# 14

# Software Maintenance

### LEARNING OBJECTIVES

- To know about well-known categories of maintenance tasks and data on their distribution
- To be able to discern major causes of maintenance problems
- To be aware of reverse engineering, its limitations, and tools to support it
- To appreciate different ways in which maintenance activities can be organized
- To understand major differences between development and maintenance and the consequences thereof

**Software maintenance is not limited to the correction of faults. A large part of maintenance deals with accommodating new or changed user requirements and adapting software to a changed environment. It is about evolution, rather than just maintenance. We discuss the various types of maintenance tasks, and how to organize them.**

> *Like living organisms and most natural phenomena, software projects follow a life cycle that starts from emptiness, is followed by rapid growth during infancy, enters a long period of maturity, and then begins a cycle of decay that almost resembles senility.*
> (Jones, 1989)

> *Software, unlike a child, does not grow smarter and more capable; unfortunately, it does seem to grow old and cranky.*
> (Lyons, 1981)

Consider UBank, a multinational bank, a typical large organization that is heavily dependent upon automation for its daily operation. UBank is the result of a number of mergers and takeovers.

UBank has hundreds of offices spread all over the world. It has a number of mainframes at a central site, as well as thousands of workstations and printers connected. It has internet connectivity, all over the world, and strives for $24 * 7$ availability. The workload is an enormous number of transactions per hour. The bank has hundreds of application systems averaging over 100 000 lines of code. Programs are written in a variety of languages, most notably COBOL, various 4GLs and JCL. The systems make use of huge databases implemented under IDMS, INGRES, and so on. Some of the basic information is shared by many systems.

Quite likely, the bank has no complete overview of its application portfolio. Because of the mergers and acquisitions, integration of applications is a big issue. There are many wrappers, bridges, and other temporary means to glue systems together. There are more people involved in maintaining UBank's information systems, than there are people involved in developing new systems for UBank.

There are many organizations like UBank, organizations whose portfolio of information systems is vital for their day-to-day operation. At the same time, these information systems are ageing and it becomes increasingly difficult to keep them 'up and running'. An increasing percentage of the annual budget of these organizations is spent on keeping installed systems functioning properly.

It is estimated that there are more than 100 billion lines of code in production in the world. As much as 80% of it is unstructured, patched, and badly documented. It is a gargantuan task to keep these software systems operational: errors must be corrected, and systems must be adapted to changing environments and user needs. This is what software maintenance is about. Software maintenance is defined as (IEEE610, 1990):

> The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

So software maintenance is, in particular, *not* limited to the correction of latent faults. The distinction between development and maintenance is fuzzy, to say the least. This makes it hard to very bold about percentages and types of maintenance categories. In section 14.1, we revisit the discussion about types of maintenance activities from chapter 1 and provide a more balanced view.

Changes in both the system's environment and user requirements are inevitable. Software models part of reality, and reality changes, whether we like it or not. So the software has to change too. It has to evolve. A large percentage of what we are used to calling maintenance, is actually evolution.

When looking for ways to reduce the maintenance problem, it is worth bearing in mind the classification of maintenance activities given in chapter 1. Possible solutions to be considered include:

- Higher-quality code, better test procedures, better documentation and adherence to standards and conventions may help to save on corrective maintenance;
- By anticipating changes during requirements engineering and design and by taking them into account during realization, future perfective and adaptive maintenance can be realized more easily. In particular, the explicit evaluation of a software architecture with respect to ease of change is to be recommended. Through its inheritance and virtual typing capabilities, the object-oriented development paradigm in particular offers opportunities for isolating parts that are susceptible to changes from those that are less so. Many design patterns are aimed at encapsulating change-prone elements;
- Finer tuning to user needs may lead to savings in perfective maintenance. This may, for example, be achieved through prototyping techniques or a more intensive user participation during the requirements engineering and design phase;
- Less maintenance is needed when less code is written. The sheer length of the source code is the main determinant of total cost, both during initial development and during maintenance. In particular, a 10% change in a module of 200 LOC is more expensive than a 20% change in a module of 100 LOC. The reuse of existing software in particular has a very direct impact on maintenance costs.

These possible actions are all concerned with initial software development. This is not surprising, since the key to better maintainable software is to be found there. All these issues have been discussed at great length in previous chapters.

Better initial development though will not automatically result in lower maintenance costs. Worse, Dekleva (1992) found exactly the opposite. He found that

development projects with analysis and design phases that produce a logical presentation of the system's function incur *higher* maintenance cost than projects that did not produce such a presentation. The explanation is that the users eventually learn what can reasonably be asked during maintenance. If they know a structured approach has been followed, they expect enhancements can be asked for, and will be realized. So they will ask for enhancements. If they know no structured approach has been followed, they expect only the necessary bug fixing is feasible, and maintenance requests will remain moderate. So higher quality may well incur higher maintenance cost.

Maintenance problems are there to stay. Some of these problems are inherent -- systems degrade when they are changed over and over again -- while others are caused by simple facts of life: real development and maintenance activities are carried out in less than perfect ways. The major causes of the resulting maintenance problems are addressed in section 14.2.

This discussion of maintenance problems suggests two approaches to improve the situation. Section 14.3 discusses various ways to rediscover lost facts ('what does this routine accomplish', 'which design underlies a given system', and the like) and restructure existing software systems in order to improve their maintainability.

The second approach, discussed in section 14.5, entails a number of organizational and managerial actions to improve software maintenance.

## 14.1 Maintenance Categories Revisited

Let us recall part of the discussion from chapter 1. Following Lientz and Swanson (1980), we distinguished four types of maintenance activity[1]:

- **Corrective maintenance** deals with the repair of faults found.
- **Adaptive maintenance** deals with adapting software to changes in the environment, such as new hardware or the next release of an operating system. Adaptive maintenance does not lead to changes in the system's functionality.
- **Perfective maintenance** mainly deals with accommodating new or changed user requirements. It concerns functional enhancements to the system. Perfective maintenance also includes activities to increase the system's performance or to enhance its user interface.
- **Preventive maintenance** concerns activities aimed at increasing the system's maintainability, such as updating documentation, adding comments, and improving the modular structure of the system.

[1] The IEEE uses slightly different definitions. In particular, they combine Lientz and Swanson's adaptive and perfective categories, and call the combination adaptive maintenance. The reader should be aware of these different definitions of maintenance categories, especially when interpreting percentages spent on the different categories.

Notice that 'real' maintenance activities -- the correction of faults -- accounts for about 25% of the total maintenance effort only. Half of the maintenance effort concerns changes to accommodate changing user needs, while the remaining 25% largely concerns adapting software to changes in the external environment (see figure 14.1).

Recall also that the total cost of system maintenance is estimated to comprise at least 50% of total life cycle costs. Similar figures hold for the personnel involved. Figure 14.2 gives an estimate of the number of people working in software development compared to software maintenance according to (Jones, 2006).

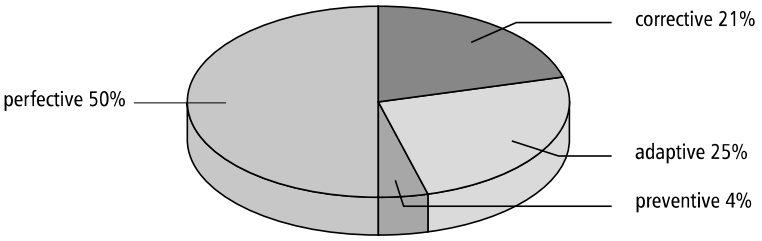


Figure 14.1 Distribution of maintenance activities

| Year | Development | Maintenance | Maintenance percentage |
|---|---|---|---|
| 1975 | 350,000 | 75,000 | 17.65 |
| 1990 | 900,000 | 800,000 | 47.06 |
| 2005 | 775,000 | 2,500,000 | 76.34 |

Figure 14.2 US distribution of developers and maintainers

The data in figure 14.1 are based on (Lientz and Swanson, 1980) and reflect the state of the practice in the 1970s. Later studies have shown that the situation has not changed for the better. Nosek and Palvia (1990) raised the major maintenance issues once again and came to the disturbing conclusion that maintenance problems have remained pretty much the same, notwithstanding advances in structured development methodologies and techniques. Other studies, such as Basili et al. (1996) give roughly the same results. The relative distribution of maintenance activities is about the

same as it was 20 years ago. Systems though have become larger, maintenance staff has grown, there are more systems, and there is a definite trend to an increase in maintenance effort relative to development effort.

Some studies give results that are quite different from the general picture sketched above. Schach et al. (2003), for instance, investigated maintenance effort in three systems and found corrective maintenance percentages of 50% and more, and very low percentages for adaptive maintenance. There is no convincing argument for these differences.

In many organizations, the definition of software maintenance does not follow the IEEE definition. Some organizations for instance define change efforts larger than, say, three months, as development rather than maintenance. This blurs the picture even further. In practice also, people find it difficult to distinguish between adaptive and perfective maintenance. What remains then is a distinction between correcting fault and 'the rest'. The latter mostly caters for 75% or more of the maintenance effort.

The maintenance categories from (Lientz and Swanson, 1980) refer to the software only. Keeping software alive incurs other costs too, though. For instance, new users must be trained, and the helpdesk needs to be staffed. Nowadays, it is not uncommon that these supporting costs account for around 25% of the cost of keeping a system deployed.

Another way to look at the distribution of maintenance cost and prevailing types of maintenance tasks is along the time dimension. We may distinguish the following maintenance life cycle stages:

- During the **introductory** stage of a new system, most of the effort is spent on user support. Users have to be trained, and they will often contact the helpdesk for clarification.

- Next follows a **growth** stage in which more and more users start to explore the system's possibilities. As far as maintenance is concerned, emphasis during this stage is on correcting faults.

- The growth stage is followed by a period of **maturity**. Users know what the system can and cannot do, and ask for enhancements.

- Finally, a period of **decline** sets in. Technology replacement, such as another platform or user interface kit, constitutes a major category of maintenance tasks during this period.

Successful maintenance requires knowledge of the application. After initial delivery, this knowledge usually is available. Either knowledge of the application is explicitly transferred to the maintenance organization via documentation, training, and the like, or the developers have become maintainers of the application they just developed. But over time, this knowledge vaporizes, and at some point in time, it has become scant. This point in time more or less coincides with the transition from the mature stage to the declining stage. The "if it ain't broken don't fix it" adagium then becomes

prevalent. Bennett and Rajlich (2000) use the terms **evolution stage** and **servicing stage** to distinguish between the period in which the system can successfully evolve and the subsequent period where this is no longer the case. In the latter stage, changes become tactical. For example, necessary changes are realized through patches and wrappers.

