How to Implement the Future?

C. Verhoef
University of Amsterdam, Programming Research Group,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
x@wins.uva.nl

Abstract

This paper sheds light on the realities of implementing spanking new ideas in existing software systems. Information is provided both on the advantages and drawbacks of starting from scratch and basing yourself on an existing software asset to implement your IT future. The paper touches upon the need for tool support to refactor existing software systems so that implementing their IT future is enabled. An idea of how such refactory tools look like is given and pointers to technical research contributions are provided. Those tools aid in overcoming the maintenance debt built up over the years. It is argued that implementing the future in IT amounts to dealing with the past to a large extent.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.3.4. [**Processors**]: Parsing.
Additional Key Words and Phrases: Reengineering, System renovation, Software renovation factories.

1 Introduction

Inventing the future is one thing, and should maybe best left to the Three Princes of Serendip, who according to legend [85], during their travels were always making discoveries, by accident and sagacity, of things which the highnesses were not in quest of. Implementing such marvelous inventions is another matter, and how to do it is the subject of this paper. New inventions are often implemented in existing infrastructures, for instance, when mobile phones were invented, their full-blown implementation not just consisted of building the phones and putting antenna's everywhere in our environment. The implementors also had to make significant modifications to sometimes 20 year old public telephony switches. Or imagine a very old building, like a cathedral. It was not at all designed to accommodate a sewer, tap water, electricity, heating, and what have you, but its all there, implemented in an existing infrastructure that apparently survived countless technological generations. When you think of it, nearly all implementations of new inventions need significant adaptations of existing artifacts; artifacts that were definitely not designed to smoothly integrate with the new invention. In fact, when you try to implement a central heating system in an old building you have to understand the structure of the building, you have to take outmoded design decisions into account, and there are a lot of unexpected restrictions around. Such constrains are not present in a contemporary building that is designed from day one with things like central heating in mind. The same issues play a role when you want to implement new inventions in software intensive systems. In this paper you will get some information on what aspects pay a role when implementing the future in software.

Knowing what the hot issues in *implementing* the future in information technology are is a simple matter. It's all about money, money, money. Global benchmarking organizations like Gartner Group, Giga Group, META Group, Standish Group, and so on, are polling the big spenders on information technology where the money goes. Just looking at their wish lists gives you all the information you need. For instance, in Europe there is an annual benchmark where the top 100 companies spending the most on IT are polled. Last year (1999) this benchmark revealed that solving the Year 2000 problem soaked up to 30% of the budgets. See [66] for more information on cost aspects of the Year 2000 problem. Other top-of-the-list issues are the rise of Internet development and a growing fascination for knowledge management. Hey, this was not a surprise, for, we're moving from the information age to the age of immediate answers. Implementing these issues reveals another top-priority of the top 100 spenders on IT: dealing with erratic infrastructures that are hindering global networking [99]. Surprise or not, significant portions of IT budgets are spent on existing systems, adapting them to our ever changing environment where the future is being invented.

Owing to hardware engineering this kind of work is often called *software maintenance*. The word *maintenance* has its origins in the Latin phrase manu tenere, which means *to hold in the hand*. The word *software* has its origin in the 1960s and is considered to be the entire set of programs, procedures, and related documentation associated with a computer system. Software maintenance could be dubbed as to keep your software in the hand. As a continuous stream of data over the past 30 years has shown extensively: in general, software maintenance gets *out of hand*. One of the most well-known facts about software maintenance is that it is expensive in manpower and resources. Estimates of the magnitude of software maintenance costs range from 50% to slightly more than 80% of overall software life-cycle costs [39, 12, 35, 76, 86, 13, 58, 65, 83, 78]. Not only the total cost is high, but also the cost per unit of work is high. Already in the early seventies the ratio of the costs for development to costs for maintenance was more than 1 to 50 [103]. This ratio has gone up since then.

One study reveals that only 8% of the post-release work on software systems comprises the correction of errors [86]. Significant parts of working on software in the employment phase consists of implementing the future. So, the word maintenance is a misnomer: it implies simple corrective work. This corrective meaning is much more true in hardware maintenance where it refers to simple activities in comparison with the construction activity. Fact is that many of us consider working on existing software systems as unstimulating drudgery. When you buy a car you first figure out what you want, and then make a buy. It is not your intention to put in a jet engine in 3 months, move the steering wheel to the back seat, have it completely remote controlled, or shift gears with your rear view mirror. By contrast, the first release of a software intensive system is only the beginning of its evolution. Even stronger, as soon as the requirements phase is finalized the software grows *during development* at a rate of 2% undocumented features per month due to requirements creep [64, p. 211]. An example of the software equivalent of a "rear view mirror kludge" is the web-enabling of mainframe applications: originally designed to run on dumb 3270 data stream terminals using function keys, the new requirement is to run the front-end from a personal computer, from a remote location anywhere in the world, using a web browser and a mouse instead of function keys. While this kind of dramatic change sounds silly for a car, this is not at all the case for an evolving software system, in fact it is the most wanted implementation of our electronic future.

Some people think of shrink-wrapped software when they are confronted with the abstract notion of a software system. Or do you think of the software running on about 60 computer chips when you step into your car? In other words, what kind of systems should you have in mind for this paper? Think of a sophisticated Magnetic Resonance Imaging (MRI) Scanner, which is employed between 10 and 20 years. Of course, no hospital can afford to abandon such a system just because they want to integrate their medical information systems with the output of this MRI Scanner so that the time between scanning and diagnosis is shortened drastically, saving a lot of money. An MRI scanner is a system typically containing 2 million lines of code, written in 15 languages, running on 7 CPUs containing 4 different operating systems. Obviously complex systems, and implementing unforeseen inventions into them amounts to major renovation efforts. Another example: take a public telephone switch; they also have a life-time between 10 and 20 years. After its first release many years ago, it needs significant modifications to incorporate matters such as: ISDN, Internet telephony, mobile phones, automatic number identification, automatic location identification, 112, 911, 411, 1-800, call waiting, call redirect, voice mail, interactive voice response, computer telephony integration, the Year 2000 fix, and the Euro conversion. Note that I call the Year 2000 fix an invention. This may sound strange at first sight: isn't is just an error? No. It was in many software projects even part of the requirements. The Year 2000 fix has exactly the same characteristics as a so-called *preventive innovation*, just like birth control is a preventive innovation [87, 88]. They are both hard to understand, hard to sell, and when it is finally applied the effect is that there is, well, no effect. "Nothing happens" is the best result of a preventive innovation, for, the nature of preventive measures is to lower the probability that some future unwanted event will occur. The unwanted future event might not happen anyway, even without adoption of the measure, and so the benefits of adoption are not clear-cut. Also the prevented events, by definition, do not occur, and so they cannot be observed or counted [88]. Anyways, the main point is: think of these kind of bespoke business-critical systems, systems in constant need to be significantly extended over time, unlike a car or shrink-wrapped software.

