Legacy to the Extreme

Arie van Deursen

Tobias Kuipers

Leon Moonen
http://www.cwi.nl/~{arie,kuipers,leon}/
{arie,kuipers,leon}@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
We explore the differences between developing a system using extreme programming techniques, and maintaining a legacy system. We investigate whether applying extreme programming techniques to legacy maintenance is useful and feasible.

1998 ACM Computing Classification System: D.2.7, D.2.3, K.6.3
Keywords and Phrases: Software renovation, program understanding, reverse engineering, legacy systems, extreme programming
Note: This work was carried out in department SEN1: Interactive Software Development and Renovation
Note: This work appeared in the Proceedings of the first international conference on Extreme Porgramming (XP2000)

1. Introduction

One of the key elements of *extreme programming* (XP) is *design for today*, so that the system is equally prepared to *go any direction tomorrow*. As Beck argues, XP gets away with this minimalist approach because it exploits the advances in software engineering technology, such as relational databases, modular programming, and information hiding, which all help to reduce the cost of changing software [Bec99]. The result of this is that the software developer does not need to worry about future changes: the change-cost curve is no longer exponential, but linear. Making changes easily is further supported by XP in various ways:

- Releases are small and frequent, keeping changes small as well;
- The code gets refactored every release, keeping it small and adaptable;
- Testing is at the heart of XP, ensuring that refactored code behaves as it should.

The assumption that the system under construction is easily modifiable rules out an overwhelming amount of existing software: the so-called *legacy systems*, which by definition *resist change* [BS95]. Such systems are written using technology such as Cobol, IMS or PL/I, which does not permit easy modification. (As an example, we have encountered a 130,000 lines of code Cobol system containing 13,000 go-to statements.) Moreover, their internal structure has degraded after repeated maintenance, resulting in systems consisting of duplicated (but slightly modified) code, dead code, support for obsolete features, and so on. The extreme solution that comes to mind is to throw such systems away — unfortunately, it takes time to construct the new system, during which the legacy system will have to be maintained and modified.

Now what if an extreme programmer were to maintain such a legacy system? (Which means he was either forced to do so or seduced by an extreme salary). Would he drop all XP practices because the legacy system resists change? Of course not. He would write test cases for the programs he has to modify, run the tests before modification, refactor his code after modification, argue for small releases, ask for end-user stories, and so on, practices that are all at the heart of XP.

In this paper, we explore the relationship between legacy systems and extreme programming. We explain how the use of (reverse engineering) tools can help to reduce the cost of change in a legacy setting, and illustrate the use of these tools. Subsequently, we discuss how and which XP practices can be incorporated into the maintenance of legacy software systems, and we analyze how and why the positive effects for regular and legacy XP projects are different. We conclude with an episode in which a pair of XP programmers face the task of changing hostile Cobol code (examples included), and are able to do so thanks to their tools and bag of XP practices.

2. Tools Make It Possible

Radically transforming a program while preserving its semantics (i.e., *refactoring*) is not hard . . . as long as you know what those semantics were in the first place. Extreme programming feats of this kind (think of the change in the transaction interface described by [Bec99, Chapter 5]) are only possible when you have every detail of the system in your head, or if you have the tools to get to these details quickly and accurately.

This detailed information is provided by modern development systems. They provide all sorts of development-time and run-time information which lets you verify hunches within seconds. Some people, however, are stuck on a mainframe and have to control their mix of JCL, Cobol and others without even the help of a decent *grep* tool. These people usually get by because part of the team has been working on the system for years (ever since that mainframe was carried into the building). New people are introduced to the system on a need-to-know basis.

More and more often, these systems get “outsourced”, (the development team is sold off to another company, and the maintenance of the system is then hired from this new company). After such an outsourcing the original development/maintenance team usually falls apart, and knowledge of the system is lost. And it is still running on that same mainframe, without *grep*.

