The Realities of Large Software Portfolios

Chris Verhoef

Programming Research Group, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

x@wins.uva.nl

Abstract

Large software portfolios (and by large portfolios we mean millions of lines of code) pose a unique category of non-intuitive problems for organizations dealing with massive software modifications. These problems lay dormant, gathering disruptive capability, until projects such as the Euro conversion, or the software integration of merging companies bring them to light, often with devasting consequences. This paper will identify the problems, the difficulties they present, and touch upon some of the remedies.

Categories and Subject Description: D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring.
Additional Key Words and Phrases: Reengineering, System renovation, Software renovation factories, COBOL.

1 Introduction

Owning a large software portfolio does not merely mean having a lot of software, just the same way that being deaf does not merely mean that you cannot hear. Deaf people need a special alarm clock, pay on the average more for a car insurance, they need a special text-telephone, they use sign-language, and so on. For large software portfolios similar special attention is needed. This paper illustrated some of the major issues that come into play when modification of large software portfolios is necessary. If the size of a portfolio is in excess of 2 million lines of code, we speak of a large software portfolio. We note however, that many large software portfolios easily top hundreds of millions of physical lines of source code.

Many organizations made major investments in automating their business processes in the past. They did this to stay competitive, or to deliver better service to the public. Such investments often reflect the long-term strategy of these organizations. Those decisions gradually turned such enterprises in software intensive organizations. Enterprises with a long tradition of implementing their business often do not realize that their software portfolios has grown very large and that special care and attention is necessary to enable further enhancements.

There is an important dynamic that masks the size issue. Of course, the software is not static: it is constantly modified. Since people are natural learners, their competence for making modifications to a particular system tends to grow over time. This increase in learning masks the size increase and deterioration of the code itself. Therefore, management does not realize that the quality attribute maintainability itself needs to be maintained and the entire software portfolio grows and deteriorates in an uncontrolled manner. To make the issue a bit more concrete let us give an example. One measure of maintainability is the so-called ripple effect, being the number of separate code areas that have to be changed to effectuate a single modification to the code. One hardware manufacturer studied this ripple effect to modifications to its operating system. They found that each modification led to about 300 other modifications. In the words of Gerald Weinberg: this operating system was equivalent to a nuclear reactor, one that was on the verge of turning into a nuclear bomb [15, p. 237, 243]. The metaphor of a nuclear bomb is not only true for this particular operating system, but also applies to some large software portfolios. Software grows and deteriorates over time. This phenomenon has been empirically measured by Belady and Lehman [1] and was formulated in their famous Laws of Software Evolution. Some managers do not know that aging software becomes brittle. This can be due to the competence/masking dynamic mentioned earlier. Based on earlier modification experiences, their demands for rather sophisticated enhancements increases whereas the potential to do so decreases over time. Pushing the maintenance crew harder will ultimately result in a productivity collapse and high turnover rates. As soon as the competent crew leaves the organization, the maintenance iceberg becomes visible. Initial reactions, like buying functionality, componentization of the portfolio, outsourcing, converting everything to Java, and so on, are understandable when faced with large and often acute problems. However, such solutions are as fictional as curing deaf people by decree.

In this paper we sketch some of the realities that one faces when dealing with a large software portfolio. Especially when portfolio-wide modifications are necessary, a lot of problems have to be taken care of *before* we can start with the actual task. We will indicate a number of such tasks, some of them are entire projects in their own virtue. We mention the Year 2000 problem, the Euro conversion or a merge between two companies and their software as the standard examples of portfolio-wide modifications. There are many more modifications that affect entire software portfolios, such as new compilers, or new coding standards. Apart from that, for more restricted change efforts similar preparatory efforts are necessary, although the scale may be more modest.

Normally, the story of a large software portfolio starts with a very successful initial system bespoke system. If software is a success we often see what could be called *software mitosis*, that is, multiple versions of the software come into existence, each with their own special features. Normally, there is no initial *product-line* thinking, so before anyone notices various versions are in use, each digressed so far from the base idea that all of a sudden multiple versions have to be maintained and enhanced whereas only for one system was planned (if there was a planning at all). Of course, all the versions have problems, and need enhancements. This causes a multitude of customer requests. Those requests in turn can cause a dramatic productivity disruption of the development and maintenance teams. Then, all of a sudden, corporate management realizes that there is some sort of a problem. Most of the times they do not realize that it is a case of software mitosis. Either way, an expensive process to bring the situation back to maintainable proportions has to commence, if at all possible. This is comparable to working in a noisy environment without ear protection. Each day without protection contributes to the hearing problems until after some years an irreversible situation is reached. Then everything you do is expensive but will not bring back the original situation.