The goal of this paper is to inform you about the way the future is implemented in such software intensive systems. Being informed will not make things easier, just more realistic. As Dörner puts it eloquently [36, pp.98-99]:

> Anyone who has a lot of information, thinks a lot, and by thinking increases his understanding of the situation will have not less but more trouble coming to a clear decision. To the ignorant, the world looks simple. If we pretty much dispense with gathering information, it is easy for us to form a clear picture of reality and come to clear decisions based on that picture.

May the informed decision be with you!

organization The rest of this paper is organized as follows. Section 2 deals with some common misunderstandings about restarting projects that need to incorporate the future. Section 3 illustrates that reusing existing software assets is also not an easy task. In Section 4 tool support to aid in reworking existing systems is highlighted. Finally, in Section 5 some sobering words, so that the innovators among us do not become too thrilled and have a little more patience before their great innovations make the transfer from future to present. Extensive references to the relevant literature help you find your way mastering the intricacies of the fine art of implementing the future.

Note This invited paper served as background material for a keynote address to the 26th Euromicro Conference, Maastricht, the Netherlands, September 5–7, 2000. The theme of the flagship meeting of Euromicro was: *Informatics: Inventing the Future.* Many thanks to the General Program Chairman of Euromicro 2000, Ferenc Vajda from the Computer and Automation Research Institute of the Hungarian Academy of Sciences (Budapest) for asking me to explain how to go about *implementing* the future once the attendees invented it.

2 Restarting from Scratch

Some people think that the best way to implement the future is to start from scratch and do everything right this time. Just stop thinking of the crusty old systems written in outdated languages, running on obsolete platforms, and designed using superseded methods. Use all the new and hot methods and technologies to materialize the fantastic innovations and all problems of the past will vanish. Several problems stand between these desires and the reality. Here's a short list:

- resistance to change
- second system effect
- silver bullet syndrome
- management responsibility cycle
- large chance of failure

The rest of this section deals with these issues. All (human) life forms have an instinct that protects them from the possibly dangerous unknown. Just remember that time you got sick of some bad food: years and years later the scent of that particular food immediately rings the alarm-bell! Overall it is very natural to have a healthy *resistance to change.* As soon as software developers are told to skip whatever they learned: use a different language, a different method, different technology and so on, be prepared for a severe reaction resisting your plans [6, 14, 82, 108]. For instance the introduction of a single technology like the valuable idea of software inspections [40, 41, 49, 47] is so resistant to change that despite a continuous stream of proof that this is a best practice [78] it is hardly possible to introduce this in an enterprise. And even when you succeed, you can shoot yourself in the foot with it, as you can read in the post-mortem of the Ariane 5 disaster [77]:

> Nevertheless, it is evident that the limitations of the SRI [Inertial Reference System] software were not fully analysed in the reviews, and it was not realised that the test coverage was inadequate to expose such limitations. Nor were the possible implications of allowing the alignment software to operate during flight realised. In these respects, the review process was a contributory factor in the failure.

So the inquiry board contributes the failure in flight 501 in part to the reviews. They do recommend to have better reviews for all software including embedded software.

Then there is the problem also known as the *second system effect*. As Fred Brooks puts it [31, p. 55]:

> This second is the most dangerous system a man ever designs. [..] The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.

The qualification *second* should not be taken literally. You can sometimes diagnose a case of second system effect by spotting irregularities in version numbers. I was once involved in a consultancy project for one of the largest financial enterprises in the world. The main version numbers of the system that needed refactoring badly [46] were 1, 2, and 4. There was no version 3. Version 3 was an attempt to start from scratch with this very successful system using a new language, employing a new platform, and adding many new features. A clear example of the infamous second system. As it turned out, version 4 was the reincarnation of the failed effort, and indeed it ran on the new platform, was written in the new language, but it did not contain by approximation the features of the much more powerful version 2. After these failures the CIO decided to embark on a multi-million dollar reengineering effort that would evolve version 2 in many small steps towards the changed business needs. Needless to say that huge amounts of money were waisted on this "second" system, and that extremely high-turnover rates were caused by this failure, adding to the loss of valuable knowledge about the existing systems.

Apart from these problems the well-known *silver bullet syndrome* is always lurking. Brooks postulated in an invited paper at the IFIP 86 conference in Dublin (later reprinted in [30]) that there is no single development in technology or management that can improve productivity, reliability, or simplicity even one order-of-magnitude. The well-known myth that dramatic improvements are possible with simple means, is still omnipresent in the current software engineering practice. What is maybe less well-know is why this pitfall is roaming around us. I think that one of the major causes why people keep falling in this trap is that, they do not feel the pain it causes. Normally when you make a decision, you are responsible for the consequences. Especially in the case of failure you are to blame, and next time, like with rotten food, the preventive alarm-bell will sound. In the IT business things are different. There is a dynamic that could be called the *management responsibility cycle* that plays a major role. Often, managers are responsible for sparking off the use of new technology, new methods, or new tools, and so on. There is a huge shortage of IT personnel. As a consequence, the average time that a manager is responsible for a certain project is in some companies about 8 months. Software projects that use unfamiliar technologies are normally taking longer than 8 months, so the managers that initiated the new technology are gone before it is time to take responsibility for the consequences of their decisions. They moved up in the management responsibility cycle and are not held responsible. It is interesting to see how the new manager reacts on a project that is under way: if he or she approves the use of unfamiliar technology initiated by the former manager, this implies taking over the responsibility. Therefore, often another (new) technology is proposed. This causes again a delay, and probably just long enough so that the second manager can also escape the project without taking the responsibility. Being a good manager then amounts to successfully avoiding taking responsibility and assigning management tasks to inexperienced developers just before things go awry (which closes the negative feedback loop). In this way the silver bullet syndrome is self-sustaining and becomes a chronic disease within an organization. Apart from this unhealthy corporate situation also the people themselves change due to the constant organizational reengineering they are exposed to. The continuous focus on short term gain learns them the lesson that detachment and superficial cooperativeness are better armor for dealing with the current realities of short term emphasis than behavior based on values of loyalty and service [97, p. 25]. This short-term philosophy threatens to corrode the character of human beings, particularly those qualities of character which bind them to one another and furnishes each with a sense of sustainable self [97, p. 27].