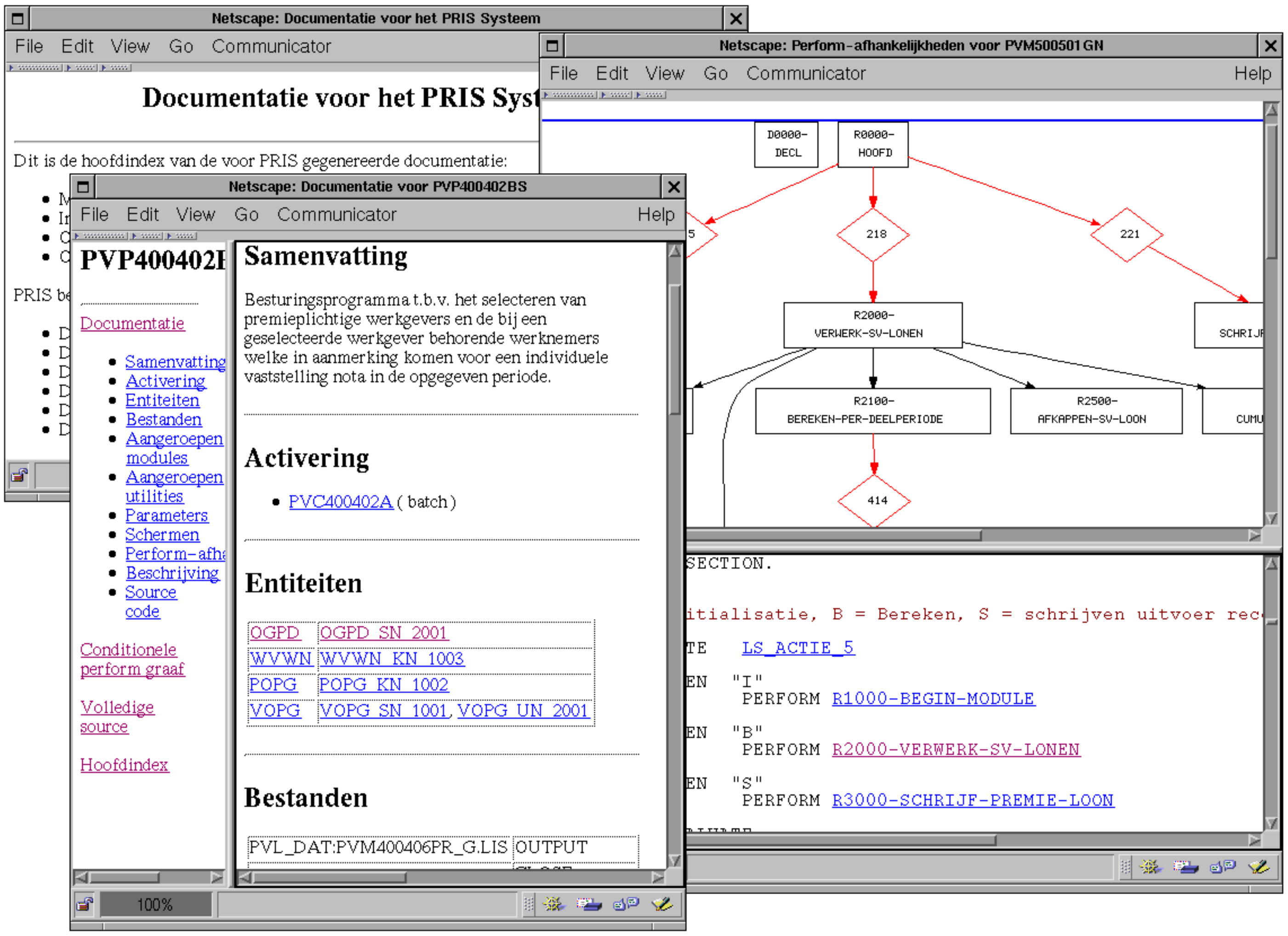
Consequently, maintenance on these systems will be of the break-down variety. Only when things get really bad, someone will don his survival suit and venture inside the source code of the system, hoping to fix the worst of the problems. This is the mode most administrative systems in the world are in.