Of course, with good management practice, such situations might be avoided. One possible solution to prevent multiple versions could be to recover a core asset architecture from which the various versions can be derived. However, the reality is different. The reason for our experience with large software portfolios is that we develop technology to modify large amounts of software in an automated fashion. One could call such software aids *Software Renovation Factories*. It is our experience that the majority of modifications to large software portfolios is so company and/or system specific that no-one can expect to buy an off-the-shelf tool for the specific task. Therefore, sophisticated software aiding in rapid development of company specific renovation tools is relevant as a research issue. Moreover, software renovation factories become more and more a necessary part of IT industry.

It is very difficult to *fast forward* large amounts of business critical software 30 years in time, for instance, by migrating a code base from COBOL to (native) Java. Such modernizations can only be done in an evolutionary manner, with significant human interaction. In this paper we illustrate such issues by giving insight in the current state of large portfolios, their problems, and their brittleness.

the COBOL factor Today we are in a situation that about 70% of all the business critical applications is running on mainframes. Most of those mainframes are IBM AS/400 machines. According to the OVUM-Group about 60% of the mainframe applications in Europe is written in COBOL, 11% is written in PL/I, 9% uses Assembler, and the rest is written in some fourth generation language (Natural, ADS-Online, Ideal, CSP, Focus, Telon, and so on) [8]. Globally, it is estimated that between 30 and 35% of the software is written in COBOL. It is estimated that the installed base of software is about 7 billion function points. So 30–35% of COBOL amounts then to between 225 and 260 billion logical source lines of COBOL code. These figures clearly show that COBOL is a very important factor in the IT industry. Therefore, one of Edsger Dijkstra's advices not to teach COBOL [3]:

> The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

seems not to be based on global benchmarking studies.

Who is using COBOL? Typically, the financial world has a long tradition in automating their corporate assets culminating in software portfolios that top hundreds of millions of physical lines of source code. In this paper we describe our experience with large software portfolios that we encountered over the years. Our experience is most prominent for portfolios in large financial corporations, in the telecommunications industry, and defense applications. We disguised all people and corporations from whom we have obtained confidential information.

Organization The rest of this paper is organized as follows. In Section 2 we make the connection with large software portfolios and a typical enterprise-wide modification effort: the Year 2000 problem. In Section 3 we discuss some of the major problems one has to overcome when large-scale modifications are commencing on large software portfolios. Finally, we in Section 4 we make some concluding remarks and we give directions for further reading on technical infrastructures enabling mass-changes to software portfolios.

2 Large Software Portfolios and the Year 2000 Problem

We briefly discuss the Year 2000 problem being our running example and its consequences for large software portfolios. Before we loose all the readers who have had enough Y2K talk, we stress that the issues we will address are not what you might think they are. They are nonintuitive problems that one encounters when managing modifications to large software portfolios. We briefly explain what the typical issues are that need to be taken care of when commencing with large-scale modification efforts, such as the Year 2000 conversion effort.

Solving the Year 2000 Problem, abbreviated Y2K[1] is an excellent example of an evolutionary modification that needs implementation on a large scale. It is our experience that with this example the most prominent issues are addressed in the realm of changing large software portfolios. Therefore, we use it as a running example throughout this paper. Some people think that the Y2K problem is rather a trivial problem, since it only addresses a simple conversion from two digits to four. First of all, this is not always possible. For, a data type change typically will ripple through the entire system. Such a solution will not only affect the program code but also demand migration of all the data bases.

[1] In our opinion it is a very unfortunate idea to shorten *Year 2000* to *Y2K*. It was this kind of thinking that caused the problem in the first place.

Secondly, this is only a tiny part of the problem as this paper will illustrate. Another often heard proposal for solving the Y2K problem is to abandon all the old software and reprogram everything. It is not technically feasible to develop major new applications in a few years, and for many business critical applications no commercial replacement exists. As Capers Jones put it in 1998 [6, p. 161]:

> [..] the maximal size of applications that might be constructed prior to the end of the century is roughly 2000 function points [..] Many of the key corporate systems top 10.000 function points and could not be redeveloped in 48 calendar months.