Another problem with a restart from scratch is that large software projects are risky and have a *large chance of failure*. To give you an idea, in one research report 8000+ projects were taken as input to get an idea of the situation in the USA. One of the outcomes was that each year about 81 billion US dollars were spent on cancelled projects. Moreover, 59 billion was spent each year for challenged projects. Think of cost/time overruns, or a change of scope [61, 55]. These figures are not carved in stone: some people do not believe these figures [50, p. 2]. Based on my own experience I do believe that the losses are huge. Of course, we all know about the infamous software related disasters like the exploding space shuttle Challenger [81], the (earlier mentioned) exploding Ariane 5 rocket [77] and so on. We are not talking about such disasters per se. Let me give a few examples to make clear what I mean.

The Department of Motor Vehicles (DMV) in California owns an information system originating from 1965. The system had become so brittle that it took the equivalent of 18 programmers working for an entire year to add a social security number file to the drivers license and vehicle registration file [89]. Then they decided to thoroughly renew the system in the early 1990s. Seven years later not a single usable program was produced and the state had to cancel the project with a loss of 44 million US dollars [16, 54, 48, 38, 51]. Now you think okay, but we learned from this major disaster and this will not happen again. Au contraire. In 1993, the Oregon Department of Motor Vehicles embarked on a five-year project to computerize its paper-based records. It was estimated that they could downsize the DMV workforce by one-fifth and save 7.5 million US dollar annually. In a few years, the delivery date slipped to 2001 and the estimated budget rose from 50 to 123 million US dollars. In 1996 a prototype was installed that was a total failure [42]. Another 123 million US dollars shot to shambles.

Needles to say that the 81 billion US dollars is not solely spent on disasters in risky space related software intensive projects but also in daily information systems practice. Many more failure stories can be found in books devoted to the topic of failure in software engineering [50, 51].

So, starting from scratch is a risky business and not always the best idea, although it appeals to many of us. An alternative to restarting from scratch to implement the future is to renovate the existing software system first.

3 Restarting from an Existing System

Many of the problems that were mentioned for brand-new systems also apply to projects where existing systems play a prominent role. Just don't start flying ribbons in the sky and think that starting with an existing system would solve all your problems. Several major impediments are waiting for you here as well:

- failure of (business process) reengineering
- underestimating the problems
- business-rules are well-hidden
- paying the maintenance debt first
- lack of tool support

Often, a software reengineering project is a consequence of a reshuffling of the business, also called *business process reengineering* [57]. These business reengineering projects do not necessarily end in successfully reengineered enterprises. As Scott Adams puts it: a good example for which you can confidently predict failure is any large-scale reengineering effort [1, p. 73]. Or take Clemons who states that many, even most reengineering efforts fail [34]. The existing system of the Californian Department of Motor Vehicles system is a good example: it was so problematic that further modifications seemed impossible in a cost-effective way. But it was also not possible for them to restart from scratch. So they seem to be trapped. In fact, some companies found themselves as early as 1993 in such a situation where the entire workload is related to updating, enhancing, and fixing problems in existing legacy applications [63, pp. 145-6]. So it is not even possible to do *any* new development for those companies.

Another problem that is associated with major enhancements of an existing system is that people often grossly *underestimate* such efforts. When expectations are not well-managed this can cause as severe failures as you can encounter on a green-field software engineering project. If reengineering projects are not carefully planned, do not get the proper expert staffing, and no realistic budget they are a sure candidate for failure. In fact, the more easy you think a task is, the more likely it is that you make errors. An illustration will help here. Suppose you want to downsize an old system from the mainframe to a PC environment [98]. Since you are into component-based software engineering [37, 4], and the Visual Basic Component market is very promising, you decide to migrate the obsolete OS/VS COBOL code to Visual Basic while you are busy downsizing the system. Then one of the software engineers converts the code below (both fragments are taken from [100]):

```
DATA DIVISION.
 01 A PIC S9V9999 VALUE -1.
PROCEDURE DIVISION.
 ADD 1.005 TO A.
 DISPLAY A.
```

into its equivalent in Visual Basic:

```
Dim A As Double
A = -1
A = A + 1.005
MsgBox A
```

Are you alarmed? Probably not. However, the Visual Basic code yields +0.0049, indicating a rounding error. Would you want to be the manager of this conversion project when the accounting department finds out that millions of US dollars are missing due to some obscure rounding error in the freshly converted system? If you are exposed to the manager responsibility cycle probably someone else gets the blame (but remember that your character might corrode). Apart from that, this example illustrates how easy it is to underestimate the difficulties of reengineering existing systems. Now imagine an accumulation of such countless unfathomable details scattered over millions of lines of code. That, now, is a reengineering project.

Remember that existing systems have been modified often. Most of the times the updates and enhancements have not been mirrored in user and or system documentation so that the requirements of a new system are well-hidden in the code of the existing system. Huge amounts of critical knowledge often called *business-rules* are *encoded* over the years in such systems. I say encoded and I mean encoded. Often I get the question: how is it possible that you write a software system that you cannot understand once it is written? Fair question. How is it possible that we cannot solve every imaginable medical problem whereas all the relevant information is just in our genes? The answer is: the information may be there, but inaccessible for human comprehension. Of course, deciphering the human genome is much more complicated than to decode a software system and extract its actual business rules. But both humans and serious software systems materialized in an evolutionary manner [7, 70].

When commencing with a major reengineering effort of evolved systems it is necessary that the software is fast-forwarded 30 years in time on a very sort notice. In order to be successful the enormous *maintenance debt* has to be paid, and extensive reworking of the system is mandatory before any new invention can be implemented in a save manner. This may be costly but bear in mind that the payoff especially for such business-critical indefinitely-lived systems is commensurately large [53].

Luckily, more and more people are recognizing that reworking existing systems to enable change is a good idea. A recent book calls this effort *refactoring* [46]. The book advocates Ovid's "Adde parvum parvo magnus acervus erit[1]". You should make

[1] Add little to little and there will be a big pile

many small systematic modifications to an existing software asset and the result will be a system that is better understandable, and enabled for change—the ultimate goal of extensive rework so that the future can be implemented. In the book, the usual arguments of program transformation aficionados are extensively used. Already in 1970 [15] Boyle claimed that many small transformations to computer programs (supported with a tool) can cause improvements of many kinds to the programs. Indeed, although the refactoring book emphasizes reworking by hand, the book contains a brief chapter on refactoring tools whose introduction reads as follows [46, p. 401]:

> One of the largest barriers to refactoring code has been the woeful lack of tool support for it. Languages in which refactoring is part of the culture, such as Smalltalk, usually have powerful environments that support many of the features necessary to refactor code. Even there, the process has been only partially supported until recently, and most of the work is still being done by hand.

