Computer Risks and Insurability of Software

Prof. J.A. Makowsky
Department of Computer Science,
Technion - Israel Institute of Technology, Haifa, Israel
and
Scientific Consulting, Zurich, Switzerland

April 23, 2009

Contents

1 Introduction

This report is the result of my investigations into the questions discussed with Dr. Jenny and Mr. Augustin during our several meetings in spring and summer 1989. It was then agreed that I should invest 25 man/days of work to prepare this report. I have enlisted the help of Prof. M. Hofri and A.Herzberg, a Ph.D. student, both at the Computer Science Department of the Technion, and of Dr. J.-C. Grégoire, whom I met at the Swiss Federal Institute of Technology in Lausanne and who currently is a post-doctoral fellow at the Center for Intelligent Systems and Robotics at the Technion. I am currently the scientific head of the center. The report is written in English, so my collaborators could also read it.

I want to repeat here the task of my investigation as I have understood them:

- Based on the information provided by **ZURICH** about their computer network, I was to provide a few scenarios exemplifying the risks, to which **ZURICH** might be exposed by hostile attacks tampering with software only. The results of my investigation are contained in section 2 of this report, which was drafted by A.Herzberg.
- Based on the draft documents concerning liability insurance for software producers (especially concerning software for auditing purposes), I was to evaluate the feasability of these draft contracts and to provide a basis for the formulation of future software liability insurance contracts. The results of my investigations are presented in section 3 and 4. They were obtained in collaboration with Prof. Hofri and Dr. Grégoire.
- I was to provide several proposals for the continuation of our collaboration concerning software risk engineering and the insurability of software caused risks. My long term view of software risk engineering is elaborated in section 4, and a proposal for a future collaboration for the next two years is presented in section 5.

I have on purpose exceeded the work efforts to prepare this document by at least 10 man/days. I see this additional work as an investment into further collaboration.

2 Hostile Software Risks: Illustrative Scenarios

2.1 Introduction

An interesting and important category of computer risks are risks due to *hostile software.* This section contains several *scenarios* illustrating such risks. The purpose of this section is to help the decision-makers to understand hostile software risks. For this purpose, the scenarios reported are all *realistic scenarios.*

The context of all the scenarios is the computer systems of the **ZURICH** Insurance Company, as described in the documents we received (dated 16.06.89 and 23.03.89). Apart from being based on the report of the computer facilities at the company, and on our knowledge of existing and possible risks, this report is *entirely fictitious.* In particular, this report is neither an analysis of the risks to the security of the computer systems, nor a complete and exhaustive set of examples for all the possible risks. There are many risks which we did not exemplify. Also, the examples are limited by our *very incomplete understanding* of the data-processing operations of the company. Furthermore, this report does not try to give a comprehensive account of the different risks and the methods to defend against them; we included only brief remarks about the most obvious weaknesses exploited by the scenarios, and a very high level introduction to the basic risks and countermeasures.

It is clear that the use of software, as in the case of any sophisticated tool, carries risks even when no hostility is involved. Discussing those is beyond the scope of the present document.

2.2 Organization of this Chapter

In the next section we give a general introduction to the risks from hostile software (and badly protected hardware), and the protection against them. The following sections contain three scenarios: a 'typical' virus, a 'directed' virus, and a 'classical' system penetration. In the conclusions we discuss some weaknesses of the computer systems of the **ZURICH** Insurance Company, as described in the documents we received (dated 16.06.89 and 23.03.89).

2.3 Introduction to Hostile Software Risks

Presently, there is no one accepted taxonomy of software risks. We will try to use a coherent taxonomy which seems widely accepted.

Sources and types of hostile software The general term for a program which is supposed to do a certain task, and (intentionally) also does some other (hidden) task is a *trojan horse.* When the hidden task is invoked only at a specific date, or after specified elapsed time, the software is called a *time bomb.* A trojan horse may be locally created by an (otherwise authorized and trusted) authorized programmer, or be imported, installed inside a new software product, with no one at the site the wiser. However, there are two other methods of infection, which may affect a computer without direct contact with the location of the trojan horse. When such is the case we obtain the types that are called a *virus* and a *worm.* Recently, software viruses became a subject of much concern.

A virus is a part of a program that adds itself to a program, and propagates by attaching itself later to other programs as well. A specific virus may add itself to different kinds of programs, where a program is any kind of code which is (eventually) executed, including machine-loadable programs (object-code modules), source programs, command files and macro files. The virus may combine with system programs which normally do not appear in the file and disk status reports that are routinely produced.

Programs to which a specific virus may add itself (infect) are called the *hosts* of that virus. When an infected host program runs, and only then, the virus code is executed. One of its functions is to add the virus code to additional host programs; other functions may be less frequent and more harmful, e.g. to erase all data. While the virus copies itself to a new program, it may change itself (mutate) or change the host (e.g. to avoid detection or to adapt itself to a different kind of host). For example, the virus may compress the host program, so that the length of the infected program remains identical to the length of the original program.

All viruses publicized so far, based their operation on the weakness of the protection mechanisms of microcomputers. Namely, in most microcomputers and their operating systems, there is no restriction on the file operations of a program. In particular, a program may change (write into) the operating system or other programs. Therefore, the virus can easily copy its code to additional host programs. All anti-viral software products known to us attempt to prevent or detect such unauthorized modifications in filed programs or in main memory.