So, due to such (and more) constraints it is not possible for the majority of enterprises to fall back on the simple scenarios that are often envisaged by people who have no idea what it means to manage large software portfolios.

We experienced that in many cases when a significant change effort is commencing, myriads of other problems and issues first have to be resolved. For instance, an inventory of the portfolio, or adapting the used languages and operating systems, or dealing with very specific Y2K issues, or very special testing efforts. Although some of these items may look unproblematic initially, we will elaborate on them in more detail in the next section. We focus on the nonintuitive issues that come into play in the context of a large software portfolio. We hope that the subsequent discussion will increase insight in the realities of large software portfolios.

3 Large Software Portfolios and Large-scale Modifications

In this section we elaborate on a number of problems that have to be addressed before we can start to make modifications on an enterprise-wide scale. Imagine for the time being that we are trying to start with a modification effort like a Y2K conversion for an entire software portfolio of a typical data processing organization. This could be a large financial institution, a government organization, a defense organization, or a large telecommunications organization.

3.1 Inventory of the Portfolio

In order to change code, we need to know in which files. This seems obvious. Until you obtain 10 million lines of source code, in an unknown dialect of COBOL, containing files in Assembly language, missing include files (copy books in COBOL terminology), unknown compiler options, and so on. We discuss some striking examples to give an idea on what to expect—and what not—when a large software portfolio needs inventory. We discuss the production environment versus the development environment, version management, dormant applications, missing sources, how to rebuild the objects after modification, and some sociological issues. The entire inventory problem alone can easily take a year before the largest problems have been solved.

production versus development Most of the times it is not at all clear what an entire software portfolio comprises. Sometimes the separation between the development environment and production environment is not clear. In one case a production system called many load modules residing in personal directories of various developers. As soon as a developer changed a load module, modified some permissions, or the account was removed due to turnover, parts of the software portfolio started behaving erroneously. Not until this problem was solved with procedures for checking in systems in the production environment, it was not possible to start with structural modification efforts on an enterprise-wide scale.

version management Usually, there is no strict version management (except in telecommunications industry). If we are dealing with organizations whose software is running on many sites, for instance there is one core version with specializations for various countries, then usually the problem is even worse. We had a case with about 100 versions of the software that was originally based on a single core system. Software mitosis is endemic when version management lacks. Also here multiple versions digressed from the base system in such a way that local uncontrollable versions emerged. A software inventory at the corporate level was very difficult. Even if those problems are solved, it might be that some local versions may have new owners due to local take overs of maintenance organizations. A local take over caused one company to loose access to the source code of an instance of their product. Even if there is solid version management, there can be severe problems due to software mitosis. How to deal with release management of a large system that asks for many changes with different timing, in an increasing number of instances that are geographically dispersed? Solving such pervasive problems is a serious project of its own.

does it ever run? Then we have the problem of dormant applications. Those are applications, maybe even maintained, but never run. A large-scale study of 8 IBM data centers found that about 50% of the applications had not been run for several years, and probably would never run again. It is a problem in itself to trace and to exclude such applications from the software portfolio. For, we should commence with large-scale modification efforts only when dormant applications are filtered out. The cause of dormant applications often is that jobs that are normally run on a regular basis, are for some reason removed from the queue. Then those systems are scheduled to run on request only. In most cases nobody dares to remove on-request-systems, resulting in half of the jobs that are only run on request in many organizations. When the original developers leave, it is hard to figure out what to do with such systems. Sometimes a software tool is installed that measures how often the software is actually executed. If systems are not executed for say 3 years, they are temporarily removed from the on request queue to see what happens. That does not sound very reassuring, especially in solving the Y2K problem if the deadline is less than three years. In one case an expensive Y2K change effort was carried out on a dormant application, since it could not be determined whether the software was still in use. In other cases, analysis of the execution frequency of applications revealed that some software ran that should not. In one case someone was still selling a financial product, and let the software run, that was no longer in the portfolio of the company (due to the fact that it was not prosperous). The lessons learned from this project were to review on a regular base the execution frequency of the software portfolio, to prevent selling financial services that are no longer profitable.