The authors of Chapter 14 of [46], Don Roberts and John Brant, mark the spot with this remark. There is an endemic *lack of tool support* to aid reworking existing code. Jones reports Year 2000 search engines support for less than 50 languages and Year 2000 repair engines are available for about 10 languages [66, p. 325]. This seems a lot, but it was also estimated that the Year 2000 problem manifests itself in systems written in at least 700 languages [65]. So tool support was available only for a small fraction of the languages. Add to this that most software systems are mixed language applications for which most Year 2000 engines come to a halt anyway, and the conclusion must be that general tool support for automatically renovating aging software systems is lacking. This is problematic since, especially with large amounts of code, it is error prone and time consuming to make changes by hand, and tool support to rework existing systems would help significantly.

So also using existing systems as a starting point of implementing the future is not an easy task. But extensive tool support to aid systematic evolutionary modifications leads to a situation where implementing the future becomes more feasible.

4 Tool support for Implementing the Future

Gartner Group advises anyone who has the responsibility for a software portfolio of 2 million lines of code or more should use a so-called software renovation factory to implement the Year 2000 updates, and/or analyses [56, 67]. Of course this does not only apply to the Year 2000 problem but to any significant software renovation problem. A *software renovation factory*, or should it be called a refactory, is a product-line architecture [5] that enables the rapid implementation of tools to support program and system analysis and/or transformations. One could say that such refactories are the future compilers of our existing software assets, meant to "compile" them into refactored systems better capable of meeting new business needs [73, 59].

What should be the properties of a refactory? You would expect it depends on the changes to the existing software are likely to occur. In the refactoring book an extensive list is presented, which could serve as a starting guideline. However, when dealing with software that has a large maintenance debt, the most likely changes that are necessary are the ones you'd expect the least. A few examples make things clear here. What about an automated COBOL 85 back to COBOL 74 conversion? It consists of `GO TO` introduction, explicit scope terminator elimination, among other unexpected code modifications [32]. Or a COBOL to PL/I conversion? Or going from SAP's ABAP/4 to VS COBOL II? Notice that all these conversions are nonintuitive: they go from more modern languages back to older paradigms. Still all these real-life reengineering projects were utterly necessary. To implement the future, there are many more examples that spring to mind, but I'll refrain from giving you a long list. The details would distract you from the main point which is: it is important for now to realize that in general no one can predict what kind of modification is useful. So, a software renovation factory should be prepared for *any* analysis or code transformation. Of course within the limits of cost-effectiveness.

Instead of bogging you down with a requirements engineering document on the ins and outs of the indispensable features that a software renovation factory should contain, it is maybe better to continue with a typical case, and discuss requirements issues when they pop up.

Let's take a typical business-critical mixed language application that runs on an IBM Mainframe and is written in COBOL with embedded SQL. Some parts of this management information system are more than 25 years old, and other parts have been added just yesterday. This example is based on research reported on in [93]. A number of typical refactorings are necessary before the new version of DB2, which implements SQL, can be installed. The example system employs hard-wired constants all over the place including literals for SQL return codes. For the new version of DB2 additional SQL return codes have to be added, and it was decided by the company that prior to this update, the hard-wired constants should be eliminated. This would also ease future modifications to the software system, for instance web-enabling this system.

We need to parse the COBOL/SQL code. It would be naïve to think that lexical tools would do the trick. Many analyses and modifications need type information, or need information that is not concentrated in one location. In [28] an entire section is spent on what devastating problems would occur when just removing `UPON CONSOLE` from statements like:

```
DISPLAY '** BEGIN PROGRAM' UPON CONSOLE.
```

This would cause the system to be broke at worst, and unreadable at best. On second thought, it is not hard to figure out that as soon as you start to modify millions of lines of code in thousands of files with lexical tools, it can easily create havoc. Believe me, it will, so a parser is what you need. Now you know more: don't suggest to use Microsoft Word for making changes to business-critical systems, but the solution does not become more clear cut, as Dörner eloquently formulated it in his book on the logic of failure (see Section 1).

Ever built a parser? Ever built a COBOL parser? Ever built a COBOL parser that does not remove comments, that can handle different dialects, embedded languages like CICS and/or SQL, that deals with (home grown) preprocessors, that does not destroy compiler directives, that can handle unexpanded macros, unexpanded include files (copy books), that knows how to deal with debugging lines, continuation lines, that moreover parses undocumented syntax, retains idiosyncratic layout, to mention a few issues? Well, these are the typical problems you need to overcome when dealing with the compiler of the future: the existing systems are written in languages containing the above mentioned properties so tools to refactor or renovate code should take them into account.

To make a long story short: the first problem to attack is to get a parser suitable for software renovation. This problem in itself sparked off an entire field of research, called grammar engineering [73], computer aided language engineering [95] or lingware engineering [80]. It turns out to be possible to obtain parsers both for very obscure languages and for well-established languages such as COBOL in a cost-effective manner by using a very systematic approach of taking small steps and applying refactorings. More information on how to accomplish this can be found in the following papers [91, 95, 94, 73]. By way of proof, a complete grammar specification for COBOL can be retrieved on the Internet [72].

It would come in handy when you can combine grammars to build new ones. For instance the combination of an SQL grammar with a COBOL grammar would be nice to have for the example project. Unfortunately, with this extra requirement it becomes infeasible to use so-called mainstream parser generator technology: this is Lex/Yacc like technology [75, 62] implementing the class of LR languages. It is known that combining two LR languages does not necessarily result in another LR language [3, 2]. Consequently, combining LR grammars does not necessarily result in a grammar that you can use as input for an LR based tool. So we need different tools for combining grammars. One solution is to use tools supporting an implementation of general context-free languages, e.g., Generalized LR parser generators [74, 102, 84, 105, 106] or approximations of them [11].

It is possible to implement a COBOL parser with all the qualifications that were presented above in a cost-effective manner [25, 73]. In fact this is possible for other languages, as well.