However, the problem of viruses runs deeper; even if the protection mechanisms are infallible, the following virus is still possible. Suppose one accepts a compiler which contains the virus. Whenever this compiler compiles a program, it appends the virus code to the normal code of the program. In particular, whenever a new compiler is being compiled, the virus will be copied into it. This problem exists even in the most protected operating systems known to us. A virus of this kind was programmed in 1984. The known hosts of this virus are (only) versions of the **C**-language compilers. Whenever the infected compiler compiles a new **C**-compiler, the virus is added to it. Whenever the compiler compiles the **login** procedure of the UNIX operating system, the virus plants a *trapdoor* in this procedure which enables its author to penetrate the system.

A worm is a program that can copy itself to other computers. A worm is a whole program, not a part of a program, and does not combine with existing programs. To infect a computer, the worm must be copied to this computer and be executed there.

Some worms spread between computers via software exchange between users. Those worms pretend to be (are described in the catalogue as) useful programs, sent electronically to the user by some correspondent known to the user. When executed, the worm sends (electronically) copies of itself to other users, for example those listed in the user's addresses file. In addition to possible harmful activities, these worms may cause so much traffic that the network may collapse.

Other worms execute completely automatically, utilizing a *bug* or some other *features* of the electronic mail services.

2.4 The Risks from Hostile Software

The following are the risks related to hostile software:

Loss of data or of data integrity. The risk of loss of data integrity is more difficult to detect than that of entire files, and hence more severe.

Disclosure of secret information. If some information is perceived as very sensitive, cryptographic protocols and encryption can be used effectively. The real danger comes from gaining access to secret information which is implicitly stored in the computer, with each explicitly stored part being of a lower degree of secrecy.

Denial or degradation of service. Office automation and other automated services are always compared to similar non-automated services. Availability of service is a major asset of a service rendering company. Availability *is a commodity* whose value cannot be underestimated, but has not yet been perceived as such in insurance contracts.

Circumvention of restrictions and security mechanisms. There are two kinds of hostile software which cause this risk:

- Some programs are authorized to do some special operations, in the normal course of operations, which are not possible to *regular* programs; but the program is supposed to use these operations only for specific ends. Such a program is said to contain a trapdoor if by using some undocumented feature, the user gains control of the restricted special capabilities. For example, the program may have access to special files (intended for a specific use), and a trapdoor may enable the user to freely modify these files.

- In most protection systems, the user usually has to supply his password when logging into the computer (and possibly more passwords later, for additional operations). A *spoof* is a program which pretends to be the password requesting program of the system, but was actually prepared by an attacker. The spoof simply reads the the password that the user types and makes it accessible to the attacker.

2.5 Protection Against Hostile Software

An immediate conclusion from the theory of computation (Rice's theorem) is that it is impossible to verify automatically the correctness of a program. A corollary is that it is also impossible to check that a program is not hostile (e.g. that it does not contain a virus). However, if the source code is available, then sometimes manual software verification is possible. By proper restrictions of the programming language, it is possible to prevent the program from accessing data which it was not supposed to access. This may be achieved also without restricting the software, by appropriate hardware mechanisms, implemented in most of the processors with protection systems. Those hardware mechanisms try to prevent or detect the harmful activity (when it is attempted by the software). They are not predictive. Usually, such hardware is designed to support a protected operating system, thereby:

- Programs can access only permitted files.

- Programs interact only in designated manners; in particular, one program cannot modify another.

- Specific I/O activities are permitted only to system software.

The protected operating system prevents the hostile software from unauthorized modification and deletion of data. Sometimes, quite simple hardware mechanisms are also used for this purpose (i.e. a write-protect tab or switch). However, all of the protection mechanisms described so far do not deal with hostile software whose effect is upon data it is permitted to handle.

To prevent spoofing, the user should be able to identify the correct software. The best way of achieving this is by having a special output signal which is active only when an authorized program requests a password. It should be impossible for other programs to read the password from the keyboard at this time. When this is impossible, the software may include a mechanism which will enable the user to detect the spoofing. To achieve this, the identification software presents to the user a pre-defined phrase whenever the user enters the correct password. The pre-defined phrase acts as an 'inverse password', informing the user that he supplied his password to the correct software (and not to a spoof).

It is impossible to prevent worms by technical means, since they appear either due to an innocent, intentional act of the user (who does not realize what he is doing), or through weaknesses in the security system. Therefore, it is critical to warn the users, especially from running programs obtained through electronic mail: it is possible to prevent the worm from sending mail automatically to other users, by asking the user for a manual intervention, possibly entering a password, before sending any executable files. However, every prevention mechanisms can be bypassed by more refined methods.

The situation is more hopeful concerning viruses. A virus propagates by infecting programs. Therefore, it is possible to detect the virus by the effect on programs. If there is insufficient file protection, the virus may change files which should not have changed. This kind of viruses may be detected by computing a check sum of such files, and verifying the check sum periodically.