missing/mystery sources During the inventory, it is usually not possible to match all object code with the accompanying source code. One reason is that version management has not been optimal so that we end up with multiple versions of source code for one object. We call that mystery sources. It is also possible that there is no source code at all. To give an indication, a world-wide average is that about 5% of the object code lacks its source code. This figure illustrates the severity of problems that one faces when an entire portfolio needs to be modified. Also this problem is large and costly. Source recovery costs a significant amount of money: not only the recovery process itself, but also the recovery of the identifier names, that have to be picked from the rest of the system. To give an idea, when 300.000 missing lines of COBOL code have to be rewritten, it costs at least 43 staff years, and about 6.7 million US$. If tools are used to recover the source code, it costs about 1.5 million US$, and about 1 staff year [4]. Things start becoming even worse, when the sources were written in a nonstandard language, such as a fourth generation language (4GL) that generates COBOL. Then at most the generated COBOL can be recovered, and the original sources in the 4GL are lost forever. In one case 50% of the 4GL sources could not be matched with running objects. So, identification of mystery sources is not always possible. An economic argument not to build tool support, is that some 4GLs are so nonstandard that the return on investment often is negative (but see [11] for an economic method to generate software renovation factories from the source code of compilers).

build process recovery Even if we have all the sources and the objects connected, the problem how these objects have been build from the sources remains. Some objects have been build over the years taking many steps. We have seen examples of object code of which the first load modules have been compiled about 30 years ago while other parts of the executable have been compiled and linked recently. Some versions of the compilers are no longer present at the site. Therefore, it is also not trivial to recover the build process for all the running software. As we can only modify source code when we can rebuild it, this problem needs to be solved as well.

rebuilding the objects There are many preprocessors for COBOL. There are many different versions and patch levels for COBOL compilers, each with many compilation options. There are also postprocessors optimizing the object code. These possibilities imply that there are hundreds of possibilities how object code could have been created. Often the exact history of building the objects is lost, and in some cases we have seen that the build-history can span three decades. In addition, there are home-grown preprocessors, compilers with special company specific features, and what have you. Even if we recovered the build process (thus we know how to build the objects from the source code), then it may be the case that it is not possible anymore to build it or enhancements of it. For, personal preprocessors of developers can be lost, versions of compilers and preprocessors that built part of the object code many years ago are no longer available, optimizers that assumed certain output of certain compilers do not deliver the desired results anymore, and so on. Rebuilding the objects from a large software portfolio is a difficult problem in itself. It is maybe not a coincidence that the new IBM operating system still supports executables that ran under an older operating system. In one case it took 3.5 year and about a 100 man year to move executables from the old operating system to the new one

bad news and the messenger Who is going to tell corporate management that some part of the source code is missing from a business critical system? Programmers who find out about these things, often prefer to leave the company rather than be blamed for bringing the bad news. Large software portfolio owners should be aware of this issue and pro-actively search for possibly missing source code problems (without blaming persons).

software attack! Large software portfolio owners would probably not hesitate to fire employees who took a few hundred dollars from the company. They should also protect their information systems from people attacking them since those systems are often their most valuable assets. This is not always practiced, as the following case shows. A disgruntled programmer wanted to blackmail his former employer by throwing away all the source code of a complicated logistics system. There was no proper software configuration management and there were no proper back-ups. His personal back-up appeared to be outdated over a year. The company is still struggling with the source code recovery process (of course a 4GL), and thereafter, the Y2K problem has to be addressed.

3.2 Tools, Languages, and Operating Systems

Another category of issues is the use of language oriented tools, such as application generators, the use of certain languages and their consequences, and the use of a particular operating system. For large software portfolio owners a number of issues come into play that may not be immediately clear to uninitiated people. Some issues are worth mentioning in order to elucidate the realities of having a lot of software.

application generators Sometimes certain parts of a large software portfolio were developed using an application generator. Most of the times a 4GL is used. For instance, Natural, ADS-Online, Ideal, CSP, Focus, Telon, and so on are often used in Europe. In particular, mainframe based user-interfaces are good candidates for generation. During development, 4GLs are sometimes an advantage: for, the application generator is geared towards the application domain and proper use will increase productivity. But beware: an illustration of a 4GL disaster where Ideal was involved is reported on in [7].

Usually, the source code of a 4GL is nonstandard. Therefore, its use creates a dependency with a specific vendor. What happens when a vendor stops supporting the application generator, when their products become suddenly too expensive, or when they go out of business? As soon as one of those problems occurs, the customers have the problem that part of the portfolio cannot be supported by standard tools, unless the vendor supplies them. The latter is usually not the case. When a particular vendor stops supporting the software that your portfolio depends on, it may encumber future modification efforts.