So it is save to assume that you can parse the COBOL/SQL example system for refactoring purposes. Next, for each program it should be checked whether the hard-wired SQL return codes are used, and if so certain modifications need to be made to the code. Hopefully the modifications are done completely automated. Okay, so you need to *analyze* and *transform* source code. Now imagine this giant parse tree residing in some repository, or in main memory. Some tool should figure out whether the Boolean condition

```
SQLCODE = -818 OR -904 OR -911 OR -922
```

occurs in the code. Of course this is only one possibility a programmer could have written it down. Many other possibilities exist including ones containing brackets (see below). So this tool should sniff along the entire parse tree in search of a specific piece of subtree representing the above code fragment. If it has found it the tool reports this. This implies that another requirement of a software renovation factory is that a set of generic analyzers is available that can combine the analysis results of small trees into large ones leading to an analysis of an entire parse tree. It is possible to *generate* such generic functionality directly from the grammar of a language [28]. Using such a generic analysis framework, it is trivial to implement a tool detecting the above displayed piece of code.

Once such a piece of code is found, it has to be refactored into something else. Whereas in the refactoring book the idea is put forward to improve the code in some way, in software renovation the code does not necessarily improve in the sense of understandability. It becomes other code. For instance, the code fragment:

```
IF SQLCODE = -818 OR -904 OR (-922 OR -911)
    PERFORM 9999-EM904-FILL
```

has to be refactored into:

```
MOVE SQLCODE TO SQL-CODE IN LINKAREA-EM948
CALL 'UT100' USING L-EM948 LINKAREA-EM948
    IF RETURNCODE = '9'
      PERFORM 9999-EM904-FILL
```

So the four hard-wired constants are exchanged for another hard-wired constant: the number `9`. With respect to the refactoring movement this kind of code modification is debatable: for, is the code really improved with respect to comprehensibility? On the other hand, we should not flex the code more than necessary since in business-critical systems failure can be quite expensive. Anyways, imagine this giant parse tree. A tool should traverse the tree and upon detection of the Boolean condition it should replace the unwanted `IF` construct by the `MOVE` and `CALL` statements. This reveals yet another requirement of a software renovation factory: you need a framework of generic traversals. It is convenient when these generic traversals implement the identity mapping on a parse tree. Then on certain locations you can override the framework to implement an actual modification you need to make. Completely analogous to the generic analysis functionality it is possible to *generate* a generic traversal framework directly from the grammar of a language [28].

Now assume that you have analyzed and refactored this entire system in accordance with the requests of the owners. How to get their source code back? After all, everything is now stored in parse tree format. We have to go back to source text, to be fed to maintenance teams and the compiler of the present. It is possible to generate from the grammar of the language a set of rules that need manual adaptation to the local company standards that takes care of formatting the code. It is the inverse of a parser generator and is often called an unparser generator [29].

So you need grammar engineering tools, sophisticated parser generators, generic analysis and transformation generators, and unparser generators to construct a product-line architecture that is suitable to refactor source code. There is more. As soon as all the tools are available to carry out the mass-changes to this COBOL/SQL software system, they have to be coordinated. As said earlier, we need to perform many small steps and combine them to accomplish significant change. The steps are small tools like the one discussed above. Imagine you have accumulated a number of such steps, and you have a certain ordering of applying them. The steps should ideally be components that you can invoke at wish. In that way it is possible to assemble a special-purpose refactoring product for a client who can run it locally. Of course a component-based development approach also lets you use third-party components such as parsers, control-flow or data-flow analyzers. So, an architecture able to smoothly integrate heterogeneous components is mandatory [70].

For the COBOL/SQL refactoring example things look as follows: 5 tools were necessary to carry out

components	eqs	bytes
Norm-cond.eqs	15	2622856
Add-EM948.eqs	7	2670264
Use-EM948.eqs	20	2673768
Eval-SQL-a.eqs	5	2638336
Eval-SQL-b.eqs	6	2638336

Table 1: Some characterizations of the components.

totals	SLOC
handwritten code:	325
Shared C code:	91777
Make files:	4651

Table 2: Totals of physical source lines of code for the components.

the refactoring tasks. The coordination was simple: a pipeline of 5 tools of which the first tool loops until a fixed point is reached. This was expressed in a coordination language with a supporting software bus called ToolBus [9, 8, 10]. Furthermore, using an expert compiler [23, 18] the small steps encoded as term rewriting systems [71] were compiled into ANSI C programs. The parser used an efficient binary format as intermediate format that is understood by the generated C programs [17].

Without going into too much detail it is worthwhile to take note of some metrics that characterize these kinds of projects. In Table 1 we summarized the 5 components. In the second column you find an entity called `eqs`. This abbreviation stands for *equations* that are encodings of conditional term rewriting rules possibly containing negative premises [71, 69, 45, 52, 79]. Such term rewriting systems are a very powerful means to succinctly express, e.g., code transformations. These term rewriting systems can be compiled into executables [68, 107, 23, 18, 43, 44]. In this example the implementation discussed in [23, 18] was used. The executables that were built are all about 2.6 Meg and have a memory footprint of 10 Meg.

The data in Table 2 gives you an idea how much coding effort the work comprised. SLOC stands for physical source lines of code. The handwritten code only took 300+ lines. Of course, this code was connected to the generated frameworks so that the resulting generated C code contained 90.000+ lines. The make files were also generated and compiling the components amounted to invoking the GNU tool `gmake`.

activity	min.
analysis and design	240
implementation	180
loading grammar and tools	10
dumping trees	10
C code generation	30
make	20
factory assembly	5

Table 3: Productivity metrics for this particular problem.

In Table 3 the accumulated effort by activity of the entire tool construction project is specified. The abbreviation min. stands for minutes. You can easily check that the total effort is a single day, after which it is possible to start running the tool on a software system. Bear in mind that these figures are representative for someone who is well-informed on software renovation, on COBOL/SQL systems, on the domain of the information systems and on program transformations. So probably an untrained software engineer needs more than a single day to accomplish this task. Nevertheless, the figures clearly show that the product-line architecture indeed is capable of delivering ad-hoc tools of very high-accuracy that can accomplish the required refactoring tasks in a 100% automated fashion.

