Techniques for understanding legacy software systems
Kuipers, T.

Citation for published version (APA):
Kuipers, T. (2002). Techniques for understanding legacy software systems

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 1

Introduction

Every software system that is being used needs to be maintained. Software is only finished when it is no longer in use. However, the rate of change may vary wildly from system to system. The number of releases for a new internet application that needs to stay ahead of the competition by adding features every week may be very high. It is typically low for embedded systems software, because of the high cost of retrofitting for instance your television with new software.

There are three main reasons for changing software, the so-called software maintenance categories [Swa76].

Corrective maintenance The first reason software needs to be changed is fault repair (or bug fixing). No software is delivered fault free, and during the lifetime of the software some of those faults will become apparent and need to be fixed.

Perfective maintenance A second reason software needs to be changed is of commercial nature. When a company brings out a software product in a competitive market, then the competition may want to change their product to stay competitive. Software may also need change to reflect the business process of a particular company. The software used inside large services organizations may be modeled upon the structure of the organization. If there is a division “Regional offices” then that division usually has its own systems, as has the division “Foreign offices”. Were the company to reorganize, and these two divisions to be merged into a new division “Offices”, then this has consequences for the two software systems.

Adaptive maintenance Finally, systems may need to change for legal or technical reasons. The system may need to run on different hardware or it may need to be upgraded to run on a different operating system. Legal changes could include the modification of currencies on which the system operates,
or a change in the time during which historical data is stored by the system. After the change the system will perform the same task as before, with hardly any visible change for the user.

An overview of different measurement studies presented in [Pig97] shows that on average about 80% of maintenance cost is in either perfective or adaptive maintenance (or non-corrective maintenance) and about 20% in corrective maintenance.

1.1 Maintaining Legacy Systems

When, for whatever reason, an existing system needs to be modified, it will invariably be seen as a legacy system (at least in part). That is, the system is the legacy of the people who have created it, or have modified it previously. In the ideal situation the original development team will stay on as maintainers of the system throughout its lifetime. This way, everyone who was ever involved in the design and building of the system is available to change it. Unfortunately, this is never the case. Traditional software engineering distinguishes strongly between developers or creators of a system, and maintainers of the same system. This follows the traditional manufacturing model, where a product, for instance a car, is developed and built by a team of people. Once the car is finished and delivered, the maintenance of the car will be performed by a different team.

The process of keeping a software system in tune with the needs of its users is referred to as software evolution. In order to be able to evolve a system, its maintainers need to know, in some detail, the inner workings of that system. Because they did not design nor develop the system, this knowledge does not come to them naturally. In an ideal case, there is ample knowledge about the system available outside of it in the form of design documents, both functional and technical, and actual technical documentation of the source code of the system. What’s more, the original designers and developers of the system may be available to answer questions. Unfortunately, for the original development team there is no need to produce accurate, factual, and concise design documents, because they can build the system without them. Moreover, in a traditional software engineering environment, the designers of a system will produce a design document that cannot be implemented as is. All sorts of details in a design will be deemed “implementation details”, and will be left to the developers to sort out. In order to sort out the implementation details, the developers may need to (and will) stretch the design a little to accommodate for their best solution. This is not a problem as long as the intention of the original design was clear to both designer and developer, because then the chosen solution will be a common one. Unfortunately, the slight change in the design is only documented in the head of the developer who made it. During the development of a non-trivial system there are hundreds, if not thousands of such implementation details to decide about, and each one may warrant their
own slight design change. These can add up to quite major changes which are not
documented anywhere.

Software developers get away with this (as opposed to car developers) because there is no intrinsic need for these documents. Software developers in general build a single software system, which, once it is finished, can be copied an infinite number of times for negligible cost. This single copy is built by a single development team. This is as true for both mass distributed software as it is for single-purpose custom built software. Building a single car in the contemporary car market can never be cost effective. The volume production of cars is what keeps a car manufacturer going, so the need for documents that trace the exact steps to create a car is paramount. Not only do all cars need to be built from scratch, there is not a single development team. A single car model may be built on several sites distributed all over the world, and at each site they may be built by several teams.

1.1.1 Software Engineering for Maintenance

The previous paragraph should have started with the sentence “software developers get away with this temporarily”, because in most of the cases software developers do not get away with it over time. Unlike cars, the finished software product can be changed to become a slightly changed, or even radically different software product. As described above, these changes are almost impossible to make without an understanding of the inner workings of the system, and may still be very hard with such an understanding.