2.6 First Scenario: A 'Typical' Virus.

We first describe a possible scenario which may result from a typical micro-computer virus. This virus was written by a teen-ager, just for the satisfaction of causing destructive consequences and of writing a *smarter* virus (than previously known). The virus is not very original: it blends the properties of two actual widespread viruses. It is reasonable that viruses with similar 'blend' of properties are already being spread.

The virus attaches itself to IBM-PC programs. Whenever a program containing the virus is executed, the virus acts as follows:

1. Attempts to copy itself to other programs.
2. Changes randomly a few digits in a random data file or memory location.
3. Checks the date. If the date is Friday the 13, the virus erases the entire contents of the disk.

One possible scenario of the effect of this virus is as follows:

1 July, 1989: a new application, NEWAPP, is installed in several PCs in remote locations (test sites).

10 July, 1989: an employee in a test site accepts from a friend a computer game containing the virus.

21 July, 1989: the employee suspects a bug in NEWAPP (possibly due to the digit changes done by the virus). He sends his copy of the application to the central site (**Z**), for debugging. This copy contains the virus.

24 July, 1989: the application NEWAPP is debugged, and a small bug is found. An updated version is sent to all test sites. During this process, the IBM-PC used (in **Z**) for debugging NEWAPP, and the updated version of NEWAPP, becomes inflicted by the virus.

10 August, 1989: Problems are reported from the test sites; NEWAPP is returned for in-house testing.

22 August, 1989: A new version of the communication software is accepted at **Z** from the vendor. The communication software is tested on a virus-inflicted computer.

15 September, 1989: The new (infected) communication software is sent to all locations (which have IBM-PCs).

Until 12 October, 1989: Several data errors have slipped undetected into the main data-bases, during file transfer and emulation. As a result, several wrong payments are made.

13 October, 1989: Most of the data stored in IBM-PC computers is erased; to reuse the computers, the fixed disk should be re-initiated (a procedure which cannot be carried out by the user). On the screens of the computers appear the following 'warning': 'Sucker, you have been infected with a virus!! You should better check your files - I wouldn't trust the data there...'. Excited workers leak the story to the press, and many clients file charges against the company claiming they have been charged too much due to virus errors.

It will take several days until the computers are operating again, and unless proper security measures are taken, the virus may appear again. We cannot estimate the damage caused due to loss of credibility, but it is surely substantial.

2.6.1 Analysis of First Scenario

An extensive damage was caused by a simple virus, due to basic faults in the protective measures. There are no clear guidelines to the employees concerning the use of external software; no anti-virus software is used; there is no separation between computers used to service application problems on PCs and computers used to service systems programs; there is no competent and aware group of programmers which could detect the danger and prevent it.

2.7 Second Scenario: A 'Directed' Worm

The following scenario involves the intentional creation of a worm, with the motivation to blackmail the insurance company. We use here the term worm although the software is simply copied by the users, to differentiate this program from the trojan horse mechanism hidden inside it (which runs on the mainframe).

The entire operation is carried out by a single system programmer, of average talent and experience. Like the other scenarios, all details are as close as possible to these of the **ZURICH** Insurance Company.

The author of the worm is a respectable, well-earning programmer, in sudden need of a large amount of money, due to medical expenses. The author is slightly familiar with the **ZURICH** Insurance Company and its computer systems, since he has a friend working in one of the larger branches. In particular, the author is aware of the following facts:

- The company is capable of satisfying his monetary needs.
- IBM-PC computers are used by system programmers for file transfer and emulation.
- Backup is performed using standard system utilities, e.g. IEBCOPY, IEBUPDT etc.

Based on these facts, and on his knowledge of the backup utilities and the emulation software, the author is able to proceed. The following scenario results:

1 July, 1989: the author determines to attempt the blackmail. He copies the terminal emulation program from his friend (during a visit). Also, he learns what are the exact backup procedures (by looking at his friend computer screen, or from hooking on the friend's communication line, and reading the contents of unprotected libraries).

15 July, 1989: the author designs the operation, and begins writing the worm. The worm is made as a new computer game.

25 August, 1989: the author finishes programming the worm. He gives the worm-game to his friend.

5 September, 1989: a system programmer copies the worm-game (for fun).

17 September, 1989: the system programmer runs the worm, while being connected to the network. The worm detects that the computer is connected to the mainframe on a system programmer's account. Therefore, it replaces the standard backup/restore utilities with a trojan-horse utility, that enciphers the data before writing it on tape and deciphers it upon recovery to disk.

Trojan horse installed period: during this period, the trojan horse operates for backup/restore instead of the standard routines. There are minor differences such as different messages, bugs which cause some unsuccessful restore, and some error messages produced on every backup. However, those indicators are not detected.

13 October, 1989: The trojan horse contains also a time bomb set to this date. On this date, the trojan horse erases all files on disk, including the code of the trojan horse itself. Before this, it outputs a message requesting 1,000,000$ for the key required to decipher the backup files.

2.7.1 Analysis of the Second Scenario

The attack involved minimal knowledge about the operation of the computer system, plus basic knowledge of system programming. The only real obstacle for the attacker in this scenario is that he was not able to debug his trojan horse program - it must work the first time it is performed. Apart from this, the attacker needs only to obtain an attractive computer game (in which to install the worm), and to plan how to escape with the cash (which is easy, since the company is unlikely to risk additional trojan horses).