Since you only looked at a single sketchy example obviously not all the requirements for software renovation factories have been highlighted, although the most important ones have been illustrated. We depict them in Figure 1. The five pillars of software renovation factories are:

- grammar engineering
- factory generation
- component development and testing
- factory assemblage
- factory operation

The first four pillars are demarcated in Figure 1 by the dashed boxes. The fifth pillar is illustrated

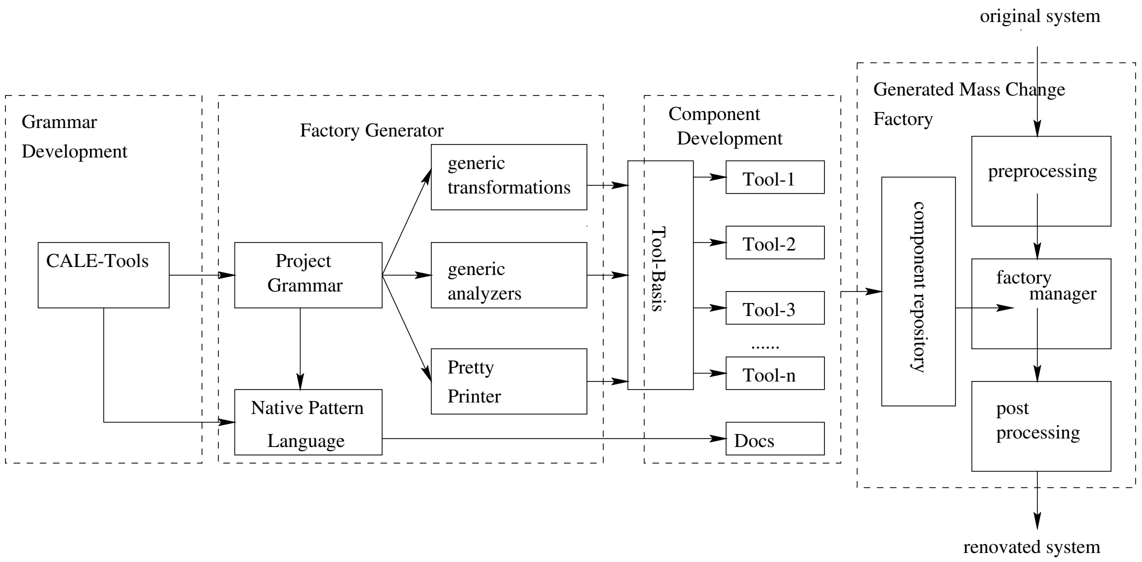

Figure 1: The five pillars of Software Renovation Factories

```
hard-wired (Lit1 Lit2 Lit3 Lit4) = true
=======================================
Use-EM948_Sentence(
Statement*1
IF SQLCODE = Lit1 OR Lit2 OR Lit3 OR Lit4
  Sentence-1
) =
Statement*1
MOVE SQLCODE TO SQL-CODE IN LINKAREA-EM948
CALL 'UT100' USING L-EM948 LINKAREA-EM948
IF RETURNCODE = '9'
  Sentence-1
```

Figure 2: Example of a conditional term rewriting rule.

by the I/O arrows indicating that an original system is the input of the assembled factory, and its output is a renovated system.

Let's discuss the remaining issues that are in the figure, but were not yet covered. The first thing that needs a clarification is the so-called *native pattern language* [92]. This is best explained by example. One of the equations dealing with the COBOL/SQL refactoring example is shown in Figure 2.

The idea of a native pattern language is that the code you have to write in transformations should resemble the world of the domain as much as possible. In that way, it is not too difficult for domain experts to develop refactoring components. So for instance the replacement code that we illustrated before, is verbatim contained in the transformation code. Of course the patterns contain variables. They resemble the variables that a programmer usually finds in user manuals. For instance `Sentence-1` is a familiar one for COBOL programmers (it stands for an arbitrary COBOL sentence, just like in the language reference manual [60]). Such a pattern language can be generated in a trivial manner [92] from the grammar of the language.

In the third dashed box `Docs` is mentioned. This is a nicely typeset version of the grammar that serves as a language reference manual for the tool developer. It has been generated using unparser generator technology [29]. In the fourth box there is a repository containing tools. The factory manager is a tool that generates the assembled end-product from the coordination script and the components needed for the renovation task. The user should provide the factory manager with information where the original files are located, in what languages the original system is written, on how many machines the various components should run, and some other vital technical data. All in all this comprises a few lines of code (say 20), from which the entire refactory is generated. By pressing a button the renovation task starts. Some languages need pre- and postprocessing, such as stripping line numbers in COBOL programs [25]. They are restored and if necessary updated in a postprocessing tool. Of course more such small technical details are taken care of during pre- and postprocessing.

It would be nice to see some figures about the performance of the mass-change to the COBOL/SQL system so that you get an impression of the fifth pillar: factory operation. I do not have such figures for that particular project. Instead I provide some figures of a similar project: a COBOL 85 to COBOL 74 conversion project. It also comprised of about 6 transformation components in addition to parsing, unparsing and pre- and postprocessing. About 10 million lines of VS COBOL II were converted back to OS/VS COBOL using a generated distributed component-based software renovation factory that ran on 9 SUN Sparc stations, which took about 24 hours real-time. This is about 400.000+ LOC per hour.

To learn more about the ins and outs of software renovation factories, their use, their implementation, case studies, and how to employ them in a commercial environment the following papers are suitable starting points [21, 33, 19, 20, 70, 22, 104]. For more involved information take a look at [25, 28, 26, 92, 27, 93, 90, 96].

5 Outlook

Despite many efforts of software professionals who invest in educating us about improving the intrinsic quality of software systems, and how to keep their quality up to date, in practice many enterprises are not at all interested in quality. Since it is very hard for such organizations to assign business value to the *intrinsic* quality of the software product itself, it is going to be a hard-sell to convince the Mongolian hordes of the relevance of building at high quality levels. This is not just a stupid attitude of such enterprises. It is really hard to calculate the risk of making a change to a software system, without adding business value but to improve its structure enabling possible future enhancements. The risk of failing today due to preventive measures is weighted against the risk of failing tomorrow due to the lack of taking such preventive measures. Bear in mind that if, e.g., a brokerage operation is down the costs are huge: 6.450.000 US dollar per hour. For a Credit Card/Sales Authorization system the costs are about 2.600.000 US dollar per hour [101, p. 5]. Therefore, the reality of today is that sooner or later in virtually every successful software intensive system a maintenance debt is created. Just like the fact that many people are keen on borrowing money, but less willing to pay the debts, software owners are interested in moving onwards with software at minimal costs but not willing to spend their budgets to preventive measures that will keep their costs low in the long run. The management responsibility cycle prohibits long-term thinking, so these preventive measures are seen as mere failure magnets. In other words, it is to be expected that maintenance and renovation are going to be among us for a long time. If there is no way out anymore, the companies who accumulated enough money by employing their software-in-debt have the funds to embark on major renovation projects. Large reengineering efforts are only initiated if there is a compelling reason. Most of the times the reason is that a Prince of Serendip invented the unavoidable future. This paper has shown you that implementing such great ideas starts with paying the debts. So implementing the future, implies dealing with the past. Therefore, software renovation is *the* technology to implement the future.