Finally, we may consider the distribution of effort over the activities of a single maintenance task. For the code-related tasks, the main activities are:

- **Isolation** The first activity is concerned with determining the part of the system (modules, classes) that needs to be changed.
- **Modification** This concerns the actual changes. One or more components are adapted to accommodate the change.
- **Testing** After the changes have been made, the system has to be tested anew (regression testing).

As a rule of thumb, isolation takes about 40% of effort, while the other two activities each take about 30%. This distribution is not the same for all types of maintenance. For corrective maintenance, isolation often takes an even larger share, while for adaptive maintenance tasks, the actual modification takes longer. During corrective maintenance, the fault that caused the failure has to be found, and this may take a lot of effort. Once it is found, the actual modification often is fairly small. For adaptive maintenance tasks, the reverse holds.

## 14.2 Major Causes of Maintenance Problems

The following story reveals many of the problems that befall a typical software maintenance organization. It is based on an anecdote once told by David Parnas and concerns the re-engineering of software for fighter planes.

The plane in question has two altimeters. The onboard software tries to read either meter and displays the result. The software for doing so is depicted in figure 14.3. The code is unstructured and does not contain any comments. With a little effort though its functioning can be discerned. A structured version of the same code is given in figure 14.4. What puzzles us is the meaning of the default value 3000. Why on earth does the system display the value 3000 (which, at first sight is not very peculiar) when both altimeters cannot be read?

The rationale for the default value could not be discerned from the (scarce or nonexistent) documentation. Eventually, the programmer who had written this code was traced. He said that, when writing this piece of code, he did not know what to display in case both altimeters were unreadable. So he asked one of the fighter pilots what their average flying altitude was. The pilot made a back-of-the-envelope calculation and came up with the above value: the average flying altitude is 3000 feet. Hence this fragment.

```
IF not-read1 (V1) GOTO DEF1;
display (V1);
GOTO C;
DEF1: IF not-read2 (V2) GOTO DEF2;
display (V2);
GOTO C;
DEF2: display (3000);
C:
```

Figure 14.3 Unstructured code to read altimeters

```
if read-meter1 (V1) then display (V1) else
if read-meter2 (V2) then display (V2) else
     display (3000)
endif;
```

Figure 14.4 Structured code to read altimeters

The person reengineering the software rightfully thought that this was not the proper way to react to malfunctioning hardware. Fighter planes either fly at a very high altitude or very close to the ground. They don't fly in between. So he contacted the officials in charge and asked permission to display a clear warning message instead, such as a flashing 'PULL UP'.

The permission to change the value displayed was denied. Generations of fighter pilots were by now trained to react appropriately to the current default message. Their training manual even stated a warning phrase like 'If the altimeter reader displays the value 3000 for more than a second, PULL UP'.

This story can't be true. Or can it? It does illustrate some of the major causes of maintenance problems:

- unstructured code,
- maintenance programmers having insufficient knowledge of the system or application domain. Understanding the rationale behind code is one of the most serious problems maintainers face.
- documentation being absent, out of date, or at best insufficient.
- software maintenance has a bad image (this is not illustrated by the anecdote but is definitely a maintenance problem).

Unstructured code is used here as a generic term for systems that are badly designed or coded. It manifests itself in a variety of ways: the use of gotos, long procedures, poor and inconsistent naming, high module complexity, weak cohesion and strong coupling, unreachable code, deeply-nested if statements, and so on.

Even if systems were originally designed and built well, they may have become harder to maintain in the course of time. Much software that is to be maintained was developed in the pre-structured programming era. Parts of it may still be written in assembly language. It was designed and written for machines with limited processing and memory capacities. It may have been moved to different hardware or software platforms more than once without its basic structure having changed.

This is not the whole story either. The bad structure of many present-day systems at both the design and code level is not solely caused by their age. As a result of their studies of the dynamics of software systems, Lehman and Belady formulated a series of Laws of Software Evolution (see also chapter 3). The ones that bear most on software maintenance are:

**Law of continuing change** A system that is being used undergoes continuing change, until it is judged more cost-effective to restructure the system or replace it by a completely new version.

**Law of increasing complexity** A program that is changed, becomes less and less structured (the entropy increases) and thus becomes more complex. One has to invest extra effort in order to avoid increasing complexity.

Large software systems tend to stay in production for a long time. After being put into production, enhancements are inevitable. As a consequence of the implementation of these enhancements, the entropy of software systems increases over time. The initial structure degrades and complexity increases. This in turn complicates future changes to the system. Such software systems show signs of arthritis. Preventive maintenance may delay the onset of entropy but, usually, only a limited amount of preventive maintenance is carried out.

Eventually, systems cannot be properly maintained any more. In practice, it is often impossible to completely replace old systems by new ones. Developing completely new systems from scratch is either too expensive, or they will contain too many residual errors to start with, or it is impossible to re-articulate the original requirements. Usually, a combination of these factors applies. Increasing attention is therefore given to ways to 'rejuvenate' or 'recycle' existing software systems, ways to create structured versions of existing operational systems in order that they become easier to maintain.

Entropy is not only caused by maintenance. In agile methods, such as XP, it is an accepted intermediate stage. These methods have an explicit step to improve the code. This is known as **refactoring**. Refactoring is based on identifying 'bad smells' and rework the code to improve its design (see also section 14.3.1).

At a low level the code improvement process can be supported by tools such as code restructurers and reformatters. To get higher-level abstractions generally requires human guidance and a sufficient understanding of the system.

This leads us to the second maintenance problem: the scant knowledge maintenance programmers have of the system or application domain. Note that the lack of application domain knowledge pertains to software development in general (Curtis et al., 1988). The situation with respect to software maintenance is aggravated by the fact that there are usually scarce sources that can be used to build such an understanding. In many cases, the source code is the only reliable source. A major issue in software maintenance then is to gain a sufficient understanding of a system from its source code. The more spaghetti-like this code is, the less easy it becomes to disentangle it. An insufficient understanding results in changes that may have unforeseen ripple effects which in turn incurs further maintenance tasks.

Maintenance is also hampered if documentation is absent, insufficient, or out-of-date. Experienced programmers have learnt to distrust documentation: a disappointing observation in itself, albeit realistic. During initial development, documentation often comes off badly because of deadlines and other time constraints. Maintenance itself often occurs in a 'quick-fix' mode whereby the code is patched to accommodate changes. Technical documentation and other higher-level descriptions of the software then do not get updated. Maintenance programmers having to deal with these systems have become part historian, part detective, and part clairvoyant (Corby, 1989).

Careful working procedures and management attention could prevent such a situation from occurring. But even then we are not sure that the right type of documentation will result. Two issues deserve our attention in this respect:

- A design rationale is often missing. Programmers and designers tend to document their final decisions, not the rationale for those decisions and the alternatives rejected. Maintenance programmers have to reconstruct this rationale and may easily make the wrong decisions.

- In trying to comprehend a piece of software, programmers often operate in an opportunistic mode. Based on their programming knowledge, in terms of programming plans and other stereotyped solutions to problems, they hypothesize a reasonable structure. Problems arise if the code does not meet these assumptions.

Finally, the noun 'maintenance' in itself has a negative connotation. Maintaining software is considered a second-rate job. Maintenance work is viewed as unchallenging and unrewarding. Preferably, new and inexperienced programmers are assigned to the maintenance group, possibly under the guidance of an experienced person. The more experienced people are to be found working on initial software development. In the structure of the organization, maintenance personnel ranks lower, both financially and organizationally, than programmers working on the development of new systems.

This tends to affect morale. Maintenance programmers are often not happy with their circumstances and try to change jobs as fast as possible. The high turnover of

maintenance programmers precludes them from becoming sufficiently familiar with the software to be maintained which in turn hampers future maintenance.

It would be far better to have a more positive attitude towards maintenance. Maintaining software is a very difficult job. The job content of a maintenance programmer is more demanding than the job content of a development programmer. The programs are usually written by other people, people who can often not be consulted because they have left the firm or are entangled in the development of new systems. When making changes in an existing system, one is bound by the very structure of that system. There is generally a strong time pressure on maintenance personnel. Maintenance work requires more skills and knowledge than development does. It is simply more difficult (Chapin, 1987).

The maintenance group is of vital importance. It is they who keep things going. It is their job to ensure that the software keeps pace with the ever-changing reality. Compared to software development, software maintenance has more impact on the well-being of an organization.

## 14.3 Reverse Engineering and Refactoring

> *What we're doing now with reverse engineering is Archeology. We're trying to gain an understanding of existing systems by examining ancient artifacts and piecing together the software equivalent of broken clay pots. Then we look to restructuring and reengineering to save the clay.*
> (Chikofsky, 1990)

It is fashionable in our trade to coin new terms once in a while and offer them as a panacea to the software crisis. One of the magical terms is **reverse engineering**. It comes under different guises and means altogether different things to different people. In the discussion below we will use the terminology from (Chikofsky and Cross II, 1990). The different terms are illustrated in figure 14.5.

Chikofsky defines reverse engineering as 'the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.'

According to this definition, reverse engineering only concerns inspection of a system. Adaptations of a system and any form of restructuring, such as changing gotos into structured control constructs, do not fall within the strict definition of reverse engineering. Reverse engineering is akin to the reconstruction of a lost blueprint. Retiling the bathroom or the addition of a new bedroom is an altogether different affair. If this distinction is not carefully made, the meaning of the term reverse engineering dilutes too much and it reduces to a fancy synonym for maintenance.

