a good price does not imbue a product with reliability. This is already happening on a large scale and is particularly worrying. Draft international standards such as IEC 61508 encourage end-users to *verify* that a product will achieve an acceptable level of reliability to a level commensurate with the risks associated with the system.

---

[41] An event greeted by undisguised relish by myself and about 300 million other computer users.

[42] This is the trend in modern medical imaging systems to reduce costs.

In general, however, most software tends to behave reasonably most of the time and users would prefer to have it than not. This is arguably due to the Darwinian effects exemplified in Figure 1.8, in that most of the really bad stuff never sees the light of day. Certainly modern telecommunications and the World Wide Web would cease to be if software were to be removed as would a number of other fundamental areas, which society as a whole in the information age would not be prepared to forgo. Any approach to liability for failure must then reflect this benefit and this therefore places particular emphasis on the need for industry-acceptable levels of competence to be used as a basis for a best practice.

Of course, trade-offs between benefit and risk have been around in engineering for many years. This is true even for critical systems. A classic example of this can be found in the commercial aviation industry where a treaty known as the Warsaw Convention attempts to limit the liability of a carrier in the event that passengers lose their lives. The idea is that the huge benefits of air travel should not be threatened by unlimited liability. In fact, liability is set at derisory levels with a life being worth around $75,000 dollars including legal fees. Some countries such as Japan do not accept this as exemplified in the Airbus crash at Nagoya a few years ago where liability was set at $1.8 million per passenger. As another example, consider the errant ABS system discussed in Chapter 1, Table 2. In general such systems are manifestly more effective than manual techniques and although they existed before computerisation, they have become far more affordable and therefore available with the advent of computerisation. Clearly, if someone is physically injured as a result of the failure of such a system, liability is strict, however, no court could afford to ignore the general effectiveness of such systems. As was pointed out earlier, systems which fail to provide a benefit at least commensurate with their failings rarely see the light of day and then not for long.

As another example, airbags are thought to be responsible for around 90 deaths in the US, usually the elderly or small children. Against

these tragedies they are also believed to have saved a rather larger number of lives in the same time interval. Not very long ago, the author trained a number of the engineers responsible for airbag deployment for a major car manufacturer. They explained that airbags originally deployed with an explosive force equivalent to a stick of dynamite[43]. Advances in technology both in hardware and software have reduced this to around a quarter stick of dynamite. This is clearly an example of a beneficial technology with a risk - the driver and front seat passenger sit facing a software-controlled motion-induced bomb, but the statistics are weighted in favour of the beneficial effect. Of course such trade-offs are not unknown for example as frequently occur in the medical world, but a surgeon does not warrant to save his patient and the law has considerable experience in dealing with this.

The only problem we face here in the long run is the author's belief that for software-controlled systems, things will get worse again before they get better owing to the dramatic rise in software content in consumer devices which will erode at least temporarily the very high levels of hardware reliability achieved using modern manufacturing methods. This is evidenced by the answering machine example also referred to in Chapter 1, Table 2, which is considerably more unreliable than the technology it replaced and manifestly unpopular in the author's family.

[43] The detonation chemical used requires an explosives licence for transport.

Chapter 5: Summary and suggestions for further work

This thesis has attempted to build bridges between the legal perspective and the perspective of a pragmatic software engineer. To some extent it has succeeded and in other ways it has fallen short of the author's expectations.

For example, when the author started he believed that he could resolve the 'goods v. services' issue rather more satisfactorily. The more he studied it however, the more he believed that the issue is essentially unresolvable as a series of legal formulae appropriate to statutory legislation. This is because of the continuous nature of the spectrum between COTS and fully bespoke software which as demonstrated in Chapter 3, can be related to an equivalent continuous spectrum of duty of care, a delictual issue and concurrently to a continuous spectrum between the treatment of goods on the one hand and services on the other. Rather, the author now believes that software would be much better treated *sui generis* and requires statutory guidance, probably under an extension to the Sale of Goods Act, to avoid inconsistent interpretation even though case law so far has led to eminently reasonable conclusions, albeit essentially derived either explicitly or implicitly from contractual issues rather than any form of delictual or product liability. There is some general argument in favour of a *sui generis* approach as evidenced by the *Beta Computers (Europe) Ltd. v. Adobe Systems (Europe) Ltd. (1995)* although this case revolved around an interpretation based on an element of Scottish law with no equivalent in English law.

One aspect becomes very clear in all of this. The greatest burden for any other than pure COTS software appears to fall on contract until statutory issues become clearer. There is therefore an urgent need to draft software contracts covering issues as detailed in Chapter 4. This may require a fundamental change of emphasis on behalf of lawyers. In general, contracts are written not to be used, in full expectation that the end-product

will be achieved without too many problems. Of course, contracts still fail and lawyers are kept busy as a result. Software development projects are manifestly different however. They *usually* fail, so contracts will need to be much more aimed at recovery based on partnership between those privy to the contract. Conventional remedies are not very useful to either party in the wake of a software project failure.