The weaknesses exploited were: no awareness of the possible meaning of deviations from normal operation of the backup software; keeping no history of modifications of system files; use of personal computers as terminal-emulators by system programmers, without proper safety procedures.

2.8 Third Scenario: A 'Classical' System Penetration

In the following scenario, a crime organization extracts information on possible targets from the data stored in the company. Intentionally, this scenario involves minimal technical sophistication - the computer know-how required does not even require academic training or years of experience. Furthermore, the operation is not very complex also in non-computer details, and hence it is conceivable that many crime organizations are capable of carrying it out.

The aim of the attackers is simply to obtain details about very expensive life and valuables insurance policies. Using these details, the attackers will be able to select burglary targets or kidnapping targets; alternately, the attackers may blackmail the insurance company (e.g. if not given a certain sum, we'll kill Mr. X and publish the fact we got the information from you and you didn't pay...; or: we will burn down a property insured at a huge sum).

The scenario is quite simple:

1 July, 1989: the organization selects the **ZURICH** Insurance Company. It carries a rudimentary investigation and finds out that the company does not use encryption on communication lines, and that maintenance is done by remote call.

7 July, 1989: the organization starts tapping the line between the company and the maintenance personnel, to learn the protocol. Also, they prepare to circumvent the dial-back mechanism by manipulating the exchange box in the street near the maintenance office. For this they need some qualified telephone technicians.

19 July, 1989: the programmer of the organization starts attempting to break into the computer through the maintenance port.

During this period, the moderatly skilled attackers left several traces of the penetration attempt - but they are not taken care of.

13 August, 1989: The programmer finally succeeds, and copies the entire disk contents of the computer.

25 August, 1989: The programmer finished processing the copied data, and delivers the lists to the crime organization. The operation is complete!

2.8.1 Analysis of the Third Scenario

The programmer performing this attack was quite incompetent. However, his task was very easy; therefore, after some time, he succeeded. There was no programming challenge in the entire operation. The only technical challenge here is the manipulation of the switchbox required to circumvent the dial-back unit. This is much easier then commonly thought. The dial-back unit gives only some protection under the assumption that the physical lines remain untouched.

This scenario was possible only through the possibility of remote maintenance, which gives complete control of the computer, protected only by dial-back. Note that a similar attack is possible even without this weakness, by using the facts that the communication lines are not encrypted, and hence by eavesdropping on these lines. However, this will be a less efficient attack, since a substantial effort will be required to connect to all lines and analyze the resulting data.

2.9 Conclusions

We have presented three scenarios perpetrated quite easily, involving huge potential damages, either direct ones or through long-range consequences. All of these scenarios are quite realistic. In fact, it is quite likely that the unintentional virus damage will happen (to some companies) in the near future, or is even occuring right now. It seems that the probability of an intentional damage is still small, though it does occur occasionally. However this will change gradually as computer expertise becomes more common, and the vast potential of data-processing crimes becomes more prevalent.

Those scenarios are probably only a pale shadow of the actual risks, since we based them only on configuration information of the computer center. In particular, we were unable to illustrate the risks involved in specific data transactions. For example, it is conceivable, that the company has some confidential information in storage which could be useful to other insurance agents (with a very high penalty to the company). Similarly, it is conceivable, that fraud attempts could be aided greatly by proper manipulations of the data stored (statistical measurements and transactions log). In both last scenarios, the attackers could have done any of these break-ins. We did not include such risks, since we do not know the details of the procedures, and the current use of automation in the company.

Let us summarize some of the weaknesses of the computer system, what are the subsequent risks, and how these several weaknesses could be avoided:

- There is no completely safe protection against viruses. Reasonably secure anti-viral protection software should be installed, at least in all micro-computers. Clear and appropriate procedures should be published and enforced, restricting operations which may cause spread of viruses. System programmers should be aware of the risk of viruses, and of the various forms in which they might appear. Great emphasis should be placed on the sensitivity of the personnel to any departure of the system operation from the normal or expected one. It may be the first sign of some malfeasance. Especially, they must pay attention to strange errors in the computer output; any such occurrence should be traced and checked until it is satisfactorily explained. Doing this on a routine basis (though each event is essentially a crisis, and anything but routine), is probably the best defense mechanism. Also, the use of micro-computers as terminals of the mainframe should be restricted.

- No encryption is used on communication lines; dial-up lines are protected by a dial-back unit. Dial-back unit provides only minimal protection (while reducing reliability and causing overhead at connection set-up). This is most critical for the remote diagnostics port; if such port must exist, an encryption device must be used there. In addition, dial-back protection should be combined, on all lines, by low-cost cryptographic solutions. Probably, it is even sufficient to use only a cryptographic software solution, which is also more reliable (compared to dial-back).
- In the document of 16.6.89, it is specified that there is no system backup site, only a backup agreement with IBM and data saved at an external site. It is unclear, if a problem occurs, if it will be possible to use the backup data with the software in IBM; in particular, where are the applications and the other non-IBM software kept? Obviously, all data and programs must be archived at one or two external locations, with at least an agreement with IBM (or others) that will ensure (through periodic checks) that the company will be able to run the programs in the external location.
- There is no defence ready against software risks. In particular, no tools nor personnel are available to trace penetrators. Therefore, even when it is possible to detect the penetration, it will probably escape detection until causing harm.
- New software should be tested prior to use, to detect viruses and trojan horses. It is desirable to inspect the source code. If this is not possible, the origin of the software should be well evaluated.
- Proper procedures should be issued and kept by all employees; for example, not to tell any details about the operation of the computer system; not to run any untested software in the central site; never let other people use the computer.