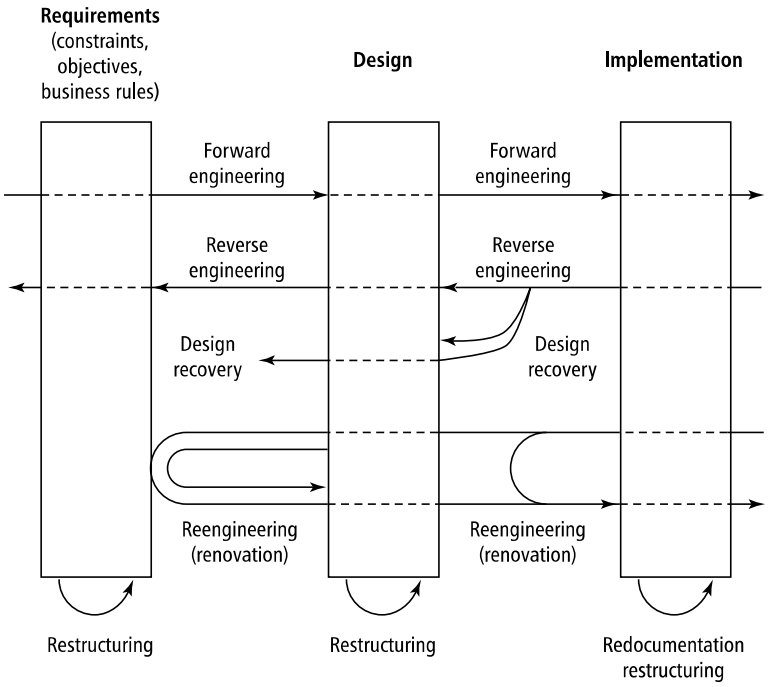


Figure 14.5 Reverse engineering and related notions (*Source: E.J. Chikofsky & J.H. Cross II, Reverse engineering and design recovery*, IEEE Software **7**, *1 (1990) pp 13--18, 1990 IEEE.*)

The above definition still leaves open the question whether or not the resulting description is at a higher level of abstraction. To emphasize the distinction, Chikofsky uses the notions of **design recovery** and **redocumentation**, respectively.

Redocumentation concerns the derivation of a semantically-equivalent description at the same level of abstraction. Examples of redocumentation are the transformation of a badly-indented program into one having a neat lay-out or the construction of a set of flowcharts for a given program.

Design recovery concerns the derivation of a semantically-equivalent description at a higher level of abstraction. Some people limit the term reverse engineering to efforts that result in higher level descriptions and thus equate the term to what we have termed design recovery.

Note that a 100% functional equivalence is difficult to achieve in reverse engineering. The person carrying out the process (the reengineer) may encounter errors in the original system and may want to correct those. Such errors may be deeply

hidden in the original system and become much more troublesome in the reverse engineered system. The programming language may be incompletely defined and its implementation may depend on certain machine characteristics. Data equivalence may be difficult to achieve because of typing issues, approximations, data conversions, etc. In practice, it seems sensible to solve this issue by agreeing on some acceptance test for the reengineered system, thereby relaxing the 100% functional equivalence requirement.

Obviously, reverse engineering is often done in circumstances where the target system is adapted as well. Two important subclasses are **restructuring** and **reengineering**.

Restructuring concerns the transformation of a system from one representation to another, at the same level of abstraction. The functionality of the system does not change. The transformation of spaghetti-code to structured code is a form of restructuring. The redesign of a system (possibly after a design recovery step) is another example of restructuring. In agile methods, restructuring the code to improve its design is an explicit process step. There, it is known as **refactoring**. Refactoring is discussed in section 14.3.1.

Refactoring is a white-box method, in that it involves inspection of and changes to the code. It is also possible to modernize a system without touching the code. For example, a legacy system may be given a modern user interface. The old, text-based interface is then wrapped to yield, for example, a graphical user interface or a client running in a Web browser. This is a black box method. The code of the old system is not inspected. The input and output are simply redirected to the wrapper. Usability is increased, although the capabilities of the new type of interface are often not fully exploited. A similar black box technique can be used to switch to another database, or integrate systems through intermediate XML documents. A third black box wrapping technique is applied at the level of components, where both business logic and data are wrapped and next accessed through an interface as if it were, say, a JavaBean.

These wrapping techniques do not change the platform on which the software is running. If a platform change is involved in the restructuring effort of a legacy system, this is known as **migration**. Migration to another platform is often done in conjunction with value-adding activities such as a change of interface, or code improvements.

Restructuring is sometimes done in conjunction with efforts to convert existing software into reusable building blocks. Such reclamation efforts may well have higher (indirect) payoffs than the mere savings in maintenance expenditure for the particular system being restructured, especially if the effort concerns a family of similar systems. The latter is often done in combination with domain engineering and the development of a (reusable) architecture or framework.

With reengineering, also called **renovation**, real changes are made to the system. The reverse engineering step is followed by a traditional forward engineering step in which the required changes are incorporated.

Each of the above transformations starts from a given description of the system

to be transformed. In most cases this will be the program code, which may or may not be adequately documented. However, it is also possible to, say, restructure an existing design or to reconstruct a requirements specification for a given design. For these transformations too, the term reverse engineering applies.

Both reverse engineering and restructuring can be done manually, but it is a rather tiresome affair. Quite a number of tools have been developed to support these processes. These tools are discussed in section 14.3.3. There are, however, some inherent limitations as to how much can be achieved automatically. These limitations are discussed in section 14.3.2.

### 14.3.1 Refactoring

The modern name for restructuring is **refactoring**. Refactoring has become popular as one of the practices from XP (see section 3.2.4). Of course, programmers have applied the technique in some form since the beginning of programming.[2]. Quite often, refactoring activities are not explicitly planned, and occur somewhat unnoticed in the daily work of software developers. In XP and other agile methods, they are an explicit method step.

There are both arguments for and against refactoring. The classic engineering rule "if it aint't broken don't fix it" is a compelling argument against refactoring. On the other hand, the second law of software evolution tells us that software becomes increasingly complex over time. So we are forced to apply refactoring to keep the software maintainable. The arguments pro and con are both valid. It depends on the phase the software is in which argument is the decisive one. During the evolution stage, when knowledge about the system is still around, refactoring is a viable option. During the servicing stage, knowledge will have vaporized to some extent, and refactoring then may well introduce more problems than it solves. During that stage, one may decide for example to add a wrapper, and not touch the software anymore.

Refactoring is applied when the structure of the software is of substandard quality. Fowler (1999) used the term **bad smells** to indicate occurrences of substandard code quality. Fowler (1999) lists the following 22 bad smells[3]:

- **Long Method** A method that is too long.
- **Large Class** A class that is too big, in terms of instance variables or methods.
- **Primitive Obsession** The use of primitives, such as numbers, instead of small classes such as Dollar.

[2]My earliest recall of a refactoring activity goes back to 1970. I was at that time fulfilling my compulsory military service. I couldn't properly fire a gun, so I got assigned to the army's computer center as a programmer. A couple of weeks before I got demobilized, I finished my last FORTRAN program. To fill up the time, I decided to improve it a bit: improve the structure, remove redundant goto's, and the like. Configuration control was unheard of at the time and I only retained the last copy of the source code. By the time I left the army, I had introduced some faults, and I could not make the program work anymore. They were probably glad I left, but not this way.

[3]Fowler also gives detailed instructions on how to improve the bad code.

- **Long Parameter List** A parameter list that is too long.
- **Data Clumps** Groups of data items that are often used together, such as the height, width and color of a graphical object. If you remove one of the items, the rest doesn't make sense any more.
- **Switch Statements** The use of type codes instead of polymorphism. This results in case statements all over the code
- **Temporary Field** A class has a variable which is only used in some circumstances.
- **Refused Bequest** A subclass which does not support all of the methods or data it inherits.
- **Alternative Classes with Different Interfaces** Methods that do the same things but have different interfaces. For example, methods DisplayRectangle next to DisplayCircle.
- **Parallel Inheritance Hierarchies** Two class hierarchies exist, and if one of them has to be extended, so has the other.
- **Lazy Class** A class that isn't doing all that much. This might be the result of a previous refactoring operation.
- **Data Class** A class that holds data, but little else.
- **Duplicate Code** According to Fowler (1999), this is "number one in the stink parade".
- **Speculative Generality** Code has been created for which you *anticipate* some future need. But this future is unknown.
- **Message Chains** An object asks for an object from another object, which in turn asks for an object from yet another object, and so on.
- **Middle Man** A class delegates most of its tasks to other classes. Delegation is OK, but too much delegation is not.
- **Feature Envy** A method is more tightly coupled to, i.e. interested in, other classes than the class where it is located.
- **Inappropriate Intimacy** Two classes are coupled too tightly.
- **Divergent Change** The same class needs to be changed for different reasons, e.g. each time a new database is added, and each time the user interface changes.

- **Shotgun Surgery** This is the opposite of the Divergent Change smell: for every change, many classes have to be changed.
- **Incomplete Library Class** A library doesn't offer all the functionality needed for the task at hand.
- **Comments** Comments are bad if they compensate for low-quality code.

This is quite a long list, and somewhat hard to go through manually in each and every refactoring situation. Mäntylä et al. (2003) gives a useful categorization of these bad smells into seven broad categories; see figure 14.6. These categories give handles as to the kind of situation one should look for when refactoring code.

| Category | Bad smells |
|---|---|
| Bloaters | Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps |
| Object-Oriented Abusers | Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces, Parallel Inheritance Hierarchies |
| Change Preventers | Divergent Change, Shotgun Surgery |
| Dispensables | Lazy Class, Data Class, Duplicate Code, Speculative Generality |
| Encapsulators | Message Chains, Middle Man |
| Couplers | Feature Envy, Inapprorpiate Intimacy |
| Others | Incomplete Library Class, Comments |

Figure 14.6 Categories of bad smells

The first category, the **Bloaters** denote situations in which something has grown too large to handle effectively. The Primitive Obsession is placed there, because the functionality to handle the primitives has to be placed in some other class, which may then grow too large. The **Object-Oriented Abusers** denote situations where the possibilities of object orientation are not fully exploited. The **Change Preventers** hinder further evolution of the software. The **Dispensables** represent things that can be removed. The **Encapsulators** deal with data communication. The two smells in this category are opposite: decreasing one will increase the other. The **Couplers** represent situations where coupling is too high. The **Others** category finally contains the smells that do not fit another category.

Bad smells not only occur at the code level. At the design level, the evolution of system through successive releases may provide valuable information about bad

smells. For example, if certain classes often change, or classes get introduced in one version, disappear in the next, and then reappear again, such merits closer inspection.