In terms of further work, there is clearly going to be a flurry of litigation in the wake of the Year 2000 problem with much more public attention focussed on software failure. It is hard to predict how this will turn out, although it is hoped that things will improve as a result rather as the disaster of the Tay Bridge collapse in 1879 led to a little later to the extraordinarily over-engineered Forth Bridge which could probably survive a direct impact from a comet. The first cases against COTS software will help to clarify opinion, but the recommendation of this thesis is that for consistency, (for example to parallel the courts' willingness to imply reasonably fitness for purpose in contracts), they be firmly treated as goods under SGA79 taking account of those problem areas mentioned in the body of the thesis.

In software engineering just as in other areas of human endeavour, disaster is regrettably the mother of innovation in engineering and litigation is an important control in prescribing what is a reasonable trade-off between benefit and risk for the end-user. As in all engineering, the balance between benefit and risk changes quickly as the discipline improves.

Appendix A: Proposed changes to the US UCC, (Universal Commercial Code), and implied terms in UK law.

The central reason for including a discussion of this US statute is quite simply that most software in use in the world originates there. It is therefore of great interest to international users of US software as to how US law compares with for example UK or European law, given that US software contracts usually specifically state that the contract falls under the jurisdiction of US courts.

The UCC is the fundamental underlying source of American commercial law and a new law, Article 2B is currently being drafted to help address disputes arising over software, [35]. The UCC plays a similar role in many ways to both the Sale of Goods Act (1979) and the Goods and Services Act (1982). Although only in draft form, and already very complex at 216 pages at the time of writing, it is already exerting great influence on the US legal system. Its contents are of great concern and clash in some cases rather violently with equivalent UK and European laws. This appendix will discuss these concerns briefly.

In essence, according to [35], *inter alia*, the new law will:-

a) Legalise Draconian restrictions on use, reverse engineering and improvement of interoperability of software products

b) Organise the law around a packaged (i.e. COTS) software model, an area which casts little light on bespoke or consultatively supplied software, which will nevertheless have to conform.

c) Make it easier for the supplier to reduce their responsibilities to the purchaser.

These are clearly in considerable antipathy with significant aspects of UK and European law and each section will now be discussed with reference to relevant UK and European law.

Restrictions on use.

Article 2B's sections 2B-312 and 313 allow the software supplier (licensor) to say where the program can be used, what machines it can run on, who can use it and what purposes the customer can use the program for. In effect, these sections would allow a supplier to deny the purchaser any rights to third-party maintenance, as has already been upheld in [36].

In the UK, section 7 (4) (a) of the UCTA, 1977 states

> "liability in respect of the right to transfer ownership of the goods, or give possession, cannot be excluded or restricted by reference to any such term except in so far as the term satisfies the requirement of reasonableness".

It is highly debatable whether the courts in the UK would consider such restrictions reasonable.

The packaged software model

Article 2B attempts to describe all legal relationships concerning software in terms of a software publisher and an end user. We have already seen in earlier discussions, that this COTS model is fundamentally different than other types of software and no legal precedent yet exists. On the other hand, there is strong legal precedent in the form of the Saphena and St. Alban's rulings discussed in Chapter 3 for the delivery of software in which there is a degree of consultancy. It can be expected that there will be fundamental differences in litigation arising from Article 2B and what has already arisen in the UK. Prospective UK purchasers of US software should take heed.

Reduced responsibility to customers

Existing US law in the shape of Article 2 of the UCC takes a similar view of the so-called “AS IS” clause, in which the purchaser assumes all of the responsibility, to that in the UK, in the sense that it allows for disclaimers but against a background judgement that they are a disreputable practice. Although in the UK, they are subject to the test of reasonableness as described by the UCTA 1977. An unreasonable clause will be deleted by the courts.

The proposed replacement clause 2B specifically allows for the model whereby the customer does not see most contract terms until after the product is purchased, for example, by being forced to click on an “I agree” button during installation. Of course the purchaser can decline and take the product back to the supplier, but if he or she clicks on the “I agree” button[44], then the terms become part of the licence however harsh.

Again, it is difficult to foresee how this would be treated in the UK. It may well be necessary for statutory provision to be made for some equivalent to the breathing period allowed in financial contracts in the *Financial Services Act (1985)*. It is certainly true from an engineering point of view that it makes it easier for a software company to sell low-quality products without legal recourse.

[44] This can actually happen inadvertently as modern graphical user interfaces store an arbitrary and sometimes load-varying number of clicks and supply them to the serviced application in some time-dependent manner. It is all too easy to click once too often and have this stored and applied later to something which the user did not intend.

References

1. Hatton, L., M.H. Worthington, and J. Makin, *Seismic Data Processing: Theory and Practice*. 1986, Oxford: Blackwell Scientific Publications. 177.