3 Insurability of Software

3.1 The Nature of Software

Software is structured set of programs designed to turn a universally programmable computer into a special purpose environment, tool or device. It serves to automatize certain features or functions, which, conceptually, could be performed by humans. Therefore the nature of software may vary widely: It may have the character of a *medium* (for example a word processing environment), of a *service* (in the case of databases), of a *device* (in the case of a tomograph), of a *scheduler* (in the case of an elevator) or a *tool* (in the case of program in an CNC). We shall subsume the three latter cases under the term *black box software.* In complex applications (the software installed in an Airbus, a power plant), all aspects may be combined together and it may be difficult to identify each part according to the above categories. When we want to guarantee the performance of a software package, it is important to keep this ambiguity of the character of software in mind.

Software which serves to write and edit text (letters, books, programs or solid modeling) is a medium. The only thing it guarantees to do is to perform the editing commands according to specifications and (hopefully) not to loose the data entered. Such software only creates a specific *format* of the data entered. Depending on the applications, format refers to *input format*, *storage format*, *display format* or all of these.

In complex applications, such as solid modeling or accounting, the format has to satisfy *rigid criteria*, required by other programs, which will process the data (storage format), or by external institutions, such as the tax revenue office or other legal authorities (display format). In these cases, the software should also guarantee that these formats correspond to the various requirements.

Editing software (word processors, input to CAD-systems, accounting) is highly interactive and serves foremost to facilitate input and format it. But many software packages create a genuine dialogue between the user and the program. They allow the user to consult the stored data. Examples of these kind are databases, expert systems and interactive simulations. It is worthwhile to look at software of this kind as *interactive books*, or better, handbooks and encyclopedias. The software then provides the user with new techniques of *cross referencing, deduction and abduction, and execution of instructions, recipes, algorithms and simulations.* The book has become dynamic, more like an interactive movie, it is an *animated encyclopedia.* What such software can guarantee in the best case, is the correctness of the basic data, the correctness of the cross referencing mechanisms and the correctness of the animation and simulation. The author of a handbook of medicine is liable for the relevant data (quantities of medication, listing of symptoms) up to a certain degree defined by the *state of art of his discipline*, but not for careless use of the handbook. It is for this reason that the analogy of software with animated books is relevant for the issue of software liability.

In both cases treated so far, the software had to guarantee the accurate functioning of specific operations. Saving of data has to be reliable, browsing in a database should not affect data, running a specific computation (simulation) should really compute (simulate) what it pretends to compute (simulate). But the danger of misusing this software, either due to lack of understanding of the user or due to human error, is far greater than the danger of damage to third parties created by the software itself. Software, however, which schedules processes or drives a device (numerical controllers, medical instruments, elevators, airplanes, or any other complex device) is only *reactive* and *not interactive* in the sense, that the user is not necessarily aware of the consequences of his input beyond what he expects the device to do from his point of view. In such cases the specific operations the software performs when running on a specific machine are the *essence* of the software, and therefore have to be not only reliable, but also free of unexpected side effects.

Many of the operations available in a software package with a dominantly medium character reflect *experience, style, habits* of an expert using that medium. Automated reasoning in databases and expert systems can still be checked by the common sense of the expert user. Much of the liability concerning the use of such software relies on the expert user. But in the case of computations and simulations which are hidden from the user (as is the case in a tomograph) the liability lies with the author of the underlying programs and algorithms.

To summarize, we find that software may be a *product* (most black box software) or a *service* (consultation systems), and sometimes the distinction is hard to maintain.

3.2 The Software Life Cycle

A software product is never generated spontaneously from the fingers of a programmer. As any other engineered product, its realization goes through the steps of *requirements analysis*, *specification*, *implementation*, *test*, *validation* and *certification.*

Requirements are written with the client. Their point is to establish *what* the software is supposed to do and, most importantly for the type of software we consider here, what kind of *user interface* should be provided. *Usage scenarios* are typically used to clarify such requirements. In some cases, a prototype of the sole user interface will be created to get an early feedback on the quality of the requirements.

A specification of the software is derived from the requirements. The specifications give a general structure of the software, describe its different modules and their associated functionalities. Specifications must guarantee the *completeness* of the modules, that is, that all the basic operations required to perform any kind of transformation or recovery of the data are defined.

Once specifications are complete, the software is implemented. It is coded in some programming language, to run on some computer, both of which can be specified in the requirements. Otherwise, they will be specified by the software company. At this stage, the quality of the whole product has to be assessed.

Testing is done to ensure that the implementation is correct with regard to the specifications. There is a wide range of testing techniques to choose from but, most important, we must underline that testing is never exhaustive, except maybe for the most trivial cases. A well tested software is dubbed verified.