A number of methods have been proposed and applied in the past to allow for people new to the system to understand it in the amount of detail they need to do their work. These people have been called “Software Immigrants” in the literature [SH98].

The source code of a software system can be seen as a document that describes exactly what the system does, by definition. Unfortunately, this is done in such enormous detail that it is hard for any individual to keep an overview of the whole system. In order to maintain an overview, and help introduce “software immigrants” to the system, some other, less detailed, source of information is needed. The methods described below all propose a different way to create, and maintain that other source of information about the system.

Traditional Software Engineering The traditional “waterfall” software engineering model starts from the premonition that software is only developed once, and that, once it is developed, it will enter into a maintenance state. When the software enters from the development stage into the maintenance stage, a number of documents should be delivered as well. Different methodologies here require different documents, but in general there should be a functional design document, which describes the overall functionality of the system, there should be a technical
design document, which describes the overall technical layout of the system, there should be in-depth technical documentation on a per-module, or per-program level, and the source code itself could be required to adhere to certain coding standards and contain information about the code.

Evidence suggests that the average software system designed and built following this method spends about 70% of its time and effort and cost in the maintenance stage [Pig97], and in that stage the system may be changed quite drastically. However, after 5 years of maintenance, only the documents from the date of delivery are available, which tend to be horribly outdated, particularly the technical documentation.

More cyclical methodologies have been developed, to iterate over the design and development phase. Errors discovered during the development phase can be reset in the following design refinement phase. Unfortunately, these methodologies, for example Boehm's Spiral Model [Boe88], also end with the implementation phase, and do not consider themselves with maintenance at all.

What is apparent in these models is that a strict dichotomy exists between the actual software system that is running on a computer (the combined source code) and the documents available about the system. That is, the actual software system may or may not do what is in the documentation, and there is no way to guarantee that the two are in any way related.

**Literate Programming** Literate programming [Knu84] has been proposed as a solution for this problem: literate programming requires the developer to have a single document containing both the program text, as well as the documentation that describes the program text. In fact, literate programming in its purest form requires the developer to produce a book describing the problem the software system is going to solve, how this problem can be broken down into smaller problems, and how all these small problems can be solved individually by a very small piece of software, which is so small that it can be printed in the book and understood instantaneously. The advantage of having both documentation and code in the same document is that the engineer who has to modify the code has the documentation right there on his screen. When he needs to modify the system, he should first decide where the error is: Is it in the narrative of the book? Then he needs to fix the narrative, and then the source code. If the narrative is correct, then there somehow is a fault in the code, and he should fix it. This method requires a lot of discipline from developers to fix the narrative before the code. Even though the two are in the same document, the relation between the two is purely circumstantial: the narrative may tell a completely different story from the code.

**Executable Specifications** Another proposed solution comes in the form of executable specifications. Here the design of a system is laid down in a formal system specification document. The actual implementation of the system is then de-
rived from that specification automatically. Because the specification language is, in fact, a (very) high-level programming language, the design, or specification, cannot skip over the already mentioned implementation details. This means that formal specifications of non-trivial problems may contain much more detail than would be written down in a natural language design document. To introduce a new engineer to the system, natural language documents are needed, and the advantage of formal specifications over “regular” programming languages with respect to understandability and maintainability are lost.

**Domain Specific Languages**  
Domain Specific Languages are usually extremely high level languages that borrow parts of their syntax from the vocabulary of a certain problem domain. Having “programmers” who are well-versed in this particular problem domain solves the problem of having software immigrants. Being domain experts, these people are already familiar with the same vocabulary that the domain specific language is built from. Having, or creating a domain specific language is, by definition, only feasible in a situation where the problem domain is extremely well defined. Furthermore, having a domain specific language can only be profitable if a number of systems will be developed in the same problem domain, to warrant the initial investment in the development of the domain specific language itself [DK98].