Fowler (1999) states that "no set of metrics rivals informed human intuition". On the other hand, several of the metrics defined in section 12.1 do relate to a number of the bad smells listed above. For example, a high value for McCabe's cyclomatic complexity could indicate the Switch Statement bad smell, while a high value for the Coupling Between Object Classes (CBO) metric could indicate a Feature Envy bad smell. Metrics thus may augment human intuition in the search for bad smells.

### 14.3.2 Inherent Limitations

> *If you pass an unstructured, unmodular mess through one of these restructuring systems, you end up with at best, a structured, unmodular mess.*
> (Wendel, 1986)

Reverse engineering will mostly not be limited to redocumentation in a narrow sense. We will often be inclined to ask why certain things are being done the way they are done, what the meaning is of a certain code fragment, and the like. We must therefore investigate how programmers go about studying program text. The relevance of these issues shows from results of a study into maintenance activities (Fjelstad and Hamlen, 1979), confirmed by Yu and Chen (2006):

- maintenance programmers study the original program code about one and a half times as long as its documentation;
- maintenance programmers spend as much time reading the code as they do implementing a change.

Insights into the discovery process which takes place during maintenance activities will give us the necessary insight to put various developments regarding reverse engineering and refactoring into perspective.

In forward engineering activities we usually proceed from high-level abstractions to low-level implementations. Information gets lost in the successive steps involved in this process. If we want to reverse the route, this information must be reconstructed. The object we start with, a piece of source code in general, usually offers insufficient clues for a full reconstruction.

The programmer uses various sources of information in his discovery process. For example, if the design documentation is available, that documentation will reveal something about the structure of the system. A characteristic situation in practice is that the source code is the only reliable source of information. So this source code has to be studied in order to discover the underlying abstractions. The question is how the programmer goes about doing this.

Several theories have been developed to describe this comprehension process. Common to these theories is that expert programmers may draw on a vast number of knowledge chunks. These knowledge chunks are called in when software is developed.

Within the realm of programming, it is postulated that experts know of programming plans or beacons. A **programming plan** is a program fragment that corresponds to a stereotypical action. For example, to compute the sum of a series of numbers, a programmer uses the 'running total loop plan'. In this plan, some counter is initialized to zero and incremented with the next value of a series in the body of a loop. A **beacon** is a key feature that typically indicates the presence of a particular structure or operation. Beacons seem to be very diagnostic of program meaning. For example, the kernel idea or central operation in a sorting program is a swap operation. If we are presented with a program that contains a swap operation, our immediate reaction would then be that it concerns some sorting program.

This type of program comprehension process occurs when studying existing software. Meaningful units are isolated from the 'flat' source text. Knowledge from human memory is called in during this process. The more knowledge the reader has about programming or the application domain, the more successful this process will be. The better the source code maps onto knowledge already available to the reader, the more effective this process will be.

During the comprehension process, the reader forms hypotheses and checks these hypotheses with the actual text. Well-structured programs and proper documentation ease this process. If application domain concepts map onto well-delineated program units then the program text will be more easily understood. If the structure of a program shows no relation with the structure of the application domain, or the reader cannot discern this structure, then understanding of the program text is seriously hampered.

As a side remark we note that there are two extreme strategies for studying program text:

- the as-needed strategy, and
- the systematic strategy.

In the as-needed strategy, program text is read from beginning to end like a piece of prose and hypotheses are formulated on the basis of local information. Inexperienced programmers in particular tend to fall back onto this strategy. In the systematic strategy, an overall understanding of the system is formed by a systematic top-down study of the program text. The systematic approach gives a better insight into causal relations between program components.

These causal relations play an important role when implementing changes. So-called delocalized plans, in which conceptually related pieces of code are located in program parts that are physically wide apart, may seriously hamper maintenance activities. Excessive use of inheritance increases the use of delocalized plans. If our understanding is based on local clues only, modifications may easily result in so-called ripple-effects, i.e. changes that are locally correct but lead to new problems at different, unforeseen places. Use of the as-needed strategy increases the probability of ripple effects.

During the comprehension process the programmer uses knowledge that has its origin outside the program text proper. To illustrate this phenomenon, consider the program text from figure 14.7.

```
for i:= 1 to n do
    for j:= 1 to n do
        if A[j, i] then
            for k:= 1 to n do
                if A[i, k] then A[j, k]:= true endif
            enddo
        endif
    enddo
enddo
```

Figure 14.7 Warshall's algorithm to compute the transitive closure of a graph

The program fragment of figure 14.7 manipulates a boolean matrix A. Before this fragment is executed the matrix will have a certain value. The matrix is traversed in a rather complicated way (potentially, each element is visited n times) and once in a while an element of the array is set to true. But what does this fragment *mean*? What does it *do*?

An expert will 'recognize' Warshall's algorithm. Warshall's algorithm computes the transitive closure of a relation (graph). The notions 'transitive closure', 'relation' and 'graph' have a precise meaning within a certain knowledge domain. If you don't know the meaning of these notions, you haven't made any progress in understanding the algorithm either.

At yet another level of abstraction the meaning of this fragment could be described as follows. Suppose we start with a collection of cities. The relation $A$ states, for each pair of cities $i$ and $j$, whether there is a direct rail connection between cities $i$ and $j$. The code fragment of figure 14.7 computes whether there is a connection at all (either direct or indirect) between each pair of cities.

Warshall's algorithm has many applications. If you know the algorithm, you will recognize the fragment reproduced in figure 14.7. If you don't know the algorithm, you will not discover the meaning of this fragment either.

As a second example, consider the code fragment of figure 14.8, adapted from (Biggerstaff, 1989). The fragment will not mean much to you. Procedure and variable names are meaningless. A meaningful interpretation of this fragment is next to impossible.

The same code fragment is given in figure 14.9, though with meaningful names. From that version you may grasp that the routine has something to do with window management. The border of the current window is depicted in a lighter shade while

```
procedure A(var x: w);
    begin b(y, n1);
        b(x, n2);
        m(w[x]);
        y:= x;
        r(p[x])
    end;
```

Figure 14.8 An incomprehensible code fragment

the border of another window gets highlighted. The cursor is positioned in the now highlighted window and the process of that window is restarted. If we add a few comments to the routine, its text becomes fairly easy to interpret. Meaningful names and comments together provide for an informal semantics of this code which suffice for a proper understanding.

This informal semantics goes much further than building local knowledge of the meaning of a component. Developers use naming conventions also to find their way around in a large system. Organizations often prescribe naming conventions precisely for this reason. When design and architecture documentation is not updated, these naming conventions serve as a proxy for that documentation.

```
procedure change_window(var nw: window);
    begin border(current_window, no_highlight);
        border(nw, highlight);
        move_cursor(w[nw]);
        current_window:= nw;
        resume(process[nw])
    end;
```

Figure 14.9 Code fragment with meaningful names

Common to these two examples as well as the altimeter anecdote from section 14.2 is that we need *outside* information for a proper interpretation of the code fragments.

The outside information concerns concepts from a certain knowledge domain or a design rationale that was only present in the head of the programmer.

The window management example is illustrative for yet another reason. Tools manipulate sequences of symbols. In principle, tools do not have knowledge of the (external) meaning of the symbols being manipulated. In particular, a reverse engineering tool has no knowledge of 'windows', 'cursor' and the like. These notions

derive their meaning from the application domain, not from the program text itself. From the tool point of view, the texts of figures 14.8 and 14.9 are equally meaningful.

The above observations have repercussions for the degree to which tools can support the reverse engineering and restructuring process. Such tools cannot turn a badly-designed system into a good one. They cannot infer knowledge from a source text which is not already contained in that text without calling in external knowledge as an aid. In particular, completely automatic design recovery is not feasible. You can't make a pig out of a sausage.

### 14.3.3 Tools

During the reverse engineering process, the programmer builds an understanding of what the software is trying to accomplish and why things are done the way they are done. Several classes of tools may support the task of program understanding:

- Tools to ease perceptual processes involved in program understanding (reformatters). Tools may for example produce a neat lay-out in which nested instructions are indented and blank lines are put between successive methods. More advanced tools print procedure names in a larger font or generate page headers which contain the name of the component, its version number, creation date, and the like.

- Tools to gain insight into the static structure of programs. For example, tools that generate tables of contents and cross-reference listings help to trace the use of program elements. Browsers provide powerful interactive capabilities for inspecting the static structure of programs. Hypertext systems provide mechanisms to extend the traditional flat organization of text by their capabilities for linking non-sequential chunks of information. If system-related information is kept in a hypertext form, this opens up new possibilities for interactive, dynamic inspection of that information. Code analyzers may be used to identify potential trouble spots by computing software complexity metrics, highlighting 'dead code', or indicating questionable coding practices or bad smells. Finally, tools may generate a graphical image of a program text in the form of a control graph or a calling hierarchy. Tools may analyze the surface structure of large systems, e.g. by considering variable names, and cluster parts that seem to be highly related. The result of this clustering provides a first guess at a restructuring for the system.

- Tools that inspect the version history of a system (see also section 14.4). These tools may for example highlight components that have been changed very often, indicating candidates for reengineering. Tools that identify pairs of components that have often changed together but are not logically related may indicate weak spots in the software architecture.

- Tools to gain insight into the dynamic behavior of programs. Next to traditional text-oriented debugging systems there are systems which provide graphical capabilities to monitor program execution, e.g. to animate data structures or the execution flow.

Note that these tools provide support for maintenance tasks in general (alongside tools such as test coverage monitors, which keep track of program paths executed by a given set of test data, and source comparators, which identify changes between program versions). With respect to reverse engineering, the above tools may be classified as redocumentation tools. By far the majority of reverse engineering tools falls into this category.

Tools which result in a description at a higher level of abstraction (design recovery tools) have some inherent limitations, as argued in the previous section. Tools for design recovery need a model of the application domain in which the concepts from that domain are modeled in an explicit way, together with their mutual dependencies and interrelations. Completely automatic design recovery is not feasible for the foreseeable future. Concepts from an application domain usually carry an informal semantics. Tools for design recovery may, in a dialog with the human user, search for patterns, make suggestions, indicate relations between components, etc. Such a tool may be termed a 'maintenance apprentice'.

Many tools exist for restructuring program code. The history of these restructuring tools goes all the way back to the late 1960s. In 1966, Böhm and Jacopini (1966) published a seminal paper in which it was shown that gotos are not necessary for creating programs. The roots of restructuring tools like Recoder (Bush, 1985) can be traced to the constructive proof given in Böhm and Jacopini's paper. Recoder structures the control flow of Cobol programs. There is a wide choice of such Cobol restructuring tools.