After this, a validation of the software is made by assessing its compliance with the requirements. The quality of the documentation must also be controlled. This is best done by an independent third party.

The software will then be put in practical use on the user's site for a trial period. At the end of this phase, if passed successfully, it will be certified.

In practice, after some time of field use, the user wants some changes to be made to the system. They can be related to different causes, such as an required increase in functionality, some changes in company procedures that must be reflected in the software, even a change or an upgrade of the computer system.

Minor changes, such as changes in parameters, in functions or simple 'bug' fixes will necessitate only small revisions of the code, very seldom a redesign. On the other hand, major changes will require a complete iteration through the whole process, hence the notion of *life cycle*: the software is an *evolving entity.* Ideally, any kind of change should require a new certification. With the state of the art in industrial practice, it as indeed been noticed that even fixing 'bugs' may lead to the introduction of new problems in the software. Validation and certification may therefore be necessary even for minor changes in the software.

3.3 The State of Art in Software Evaluation

Traditional methods produce software from user requirements through specification, design, implementation and testing. Testing can never be exhaustive, because of the complexity of large software. It is related to modules of the whole product, and to some extend to their integration. Done loosely, it does not permit to establish any meaningful level of confidence on the resulting software.

Various methods have been proposed, with more under study, or used in practice to validate software. We describe now one such method which englobes the whole life cycle. It has the interesting property to give a *figure of merit* on the software product. We present here this method as an example of state of the art of software procurements technology.

The so-called *cleanroom software engineering* method is based around a mathematical theory of error statistics. A certification model has been established for software. In the testing phase, recorded failures are matched against that model, to establish an estimate of the reliability of the software. With this (proven) method, it is possible to decide by contract, and certify, how reliable the product should be.

The key is to use realistic usage scripts from users as the basis for testing. Random variations around those scripts form the body of test data. The product is tested as a whole unit, rather than component by component, to make sure the result figures from the tests are relevant to the whole software. Note that here the testing refers to a certain user profile, and may be meaningless for user behaviour deviating dramatically from that profile.

The underlying assumption for the validity of this method is to make sure that the software already has a solid reliability factor before testing is started. This is no longer debugging, but testing. To achieve this, other methods are used in the development process.

Over the years, a lot of formal tools have been produced to handle specific aspects of computer based processing. Formal grammars are used to describe input format, with automatically generated recognizers. Regular expressions are used extensively for pattern matching. Mathematical logic allows to reason about programming structures such as loops or selections and prove termination properties.

Wherever they apply, such techniques are used, to improve the quality and the verifiability of different parts of the system. Different teams review each other's work, comment on it and suggest corrections or improvements.

Data and programming structures, with formal and provable mathematical properties, are used throughout the specification phase. Only at the implementation level are these structures converted into some programming language's statements, while preserving their semantic meaning. This allows to gain confidence on the correctness of the specifications, without sacrificing implementability.

However, no mathematical technique supports the full specification of a complex software. Some parts of the system remain based on heuristics only. Needless to say, that these parts may contain the crucial bug.

Cleanroom software engineering is an incremental method. The final product is reached through several incremental steps of potentially variable size, established via functional decomposition of the problem.

Experience with this method shows that human fallibility in the design process ranks low for responsibility in remaining errors.

3.4 Software Certificates

With the ever-increasing reliance of companies on computer based products, the potential impact of errors or problem inherent in the use of such systems has come in critical focus.

In this perspective, the *Schweizerischen Treuhand- und Revisionskammer* has issued recently a document on the certification of computer-based accounting (book-keeping) systems.

We analyze this document in some detail to exemplify our understanding of the relevant aspects for such certification. Such a certification document could serve as the minimum requirements for future software liability insurance contracts. We want to establish here that, unfortunately, the document in question does *not highlight the real issues, nor the relevant software engineering aspects.*

Four different domains of relevance for certification are identified:

- rules of the trade,
- internal control,
- documentation,
- release, testing and maintenance.

The guidelines are embodied in a *checklist*, a set of questions which must all receive a positive answer for the system to be certified.

Each question is relevant to one or more domains, not clearly separating the different issues. From the answers, a document can be issued, in the form of a certificate. Several examples of such certificates are given, and there is no standard format.

The questions are meant to guarantee that the software really does what it is supposed to do, that is book-keeping operations, in accordance to the recognized practice of the trade. No 'illegal' manipulation of the data should be allowed. The software should come with a documentation that clearly describe not only its use, but also its structure. The data should also be protected from illegitimate access and manipulation. An empirical evaluation of the ease of use of the software and some testing on numeric values must also be done.

From our previous description of software engineering practice, we see that we have a mix of unrelated issues, most of which are already handled through different phases of the life cycle. Operations, protection and user interface are parts of the requirements of any transactional system. The guidelines handle at the product level matters that should be specified, understood and agreed on well before coding gets underway.

Documentation is also treated in the life cycle. It is standard practice to provide the system user's manual as part of the requirements. Its validation through fast prototyping of that part of the system is also considered in some cases.

No difference is made in the document between requirements on the system, and on the machine it will run on. Any application runs in symbiosis with an operating system, and the system's resources. The application uses facilities provided by the operating system, and therefore has to make assumptions about their behaviour. Such assumptions must be revised and checked when there is an update of the system.