2.1 The SUE System Understanding Environment

We have been developing a tool set over the last few years which integrates a number of results from the areas of reverse engineering and compiler construction. The tool set is the System Understanding Environment or SUE. It consists of a number of loosely coupled components. One of the components is DocGen [DK99a], (so called because of its basic ability to generate documentation from the source code). DocGen generates interactive, hyperlinked documentation about legacy systems. The documentation is interactive in that it combines various views of the system, and different hierarchies, and combines those with a code browser (see Figure 1 for an example session). DocGen shows call graphs for the whole system, but also per program. It shows database access, and can visualize data dependencies between different programs. Here, we try to provide the programmer with as much information as we can possibly get from the source. (One of the problems is that the source may be written in a vendor specific dialect of a more conventional language, of which no definition is published.)

We augment the basic DocGen code browsing facility with TypeExplorer [DM99], a system which infers types for variables in an untyped language (typically Cobol), and lets the programmer browse the code using those types. TypeExplorer can be used, for instance, to aid in impact analysis sessions. When the requirements of a financial system change from “make sure all amounts are British Pounds” to “make sure all amounts are Euros” this will inevitably have an impact on all data (both variables in the code, and data in databases) which are of type “amount”. Because TypeExplorer can come up with a list of all variables that are in the same *type equivalence class*, the programmer only has to identify a single variable which deals with amounts to identify all variables in the same type equivalence class (and therefor also dealing with amounts).

The combination of DocGen and TypeExplorer proves to be a powerful tool for gaining insight into the details of a system. DocGen and TypeExplorer get their information from a repository, which is filled by a


Netscape: Documentatie voor het PRIS Systeem
File Edit View Go Communicator
Netscape: Perform-afhankelijkheden voor PVM500501GN
File Edit View Go Communicator Help
D0000-DECL
R0000-HOOFD
218
221
R2000-VERWERK-SV-LONEN
SCHRIJF
R2100-BEREKEN-PER-DEELPERIODE
R2500-AFKAPPEN-SV-LOON
CUMU
414
Netscape: Documentatie voor PVP400402BS
File Edit View Go Communicator Help
100%

Documentatie voor het PRIS Syst

Dit is de hoofdindex van de voor PRIS gegenereerde documentatie:

- M
- In
- C
- C

PRIS be

- D
- D
- D
- D
- D
- D

PVP400402I

Documentatie

- Samenvatting
- Activering
- Entiteiten
- Bestanden
- Aangeroepen modules
- Aangeroepen utilities
- Parameters
- Schermen
- Perform-afh
- Beschrijving
- Source code

Conditionele perform graaf

Volledige source

Hoofdindex

Samenvatting

Besturingsprogramma t.b.v. het selecteren van premieplichtige werkgevers en de bij een geselecteerde werkgever behorende werknemers welke in aanmerking komen voor een individuele vaststelling nota in de opgegeven periode.

Activering

- PVC400402A (batch)

Entiteiten

OGPD	OGPD_SN_2001
WVWN	WVWN_KN_1003
POPG	POPG_KN_1002
VOPG	VOPG_SN_1001, VOPG_UN_2001

Bestanden

PVL_DAT:PVM400406PR_G.LIS	OUTPUT

```
SECTION.

itialisatie, B = Bereken, S = schrijven uitvoer rec

TE   LS_ACTIE_5

EN  "I"
    PERFORM R1000-BEGIN-MODULE

EN  "B"
    PERFORM R2000-VERWERK-SV-LONEN

EN  "S"
    PERFORM R3000-SCHRIJF-PREMIE-LOON
```

Figure 1: An example DocGen session.

combination of parsing (if we have a grammar for the legacy system's language) and lexical analysis. This repository can also be queried directly, using standard SQL queries.