Restructuring tools can be very valuable -- a well-structured program is usually easier to read and understand. A study reported by Gibson and Senn (1989) provides evidence that structural differences do affect maintenance performance. Specifically, it was found that eliminating gotos and redundancy appears to decrease both the time required to perform maintenance and the frequency of ripple effects.

Yet, the merit of restructuring tools is limited. They will not transform a flawed design into a good one.

## 14.4 Software Evolution Revisited

Lehman and Belady studied the evolution of software systems and formulated their well-known laws of software evolution. Empirical studies have given general support for these laws.

Apparently, there is quite a bit of regularity in the evolution of software. We can use this insight and try to predict the *future* evolution of a specific system by looking at the *actual* evolution of that system till now. We then base our next action

on information from the past. We may for example decide which components to reengineer by looking at components that changed a lot in the recent past. The assumption then is that components that changed a lot in the recent past, are likely to change in the near future too. Gîrba et al. (2004) used the term *yesterday's weather* to characterize this idea: if we have no further information, we may guess that today's weather will be like yesterday's.

Gîrba and Ducasse (2006) distinguish two types of analysis of evolutionary data: **version-centered analysis** and **history-centered analysis**. In version-centered analysis, differences between successive versions of a system are studied. The results are typically depicted in a figure with time (i.e. successive versions) along one axis and the relevant aspects of the system on another. For example, we may consider the relative size of the different components of a system over time, as illustrated in figure 14.10 (adapted from (Gîrba and Ducasse, 2006)).

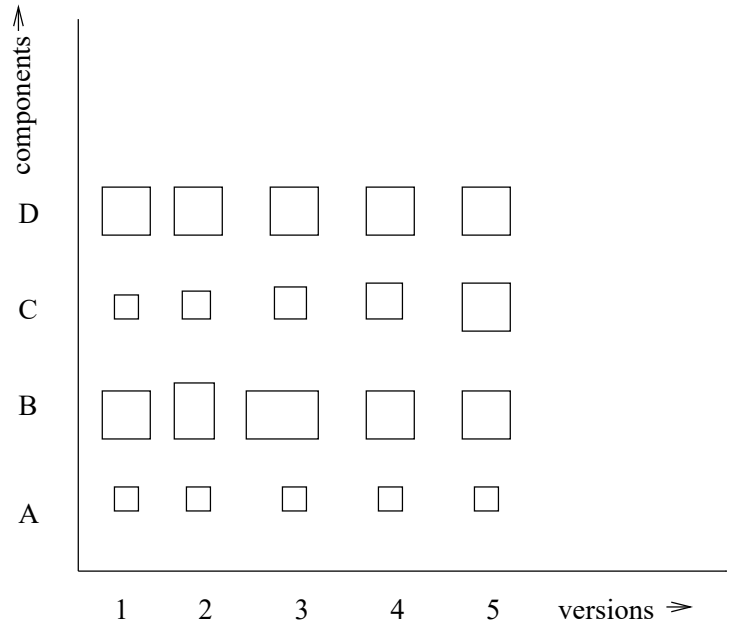


Figure 14.10 Size versus version

Each rectangle in figure 14.10 denotes a component. The width and height of a rectangle each stand for an attribute of that component. The width may for instance denote the number of classes of a component, while the height denotes its number of interfaces. Figure 14.10 tells us that component A is stable and small,

while component D is stable and big. Component C shows a steady growth from one version to the next, and component B exhibits some ripple effects in versions 2 and 3, and is stable since then.

In a history-centered analysis, a particular viewpoint is chosen, and the evolution of a system is depicted with respect to that viewpoint. For example, figure 14.11 shows how often different components are changed together. Each node denotes a component, and the thickness of the edges denotes how often two connected components are changed together (so-called co-changes). A thicker edge between components indicates more frequent co-changes. The latter information may for instance be derived from the versioning database.

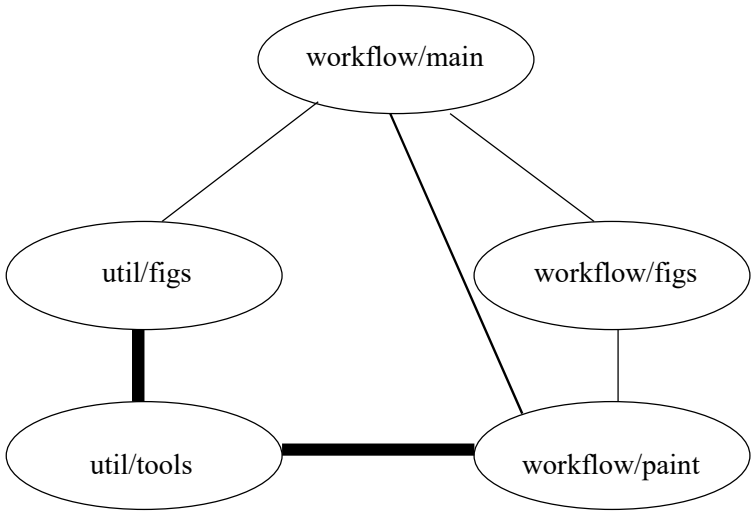


Figure 14.11 Components that change together

From figure 14.11 we learn that components /util/figs and /util/tools are changed together frequently. The same holds for components /util/tools and /workflow/paint. The names of the components suggest that components /util/figs and /util/tools are structurally related, while /util/tools and /workflow/paint are structurally unrelated. From this additional information, we might infer that the interaction between components /util/tools and /workflow/paint deserves our attention. Alternatively, we may label the components with the (external) features they participate in, and the view then shows whether changes frequently affect different features.

A version-centered analysis depicts the version information as-is. It is up to the user to detect any pattern. In figure 14.10, it is the user who has to detect growing or shrinking components; the picture just presents the facts. In a history-centered

analysis, some hypothesis guides the representation, and the patterns are then encoded in the representation, as in figure 14.11.

## 14.5 Organizational and Managerial Issues

The duties of maintenance management are not different from those of other organizational functions, and software development in particular. In chapter 2 we identified five entities that require continuous attention of management:

- time, i.e. progress towards goals;
- information, in particular the integrity of the complete set of documents, including change requests;
- organization of the team, including coordination of activities;
- quality of the product and process;
- money, i.e. cost of the project.

In this section we address these issues from a maintenance perspective. We pay particular attention to issues that pose specific problems and challenges to maintenance. These issues are: the organization of maintenance activities, major differences between development and maintenance, the control of maintenance tasks, and quality assessment.

### 14.5.1 Organization of Maintenance Activities

The primary question to be addressed here is whether or not software maintenance should be assigned to a separate organizational unit. The following discussion is largely based on an insightful study of different forms of systems staff departmentalization presented in (Swanson and Beath, 1990). The authors of this article explore the strengths and weaknesses of three alternative bases for staff departmentalization. The three organizational forms with their focal strengths and weaknesses are listed in figure 14.12. We will sketch the W- and A-Type organizations and discuss the L-type organization with its pros and cons more elaborately.

Traditionally, departmentalization in software development tended to be according to work type (a W-Type scheme). In such a scheme, people analyze user needs, or design systems, or implement them, or test them, etc. Even though they cooperate in a team, each team member has quite separate responsibilities and roles.

In a W-Type scheme, work assignments may originate from both development and maintenance projects. For example, a designer may be involved in the design of a (sub)system in the context of some development project or in the design of a change to an existing system. Likewise, a programmer may implement an algorithm for a new system or realize changes in an operational program.

| | |
|---|---|
| **W-Type** | Departmentalization by work type (analysis versus programming)<br>*Focal strength*: development and specialization of programming knowledge and skills<br>*Focal weakness*: costs of coordination between systems analysts and programmers |
| **A-Type** | Departmentalization by application domain (application group A versus application group B)<br>*Focal strength*: development and specialization of application knowledge<br>*Focal weakness*: costs of coordination and integration among application groups |
| **L-Type** | Departmentalization by life-cycle phase (development versus maintenance)<br>*Focal strength*: development and specialization of service orientation and maintenance skills<br>*Focal weakness*: costs of coordination between development and maintenance units |

Figure 14.12 Trade-offs between alternative organizational forms (*Source: E.B. Swanson & C.M. Beath, Departmentalization in software development and maintenance*, Communications of the ACM **33**, *6 (1990) pp 658-667. Reproduced by permission of the Association for Computing Machinery, Inc.*)

Note that the development of new systems does not occur in a vacuum. Designers of new systems will reuse existing designs and must take into account constraints imposed by existing systems. Programmers involved in development projects have to deal with interfaces to existing software, existing databases, etc. In the W-Type scheme, the distinction between development and maintenance work is primarily a distinction between different *origins* of the work assignment.

A second form of departmentalization is one according to application areas, the A-Type scheme. Nowadays, computerized applications have extended to almost all corners of the enterprise. Systems have become more diversified. Application domain expertise has become increasingly important for successful implementation of information systems. Deep knowledge of an application domain is a valuable but scarce resource. Nurturing of this expertise amongst staff is one way to increase quality and productivity in both development and maintenance. In larger organizations, we may therefore find units with particular expertise in certain application domains, like financial systems, office automation, or real-time process control.

Finally, we may departmentalize according to life-cycle phases, as is done in the L-Type scheme. In particular, we may distinguish between development

and maintenance. With an increasing portfolio of systems to be maintained and the increasing business need of keeping the growing base of information systems working satisfactorily, the division of development and maintenance into separate organizational units is found more often.

Separating development and maintenance has both advantages and disadvantages. The major advantages are:

- Clear accountability: we may clearly separate the cost and effort involved in maintenance activities from investments in new developments. If personnel are involved in both types of work, they have some freedom in charging their time. It is then more difficult to measure and predict the 'real' cost of software maintenance.

- Intermittent demands of maintenance make it difficult to predict and control progress of new system development. If people do both maintenance work and development, some control can be exercised by specifically allocating certain periods of time as maintenance periods. For instance, the first week of each calendar month may be set aside for maintenance. But even then, maintenance problems are rather unpredictable and some need immediate attention. Many a schedule slippage is due to the maintenance drain.

- A separation of maintenance and development facilitates and motivates the maintenance organization to conduct a meaningful acceptance test before the system is taken into production. If such an acceptance test is not conducted explicitly, maintenance may be confronted with low-quality software or systems which still need a 'finishing touch' which the development team has left undone for lack of time.