Protection can be done at several levels, and it is not clear from the guidelines where they would be best met. Access protection can be built-into the application, or simply depend on the facilities provided by the supporting machine. One should also specify the physical context in which the system would be used, in light of recent, well popularized, attacks on systems performed through computer networks or phone lines.

To summarize, let us say that such guidelines are incomplete, confusing and generally inadequate. Certification has to be done in perspective of software engineering practice, of system dependency, and overall security. Certification starts at the requirements level, where a clear statement of the system's functions must be made. What is presented in this document is more a set of standard requirements for book-keeping applications, together with some security and documentation requirements. Rather than asking the question, 'does this software perform book keeping operations correctly', it would be more simple to certify that 'does this book-keeping software perform according to its requirements?'. Good practice leads to few errors, and must therefore be explicitly taken into account in the certification process.

3.5 Proficiency of Software Users

In the previous sections we have mentioned the importance of the user and the user interface. It is critical that the proficiency of the users be taken into account when one evaluates the risks associated with the operation of any software. Let us take the example of a data removal, or *delete* operation. In commercial systems, one finds several ways of accommodating such an operation: we have the delete without acknowledgement, the delete with user acknowledgement, or the delete with undo facility. It makes no doubt that human failures will have a lower impact in the latter case than in the former.

Operators must be trained to operate any form of application. However, there is a built-in assumption on the professionalism and reliability of the user in the system's interface. Just the way there are different licenses for drivers, from cars to trucks or even racing, certification should make a clear assessment made of the proficiency required of any user, for the different class of operations the system can execute. A privilege system could be used to enforce such policies.

3.6 The Draft Insurance Policies of ZURICH

The draft insurance policies offered by ZURICH as a supplement to existing professional liability insurance contracts for CPA's seem to us very *problematic*. They do not pose a risk for the insurance company, as they cover only claims occuring *one year after final delivery* of the software in question. This almost surely will exclude all possible claims, because revised releases of the software are considered changing the delivery date. It is a matter of time, until clients understand this feature, but in the future this will prevent them from buying such a supplement.

However, the proposed drafts do address some of the features we mention before, though rather vaguely. As we strongly emphasize, insurance has to take into account the software life cycle, and should *vigorously reject* insuring 'amateur software' distributed among professional colleagues. It should be recalled here, that non-professional software is particularly suspicious as a potential carrier of viruses, worms and other malevolent features.

We expected that existing software certification procedures, such as the one mentioned in the previous section (Schweizerische Treuhand- und Revisionskammer), could serve as guidelines for future software insurability criteria. However, the critique we formulated in the previous section shows that this is not the case under the current version of the Software Certificate. On the contrary, we have to warn all parties involved about the misleading character of the Software Certificate of the Schweizerische Treuhand- und Revisionskammer. We still suggest, that a thoroughly revised version of this Software Certificate could and should be a sine qua non for insurability of software.

4 The Future of Software Risk Engineering at ZURICH Insurance

4.1 Professional and Product Liability

Software liability insurance and professional liability insurance for software engineers is only possible, when the scope of the insurance contract is well defined. Traditionally, liability insurance contracts are based on an implicit *established standard of care* and on *prevailing laws and principles of product liability.* The application of the concept of standard of care is influenced by the level of professional authority of the producer. Contrary to traditional engineering situations, the following problems arise in the case of software engineering:

- The standards of care and principles of product liability are still in the *formative stage.*
- Standards and principles relate in an *inseparable* way to the quality assurance of the software in question, to the quality of the maintenance of the software as well as to its potential users and concern all levels of the software life cycle and its environment.
- In software engineering the production process cannot be effectively separated from the properties of the final product, as the production tools consist again of software which needs certification.
- Frequently, the software engineering team does not act in full competence of the problem it solves, as it may not understand the application of the software in its full scope. It should be supplemented by *application specialists* capable of providing the team with proper requirements.
- Therefore, the software engineering team *cannot* be perceived as acting on the basis of *exclusive professional authority.*

4.2 Software Risk Engineering

Based on our considerations on professional and product liability and the relevant aspects of the nature and life cycle of software, which we described in the previous chapter, we recommend the creation of a Software Risk Engineering Group (SREG) within **ZURICH** Insurance Company. The activities of the SREG are described as follows:

- Software risk engineering encompasses all aspects of software engineering, and the SREG of an insurance company should be well aware of the developments in this field.
- If software liability insurance contracts are sold for specific applications (auditing, medical equipment, etc), collaboration of the SREG with the professionals of the application domain is mandatory.
- Insuring software *after it has been developed and released*, seems rather dangerous and inappropriate, as the checking of quality control criteria may become almost impossible.
- The more the insurer can *intervene during the development phase of the software* to be insured, the more precisely can he formulate the policy. Intervention can be in form of direct monitoring (auditing) of the software development by the SREG, or by distributing general, but precise and binding guidelines to be followed by the producer of the insured software.

- It is to be expected, that some software producers might fiercely object any involvement of external bodies in their production process. In such occasions, the SREG should serve as a mediator to help overcoming these objections in the interest of all parties involved.