One of the key properties of SUE is that is an *open* environment: external tools can be easily integrated. An example tool is CloneDr, from Semantic Designs, which detects (near) *clones* (or "copy-paste code") in sources, and removes them (by replacing them with a single procedure and a number of calls to that procedure, for instance) [BYM$^+$98]. Code clone removal can be seen as an automated refactoring operation that adheres to the XP principle of *say it once and only once*. Apart from the obvious benefit of reducing the amount of code to be understood, a less expected benefit comes from having to give a name to the newly created procedure. This obviously is a human activity, and helps to focus the thoughts of a maintainer on a particular piece of code, which, since it was duplicated in the original, must be of some use. . .

Newer (less developed) components of SUE are *concept analysis*, which aids in the remodularization of legacy systems [DK99b], and *data flow analysis* which aids in tracking data through the system.

Using SUE, maintenance programmers can learn about the legacy system. They gain confidence about their knowledge by verifying that indeed this database table is only written to and never read from, and therefore it can be removed. They can see that two variables do not occur in the same type equivalence class, so values of the first variable never get passed to the second, and so on. As they use SUE initially to hunt down specific problems, they automatically increase their knowledge of the system, much like they would have when they were brought in during the development of the system.

2.2 More Tools

SUE is the result of research in the area of reverse engineering and program understanding, and builds upon related work in those areas (see [DK99b, DM99] for a detailed comparison). Two tools that are similar in nature to SUE are Rigi [WTMS95] and PBS [BHB99], which also can extract various pieces of data from the sources, and which can present them in various ways. Rigi and PBS have been used more for C then for Cobol, which inolves significant differences (for example, the lack of types and a parameter mechanism in Cobol, and the data-intensive nature of typical Cobol systems). At the commercial side, related Cobol tools are Viasoft's Existing Systems Workbench, Reasoning's InstantQA tools, and McCabe's testing and understanding tools. These tools tend to be closed, making it not only difficult to integrate them with other tools, but also to deal with customer or application-specific issues (think of dialects, coding conventions, I/O utilities, and so on), which occur very frequently in Cobol applications. Out of the Cobol arena, there are various tools to analyze C, C++, or Java code, such as TakeFive's Sniff+ tools.

3. Adopting XP Step by Step

Adopting the XP-approach in a legacy setting can only mean one thing: aim at simplicity. How does this affect us when we decide to introduce XP in an existing legacy maintenance project?

First of all, we have to get a picture of the existing code base. This means that we generate on line, hyper-linked documentation, using the DocGen technology discussed in the previous section. This allows us to browse through the legacy system, and to ask queries about the usage of programs, copybooks, databases, and so on. Moreover, it can be re-generated after any modification, thus ensuring up-to-dateness and consistency.

Next, we have to get into contact with the end-user. We need to collect end-user stories for modification requests. Given the current state of the system, such modification requests are likely to include technical requests as well, such as increasing the stability of the system.

Then we have to divide the modification stories into small iterations. For each modification, we identify the affected code, and estimate the effort needed to implement the request. Observe that such an "impact analysis" can only be done with some understanding of the code, which is provided by the TypeExplorer technology presented in the previous section. As in regular XP, the effort estimates are made by the developers, whereas the prioritization (which story first) is done by the end-user.

We then start working release by release. Each release goes through a series of steps:

- We write test cases for the code that is to be affected by the change request, and run the tests.
- We refactor the affected code so that we can work with it, using the reverse engineering tools described

earlier: this means removing extreme ugliness, duplicated functionality, unnecessary data, copy-paste clones, standardizing the layout, and so on. We then re-run the test cases just constructed, in order to make sure that no damage has been done while refactoring.

- After that, the code is in such a shape that we feel sufficiently confident that we can modify it. If necessary, we adapt the test cases to reflect the modified features, implement the modification request, and re-run the test cases.
- Finally we refactor again, re-test, and re-generate the system documentation.

For XP-programmers these steps will sound extremely familiar. So what are the differences with regular XP?

First of all, the productivity per iteration is lower than in regular XP. This is because (1) there are no test cases, which will have to be added for each refactoring and modification; (2) the code has not been previously refactored; and (3) the programming technology used is inherently more static than, for example, Smalltalk.

Second, the code base itself is not in its simplest state. This means that program understanding, which constitutes the largest part of actually changing a program, will take much more time. Luckily, XP-programmers work in pairs, so that they can help each other in interpreting the code and the results from invoking their tool set. The code not being in its simplest state also means that while studying code (during impact analysis, for example), the pair is likely to identify many potential ways of refactoring, for example when browsing through dead code.

One might consider doing a one-shot, up front refactoring of the entire legacy system to avoid such problems. However, successful refactoring is not an automatic process, requiring (expensive) human intervention. Moreover, there are no test cases available a priori. Last but not least, a total refactoring may be unnecessary anyway if parts of the system do not need modification or are likely to be removed (simplicity requires us not to worry about things we are not going to need).

Another observation is that in normal XP the positive effects of refactoring are accumulated – keeping the system flexible at all times. When applying XP to a legacy system, only after starting to follow XP principles parts of the system get refactored. The accumulated effect of this is much lower than in regular XP.

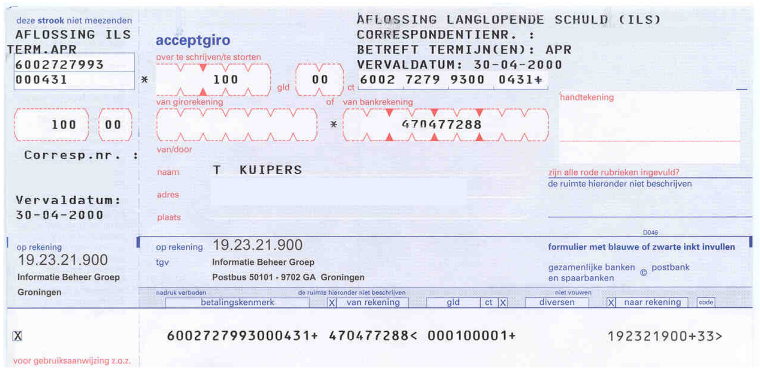
A final question to ask is whether the scenario sketched is realistic. If it is so good, why has it not been done before? Reasons may be a lack of awareness of the XP-opportunities, fear of the overwhelming amount of legacy code leading to paralysis, confusion with the expensive and unrealistic one-shot refactoring approach, or the plain refusal to invest in building test cases or refactoring. The most important reason, however, is that it is only during the last few years that reverse engineering technology has become sufficiently mature to support the XP approach sketched above. Such technology is needed to assist in the understanding needed during planning and modification, and to improve existing code just before and after implementing the modification.

4. XP on Legacy Code

So how would all of this benefit a bunch of maintenance programmers facing a mountain of Cobol code? We will try to answer that question by describing a concrete step-by-step maintenance operation. The example is from the invoicing system of a large administration. All code used in this example is real. We have, however, changed it slightly as to camouflage actual amounts and account numbers.

But first, some culture: In the Netherlands, bill paying is largely automated with companies sending out standardized, optically readable forms called “accept-giro”. Normally, all information including the customers account number, the companies account number and the amount to be paid, is already on these accept-giros, and all the customer has to do is sign them and send them back to accept the mentioned amount being charged off his account. These forms then are read automatically by a central computer operated by all associated Dutch banks, and the appropriate amount is transfered from one account to the next, even between different banks. An example of an accept-giro is in Figure 2.

The task at hand for the programmers is: we have changed banks/account numbers, and all invoices printed from next month should reflect that. That is, all bills should be paid to our new account number.



deze **strook** niet meezenden
AFLOSSING ILS
TERM.APR
6002727993
000431

100 00

Corresp.nr. :

Vervaldatum:
30-04-2000

op rekening
19.23.21.900
Informatie Beheer Groep
Groningen

acceptgiro
over te schrijven/te storten
* 100 gld 00 ct
van girorekening of van bankrekening
* 470477288
van/door
naam T KUIPERS
adres
plaats

AFLOSSING LANGLOPENDE SCHULD (ILS)
CORRESPONDENTIENR. :
BETREFT TERMIJN(EN): APR
VERVALDATUM: 30-04-2000
6002 7279 9300 0431+

handtekening

zijn alle rode rubrieken ingevuld?
de ruimte hieronder niet beschrijven

D046

op rekening 19.23.21.900
tgv Informatie Beheer Groep
Postbus 50101 - 9702 GA Groningen

formulier met blauwe of zwarte inkt invullen

gezamenlijke banken en spaarbanken © postbank

nadruk verboden — de ruimte hieronder niet beschrijven — niet vouwen

betalingskenmerk | [X] van rekening | gld | ct [X] | diversen | [X] naar rekening | code

[X] 6002727993000431+ 470477288< 000100001+ 192321900+33>

voor gebruiksaanwijzing z.o.z.

Figure 2: An example accept-giro

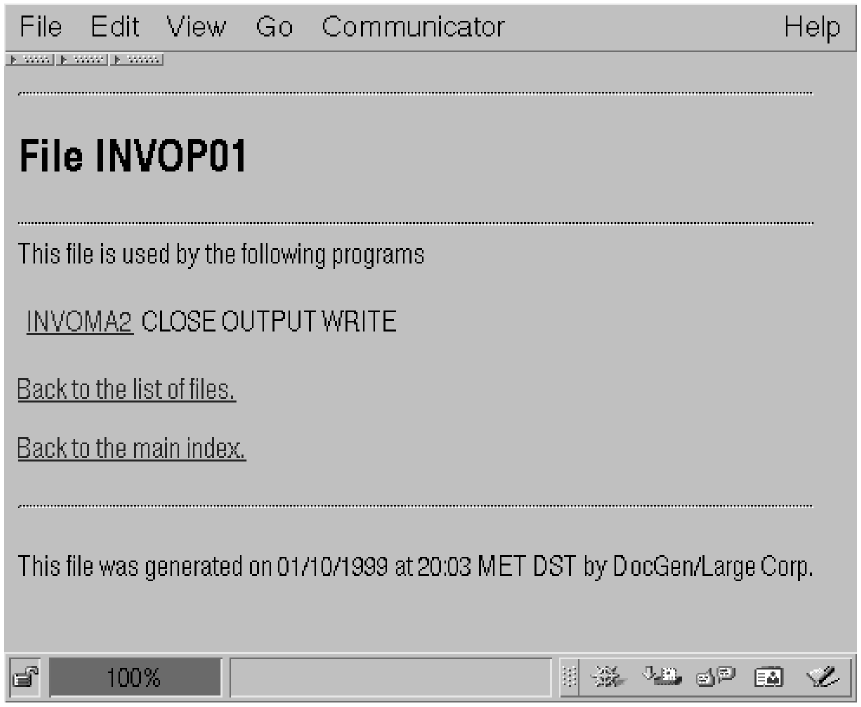
```
xxxx yyyy
zzzz wwww                 100  00     xxxx yyyy zzzz wwww

   100 00

                      TESTNAME T
                      TESTST 12
                      9999 XX TESTCITY

555.12.12              555.12.12
LARGE                  LARGE CORP.
CORP.                  AMSTERDAM


X                xxxxyyyyzzzzwwww+      10000x+5551212+37>
```

Figure 3: INVOP01 from the initial system

Once the team understand the task, they start to work. First they need to find out what file is being printed on the blank forms. They know that their system only creates files, and that these files are then dumped to a specialized high volume printer somewhere. After asking around and looking at the print job descriptions, it turns out that all data for the invoices is in a file called INVOP01. As INVOP01 is the end product of this particular task, the team run the system on their test data and keep a copy of the INVOP01 they have just created. Now they know that when they are finished with the task, the INVOP01 file they generate should be the same as the current INVOP01, apart from the accountnumber. The INVOP01 on the test data can be seen in Figure 3.