2. Sommerville, I., *Software Engineering*. Fourth ed. International Computer Science Series. 1992, Wokingham England: Addison-Wesley. 334.

3. Boehm, B.W., *Software Engineering Economics*. 1981, Englewood Cliffs, New Jersey: Prentice Hall.

4. Arnold, R.S., *On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality*, . 1983, University of Maryland.

5. Kaletsky, A., *Snakeoil, software and Gates*, in *The Times*. 1997: London.

6. Wayt Gibbs, W., *Software's Chronic Crisis*, in *Scientific American*. 1994. p. p. 72-81.

7. Mellor, P., *CAD: Computer-Aided Disaster*, . 1994, Centre for Software Reliability, City University, London.

8. Hatton, L., *Computer programming languages and safety-related systems*, in *Proceedings of 3rd. Safety-Critical Systems Symposium*, F. Redmill and T. Anderson, Editors. 1995, Springer-Verlag.

9. Woodward, M.R., D. Hedley, and M.A. Hennell, *Experience with path analysis and testing of programs.* IEEE Transactions, 1980. **6**(3): p. 278-286.

10. O'Donnell, J., *GM: Computer glitch may cause brake problems*, in *USA Today*. 1998: Detroit. p. B1.

11. Leveson, N.G., *Safeware: System Safety and Computers*. 1995, Reading, Mass: Addison-Wesley.

12. Neumann, P.G., *Computer-Related Risks*. 1995, New York: Addison-Wesley. 367.

13. Hatton, L., *Software failures - follies and fallacies.* IEE Review, 1997. **43**(2): p. p. 49-54.

14. Collins, A. and D. Bicknell, *Crash: Learning from the World's Worst Computer Disasters*. 1998, Sydney: Simon & Schuster. p. 428.

15. Hatton, L. *The Year 2000: How much should we worry ?* in *Eurostar '97*. 1997. Edinburgh.

16. EC-85/374, *Directive on Product Liability*, . 1985, EC: Brussels.

17. Yourdon, E., *Decline and Fall of the American Programmer*. 1992, Englewood Cliffs, N.J.: Prentice-Hall. 352.

18. Adams, N.E., *Optimizing preventive service of software products.* IBM Journal Research and Development, 1984. **28**(1): p. 2-14.

19. Hatton, L., *The T-experiments: errors in scientific software*, in *Quality of Numerical Software, Assessment and Enhancement*, R.F. Boisvert, Editor. 1997, Chapman & Hall: London. p. p. 384.

20. Hatton, L., *Re-examining the fault density - component size connection.* IEEE Software, 1997. **14(2)**(March/April 1997): p. p. 89-97.

21. Hatton, L., *The T experiments: errors in scientific software.* IEEE Computational Science & Engineering, 1997. **4**(2): p. 27-38.

22. Pfleeger Lawrence, S. and L. Hatton, *Investigating the influence of formal methods.* IEEE Computer, Feb. 1997, 1997. **30**(2): p. pp 33-43.

23. Hatton, L., *Safer C: Developing for High-Integrity and Safety-Critical Systems.* 1995: McGraw-Hill.

24. Humphrey, W.S., *Managing the Software Process*. 1990: Addison-Wesley. 494.

25. Genuchten, M.v., *Towards a Software Factory*, . 1991, Eindhoven.

26. Brooks, F.P., *The mythical man-month*. 1975: Addison-Wesley.

27. Lloyd, I.J., *Information Technology Law*. 2nd edition ed. 1997, London: Butterworths. 508.

28. Reed, C., ed. *Computer Law*. . 1996, Blackstone Press Ltd.: London. 396.

29. Leder, M. and P. Shears, *Consumer Law*. M+E Law Handbooks. 1991, London: Longman. 279.

30. Triaille, J.-P., *The EEC directive of July 25, 1985 on liability for defective products and its application to computer programs.* The Computer Law and Security Report, 1993. **9**: p. 214-226.

31. Lloyd, I.J. and M.J. Simpson. *Legal aspects of software quality*. in *Software Quality Management*. 1993. Southampton: Computation Mechanics Publications, Elsevier.

32. Beizer, B., *Software Testing Techniques*. Second ed. 1990: Van Nostrand Reinhold.

33. Hatton, L., *Software Failure: avoiding the avoidable and living with the rest*. 1999: Addison-Wesley.

34. Finkelstein, A., *A software process immaturity model.* ACM Software Engineering Notes, 1992. **17**(4): p. 22-23.

35. Kaner, C. and B. Lawrence, *UCC Changes pose problems for developers.* IEEE Software, 1997. **(14) 2**(march/April): p. p.139-142.

36. Reporter, F., *MAI Systems Corp. v Peak Computer Inc.*, . 1993, US Court of Appeals for the Ninth Circuit.