The main activity of the SREG therefore consists in establishing such auditing criteria and guidelines. Once such guidelines are established, checking their enforcement can by done by the SREG itself, or by a trusted third party. To establish expertise for the SREG, there are three complementary possible avenues to consider: in house risk engineering, external research and collaboration with professional organizations.

4.3 In House Risk Engineering

As we stated, the insurer should be involved in the development phase of the software, either directly or by providing binding guidelines. To make this possible, the insurer should be able to provide the buyer of an insurance policy with expert advice and auditing services. This means, that the insurer disposes of a qualified team of software quality insurance experts. These people form the core of the SREG, which directs others, more closely involved in the selling of insurance policies and checking the claims, either as direct employees or as independent brokers, agents and consultants.

The core of the SREG requires a critical size to be effective. It should have enough time to absorb also the rapidly evolving new technologies and changing standards of software engineering practice. The SREG should not only be aware of the software engineering issues, but also of the particular issues of the application domain for which the software is used (medicine, civil engineering, accounting, etc). The exact composition of such a team depends very much on the short and long range goals that the insurance company decides upon.

4.4 External Research

In absorbing the state of the art, the core of the SREG can be aided by commissioning survey works according to their specifications from universities, research institutes (MCC, IBM, Bell Labs, etc), research consultants (SRI, Batelle, small private firms). This concerns mostly gathering published material. It may be extended to subsidizing directed research and commissionning research and evaluations. Note that gathering of material can be delegated, whereas *digestion* of the material remains in the responsibility of the SREG.

In addition the SREG should be in close contact with various government agencies (NASA, DARPA, NBS= EMPA in Switzerland, etc), and professional organizations, which are involved in setting standards.

4.5 Collaboration with Professional Organizations

Professional organizations (CPA, Medical doctors, etc) have to be involved to coordinate the evolution of established standards of care with their perceived needs as well as to avoid that the professional organizations will attempt to certify software without thorough understanding of the software engineering aspects of the problems. Further relevant experience can be obtained by collaborating with telecommunication companies (PTT), politicians (law-makers), consumer organizations and their legal representatives as well as the media (public awareness).

5 Proposal for Future Collaboration

It is conceivable to continue this work in the form of *ad hoc consulting*: I could evaluate new draft insurance contracts, whenever they are produced at clients' request; I could get involved in the future drafting of generic insurance policies for very specific applications and I could expand this report further, such that it can serve is a guideline for internal use at **ZURICH**.

A continuation of the study of in-house risks due to hostile software would require too much work on location, to be feasible in the context of external consulting.

Ad hoc consulting, however, is not *comprehensive enough* because it cannot relate thoroughly to the issues described in the previous chapter. I would like to stress that for future software insurance activities, **ZURICH** should set up its own in-house team, which can address these questions competently.

My short term proposal, therefore, consists in helping to establish the core of the Software Risk Engineering Group (SREG) along the following lines:

- Defining a group of between 5 and 10 qualified people as well as their work for the first two years.
- Recruiting and evaluating the candidates for the SREG and initiating their activities.
- Identifying one or two very specific test domain of software insurance on which the SREG should concentrate first. I would propose one of them to be in the domain of CPA and one in the domain of software hidden in specific devices (black box application software) such as medical instruments or robots.

In each of these phases I propose to act as an external consultant, either in form of my own company or in collaboration with the Technion - Israel Institute of Technology.

During these two years, with the experience gained, more precise long term developments can be defined. The field of software insurance is a growing concern and will have to yield to public demand. I am aware of similar undertakings in the US and England, and with the growing awareness of consumers of software dependent services, I am sure that some form of software insurance will be a needed product and that a potentially important market will evolve.

6 References

1. S.R Wolk and W.J.L Luddy, Jr.; Legal aspects of computer use, Prentice-Hall 1986.
   Useful reference on legal practice, including case studies from American courts.

2. Fachmitteilungen der Schweizerischen Treuhand- und Revisionskammer, 1988 Nr.9: Software-Zertifizierung.
   This is the document discussed in section 3.4.

3. T.P. Keenan; Emerging vulnerabilities in office automation, Computers & Security, 8 (1989) pp. 223-227.
   H. B. DeMaio; Viruses - A management issue, Computers & Security, 8 (1989) pp. 381-388.
   Two background articles for chapter 2.

4. R.A. DeMillo, W.M. McCracken, R.J. Martin and J.F. Passafiume; Software testing and evaluation, The Benjamin/Cummings Publishing, Menlo Park, 1987
   State of the art report on software testing, containing samples of the American DARPA guidelines for software production together with critical evaluation.

5. R. Fairley; Software Engineering Concepts, McGraw-Hill, 1985.
   A. Macro and J. Buxton; The Craft of Software Engineering, Addison-Wesley 1987.
   State of the art textbooks of the American and British schools of software engineering. Considered by most the best textbooks in the field. H.D. Mills, M. Dyer and R.C. Linger; Cleanroom software engineering, IEEE Software, September 1987, pp. 19-24.
   Expository article on the cleanroom methodology discussed in chapter 3. Further references may be found there.

6. B. Berliner, Limits of insurability of risks, Prentice-Hall 1982.
   Our background reading concerning insurability questions.