- By specializing on maintenance tasks, a higher quality of user service can be realized. By their very nature, development groups are focused on system delivery, whereas maintenance people are service-oriented and find pride in satisfying user requests. We will further elaborate upon this issue in section 14.5.2.

- By concentrating on the systems to be maintained, a higher level of productivity is achieved. Maintenance work requires specific skills of which a more optimal use can be made in a separate organization. If people are involved in both development and maintenance, more staff have to be allocated to maintenance and the familiarity with any particular system is spread more thinly.

On the other hand, the strict separation of development and maintenance has certain disadvantages as well:

- Demotivation of personnel because of status differences, with consequential degradation of quality and productivity. Managerial attitudes and traditional career paths are the main causes for these motivational problems. Conversely,

proper managerial attention to maintenance work goes a large way towards alleviating the morale problem. For example, an organization may decide to hire new people into development only and explicitly consider a transfer to maintenance as a promotion. (Most organizations do exactly the opposite.)

- Loss of knowledge about the system (with respect to both its design and the application domain knowledge incorporated) when the system is transferred from development to maintenance. Various strategies can mitigate against this loss. For example, a future maintainer of a system may spend some time with the development team, a developer may stay with maintenance until the maintainers have become sufficiently acquainted with the system, or a designer may instruct the maintainers about the design of a system.

- Coordination costs between development and maintenance, especially when the new system replaces an existing one.

- Increased cost of system acceptance by the maintenance organization. If the system is explicitly carried over from development, certain quality and documentation criteria must be met. Within an A-type organization these requirements can often be relaxed a bit, or their fulfillment is postponed. It is by no means clear though that this really incurs an increase in cost. In the long run it may well be cheaper to only accept systems which pass a proper maintenance acceptance test.

- Possible duplication of communication channels to the user organization.

Based on an analysis of existing departmentalizations and the resulting list of strengths and weaknesses, Swanson and Beath express a slight preference for having development and maintenance as separate organizational units. We concur with that. Careful procedures could be devised that overcome some or all of the disadvantages listed. We should stress that personnel demotivation is a real issue in many organizations. It deserves serious management attention.

Combinations of departmentalization types are also possible. In particular, combinations of A-type and L-type departmentalizations are quite common. So, within the maintenance organization, smaller groups may specialize in some application domain, i.e. a specific collection of information systems. This may be termed the L-A-scheme. Conversely, in an A-L-scheme a small maintenance unit is found within a group that specializes in a certain application area. The L-A-scheme is more likely to exhibit the advantages of the L-scheme than the A-L-scheme does.

Too much specialization is a lurking danger though. A system should never become someone's private property. A variation of the reverse Peter principle applies here: people rise within an organization to a level at which they become indispensable. Job rotation is one way to avoid people from becoming too much entrenched in the peculiarities of a system. There is a trade-off though, since such a step also means that in-depth knowledge of a system is sacrificed.

### 14.5.2 Software Maintenance from a Service Perspective

> *Software maintenance organizations need to realize that they are in the customer service business.*
> (Pigoski, 1996)

Software development results in a product, a piece of software. Software maintenance can be seen as providing a service. There are notable differences between products and services, which mean that the quality of products and services is judged differently. As a consequence, the quality of software development and software maintenance is also judged differently and maintenance organizations should pay attention to service-specific quality aspects.

Apparently, this is not widely recognized yet. Within the software maintenance domain, the focus is still on product aspects. The final phases of software development supposedly concern the delivery of an operations manual, installing the software, handling change requests and fixing bugs. In practice, the role of an IT department is much broader during the deployment stage, as is illustrated by the ubiquitous help desk.

This is confirmed by Stålhane et al. (1997) who report on a survey to find those aspects of software quality that customers consider most important. The main insight to be gained from their study is the strong emphasis customers place on service quality. The top five factors found in their study are: service responsiveness, service capacity, product reliability, service efficiency, and product functionality. They also quote an interesting result from a quality study in the telecommunications domain. To the question 'Would you recommend others to buy from this company?', a 100% yes was obtained from the category of users that had complained and got a satisfactory result. For the category that had not complained, this percentage was 87%. Apparently, it is more important to get a satisfactory service than to have no problems at all.

The main differences between products and services are as follows:

- Services are **intangible**, products are tangible. This is considered the most basic difference between products and services. Services -- being benefits or activities -- cannot be seen, felt, tasted, or touched, unlike products. Consequently, services cannot be counted, stored, patented, readily displayed, or communicated, and pricing is more difficult.

- Because services are created by activities, and activities are performed by humans, services tend to be more **heterogeneous** than products. Customer satisfaction depends on employee actions during the service delivery. Service quality depends on factors which are difficult to control, such as the ability of customers to articulate their needs, the ability and willingness of personnel to satisfy those needs, the presence or absence of other customers, and the level of demand for the service. These complicating factors make it hard to know whether the service was delivered according to plan or specification.

- Services are produced and consumed **simultaneously**, whereas production and consumption of products can be separated. For example, a car can be produced first, sold a few months later, and then be consumed over a period of several years. For services, production and consumption has to take place in parallel. The production of the service creates the set of benefits, whose consumption cannot be postponed. For example, a restaurant service -- preparing a meal and serving the customer -- by and large has to be produced while the customer is receiving the service. As a consequence, customers participate in and affect the transaction, customers may affect each other, employees affect the service outcome, and centralization and mass production are difficult.
- Services are **perishable**, products are not. Services cannot be saved or stored. They cannot be returned or resold, and it is difficult to synchronize supply and demand.

The difference between products and services is not clear-cut. Often, services are augmented with physical products to make them more tangible. For example, luggage tags may be provided with a travel insurance. In the same way, products are augmented with add-on services, such as a guarantee, to improve the quality perception of the buyer. In the service marketing literature, a product--service continuum is used to indicate that there is no clear boundary between products and services. This product--service continuum has pure products at one end, pure services at the other, and product--service mixtures in between. Figure 14.13 shows some example products and services along this continuum.

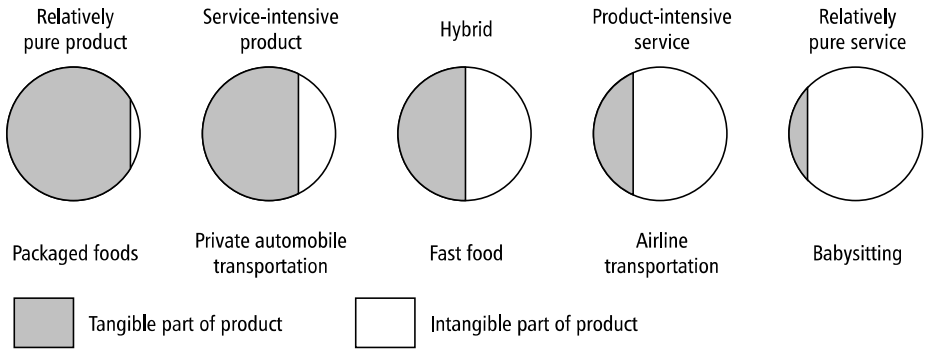


Figure 14.13 The product--service continuum (*Source: L.L. Berry & A. Parasuraman,* Marketing Services: Competing Through Quality, *1991, The Free Press*)

As this figure shows, products and services can be intertwined. In the case of fast-food, both the product, the food, and the service, quick delivery, are essential to

the customer. The quality of such a product--service mix is judged on both product and service aspects: does the food taste good and is it served quickly.

Let us return to the software engineering domain. A major difference between software development and software maintenance is the fact that software development results in a *product*, whereas software maintenance results in a *service* being delivered to the customer. Software maintenance has more service-like aspects than software development, because the value of software maintenance lies in activities that result in benefits for the customers, such as corrected faults and new features. Contrast this with software development, where the development activities themselves do not provide benefits to the customer. It is the resulting software system that provides those benefits.

As noted, the difference between products and services is not clear-cut. Consequently, this goes for software development and software maintenance as well. Figure 14.14 shows the product--service continuum with examples from the software engineering domain.

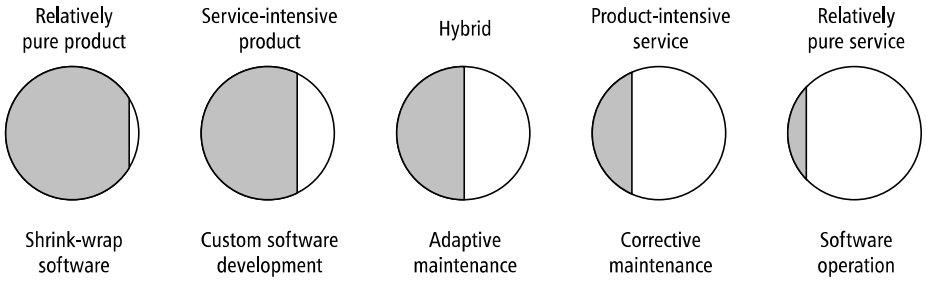


Figure 14.14 The product--service continuum for software development and maintenance

Service marketeers often use the gap model to illustrate how differences between perceived service delivery and expected service may come about. This gap model is depicted in figure 14.15. Service quality is improved if those gaps are closed. The difference between the perceived quality and the expected quality (gap 5) is caused by four other gaps. These four gaps, and suggested solutions for bridging them, are:

Gap 1 The expected service as perceived by the service provider

differs from the service as expected by the customer. In the field of software maintenance, this difference is often caused by an insufficient relationship focus of the service provider. For example, a maintenance department may aim to satisfy certain availability constraints such as 99% availability, while the actual customer concern is with maximum downtime.