**Extreme Programming**  
The most extreme solution is provided by the aptly named Extreme Programming (XP) methodology. XP does not distinguish at all between design, testing, development, or maintenance stages. Instead, a team of engineers produces a tiny, but working system in a couple of weeks, and keeps modifying and extending that system into the system that is actually requested by the customer. There are no design documents, or at least no persistent design documents; the design of the system is the source code itself. Because all engineers that are maintaining the system have helped develop and design it, their knowledge about the system comes naturally. Writing documentation or commenting code is actually discouraged, because it will be outdated the moment someone changes the code. Understandability should be achieved by creating readable code. Because all code is written in rotating pairs, there is no one engineer with exclusive knowledge of (part of) the system.

### 1.2 Changing Legacy Systems

Even though there is a lack of documentation and a lack of people with insight into the system, the legacy system still needs to be changed. The changes that can be made to a legacy system can fall into one of two categories. Both types of changes have their particular reasons, and both require a different type of knowledge about
the system to be available for the engineer. Of course the difference between the two types of change is not always clear; the two categories can be seen as the two ends of a change spectrum.

1.2.1 Minor Change

Minor changes in a software system are usually operational in nature: a system terminates on unexpected or illegal input, or it produces the wrong results for a particular input. An engineer is called in to repair the program as soon as possible, to get it up and running again. Large batch processes on mainframe computers typically run at night, with a single engineer standing by to fix things when they go wrong. It is impossible for this one engineer to know very much about all the systems that may be running at any given night. When the system breaks down, the engineer needs access to information about the system very fast, and will not be bothered by the amount of detail the information contains. The first goal of the information available should be to decrease the search space in which the problem can be found. If at a given night 25 systems are running, and each system consists of 200 modules, then having information available that within minutes pinpoints the error in one of the 5000 modules is invaluable. Further information that narrows the error down even further to a particular section of a module may be useful, but if the information takes half an hour to retrieve then its usefulness is rather limited, because an engineer can probably read the whole module in about half an hour.

Minor changes occur where an otherwise working system breaks down. Obviously, this is not a definition but rather a vague description: If a system breaks down on January 1st, 2002, because it cannot process the Euro currency, hardly any software engineer will tell you that fixing this problem would involve only a minor change.

1.2.2 Structural Change

Fixing an error such as being incapable of processing the Euro may not lead to structural change per se: The engineer could just change every occurrence of the guilder or the mark into an occurrence of the Euro. Although this may affect a large part of the system (and typically will, in legacy systems) the actual structure of the program may not have to be changed. If however the engineer decides to fix the problem once and for all, then he may decide to put all occurrences of money in a single part of the system, and have all other parts of the system reference this part if necessary. Then, at the introduction of the global unified currency he will only have to change his one money part, and the rest of the system will be unaffected, thus the problem will be reduced to a minor change.

The process of finding and relocating all references to (in this case) parts of the system that relate to the processing of a particular currency is deemed structural
change. The information needed to perform such a change is much more detailed and harder to obtain than the information needed for minor changes. For example, in the previous section, it didn’t matter very much if the information available only lead to the first occurrence of the problem: the aim of the engineer was to get the system running again. The second occurrence of the same problem may not appear that night, or only hours later: in the mean time, valuable computations might have been performed that may not have been performed if the engineer spent his time looking for other occurrences of the same problem.

When performing a structural change, the engineer would definitely have to find any and all references to currencies: if he would miss a single database retrieval operation, then guilder amounts may be subtracted from euro amounts or vice versa with all commercial and possibly legal consequences that may have.

When performing structural changes, time is usually less of a constraint than with minor changes. Accuracy is instead of extreme importance. Especially considering that the above example is one of the less complex forms of structural change.

More complex structural changes would be to change a system consisting of an online and a batch subsystem (for instance a checking account management system) into a client/server system. Here all the parts of the system that have to do with batch processing (typically the transferal of money from one account to the other) and the parts that have to do with online querying (balance retrieval, adding and removal of transfers, etcetera) are separated completely. At a certain moment each day (say 7 PM) the system moves from online mode to batch mode. At that time, balances can no longer be requested, and transfers can no longer be added or removed. Changing such a software system to (for instance) a client/server system that does not have harsh restrictions on which action can be performed at what time is a very complex structural change.

A final example of complex structural change of a software system is a change of implementation language. Legacy systems are typically implemented in languages that lack certain features that make software written in modern languages better understandable and maintainable. A form of modernization can be to re-implement the system in a different language.