About the author Chris Verhoef is affiliated with the University of Amsterdam and the Software Engineering Institute of the Carnegie-Mellon University. He is an elected Executive Board member of the IEEE Technical Council on Software Engineering. He serves in Steering Committee, General Chair and Program Chair positions for several important juried research conferences, including the IEEE Working Conference on Reverse Engineering and the European Conference on Software Maintenance and Reengineering. He co-founded the upcoming IEEE International Conference on Software Architecture. His research interests are software engineering, maintenance, renovation, software architecture, and theoretical computer science. He is a frequent speaker on international conferences. He contributed to numerous papers in conference records and journals. He co-authored two chapters in computer science handbooks. He co-edited conference proceedings and special issues. He has acted as an industrial consultant in several software intensive areas, notably hardware manufacturers, telecommunications companies, financial enterprises, leading software renovation companies, and large service providers. Contact him at the Programming Research Group, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands; `x@wins.uva.nl`. His research is available via `http://adam.wins.uva.nl/~x`.

References

[1] S. Adams. *The Dilbert Principle.* MacMillan Publishers Ltd, 1996.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling.* Prentice-Hall, Englewood Cliffs (NJ), 1972–73. Vol. I. Parsing. Vol II. Compiling.

[4] P. Allen and S. Frost. *Component-Based Development for Enterprise Systems.* Cambridge University Press, 1998.

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

[6] M. Bauer (ed.). *Resistance to New Technology: Nuclear Power, Information Technology, and Biotechnology.* Cambridge University Press, 1997. reprint edition.

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

[8] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88, 1996.

[9] J.A. Bergstra and P. Klint. The discrete time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, 1996.

[10] J.A. Bergstra and P. Klint. The discrete time ToolBus—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[11] D. Blasband. *Automatic Analysis of Ancient Languages.* PhD thesis, Free University of Brussels, 2000. Available via http://www.phidani.be/homes/darius/thesis.html.

[12] B.W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25:1226–1241, 1976.

[13] B.W. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[14] B.M. Bouldin. *Agents of Change – Managing the Introduction of Automated Tools.* Yourdon-Press, 1989.

[15] J.M. Boyle. A transformational component for programming language grammar. Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, 1970.

[16] J.S. Bozman. DMV Disaster: California Kills Failed $44M Project. *Computerworld*, page 1 and 16, May 1994.

[17] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software—Practice and Experience*, 30:259–291, 2000.

[18] M.G.J. van den Brand, P. Klint, and P. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Proceedings of the eight International Conference on Compiler Construction*, volume 1575 of *LNCS*, pages 198–213. Springer-Verlag, 1999.

[19] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[20] M.G.J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997. Available at http://adam.wins.uva.nl/~x/sigplan/plan.html.

[21] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. *ACM Software Engineering Notes*, 22(1):57–68, 1997. Available at http://adam.wins.uva.nl/~x/reeng/REanno.html.

[22] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Springer-Verlag, 1998. Available at: http://adam.wins.uva.nl/~x/sale/sale.html.

[23] M.G.J. van den Brand, P. Olivier, J. Heering, and P. Klint. Compiling rewrite systems: The ASF+SDF compiler. Technical report, CWI/University of Amsterdam, 1998. In preparation.

[24] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[25] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer-Verlag, 1997. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html.

[26] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 11–19, 1998. Available at http://adam.wins.uva.nl/~x/cfn/cfn.html.

[27] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[28] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000. Available at http://adam.wins.uva.nl/~x/scp/scp.html. An extended abstract with the same title appeared earlier: [24].

[29] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[30] F.P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[31] F.P. Brooks Jr. *The Mythical Man-Month – Essays on Software Engineering.* Addison-Wesley, 1995. Anniversary Edition.

[32] 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.

[33] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[34] E.K. Clemons. Using Scenario Analysis to Manage the Strategic Risks of Reengineering. *Sloan Management Review*, 36(4):62, 1995.

[35] E.B. Daly. Management of Software Engineering. *IEEE Transactions on Software Engineering*, SE-3(3):229–242, 1977.

[36] D. Dörner. *The Logic of Failure – Recognizing and Avoiding Error in Complex Situations.* Perseus Books, 1996.

[37] A.W. Brown (ed.). *Component-Based Software Engineering.* IEEE Computer Society Press, 1996.

[38] V. Ellis. Audit says DMV ignored warning. *Los Angeles Times*, pages A3–A34, Augustus 18 1994.

[39] J.L. Elshoff. An analysis of some commercial PL/I programs. *IEEE Transactions on Software Engineering*, SE-2(2):113–120, 1976.

[40] M.E. Fagan. Design and code inspections to reduce errors in programs. *IBM Systems Journal*, 15(3):182–211, 1976.

[41] M.E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, 1986.

[42] T. Field. When bad things happen to good projects. *CIO*, October 15 1997. Retrieved via: `http://www.cio.com/archive/101597_bad.html`.

[43] W.J. Fokkink, J.F.Th. Kamperman, and H.R. Walters. Within ARM's reach: compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Transactions on Programming Languages and Systems*, 20(3):679–706, 1998.

[44] W.J. Fokkink, J.F.Th. Kamperman, and H.R. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1), 2000. To appear.

[45] W.J. Fokkink and C. Verhoef. Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories. In J.-P. Finance, editor, *Proceedings 2nd Conference on Fundamental Approaches to Software Engineering*, volume 1577 of *LNCS*, pages 98–113, Amsterdam, 1999. Springer-Verlag.

[46] M. Fowler. *Refactoring – Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, 1999.

[47] D.P. Freedman and G.M. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews.* Dorset House, 3rd edition, 1990. Originally published by Little, Brown & Company, 1982.

[48] W.W. Gibbs. Software's chronic crisis. *Scientific American*, 273(3):86–95, 1994.

[49] T. Gilb and D. Graham. *Software Inspection.* Addison-Wesley, 1993.

[50] R.L. Glass. *Software Runaways – Lessons Learned from Massive Software Project Failures.* Prentice Hall, 1998.

[51] R.L. Glass. *Computing Calamities.* Prentice Hall, 1999.

[52] J.A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems (CTRS '88)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 1988.

[53] R. Gray, T. Bickmore, and S. Williams. Reengineering Cobol Systems to Ada. In *The Proceedings of the Seventh Annual Air Force/Army/Navy Software Technology Conference*, Salt Lake City, April 1995.

[54] R.X. Gringely. When disaster strikes IS. *Forbes ASAP*, pages 60–64, Augustus 29 1994.

[55] The Standish Group. CHAOS, 1995. Retrievable via: http://standishgroup.com/visitor/chaos.htm.

[56] B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*. GartnerGroup, 1996.

[57] M. Hammer and J. Champy. *Reengineering the Corporation*. Harper Business, New York, US, 1993.

[58] M. Hanna. Maintenance Burden Begging for a Remedy. *Datamation*, pages 53–63, April 1993.

[59] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, March 2000.

[60] IBM Corporation. *VS COBOL II Reference Summary*, 1.2. edition, 1993. Publication number SX26-3721-05.

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

[62] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[63] C. Jones. *Assessment and Control of Software Risks*. Prentice-Hall, 1994.

[64] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, second edition, 1996.

[65] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

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

[67] N. Jones. Year 2000 market overview. Technical report, GartnerGroup, Stamford, CT, USA, 1998.

[68] J.F.Th. Kamperman. *Compilation of Term Rewriting Systems*. PhD thesis, University of Amsterdam, 1996.

[69] S. Kaplan. Positive/negative conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 129–143. Springer-Verlag, 1988.

[70] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998. Available at: http://adam.wins.uva.nl/˜x/evol-se/evol-se.html.

[71] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume II*, pages 1–116. Oxford University Press, 1992.

[72] R. Lämmel and C. Verhoef. *VS COBOL II grammar Version 1.0.3*, 1999. Available at: http://adam.wins.uva.nl/˜x/grammars/vs-cobol-ii/.

[73] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery, 2000. Available via: http://adam.wins.uva.nl/˜x/ge/ge.html.

[74] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[75] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, UNIX Programmer's Supplementary Documents, volume 1 (PS1) edition, 1986.

[76] B.P. Lientz and E.B. Swanson. *Software Maintenance Management—A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading MA: Addison-Wesley, 1980.

[77] J.L. Lions. ARIANE 5 Flight 501 Failure / Report by the Inquiry Board, 1996. Retrievable via: http://www.esa.int/.

[78] S. McConnell. *Rapid Development*. Microsoft Press, 1996.

[79] C. K. Mohan and M. K. Srivas. Conditional specifications with inequational assumptions. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems (CTRS '88)*, volume 308 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 1988.

[80] M.J. Nederhof, C.H.A. Koster, C. Dekkers, and A. van Zwol. The grammar workbench: A first step towards lingware engineering. In W. ter Stal, A. Nijholt, and H.J. op den Akker, editors, *Proceedings of the second Twente Workshop on Language Technology – Linguistic Engineering: Tools and Products*, volume 92-29 of *Memoranda Informatica*, pages 103–115. University of Twente, 1992.

[81] United States. Presidential Commission on the Space Shuttle Challenger Accident. *Report to the President / by the Presidential Commission on the Space Shuttle Challenger Accident*. DIANE Publishing Co, 1986.

[82] R.S. Pressman. *Making Software Engineering Happen*. Prentice-Hall, 1988.

[83] L.H. Putnam and W. Myers. *Measures for Excellence – Reliable Software on Time, Within Budget*. Yourdon Press Computing Series, 1992.

[84] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

[85] T.G. Remer (ed.). *Serendipity and the Three Princes of Serendip; From the Peregrinaggio of 1557*. Norman, University of Oklahoma Press, 1965.

[86] J. Reutter. Maintenance is a management problem and a programmer's opportunity. In A. Orden and M. Evens, editors, *1981 National Computer Conference*, volume 50 of *AFIPS Conference Proceedings*, pages 343–347. AFIPS Press, Arlington, VA, 1981.

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

[88] E.M. Rogers. *Diffusion of Innovations*. Free Press, 4th edition, 1995.

[89] M.B. Romney, P.J. Steinbart, and B.E. Cushing. *Accouning Information Systems*. World Student Series. Addison-Wesley, 7th edition, 1997.

[90] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 72–82. IEEE Computer Society Press, 1999. Available at http://adam.wins.uva.nl/~x/cics/cics.html.

[91] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions – extended abstract. In B. Nuseibeh, D. Redmiles, and A. Quilici, editors, *Proceedings of the 13th International Automated Software Engineering Conference*, pages 314–317, 1998. For a full version see [95]. Available at: http://adam.wins.uva.nl/~x/ase98/ase98.html.

[92] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society, 1998. Available at http://adam.wins.uva.nl/~x/npl/npl.html.

[93] 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.

[94] 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.

[95] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000. Full version of [91]. Available at: http://adam.wins.uva.nl/~x/cale/cale.html.

[96] M.P.A. Sellink and C. Verhoef. Scaffolding for software renovation. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 161–172. IEEE Computer Society Press, March 2000. Available via http://adam.wins.uva.nl/~x/scaf/scaf.html.

[97] R. Sennett. *The Corrosion of Character – The Personal Consequences of Work in the New Capitalism*. W.W. Norton & Company, 1998.

[98] H.M. Sneed and E. Nyary. Downsizing large application programs. *Journal of Software Maintenance*, 6(5):235–247, 1994.

[99] P. Tate. The Big Spenders. *Information Strategy*, pages 30–37, 1999. Retrieved via http://www.infostrategy.com/current/top100.html.

[100] A.A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 2000. To Appear. Available at http://adam.wins.uva.nl/~x/cnv/cnv.html.

[101] J.W. Toigo. *Disaster Recovery Planning – Strategies for Protecting Critical Information Assets*. Prentice Hall, 2000.

[102] M. Tomita. *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.

[103] W.L. Trainor. Software: From Satan to saviour. In *Proceedings of the National Aerospace and Electronics Conference*, 1973.

[104] C. Verhoef. Towards Automated Modification of Legacy Assets. *Annals of Software Engineering*, 9:315–336, March 2000. Available at http://adam.wins.uva.nl/~x/ase/ase.html.

[105] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. Available at http://www.wins.uva.nl/pub/programming-research/reports/1997/P9707.ps.

[106] E. Visser, J. Scheerder, and M. van den Brand. Scannerless Generalized-LR Parsing, 2000. Work in Progress.

[107] H.R. Walters. *On Equal Terms — Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[108] G.M. Weinberg. *Understanding the Professional Programmer*. Dorset House, 1988.