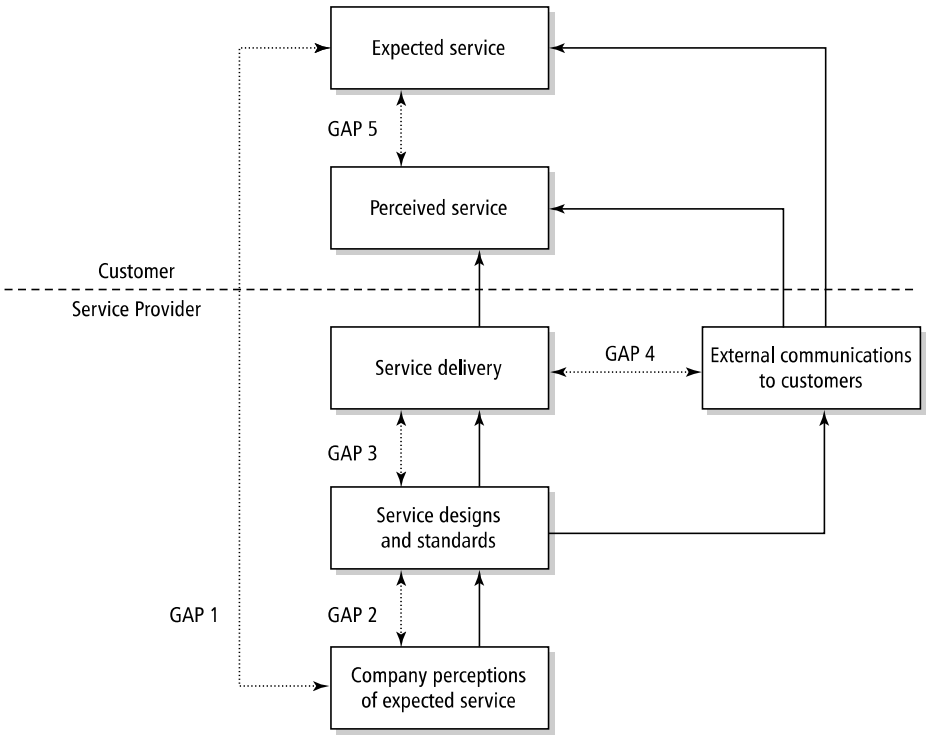


Figure 14.15 The gaps model of service quality (*Reprinted with permission from A. Parasuraman, V.A. Zeithaml & L.L. Berry*, A Conceptual Model of Service Quality and its Implication for Future Research, ***in*** *Journal of Marketing* ***49***, *Fall 1985, pp. 41-50. Published by the American Marketing Association.*)

It is important for a maintenance organization to translate customer service expectations into clear service agreements. Preferably, the maintenance service commitments are specified in a contract -- the **Service Level Agreement** -- which specifies, amongst other things, the services themselves, the levels of service (i.e. how fast and how reliably will the service be delivered), what happens if the service provider does not reach the agreed upon service levels, when and how the customer will receive reports regarding the services actually delivered, when and how the service level agreement will be reviewed, and so on.

Gap 2 The service specification differs from the expected service as perceived by the service provider. This may arise if the (internal) service designs and standards

do not match the service requirements as perceived by the service provider. For example, the customer expects a quick restart of the system, while the standard procedure of the maintenance organization is focused on analyzing the reason for the crash.

The maintenance activities as specified in the service level agreement have to be planned. This includes the planning of the activities themselves, the transfer of the results to the customer, the planning of releases, the estimation of resources needed, the scheduling of maintenance activities, and the identification of possible risks. Explicitly basing the planning of maintenance activities on the commitments as agreed with the customer helps to close this gap.

Gap 3 The actual service delivery differs from the specified services. This is often caused by deficiencies in human resource policies, failures to match demand and supply, and customers not fulfilling their role. For example, customers may bypass the helpdesk by phoning the maintainer of their system directly, thereby hindering a proper incident management process.

The service level agreement states which maintenance activities are to be carried out, and how fast, reliably, etc. this should be done. In order to be able to report on the performance of the maintenance organization in this respect, information about the actual maintenance activities must be gathered. This information can be used to monitor maintenance activities and take corrective actions if necessary.

For example, when the customer reports a bug, information about the bug itself (originator, type, etc.) is recorded, as well as the reporting time, the time when corrective action was started and ended, and the time when the bug was reported as fixed. If these data indicate that the average downtime of a system exceeds the level as specified in the service level agreement, the maintenance organization might assign more maintenance staff to this system, put maintenance staff on point-duty at the customer site, renegotiate the agreed upon service level, or take other action to realign agreement and reality.

By keeping a strict eye upon the performance of the maintenance organization and adjusting the maintenance planning or renegotiating the commitments with the customer when required, gap 3 is narrowed.

Gap 4 Communication about the service does not match the actual service delivery. This may be caused by ineffective management of customer expectations, promising too much, or ineffective horizontal communication. For example, a customer is not informed about the repair of a bug he reported.

An important instrument to help close this gap is event management. Event management concerns the management of events that cause or might cause the maintenance activities carried out to deviate from the levels as promised

in the service level agreement. An event is either a change request, such as a user request for a new feature, or an incident. Incidents are software bugs and other hazards that the maintenance organization has promised to deal with, such as, say, a server being down.

The main purpose of event management is to manage all those events. To do so, an event management library system is employed, often in the form of a 'helpdesk system'. The event management library system provides for the storage, update, and retrieval of event records, and the sharing and transfer of event records between parties involved. This event management library system supports the communication with the customer about maintenance services delivered. It is also a highly valuable 'memory' for the maintainers: they may use the event library to search for similar incidents, to see why certain components were changed before, etc.[4]

Since the fifth gap is caused by the four other gaps, perceived service quality can be improved by closing those first four gaps, thus bringing the perceived quality in line with the expected quality. Since software maintenance organizations are essentially service providers, they need to consider the above issues. They need to manage their product -- software maintenance -- as a service in order to be able to deliver high quality.

### 14.5.3 Control of Maintenance Tasks

Careful control of the product is necessary during software development. The vast amount of information has to be kept under control. Documentation must be kept consistent and up-to-date. An appropriate scheme for doing so is provided by the set of procedures that make up configuration control; see chapter 4. Configuration control pays particular attention to the handling of change requests. Since handling change requests is what maintenance is all about, configuration control is of vital importance during maintenance.

Effective maintenance depends on following a rigorous methodology, not only with respect to the implementation of changes agreed upon, but also with respect to the way change is controlled. Following IEEE Standard 1219, we suggest the following orderly, well-documented process for controlling changes during maintenance:

1. **Identify and classify change requests** Each change request (CR) is given a unique identification number and is classified into one of the maintenance categories (corrective, adaptive, perfective, preventive). The CR is analyzed to decide whether it will be accepted, rejected, or needs further evaluation. This analysis also results in a first cost estimate. The CR is finally prioritized and scheduled for implementation.

[4] Note that the focus of the event management library system differs somewhat from that of configuration management as discussed in the next section. Configuration management emphasizes the *internal* use of information about change requests and the like. Our description of event management focuses on the *external* use of essentially the same information. In practice, the two processes may well be combined.

2. **Analysis of change requests** This step starts with an analysis of the CR to determine its impact on the system, the organization, and possible interfacing systems. Several alternative solutions to implement the CR may be devised, including their cost and schedule. The results of the analysis are documented in a report. Based on this report, a decision is made whether or not the CR will be implemented. The authority for this decision is usually assigned to the configuration control board; see also chapter 4.

3. **Implement the change** This involves the design, implementation and testing of the change. The output of this step is a new version of the system, fully tested, and well documented.

The above steps indicate a maintenance model in which each change request is carefully analyzed and, if (and only if) the request is approved, its implementation is carried out in a disciplined, orderly way, including a proper update of the documentation. This control scheme fits in well with the **iterative-enhancement** model of software maintenance; see figure 14.16. The essence of the iterative-enhancement model is that the set of documents is modified starting with the highest-level document affected by the changes, propagating the changes down through the full set of documents. For example, if a change request necessitates a design change, then the design is changed first. Only as a consequence of the design change will the code be adapted.

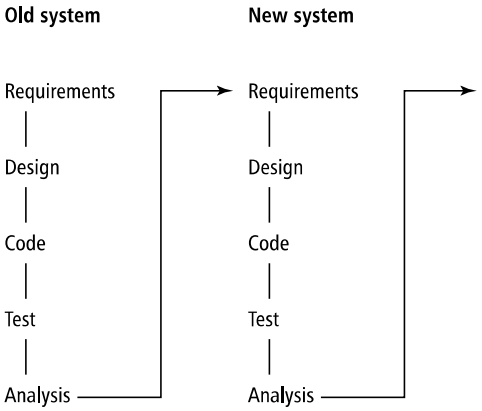


Figure 14.16 Iterative-enhancement model of software maintenance (*Source: V.R. Basili, Viewing maintenance as reuse-oriented software development*, IEEE Software **7**, *1 (1990) 19--25, 1990 IEEE.*)

Reality is often different. Figure 14.17 depicts the so-called **quick-fix** model of software maintenance. In the quick-fix model, you take the source code, make the necessary changes to the code and recompile the system to obtain a new version. The source-code documentation and other higher-level documents get updated after the code has been fixed, and usually only if time permits.

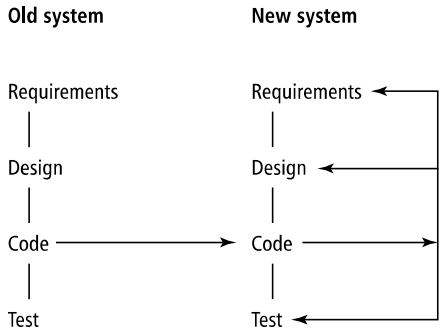


Figure 14.17 Quick-fix model of software maintenance (*Source: V.R. Basili, Viewing maintenance as reuse-oriented software development*, IEEE Software **7**, *1 (1990) 19--25, 1990 IEEE.*)

In the latter scheme, patches are made upon patches and the structure of the system degrades rather quickly. Because of the resulting increase in system complexity and inconsistency of documents, future maintenance becomes much more difficult. To be realistic, the quick-fix model cannot be completely circumvented. In an emergency situation there is but one thing that matters: getting the system up and running again as fast as possible. Where possible though, the quick-fix model should be avoided. If it is used at all, preventive maintenance activities should be scheduled to repair the structural damage done.

In a normal, non-emergency situation, change requests are often bundled into **releases**. The user then does not get a new version after each and every change has been realized, but after a certain number of change requests has been handled, or after a certain time frame. Three common ways of deciding on the contents and timing of the next release are:

- fixed staff and variable schedule. In this scheme, there is a fixed number of people available for the maintenance work. The next release date is fixed in advance. Often, the release dates are scheduled at fixed time intervals, say every six months. The next release will contain all changes that have been handled within the agreed time frame. So the next release is always on time. There also

is some flexibility as to the contents of the next release, since the maintainers and the customer do not fix the contents in advance.