We mention a typical problem with 4GLs that we encountered at one large software portfolio owner. A typical aspect of 4GL tools is that they are often updated to improve rapid development within the application domain. When version management is not optimal, it is sometimes hard to connect sources with executables. That is not too much of a problem, if you use source recovery tools indicating which compiler options have been used during compilation. Then the mystery sources can all be compiled and bit by bit compared. However, since the 4GL tools are updated so often, the generated COBOL code will not produce the same binary code if a mystery source is recompiled. This enterprise had the good practice to remove the generated code to prevent software engineers from modifying the generated COBOL code, so it was also not possible to use the intermediate form for comparisons. Also the vendor went bankrupt and there was no possibility to recover the many versions of the application generator. This resulted in the situation that it was not possible to connect 50% of the sources to the executables. Try solving the Y2K problem with such preconditions!

Suppose that life is treating you better than the abovementioned case. Then still there is potentially a large problem in case the entire software portfolio needs modification. For instance, for solving the Y2K problem it has been measured that for a small 50 languages there is Y2K search support and for about 10 languages there are automated Y2K repair engines. We recall that the usage distribution of these languages is not in accord with the availability of tools. In other words, using an application generator implies taking risk in the long term. These nonstandard parts of the software portfolio form a risk for any large-scale change effort. Moreover, getting rid of a (nonstandard) language is an intricate process of its own. And it is not so easy as some of us may think [13].

operating systems A typical far reaching decision is the choice for a particular operating system. Back in the seventies it was in vogue to use a proprietary operating system. This implies that the operating system, the compilers, and occasionally also proprietary processors had to be maintained and updated. In the telecommunications area this situation is still the norm. The rapid developments in hardware technology make clear that the decisions of the past now restrain further development so that large-scale efforts to migrate from proprietary solutions both in hard- and software to standard processors, operating systems and compilers are becoming more and more important. Also in the information systems world operating system issues play an important role. For instance, the Wang operating system and the Wang COBOL compiler are no longer supported. This means that platform migration and dialect migrations are becoming unavoidable. Even when a vendor is not going bankrupt, operating system migrations cannot always be avoided. Companies who use an IBM operating system for mainframes faced operating system migrations, since the old MVS operating system was not Y2K compliant. Moreover, the OS/VS COBOL compiler product may not be Y2K compliant. Since it is not supported for a long time, the compliancy is solved by acquiring the new version.

assembly language In all large software portfolios that we have encountered, always parts of them have been optimized using assembly language. In Germany the amount of assembly seems to be relatively high. Some experts say that about 50% of the data processing software is written in assembly. The world-wide estimate is that 10% of all the software is written in various assembly languages. This amounts to 140–220 billion logical source lines of code. In accordance with Fishers Fundamental Theorem (the better adapted a system is to a particular environment, the less adaptable it is to new environments), changes like a new operating system induce modifications to the used assembly. Sometimes assembly routines can be converted to a higher level language. In one enterprise this was done so often that the phrase *coboling the code* was used for turning Assembly/370 into COBOL. Also, old assembly is sometimes translated to a newer assembly language in order to take advantage of the new features that the upgraded operating system enables.

language upgrading An often heard language upgrade is to migrate to the latest hype language. Vendors explaining the benefits of new languages, and their productivity increments, usually serve as a silver bullet. Some people tend to believe these hypes. As a consequence, we often receive requests whether it is possible to migrate a legacy system written in COBOL, using JCL and CICS, into a web-enabled Java based multi-tier application. It takes an enormous investment to make such dramatic modifications to a portfolio, and its is out of the question that there is a simple solution that will do the "trick". We refer to [13] for a sobering discussion on the realities of automated language conversions.

Instead of converting an application we sometimes see that enterprises choose to redesign their software portfolio. This approach has a high risk of failure. In 1995 it was estimated by Standish Group that in the US alone 81 billion

US$ annually is spend on cancelled software projects [5]. By way of anecdotal evidence we mention a case where our colleague Harry Sneed is reengineering a 2.3 MLOC C++ stockbroking system *during* development [12]. Sneed estimated that it probably takes another year before the project will be cancelled.

As one consultancy company puts it: "In the old days, language migration was simple: install the new compiler and use it. Today, things are more complex. Each new version of a compiler brings with it new requirements and restrictions. Implementing the supporting environment is even more difficult." Typical language upgrades are Assembly upgrades, when migrating from one operating system to a newer version. Also upgrades from COBOL 74 dialects to COBOL 85 dialects are popular these days, but far from easy [13]. Moreover, such upgrades are unavoidable. When the COBOL 85 ANSI standard was a fact, some vendors announced that they would support the old dialect only the next ten years, so that everyone would have ten years to migrate their software to the new dialect. Large software portfolio owners often cannot afford themselves to fail, therefore, their secret motto is *be second.* Most COBOL 85 compiler defects have been repaired by now, so many sites are at the time of writing this document commencing with migrations to COBOL 85.

Projects merely focused on survival rather than new functionality are hard to sell to corporate management, like all preventive measures are difficult to comprehend for humans [9]. Therefore, in practice, such projects commence when there is no other way out. In some cases this means that they become prerequisites for other large-scale change efforts, like operating system migrations, or a Y2K conversion. As we indicated before, the new operating system for IBM mainframes also supports the older 24-bit address space so that load modules can be transposed as is. Only when enhancement of those systems is necessary, the migration to COBOL 85 makes sense. Therefore, COBOL 74 to COBOL 85 migrations will be among us for a long time.

language downgrading Normally language migrations are always from old languages to newer ones. At least that is the intuition that we often have experienced among many of our fellow renovation colleagues. However, when things grow large, nonintuitive projects are sometimes necessary. One of those projects was a language downgrading project, that is, migrating software to an *older* version of the language than currently used. Let us explain the finer details of this issue, since the amount of software and its complex interaction clearly shows one of the realities of large software portfolios.

Some parts of a large portfolio are very young. So the use of application generators to assist in the development process is then common. Normally such tools generate some 3GL code, like COBOL 85. For instance, KEY owned by Sterling Software produces a COBOL 85 dialect. For some business reason, KEY is not generating COBOL 74. In one case the bulk of a software portfolio was written in some COBOL 74 dialect. Normally it is not a problem to communicate between COBOL 85 and COBOL 74, but due to the fact that the existing software depended in an essential way on optimized call routines in assembly language, it was difficult for COBOL 85 systems to communicate with the surrounding systems. Also this was not a large problem. For, an upgrade project for the surrounding systems was on its way. As we all know, software projects are often late, so there is a chance that the existing software that needs to communicate with the generated COBOL 85 is not upgraded on time. Normally this is also not too much of a problem, since the new software is perhaps also late. However, if the COBOL 74 migration is more difficult than was anticipated (see [13] for some reasons why) the upgrade project will slip schedule so much that even a late new development project is finished sooner. To bridge the gap, temporary downgrading of the generated COBOL 85 code to COBOL 74 code was necessary. As soon as the language upgrading projects of the systems in the vicinity of the new system are finished, the temporary downgrading can stop. Although this whole downgrading issue might sound unprofessional at first sight, an inter-project critical-path analysis revealed the probable necessity of a language downgrading tool. Planning for the acquisition of such tools is a sign of sophisticated risk management. For more information on the implementation of this downgrading tool we refer to [2].

3.3 Y2K Issues

In this section we mention two issues that deserve some further attention. First of all, we stipulate the problems with the use of a Y2K compliant system from others. We also briefly mention management issues. We rather bring up the issue of size than a detailed plan, which is out of scope for this paper.

Y2K compliancy is not a release As indicated, projects that serve the purpose of surviving, are hard to sell. Therefore, many Y2K repair efforts are combined with enhancements. Those enhancements are the main issue, and the Y2K compliancy comes as a by-product. So, many software vendors combine a Y2K repair effort with the release of other features. In some cases the new features introduce software faults that are not at all related to the Y2K repair effort. Using such products leads to annoying nonintuitive problems when trying to make a large software portfolio Y2K compliant. The typical large software portfolio owner is user of such products. While upgrading for Y2K compliance, they are faced with the problem of coping with the software faults in the new version of the product. So some software systems that are made Y2K compliant cause trouble in other software systems that are upgraded with these products to solve the Y2K problem. We know of a case where the Y2K compliance of a certain subsystem took about two years for this reason. This was due to the enhancements that were made by the vendor to the product in addition to Y2K compliancy. The product users had taken advantage of certain internal interfaces of the old version of the product. One of the enhancements was that those interfaces were changed, which caused a major delay in the Y2K effort for the subsystem that used the internal interfaces.

Y2K management We have indicated a string of technical problems that have to be solved before actual Y2K repair efforts can start. In order to perform all these technical preparations, a lot of planning and vision is necessary. We recall that at least the following difficult problems need to be addressed first: an inventory, how to rebuild the original objects, problems with application generators, operating systems, language migrations, and so on. If all these problems are solved (or at least large parts of them), we can turn our attention to Y2K search and repair efforts. Of course, the managerial part of such change efforts should not be underestimated. One large software portfolio owner managed about 50 simultaneous Y2K projects, where all the managers reported to an overall Y2K team that tracked the methods used, and the progress made by several teams and served to steer the various teams when this was necessary. Some projects were appropriate for off-shore outsourcing, others were more appropriate for high-tech outsourcing, still others had to be done in-house. In addition we have seen that large software portfolio owners were heavily financing small high-tech companies, or even acquired them by way of insurance for having access to the best possible personnel available. Maybe the latter management problem is the most important one, since there is an endemic shortage of software personnel. Recall that such matters make utterly clear that things like date expansion or windowing—often mentioned as solutions to the Y2K problem—are only a tiny part of it. A cascade of other problems has to be managed and solved before a large software portfolio is prepared for any significant modification.

3.4 testing

The issue of testing deserves a bookshelf of its own. In this section we briefly mention a few testing problems that are strongly related to having a large software portfolio. One item is that the boundaries of testing become too large; the other item illustrates this with Euro conversion testing.

boundaries of testing During and after modification of software a lot of testing is necessary. Especially, when modifications are made to an entire software portfolio, it becomes rather difficult to test it. Normally, the modifications are made to very small parts, so that the boundaries for testing are limited. If part of a single system is modified, it is in general known how to proceed with testing although it can be quite involved. When more systems are modified simultaneously, the boundaries of testing become blurred. Suppose that all the software in a hospital was modified to make it Y2K compliant. How to test the hospital? In one case we have seen that the clocks of the hospital computers were put near the critical date, and a live simulation of a disaster was used as a test case. Let us take a look at the boundaries of this test. The clocks of the intensive care were not altered: real patients were laying there, so it was too risky. The clock of the telephone switch was not altered, since the risk that the hospital would not be reachable was too high. The clocks of the rescue helicopters were not modified, the clocks of the ambulances were not altered, and so on. This example shows that the boundaries for testing such large-scale modifications are maybe too limited to draw conclusions when the test does not reveal errors. In this hospital the test did not uncover a single problem. Can we now conclude that everything is save? Below, we will see that for Euro conversions this conclusion after testing was not save.

As a side remark, some people think that the deadline for the Y2K modification effort is just before 2000. However, these boundary problems are a major reason why owners of large software portfolios had their deadline of the Y2K repair effort *before* January 1, 1999 so that they would at least have a real year change as a test, and an entire year of testing time before the century change.

testing Euro conversions We provide an example of testability of another pervasive change that had its deadline January 1, 1999. It is the Euro conversion for the financial organizations in Europe. The Euro is the official currency starting with this date, so all the financial organizations of the European Monetary Union (EMU) are supposed to have abandoned all EMU-currencies in their software systems. Some enterprises took block-options on hotel rooms in the neighborhood so that bad weather conditions could not prevent the involved IT specialists from working on January 1, 1999. Such precautions were all part of careful contingency planning. On January 2, 1999, the Dutch National Bank revealed that a test showed a difference of 25 cents—an error. In six hours, the cause was found and corrected. This was an error in a system of a few million lines of code.

Banks are rather uncommunicative about problems. Only if the problems can no longer be denied, the public is informed. In January 1999 the Dutch public was notified by a large bank that there was a minor problem with their international transactions. It appeared that about 30% of all the international transactions vaporized and had to be taken care of manually. So hundreds of bank tellers were hired to handle those transactions by hand. A lot of software needed modification. Shortly thereafter, the European Central Bank complained that the banks were asking too much money for international transactions, while this was neither allowed nor necessary since there were no currency conversions anymore. There were: conversions to turn the tested Euro-compliant software into real Euro-compliant software. The banks did not mention this issue and complained in turn that the costs of the payments were much higher now than in the past, due to the fact that there was no standard format for such payments. From the information that we obtained, we extracted a possible cause of the problems. We presume that in some software systems the currency was used as a key: it determined the destination of where the money should be transferred to. Since those "keys" were destroyed by the Euro conversion also the destination was destroyed, and some transactions could no longer be made. So despite a lot of extensive testing, major errors were present in the software, with significant impact.

4 Concluding Remarks

We have shown that keeping a large software portfolio in good shape is a very challenging task, and that large software portfolios usually have a lot of problems. In order to enable the life cycle of such large amounts of software, it is necessary to take proper action. When dealing with large software portfolios, there is no such thing as "the solution". Instead, a lot of measures have to be taken to enable any kind of solution. We think that the major part of solution directions has to do with management awareness about large software portfolios. On the technical side, one of the key aspects is to automate as much as possible software development tasks, analysis tasks, testing and modification tasks. We think that technical issues should follow good management practice, for, in the best case technical aids only help in solving problems.

As with all tasks that are carried out more than once, there is a possibility to factor out the general principles so that they can be automated. This happens over and over again in any industry including the software industry. One of the most well-known tasks is the compilation to object code. Also tools that aid in the generation of standard parts of software systems, such as user-interfaces, parser generators, and the like are useful productivity improvers. Not only for the generation of code, but also for the design process various tools are available: tools for UML, OMT, and so on. For software analysis, there are also many tools, for instance, profilers, complexity metric tools, tools producing various diagrams, e.g., control-flow graphs, data-flow graphs, portfolio analyzers, and so on. For the testing of software there are also possibilities, think of the tools of McCabe Technologies, where code is automatically instrumented for test purposes. In all these cases, there is a clear goal: compilation, parsing, certain standard analyses, and particular forms of testing. Since the problem areas are more or less fixed it is possible to automate the tasks reasonably efficiently so that the tools are a cost effective means to help the people working on large software portfolios. In the realm of software modification tools, things become more difficult.

One way to deal with modifications of large software portfolios is to use software renovation factories. This is an architecture for the rapid development of tools that can perform many and diverse changes to massive amounts of code. To give an example, with such an architecture, it takes about a few days to build a component-based distributed software tool that turns 10 MLOC of COBOL 85 code into COBOL 74 in about 24 hours on 8 SUN work stations in a LAN. The development of such an architecture is one step in the direction of getting a large software portfolio under control. Interested readers are referred to an executive summary on how to move the organization towards the use of automated modification of their software portfolios [14]. For more technical information on the architecture for automated software renovation and maintenance we encourage the reader to consult [10].

References

[1] B.L. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[2] J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 83–90. IEEE Computer Society Press, 1999.

[3] E.W. Dijkstra. How Do We Tell Truths That Might Hurt? *SIGPLAN Notices*, 17(5):13–15, 1982.

[4] L.G. Freeman and C. Cifuentes. An industry perspective on decompilation. In H. Yang and L. White, editors, *International Conference on Software Maintenance*, 1999. Printed in the short paper appendix.

[5] J. Johnson. Chaos: The dollar drain of IT project failures. *Application Development Trends*, 2(1):41–47, 1995.

[6] Capers Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences.* Addison-Wesley, 1998.

[7] D. Kull. Anatomy of a 4GL Disaster. In R.L. Glass, editor, *Software Runaways – Lessons Learned from Massive Software Project Failures*, pages 117–131. Prentice Hall, 1998.

[8] Ovum Ltd. *Report on the Status of Programming Languages in Europe.* Ovum Report, London, 1997.

[9] E.M. Rogers. *Communication Strategies for Family Planning.* Free Press, 1973.

[10] M.P.A. Sellink and C. Verhoef. An Architecture for Automated Software Maintenance. In D. Smith and S.G. Woods, editors, *Proceedings of the Seventh International Workshop on Program Comprehension*, pages 38–48. IEEE Computer Society Press, 1999. Available at http://adam.wins.uva.nl/˜x/asm/asm.html.

[11] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 245–255. IEEE Computer Society Press, 1999. Available via http://adam.wins.uva.nl/˜x/com/com.html.

[12] H.M. Sneed and T. Dombovari. Comprehending a Complex, Distributed, Object-Oriented Software System – A Report from the Field. In D. Smith and S.G. Woods, editors, *Proceedings of the Seventh International Workshop on Program Comprehension*, pages 218–225. IEEE Computer Society Press, 1999.

[13] A. Terekhov and C. Verhoef. The Realities of Language Conversions. Available via http://adam.wins.uva.nl/˜x/con/con.html.

[14] C. Verhoef. Towards Automated Modification of Legacy Assets. *Annals of Software Engineering*, 9, March 2000. To appear. Available at http://adam.wins.uva.nl/˜x/ase/ase.html.

[15] G.M. Weinberg. *Quality Software Management: Volume 1 Systems Thinking.* Dorset House, 1992.