Because they have the system analysis tools described earlier, the team can now check what programs do something with the INVOP01 print file. Figure 4 shows all facts that have been derived for INVOP01.

It turns out that the only program operating on INVOP01 is INVOMA2. The information derived from INVOMA2 shows that this program only uses one input file: INVOI01. Executing the system on the test data reveals that INVOI01 does not contain account numbers, rather it contains the names and addresses of customers. If the accountnumber is not read from file, and INVOMA2 does not access any databases, then the accountnumber should be in the code! The team does a find on the string 5551212 in the code, and they find the code as shown in Figure 5.



File INVOP01

This file is used by the following programs

INVOMA2 CLOSE OUTPUT WRITE

Back to the list of files.

Back to the main index.

This file was generated on 01/10/1999 at 20:03 MET DST by DocGen/Large Corp.

Figure 4: All facts derived for file INVOP01

```
  05  P009-BEDRAG-CONTR PIC 9.
  05                    PIC X     VALUE "+".
  05                    PIC X(10) VALUE SPACE.
  05          .
*     07                PIC 9(8)  VALUE 51180761.
*     07                PIC X     VALUE "+".
* 05                    PIC X(4)  VALUE " 37>".
      07                PIC 9(09) VALUE 5551212.
  05                    PIC X(4)  VALUE "+37>".
```

Figure 5: Code found by searching for 5551212

```
xxxx yyyy
zzzz wwww                100  00     xxxx yyyy zzzz wwww

   100 00

                     TESTNAME T
                     TESTST 12
                     9999 XX TESTCITY

555.12.12             555.12.12
LARGE                 LARGE CORP.
CORP.                 AMSTERDAM


X               xxxxyyyyzzzzwwww+       10000x+1215555+37>
```

Figure 6: Test version of INVOP01 after modification

```
03  P008A.
    05              PIC X(19)     VALUE SPACE.
    05              PIC X(09)     VALUE "555.12.12".

03  P008B.
    05              PIC X(09)     VALUE "555.12.12".
    05              PIC X(07)     VALUE SPACE.
    05              PIC X(11)
                    VALUE "LARGE CORP.".
```

Figure 7: Code found by searching for 555.12.12

(Note the old account number commented out in the three lines starting with an asterisk ...) They change the account number to 1212555 (the new account number) and run the system using the test data. Much to their surprise, the test version of INVOP01 comes out like shown in Figure 6.

The last line of the test file shows the correct account number, together with the `+37>` that is also visible in the code. This is the part of the form that will be read optically. However, the part of the form that is meant for humans still shows the *old* account number. The team look at each other, shake their heads, and do a find on `555.12.12` in the source code. What shows up is in Figure 7.

They change the two(!) account numbers and run the test again. Now everything comes out as expected. They write a todo item that this part of the code needs urgent refactoring, or maybe they immediately implement a procedure that “dots” account numbers. Or maybe the system can be left to die and they can spend their time on an XP reimplementation of the whole system.

5. Conclusions

In this paper we have looked at extreme programming from the viewpoint of legacy systems. We observed that the programming environment used for regular XP projects provides capabilities not available for most mainframe-based legacy systems. At the same time we described progress in the area of reverse engineering tools that can be used to overcome these limitations. We used these findings to come up with a way to adopt XP practices during legacy system software maintenance.

References

[Bec99] K. Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 1999.

[BHB99] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *21st International Conference on Software Engineering, ICSE-99*, pages 555–563. ACM, 1999.

[BS95] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, interfaces and the incremental approach*. Morgan Kaufman Publishers, 1995.

[BYM$^+$98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance, ICSM'98*, pages 368–377. IEEE Computer Society Press, 1998.

[DK99a] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[DK99b] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

[DM99] A. van Deursen and L. Moonen. Understanding COBOL systems using types. In *Proceedings 7th Int. Workshop on Program Comprehension, IWPC'99*, pages 74–83. IEEE Computer Society, 1999.

[WTMS95] K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: a case study. *IEEE Software*, 12(1):46–54, 1995.