- variable staff and fixed schedule. Here, a release date is fixed in advance. The portfolio of change requests to be handled in this release is also negotiated and fixed in advance. Next, the number of people needed to implement the changes within the fixed time frame is decided. On the way, some renegotiation of both the contents and the schedule is possible. An advantage of this scheme is that change requests are assigned clear priorities and that communication with the customer about the contents of the next release is enforced.

- variable staff and variable schedule. As in the previous scheme, the portfolio of change requests to be handled in the next release is negotiated and fixed in advance. Then, the cost and schedule for this release are negotiated, and the number of maintainers required to achieve it is determined. This scheme requires more planning and oversight than the other two schemes. It is also likely to better accommodate the customer. As with ordinary development, schedule slippages and contents renegotiation are not uncommon in this scheme.

### 14.5.4 Quality Issues

Changing software impairs its structure. By a conscious application of software quality assurance procedures during maintenance, we may limit the negative effects. If we know the software quality factors that affect maintenance effort and cost, we may measure those factors and take preventive actions accordingly. In particular, such metrics can be used to guide decisions as to when to start a major overhaul of components or complete systems.

Quality control issues get quite some attention during software development. Software quality assurance however should broaden its scope to maintenance as well. The implementation of changes during maintenance requires the same level of quality assurance as development work. The ingredients of software quality assurance procedures, as discussed in chapter 6, apply equally well to software maintenance.

Software quality assurance can be backed up by measurements that quantify quality aspects. With respect to maintenance, we may focus on measures which specifically relate to maintenance effort, such as counting defects reported, change requests issued, effort spent on incorporating changes, complexity metrics, etc.

Relationships between such measures can then be sought. Observed trends can be used to initiate actions, such as:

- If maintenance efforts correlate well with complexity metrics like Henri and Kafura's Information Flow or McCabe's cyclomatic complexity (see chapter 12), then these complexity metrics may be used to trigger preventive maintenance. Various studies have indeed found such correlations.

- If certain modules require frequent changes or much effort to realize changes, then a re-design of such modules should be given serious consideration.
- Metrics can be used to spot bad smells, and initiate refactoring.

A particularly relevant issue during maintenance is to decide when to reengineer. At a certain point in time, evolving an old system becomes next to impossible and a major reengineering effort is required or the system enters the servicing stage. There are no hard figures on which to decide this, but certain system characteristics certainly indicate system degradation:

- Frequent system failures;
- Code over seven years old;
- Overly-complex program structure and logic flow;
- Code written for previous generation hardware;
- Running in emulation mode;
- Very large modules or subroutines;
- Excessive resource requirements;
- Hard-coded parameters that are subject to change;
- Difficulty in keeping maintenance personnel;
- Seriously deficient documentation;
- Missing or incomplete design specifications.

The greater the number of such characteristics present, the greater the potential for redesign.

Improvements in software maintenance requires insight into factors that determine maintenance cost and effort. Software metrics provide such insight. To measure is to know. By carefully collecting and interpreting maintenance data, we may discover the major cost drivers of software maintenance and initiate actions to improve both quality and productivity.

## 14.6 Summary

Software maintenance encompasses all modifications to a software product after delivery. The following breakdown of maintenance activities is usually made:

**Corrective maintenance** concerns the correction of faults.

**Adaptive maintenance** deals with adapting software to changes in the environment.

**Perfective maintenance** mainly deals with accommodating new or changed user requirements.

**Preventive maintenance** concerns activities aimed at increasing a system's maintainability.

'Real' maintenance, the correction of faults, consumes approximately 25% of maintenance effort. By far the larger part of software maintenance concerns the evolution of software. This evolution is inescapable. Software models part of reality. Reality changes, and so does the software that models it.

Major causes of maintenance problems were discussed in section 14.2: the existence of a vast amount of unstructured code, insufficient knowledge about the system or application domain on the part of maintenance programmers, insufficient documentation, and the bad image of the software maintenance department.

Some of these problems are accidental and can be remedied by proper actions. Through a better organization and management of software maintenance, substantial quality and productivity improvements can be realized. These issues were discussed in section 14.5. Obviously, improved maintenance should start with improved development. Opportunities to improve the development process are a major topic in most chapters of this book.

A particularly relevant issue for software maintenance is that of reverse engineering, the process of reconstructing a lost blueprint. Before changes can be realized, the maintainer has to gain an understanding of the system. Since the majority of operational code is unstructured and undocumented, this is a major problem. Section 14.3 addresses reverse engineering, its limitations, and tools to support it.

The fundamental problem is that maintenance will remain a big issue. Because of the changes made to software, its structure degrades. Specific attention to preventive maintenance activities aimed at improving system structure are needed from time to time to fight system entropy.

Software maintenance used to be a rather neglected topic in the software engineering literature. Like programmers, researchers are more attracted to developing new, fancy methods and tools for software development. This situation has changed. Major journals regularly feature articles on software maintenance, there is an annual IEEE Conference on Software Maintenance (since 1985), and the journal *Software Maintenance and Evolution: Research and Practice* (launched 1989) is wholly devoted to it.

## 14.7 Further Reading

(Pigoski, 1996) is a text book wholly devoted to software maintenance. Lientz and Swanson (1980) is a seminal booklet on software maintenance. It introduces the widely-known categories of maintenance tasks and provides data on their distribution. More recent data on the distribution of maintenance tasks are given in (Nosek and Palvia, 1990), (Dekleva, 1992) and (Sousa and Mozeira, 1998). Chapin et al. (2001) gives a new classification of maintenance categories, including a separate category

for user support. The maintenance life cycle stages are discussed in (Burch and Kung, 1997) and (Kung and Hsu, 1998). The distinction between an evolution stage and a servicing stage stems from (Bennett and Rajlich, 2000). The distribution of code-related maintenance activities is discussed in (Yu and Chen, 2006). The practice of software maintenance is discussed in (Singer, 1998) and (Tan and Gable, 1998).

The various types of reverse engineering are discussed in (Chikofsky and Cross II, 1990). The 100% functional equivalence issue in reverse engineering is discussed in (Bennett, 1998). Reverse engineering tools are discussed in (Biggerstaff et al., 1994), (Jarzabek and Wang, 1998) and (Bellay and Gall, 1998). Fowler (1999) is the standard text for refactorinig. Mens and Tourwé (2004) provide a survey of software refactoring. Migration is discussed in (Rahgozar and Oroumchian, 2003) and (Bisbal et al., 1999). Programming plans and beacons were originally proposed in (Soloway and Ehrlich, 1984) and (Brooks, 1983). Research addressing the role of these concepts in program comprehension processes is described in (von Mayrhauser and Vans, 1995) and (von Mayrhauser et al., 1997). LaToza et al. (2006) discusses developer work habits, including code comprehension, during development and evolution. Example tools to help software maintenance include (Singer et al., 2005) (support for browsing through software), (Rysselberghe and Demeyer, 2004) (visualize change history) and (Ducasse et al., 2006) (visualization of distribution of system properties).

Gîrba and Ducasse (2006) provide an overview of types of software evolution analysis. The distinction between version-centered and history-centered analysis is made in that article. Gîrba et al. (2004) discusses the 'yesterday's weather' approach to reverse engineering. Fischer and Gall (2004) and Greevy et al. (2006) discuss history-centered analysis.

Possible organizations of maintenance activities as well as their major advantages and disadvantages are discussed in (Swanson and Beath, 1990). Yeh and Jeng (2002) discuss the influence of departmentalization on software maintenance. The service perspective on software maintenance is discussed in (Niessink and van Vliet, 1999). The translation hereof into a Capability Maturity Model aimed at maintenance processes is described in (Niessink and van Vliet, 1998a).

The IEEE Process model for software maintenance is described in (IEEE1219, 1992). The iterative-enhancement and quick-fix models of software maintenance are discussed in (Basili, 1990). Approaches to scheduling releases are the topic of (Stark and Oman, 1997).

The cost of software maintenance, and empirical relations between quality aspects and cost are the topic of (Banker et al., 1993), (Kemerer and Slaughter, 1997), (Henry and Cain, 1997) and (Niessink and van Vliet, 1997). Indicators of system degradation are given in (Martin and Osborne, 1983).

## Exercises

1. Define the following terms: corrective maintenance, adaptive maintenance, perfective maintenance, and preventive maintenance.

2. Discuss the major causes of software maintenance problems.
3. What is reverse engineering?
4. What is refactoring?
5. Characterize the evolution and servicing stage of software maintenance.
6. What is the difference between design recovery and redocumentation?
7. Characterize the version-oriented analysis and history-centered analysis of software evolution data.
8. Why does corrective maintenance have more service-like aspects than product-like aspects?
9. Discuss the iterative-enhancement and quick-fix models of software maintenance.
10. Discuss the major impediments to fully-automated design recovery.
11. Discuss advantages of software configuration control support during software maintenance.
12. Discuss the possible structure and role of an acceptance test by the maintenance organization prior to the release of a system.
13. ♡ An alternative classification of maintenance and development activities is as follows:

    - Functional maintenance = corrective maintenance + adaptive maintenance + non-functional perfective maintenance (i.e. improving quality) + replacement of a system by a functional equivalent.
    - Functional development = functional perfective maintenance (i.e. adding new features) + development of new systems.

    Could this classification provide us with a better picture of the *real* maintenance effort? See also (Krogstie, 1994).
14. ♡ Assess opportunities of knowledge-based support for software maintenance (see (Devanbu et al., 1991) for a very interesting application of such ideas).
15. ♡ Give a primary classification of your maintenance organization as W-, A-, or L-Type (see figure 14.12). What are the major strengths and weaknesses of your particular organization?
16. ♡ Does your organization collect quantitative data on maintenance activities? If so, what type of data, and how are they used to guide and improve the

maintenance process? If not, how is maintenance planned and controlled?

17. ♠ Study the technical documentation of a system whose development you have been involved in. Does the documentation capture the design rationale? In what ways does it support comprehension of the system? In hindsight, can you suggest ways to improve the documentation for the purpose of maintenance?

18. ♡ Discuss the impact of component reuse on maintainability.

19. ♡ Discuss the possible contribution of object-oriented software development to software maintenance.

20. ♡ Can you think of reasons why a 10% change in a program of 200 LOC would take more effort than a 20% change in a program of 100 LOC?