### 1.3 Research Questions

This thesis does not deal with producing better software. It does not try to help people deliver fault-free software, or software that does not have to be changed. Instead it tries to help in making existing systems more flexible by supporting the engineers who need to change them with the appropriate technology. The concrete research questions that this thesis tries to answer are:

- How can we reduce the search space when searching for a particular artifact
in a legacy software system?

- How can we obtain sufficiently detailed information about a legacy software system to perform a structural change (semi-)automatically?

- How can we alter the software engineering process such that we no longer produce legacy systems?

**Reduce Search Space** The process of relating a certain externally perceivable feature of a software system to an internal artifact (the traceability of a feature) is one of the most frequently occurring activities in software maintenance. When a system contains millions of lines of code, automatically reducing the search space for a particular artifact can save hours if not days per search. Ideally, the search space is decreased to exactly those parts of the system that do something with the externally perceived feature.

**Information for Structural Change** Getting detailed information out of a legacy system *per se* is not necessarily hard. The problem is that the amount of detailed information may be (and usually is) so enormous, that it is useless unless processed (semi-)automatically. That is, the raw data extracted from a legacy system should be processed somehow to diminish in volume, but to increase in usefulness. An example could be that retrieving all database operations from a legacy system returns an enormous amount of data. Cross-referencing all the operations with the database schema, and filtering out only the operations on numbers gives back less data. If it would be possible somehow to filter the operation out even further to get only operations on numbers that represent a certain currency, then this would be the exact data needed for a currency conversion project.

**Altering the Software Engineering Process** As was described in Section 1.1.1 traditional software engineering hardly concerns itself with maintenance explicitly. If it does, it is seen as a completely separate activity from software development, to be performed by different people than the original developers. There are no practical steps in the software engineering process to prevent a system from becoming a legacy system. In fact, some people state that "programming leads to legacy systems" [Vis97b]. From that thesis follows that one way to prevent your system from becoming a legacy system is to never program it in the first place. However, there may be a slightly less drastic modification that can be made in the software engineering process to get rid of legacy systems.
1.4 Reader’s Roadmap

As stated earlier, there is a sliding scale from performing a minor change on a software system to performing a structural change. This thesis concerns itself with these two types of changes and the information required to perform them.

The first chapters, “Rapid System Understanding” (Chapter 2) and “Building Documentation Generators” (Chapter 3) deal with retrieving and presenting information from a legacy software system in such a way that a maintenance engineer can retrieve a particular artifact in the system with maximum speed.

The next chapter, “Identifying Objects with Cluster and Concept Analysis” (Chapter 4) examines what data from a legacy system is required to derive a so-called object-oriented design from that system. In an object-oriented design, procedures and the data they operate on are logically grouped. Chapter 4 examines two methods of automatically performing such a grouping and examines the pros and cons of each method.

One of the methods examined in Chapter 4 is used in the next chapter (“Types and Concept Analysis for Legacy Systems”, Chapter 5) to group data in a different way. Here, data is grouped based on the way it is used to calculate new data. This way, we can get answers to questions like: what pieces of data in this system represent a monetary value, or what pieces of data represent a date, or an account number.

Where Chapters 4 and 5 mainly deal with the presentation and filtering of already extracted data from a system, Chapter 6 (“Object-Oriented Tree Traversal with JJForester”) shows a technique for retrieving the actual detailed elements from the source code in an elegant and easy way.

Chapter 7 sketches a possible scenario of what could happen when a team of software maintainers tries to adopt the software engineering methodology called Extreme Programming.

The final chapter (Chapter 8) draws conclusions and examines how the research questions posed here have been answered.
Sources of the Chapters

Chapter 2, “Rapid System Understanding”, was co-authored by Arie van Deursen. It was published earlier as:


Chapter 3, “Building Documentation Generators”, was co-authored by Arie van Deursen. It was published earlier as:


Chapter 4, “Identifying Objects with Cluster and Concept Analysis”, was co-authored by Arie van Deursen. It was published earlier as:


Chapter 5, “Types and Concept Analysis for Legacy Systems”, was co-authored by Leon Moonen. It was published earlier as:


Chapter 6, “Object-Oriented Tree Traversal with JJForester”, was co-authored by Joost Visser. It was published earlier as:


Chapter 7 “Legacy to the Extreme”, was co-authored by published earlier in “Extreme Programming Examined”, published by Arie van Deursen and Leon Moonen. It was published earlier as: