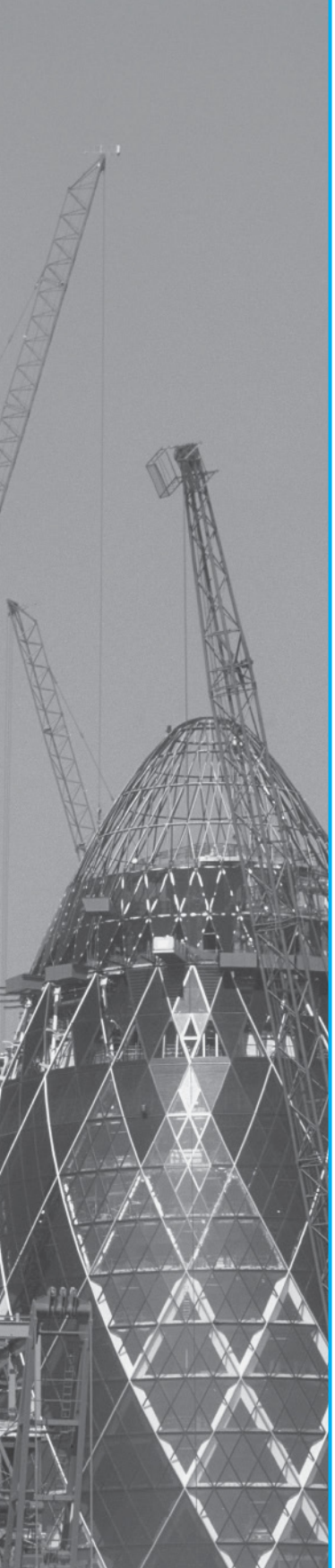

9 Software evolution

Objectives

The objectives of this chapter are to explain why software evolution is such an important part of software engineering and to describe the challenges of maintaining a large base of software systems, developed over many years. When you have read this chapter, you will:

- understand that software systems have to adapt and evolve if they are to remain useful and that software change and evolution should be considered as an integral part of software engineering;
- understand what is meant by legacy systems and why these systems are important to businesses;
- understand how legacy systems can be assessed to decide whether they should be scrapped, maintained, reengineered, or replaced;
- have learned about different types of software maintenance and the factors that affect the costs of making changes to legacy software systems.

Contents

Large software systems usually have a long lifetime. For example, military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more. Business systems are often more than 10 years old. Enterprise software costs a lot of money, so a company has to use a software system for many years to get a return on its investment. Successful software products and apps may have been introduced many years ago with new versions released every few years. For example, the first version of Microsoft Word was introduced in 1983, so it has been around for more than 30 years.

During their lifetime, operational software systems have to change if they are to emain useful. Business changes and changes to user expectations generate new requirements for the software. Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for changes to its hardware and software platform, and to improve its performance or other non-functional characteristics. Software products and apps have to evolve to cope with platform changes and new features introduced by their competitors. Software systems, therefore, adapt and evolve during their lifetime from initial deployment to final retirement.

Businesses have to change their software to ensure that they continue to get value from it. Their systems are critical business assets, and they have to invest in change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development. Historical data suggests that somewhere between 60% and 90% of software costs are evolution costs (Lientz and Swanson 1980; Erlikh 2000). Jones (Jones 2006) found that about 75% of development staff in the United States in 2006 were involved in software evolution and suggested that this percentage was unlikely to fall in the foreseeable future.

Software evolution is particularly expensive in enterprise systems when individual software systems are part of a broader "system of systems." In such cases, you cannot just consider the changes to one system; you also need to examine how these changes affect the broader system of systems. Changing one system may mean that other systems in its environment may also have to evolve to cope with that change.

Therefore, as well as understanding and analyzing the impact of a proposed change on the system itself, you also have to assess how this change may affect other systems in the operational environment. Hopkins and Jenkins (Hopkins and Jenkins 2008) have coined the term *brownfield software development* to describe situations in which software systems have to be developed and managed in an environment where they are dependent on other software systems.

The requirements of installed software systems change as the business and its environment change, so new releases of the systems that incorporate changes and updates are usually created at regular intervals. Software engineering is therefore a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system (Figure 9.1). You start by creating release 1 of the system. Once delivered, changes are proposed, and the development of release 2 starts almost immediately. In fact, the need for evolution may become obvious even before the system is deployed, so later releases of the software may start development before the current version has even been released.

In the last 10 years, the time between iterations of the spiral has reduced dramatically. Before the widespread use of the Internet, new versions of a software system

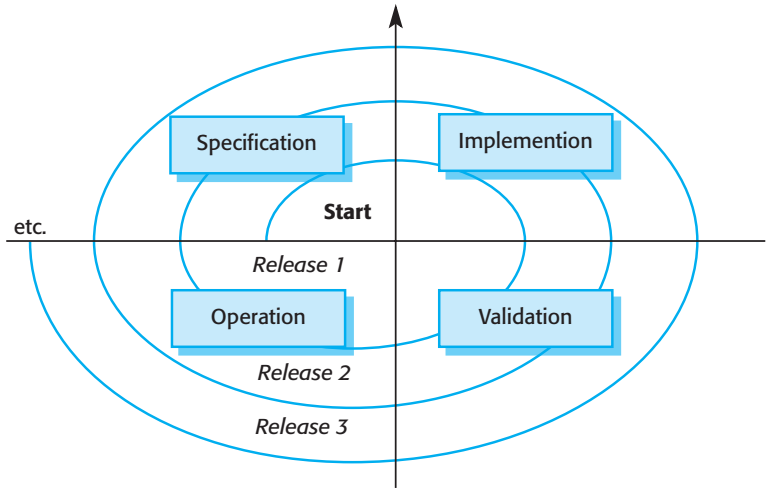

Figure 9.1 A spiral model of development and evolution

may only have been released every 2 or 3 years. Now, because of competitive pressures and the need to respond quickly to user feedback, the gap between releases of some apps and web-based systems may be weeks rather than years.

This model of software evolution is applicable when the same company is responsible for the software throughout its lifetime. There is a seamless transition from development to evolution, and the same software development methods and processes are applied throughout the lifetime of the software. Software products and apps are developed using this approach.

The evolution of custom software, however, usually follows a different model. The system customer may pay a software company to develop the software and then take over responsibility for support and evolution using its own staff. Alternatively, the software customer might issue a separate contract to a different software company for system support and evolution.

In this situation, there are likely to be discontinuities in the evolution process. Requirements and design documents may not be passed from one company to another. Companies may merge or reorganize, inherit software from other companies, and then find that this has to be changed. When the transition from development to evolution is not seamless, the process of changing the software after delivery is called software maintenance. As I discuss later in this chapter, maintenance involves extra process activities, such as program understanding, in addition to the normal activities of software development.

Rajlich and Bennett (Rajlich and Bennett 2000) propose an alternative view of the software evolution life cycle for business systems. In this model, they distinguish between evolution and servicing. Evolution is the phase in which significant changes to the software architecture and functionality are made. During servicing, the only changes that are made are relatively small but essential changes. These phases overlap with each other, as shown in Figure 9.2.

According to Rajlich and Bennett, when software is first used successfully, many changes to the requirements by stakeholders are proposed and implemented. This is

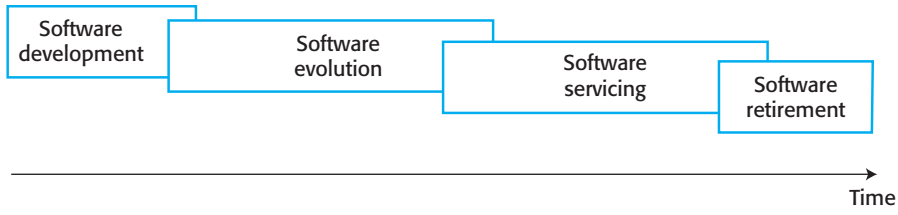

Figure 9.2 Evolution and servicing

the evolution phase. However, as the software is modified, its structure tends to degrade, and system changes become more and more expensive. This often happens after a few years of use when other environmental changes, such as hardware and operating systems, are also required. At some stage in the life cycle, the software reaches a transition point where significant changes and the implementation of new requirements become less and less cost-effective. At this stage, the software moves from evolution to servicing.

During the servicing phase, the software is still useful, but only small tactical changes are made to it. During this stage, the company is usually considering how the software can be replaced. In the final stage, the software may still be used, but only essential changes are made. Users have to work around problems that they discover. Eventually, the software is retired and taken out of use. This often incurs further costs as data is transferred from an old system to a newer replacement system.

9.1 Evolution processes

As with all software processes, there is no such thing as a standard software change or evolution process. The most appropriate evolution process for a software system depends on the type of software being maintained, the software development processes used in an organization, and the skills of the people involved. For some types of system, such as mobile apps, evolution may be an informal process, where change requests mostly come from conversations between system users and developers. For other types of systems, such as embedded critical systems, software evolution may be formalized, with structured documentation produced at each stage in the process.

Formal or informal system change proposals are the driver for system evolution in all organizations. In a change proposal, an individual or group suggests changes and updates to an existing software system. These proposals may be based on existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team. The processes of change identification and system evolution are cyclical and continue throughout the lifetime of a system (Figure 9.3).

Before a change proposal is accepted, there needs to be an analysis of the software to work out which components need to be changed. This analysis allows the cost and the impact of the change to be assessed. This is part of the general process of change management, which should also ensure that the correct versions of

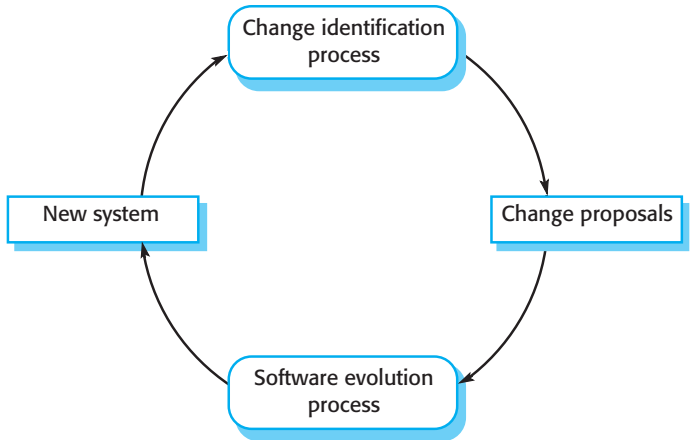

Figure 9.3 Change identification and evolution processes

components are included in each system release. I discuss change and configuration management in Chapter 25.

Figure 9.4 shows some of the activities involved in software evolution. The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.

If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

In situations where development and evolution are integrated, change implementation is simply an iteration of the development process. Revisions to the system are designed, implemented, and tested. The only difference between initial development and evolution is that customer feedback after delivery has to be considered when planning new releases of an application.

Where different teams are involved, a critical difference between development and evolution is that the first stage of change implementation requires program understanding.

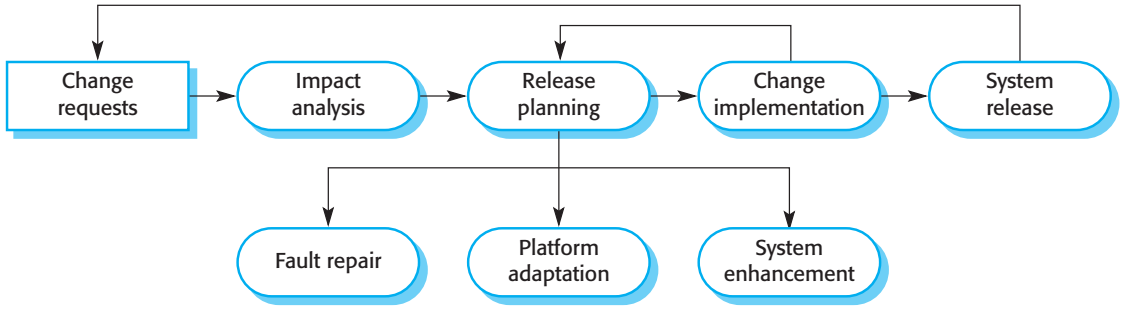

Figure 9.4 A general model of the software evolution process

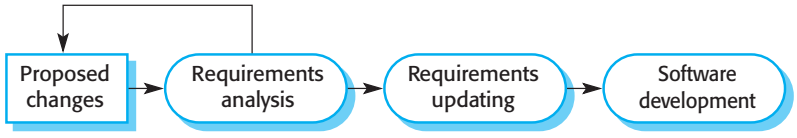

Figure 9.5 Change implementation

During the program understanding phase, new developers have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. They need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

If requirements specification and design documents are available, these should be updated during the evolution process to reflect the changes that are required (Figure 9.5). New software requirements should be written, and these should be analyzed and validated. If the design has been documented using UML models, these models should be updated. The proposed changes may be prototyped as part of the change analysis process, where you assess the implications and costs of making the change.

However, change requests sometimes relate to problems in operational systems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault is detected that has to be repaired to allow normal operation to continue or to address a serious security vulnerability.
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation.
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.

In these cases, the need to make the change quickly means that you may not be able to update all of the software documentation. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 9.6). The danger here is that the requirements, the software design, and the code can become inconsistent. While you may intend to document the change in the requirements and design, additional emergency fixes to the software may then be needed. These take priority over documentation. Eventually, the original change is forgotten, and the system documentation and code are never realigned. This problem of maintaining multiple representations of a system is one of the arguments for minimal documentation, which is fundamental to agile development processes.

Emergency system repairs have to be completed as quickly as possible. You choose a quick and workable solution rather than the best solution as far as system structure is concerned. This tends to accelerate the process of software ageing so that future changes become progressively more difficult and maintenance costs increase. Ideally, after emergency code repairs are made, the new code should be refactored

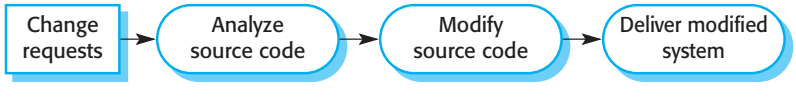

Figure 9.6 The emergency repair process

and improved to avoid program degradation. Of course, the code of the repair may be reused if possible. However, an alternative, better solution to the problem may be discovered when more time is available for analysis.

Agile methods and processes, discussed in Chapter 3, may be used for program evolution as well as program development. Because these methods are based on incremental development, making the transition from agile development to postdelivery evolution should be seamless.

However, problems may arise during the handover from a development team to a separate team responsible for system evolution. There are two potentially problematic situations:

1. Where the development team has used an agile approach but the evolution team prefers a plan-based approach. The evolution team may expect detailed documentation to support evolution, and this is rarely produced in agile processes. There may be no definitive statement of the system requirements that can be modified as changes are made to the system.

2. Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests. The code in the system may not have been refactored and simplified, as is expected in agile development. In this case, some program reengineering may be required to improve the code before it can be used in an agile development process.

Agile techniques such as test-driven development and automated regression testing are useful when system changes are made. System changes may be expressed as user stories, and customer involvement can help prioritize changes that are required in an operational system. The Scrum approach of focusing on a backlog of work to be done can help prioritize the most important system changes. In short, evolution simply involves continuing the agile development process.

Agile methods used in development may, however, have to be modified when they are used for program maintenance and evolution. It may be practically impossible to involve users in the development team as change proposals come from a wide range of stakeholders. Short development cycles may have to be interrupted to deal with emergency repairs, and the gap between releases may have to be lengthened to avoid disrupting operational processes.

9.2 Legacy systems

Large companies started computerizing their operations in the 1960s, so for the past 50 years or so, more and more software systems have been introduced. Many of these systems have been replaced (sometimes several times) as the business has changed and evolved. However, a lot of old systems are still in use and play a critical part in the running of the business. These older software systems are sometimes called legacy systems.

Legacy systems are older systems that rely on languages and technology that are no longer used for new systems development. Typically, they have been maintained over a long period, and their structure may have been degraded by the changes that have been made. Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures. It may be impossible to change to more effective business processes because the legacy software cannot be modified to support new processes.

Legacy systems are not just software systems but are broader sociotechnical systems that include hardware, software, libraries, and other supporting software and business processes. Figure 9.7 shows the logical parts of a legacy system and their relationships.

1. *System hardware* Legacy systems may have been written for hardware that is no longer available, that is expensive to maintain, and that may not be compatible with current organizational IT purchasing policies.
2. *Support software* The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.
3. *Application software* The application system that provides the business services is usually made up of a number of application programs that have been developed at different times. Some of these programs will also be part of other application software systems.
4. *Application data* These data are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent, may be duplicated in several files, and may be spread over a number of different databases.
5. *Business processes* These processes are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting an order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.
6. *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

An alternative way of looking at these components of a legacy system is as a series of layers, as shown in Figure 9.8.

Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then you should be able to make changes within a layer without affecting either of the adjacent layers. In practice, however, this simple encapsulation is an oversimplification, and changes to one layer of the system may

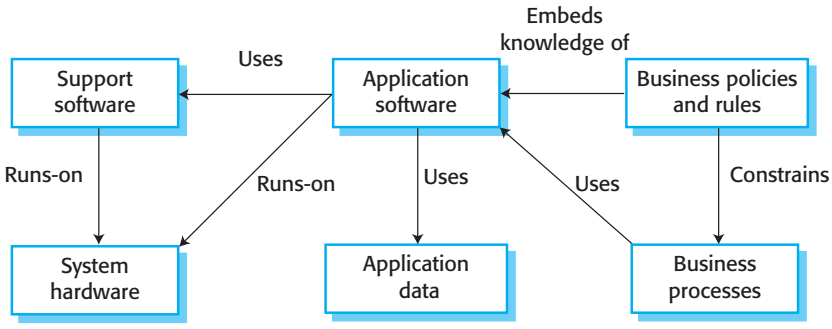

Figure 9.7 The elements of a legacy system

require consequent changes to layers that are both above and below the changed level. The reasons for this are as follows:

1. Changing one layer in the system may introduce new facilities, and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include facilities to access the data through a web browser, and business processes may be modified to take advantage of this facility.
2. Changing the software may slow the system down so that new hardware is needed to improve the system performance. The increase in performance from the new hardware may then mean that further software changes that were previously impractical become possible.
3. It is often impossible to maintain hardware interfaces, especially if new hardware is introduced. This is a particular problem in embedded systems where there is a tight coupling between software and hardware. Major changes to the application software may be required to make effective use of the new hardware.

It is difficult to know exactly how much legacy code is still in use, but, as an indicator, industry has estimated that there are more than 200 billion lines of COBOL code in current business systems. COBOL is a programming language designed for writing business systems, and it was the main business development language from the 1960s to the 1990s, particularly in the finance industry (Mitchell 2012). These programs still work effectively and efficiently, and the companies using them see no need to change them. A major problem that they face, however, is a shortage of COBOL programmers as the original developers of the system retire. Universities no longer teach COBOL, and younger software engineers are more interested in programming in modern languages.

Skill shortages are only one of the problems of maintaining business legacy systems. Other issues include security vulnerabilities because these systems were developed before the widespread use of the Internet and problems in interfacing with systems written in modern programming languages. The original software tool supplier may be out of business or may no longer maintain the support tools used to

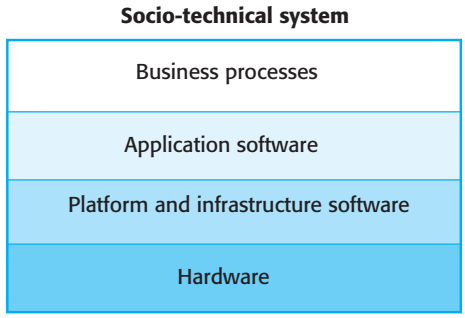

Figure 9.8 Legacy system layers

develop the system. The system hardware may be obsolete and so increasingly expensive to maintain.

Why then do businesses not simply replace these systems with more modern equivalents? The simple answer to this question is that it is too expensive and too risky to do so. If a legacy system works effectively, the costs of replacement may exceed the savings that come from the reduced support costs of a new system. Scrapping legacy systems and replacing them with more modern software open up the possibility of things going wrong and the new system failing to meet the needs of the business. Managers try to minimize those risks and therefore do not want to face the uncertainties of new software systems.

I discovered some of the problems of legacy system replacement when I was involved in analyzing a legacy system replacement project in a large organization. This enterprise used more than 150 legacy systems to run its business. It decided to replace all of these systems with a single, centrally maintained ERP system. For a number of business and technology reasons, the new system development was a failure, and it did not deliver the improvements promised. After spending more than £10 million, only a part of the new system was operational, and it worked less effectively than the systems it replaced. Users continued to use the older systems but could not integrate these with the part of the new system that had been implemented, so additional manual processing was required.

There are several reasons why it is expensive and risky to replace legacy systems with new systems:

1. There is rarely a complete specification of the legacy system. The original specification may have been lost. If a specification exists, it is unlikely that it has been updated with all of the system changes that have been made. Therefore, there is no straightforward way of specifying a new system that is functionally identical to the system that is in use.

2. Business processes and the ways in which legacy systems operate are often inextricably intertwined. These processes are likely to have evolved to take advantage of the software's services and to work around the software's shortcomings. If the system is replaced, these processes have to change with potentially unpredictable costs and consequences.

3. Important business rules may be embedded in the software and may not be documented elsewhere. A business rule is a constraint that applies to some business function, and breaking that constraint can have unpredictable consequences for the business. For example, an insurance company may have embedded its rules for assessing the risk of a policy application in its software. If these rules are not maintained, the company may accept high-risk policies that could result in expensive future claims.
4. New software development is inherently risky, so that there may be unexpected problems with a new system. It may not be delivered on time and for the price expected.

Keeping legacy systems in use avoids the risks of replacement, but making changes to existing software inevitably becomes more expensive as systems get older. Legacy software systems that are more than a few years old are particularly expensive to change:

1. The program style and usage conventions are inconsistent because different people have been responsible for system changes. This problem adds to the difficulty of understanding the system code.
2. Part or all of the system may be implemented using obsolete programming languages. It may be difficult to find people who have knowledge of these languages. Expensive outsourcing of system maintenance may therefore be required.
3. System documentation is often inadequate and out of date. In some cases, the only documentation is the system source code.
4. Many years of maintenance usually degrades the system structure, making it increasingly difficult to understand. New programs may have been added and interfaced with other parts of the system in an ad hoc way.
5. The system may have been optimized for space utilization or execution speed so that it runs effectively on older slower hardware. This normally involves using specific machine and language optimizations, and these usually lead to software that is hard to understand. This causes problems for programmers who have learned modern software engineering techniques and who don't understand the programming tricks that have been used to optimize the software.
6. The data processed by the system may be maintained in different files that have incompatible structures. There may be data duplication, and the data itself may be out of date, inaccurate, and incomplete. Several databases from different suppliers may be used.

At same stage, the costs of managing and maintaining the legacy system become so high that it has to be replaced with a new system. In the next section, I discuss a systematic decision-making approach to making such a replacement decision.

9.2.1 Legacy system management

For new software systems developed using modern software engineering processes, such as agile development and software product lines, it is possible to plan how to integrate system development and evolution. More and more companies understand that the system development process is a whole life-cycle process. Separating software development and software evolution is unhelpful and leads to higher costs. However, as I have discussed, there is still a huge number of legacy systems that are critical business systems. These have to be extended and adapted to changing e-business practices.

Most organizations have a limited budget for maintaining and upgrading their portfolio of legacy systems. They have to decide how to get the best return on their investment. This involves making a realistic assessment of their legacy systems and then deciding on the most appropriate strategy for evolving these systems. There are four strategic options:

1. *Scrap the system completely* This option should be chosen when the system is not making an effective contribution to business processes. This usually occurs when business processes have changed since the system was installed and are no longer reliant on the legacy system.
2. *Leave the system unchanged and continue with regular maintenance* This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests.
3. *Reengineer the system to improve its maintainability* This option should be chosen when the system quality has been degraded by change and where new change to the system is still being proposed. This process may include developing new interface components so that the original system can work with other, newer systems.
4. *Replace all or part of the system with a new system* This option should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation, or where off-the-shelf systems would allow the new system to be developed at a reasonable cost. In many cases, an evolutionary replacement strategy can be adopted where major system components are replaced by off-the-shelf systems with other components reused wherever possible.

When you are assessing a legacy system, you have to look at it from both a business perspective and a technical perspective (Warren 1998). From a business perspective, you have to decide whether or not the business really needs the system. From a technical perspective, you have to assess the quality of the application software and the system's support software and hardware. You then use a combination of the business value and the system quality to inform your decision on what to do with the legacy system.

For example, assume that an organization has 10 legacy systems. You should assess the quality and the business value of each of these systems. You may then create a chart showing relative business value and system quality. An example of

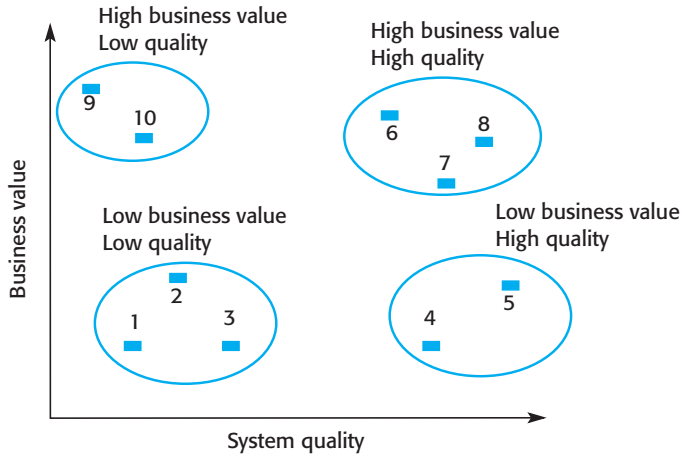

Figure 9.9 An example of a legacy system assessment

this is shown in Figure 9.9. From this diagram, you can see that there are four clusters of systems:

1. *Low quality, low business value* Keeping these systems in operation will be expensive, and the rate of the return to the business will be fairly small. These systems should be scrapped.
2. *Low quality, high business value* These systems are making an important business contribution, so they cannot be scrapped. However, their low quality means that they are expensive to maintain. These systems should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf systems are available.
3. *High quality, low business value* These systems don't contribute much to the business but may not be very expensive to maintain. It is not worth replacing these systems, so normal system maintenance may be continued if expensive changes are not required and the system hardware remains in use. If expensive changes become necessary, the software should be scrapped.
4. *High quality, high business value* These systems have to be kept in operation. However, their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

The business value of a system is a measure of how much time and effort the system saves compared to manual processes or the use of other systems. To assess the business value of a system, you have to identify system stakeholders, such as the end-users of a system and their managers, and ask a series of questions about the system. There are four basic issues that you have to discuss:

1. *The use of the system* If a system is only used occasionally or by a small number of people, this may mean that it has a low business value. A legacy system may have been developed to meet a business need that has either changed or can now be met

more effectively in other ways. You have to be careful, however, about occasional but important use of systems. For example, a university system for student registration may only be used at the beginning of each academic year. Although it is used infrequently, it is an essential system with a high business value.

2. *The business processes that are supported* When a system is introduced, business processes are usually introduced to exploit the system's capabilities. If the system is inflexible, changing these business processes may be impossible. However, as the environment changes, the original business processes may become obsolete. Therefore, a system may have a low business value because it forces the use of inefficient business processes.

3. *System dependability* System dependability is not only a technical problem but also a business problem. If a system is not dependable and the problems directly affect business customers, or mean that people in the business are diverted from other tasks to solve these problems, the system has a low business value.

4. *The system outputs* The key issue here is the importance of the system outputs to the successful functioning of the business. If the business depends on these outputs, then the system has a high business value. Conversely, if these outputs can be cheaply generated in some other way, or if the system produces outputs that are rarely used, then the system has a low business value.

For example, assume that a company provides a travel ordering system that is used by staff responsible for arranging travel. They can place orders with an approved travel agent. Tickets are then delivered, and the company is invoiced for them. However, a business value assessment may reveal that this system is only used for a fairly small percentage of travel orders placed. People making travel arrangements find it cheaper and more convenient to deal directly with travel suppliers through their websites. This system may still be used, but there is no real point in keeping it—the same functionality is available from external systems.

Conversely, say a company has developed a system that keeps track of all previous customer orders and automatically generates reminders for customers to reorder goods. This results in a large number of repeat orders and keeps customers satisfied because they feel that their supplier is aware of their needs. The outputs from such a system are important to the business, so this system has a high business value.

To assess a software system from a technical perspective, you need to consider both the application system itself and the environment in which the system operates. The environment includes the hardware and all associated support software such as compilers, debuggers and development environments that are needed to maintain the system. The environment is important because many system changes, such as upgrades to the hardware or operating system, result from changes to the environment.

Factors that you should consider during the environment assessment are shown in Figure 9.10. Notice that these are not all technical characteristics of the environment. You also have to consider the reliability of the suppliers of the hardware and support software. If suppliers are no longer in business, their systems may not be supported, so you may have to replace these systems.

Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly, but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If high costs are associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system?

Figure 9.10 Factors used in environment assessment

In the process of environmental assessment, if possible, you should ideally collect data about the system and system changes. Examples of data that may be useful include the costs of maintaining the system hardware and support software, the number of hardware faults that occur over some time period and the frequency of patches and fixes applied to the system support software.

To assess the technical quality of an application system, you have to assess those factors (Figure 9.11) that are primarily related to the system dependability, the difficulties of maintaining the system, and the system documentation. You may also collect data that will help you judge the quality of the system such as:

1. *The number of system change requests* System changes usually corrupt the system structure and make further changes more difficult. The higher this accumulated value, the lower the quality of the system.
2. *The number of user interfaces* This is an important factor in forms-based systems where each form can be considered as a separate user interface. The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
3. *The volume of data used by the system* As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data. When data has been collected over a long period of time, errors and inconsistencies are inevitable. Cleaning up old data is a very expensive and time-consuming process.

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

Figure 9.11 Factors used in application assessment

Ideally, objective assessment should be used to inform decisions about what to do with a legacy system. However, in many cases, decisions are not really objective but are based on organizational or political considerations. For example, if two businesses merge, the most politically powerful partner will usually keep its systems and scrap the other company's systems. If senior management in an organization decides to move to a new hardware platform, then this may require applications to be replaced. If no budget is available for system transformation in a particular year, then system maintenance may be continued, even though this will result in higher long-term costs.

9.3 Software maintenance

Software maintenance is the general process of changing a system after it has been delivered. The term is usually applied to custom software, where separate development groups are involved before and after delivery. The changes made to the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or to accommodate new requirements. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system.

Program evolution dynamics

Program evolution dynamics is the study of evolving software systems, pioneered by Manny Lehman and Les Belady in the 1970s. This led to so-called Lehman's Laws, which are said to apply to all large-scale software systems. The most important of these laws are:

1. A program must continually change if it is to remain useful.
2. As an evolving program changes, its structure is degraded.
3. Over a program's lifetime, the rate of change is roughly constant and independent of the resources available.
4. The incremental change in each release of a system is roughly constant.
5. New functionality must be added to systems to increase user satisfaction.

http://software-engineering-book.com/web/program-evolution-dynamics/

There are three different types of software maintenance:

1. *Fault repairs to fix bugs and vulnerabilities.* Coding errors are usually relatively cheap to correct; design errors are more expensive because they may involve rewriting several program components. Requirements errors are the most expensive to repair because extensive system redesign may be necessary.
2. *Environmental adaptation to adapt the software to new platforms and environments.* This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes. Application systems may have to be modified to cope with these environmental changes.
3. *Functionality addition to add new features and to support new requirements.* This type of maintenance is necessary when system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

In practice, there is no clear-cut distinction between these types of maintenance. When you adapt a system to a new environment, you may add functionality to take advantage of new environmental features. Software faults are often exposed because users use the system in unanticipated ways. Changing the system to accommodate their way of working is the best way to fix these faults.

These types of maintenance are generally recognized, but different people sometimes give them different names. "Corrective maintenance" is universally used to refer to maintenance for fault repair. However, "adaptive maintenance" sometimes means adapting to a new environment and sometimes means adapting the software to new requirements. "Perfective maintenance" sometimes means perfecting the software by implementing new requirements; in other cases, it means maintaining the functionality of the system but improving its structure and its performance. Because of this naming uncertainty, I have avoided the use of these terms in this book.

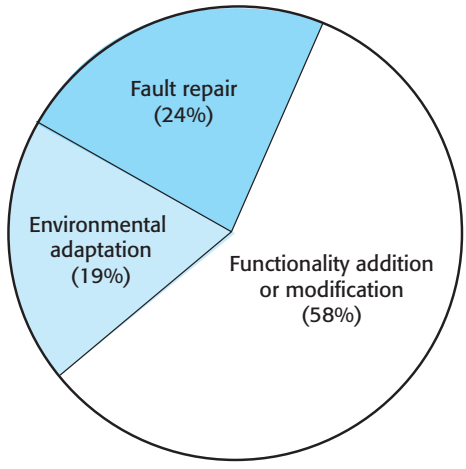

Figure 9.12
Maintenance effort distribution

Figure 9.12 shows an approximate distribution of maintenance costs, based on data from the most recent survey available (Davidsen and Krogstie 2010). This study compared maintenance cost distribution with a number of earlier studies from 1980 to 2005. The authors found that the distribution of maintenance costs had changed very little over 30 years. Although we don't have more recent data, this suggests that this distribution is still largely correct. Repairing system faults is not the most expensive maintenance activity. Evolving the system to cope with new environments and new or changed requirements generally consumes most maintenance effort.

Experience has shown that it is usually more expensive to add new features to a system during maintenance than it is to implement the same features during initial development. The reasons for this are:

1. *A new team has to understand the program being maintained.* After a system has been delivered, it is normal for the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before they can implement changes to it.

2. *Separating maintenance and development means there is no incentive for the development team to write maintainable software.* The contract to maintain a system is usually separate from the system development contract. A different company, rather than the original software developer, may be responsible for software maintenance. In those circumstances, a development team gets no benefit from investing effort to make the software maintainable. If a development team can cut corners to save effort during development it is worthwhile for them to do so, even if this means that the software is more difficult to change in future.

3. *Program maintenance work is unpopular.* Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development

Documentation

System documentation can help the maintenance process by providing maintainers with information about the structure and organization of the system and the features that it offers to system users. While proponents of agile approaches suggest that the code should be the principal documentation, higher level design models and information about dependencies and constraints can make it easier to understand and make changes to that code.

http://software-engineering-book.com/web/documentation/ (web chapter)

and is often allocated to the least experienced staff. Furthermore, old systems may be written in obsolete programming languages. The developers working on maintenance may not have much experience of these languages and must learn these languages to maintain the system.

4. *As programs age, their structure degrades and they become harder to change.* As changes are made to programs, their structure tends to degrade. Consequently, they become harder to understand and change. Some systems have been developed without modern software engineering techniques. They may never have been well structured and were perhaps optimized for efficiency rather than understandability. System documentation may be lost or inconsistent. Old systems may not have been subject to stringent configuration management, so developers have to spend time finding the right versions of system components to change.

The first three of these problems stem from the fact that many organizations still consider software development and maintenance to be separate activities. Maintenance is seen as a second-class activity, and there is no incentive to spend money during development to reduce the costs of system change. The only long-term solution to this problem is to think of systems as evolving throughout their lifetime through a continual development process. Maintenance should have as high a status as new software development.

The fourth issue, the problem of degraded system structure, is, in some ways, the easiest problem to address. Software reengineering techniques (described later in this chapter) may be applied to improve the system structure and understandability. Architectural transformations can adapt the system to new hardware. Refactoring can improve the quality of the system code and make it easier to change.

In principle, it is almost always cost-effective to invest effort in designing and implementing a system to reduce the costs of future changes. Adding new functionality after delivery is expensive because you have to spend time learning the system and analyzing the impact of the proposed changes. Work done during development to structure the software and to make it easier to understand and change will reduce evolution costs. Good software engineering techniques such as precise specification, test-first development, the use of object-oriented development, and configuration management all help reduce maintenance cost.

These principled arguments for lifetime cost savings by investing in making systems more maintainable are, unfortunately, impossible to substantiate with real

data. Collecting data is expensive, and the value of that data is difficult to judge; therefore, the vast majority of companies do not think it is worthwhile to gather and analyze software engineering data.

In reality, most businesses are reluctant to spend more on software development to reduce longer-term maintenance costs. There are two main reasons for their reluctance:

1. Companies set out quarterly or annual spending plans, and managers are incentivized to reduce short-term costs. Investing in maintainability leads to short-term cost increases, which are measurable. However, the long-term gains can't be measured at the same time, so companies are reluctant to spend money on something with an unknown future return.
2. Developers are not usually responsible for maintaining the system they have developed. Consequently, they don't see the point of doing additional work that might reduce maintenance costs, as they will not get any benefit from it.

The only way around this problem is to integrate development and maintenance so that the original development team remains responsible for software throughout its lifetime. This is possible for software products and for companies such as Amazon, which develop and maintain their own software (O'Hanlon 2006). However, for custom software developed by a software company for a client, this is unlikely to happen.

9.3.1 Maintenance prediction

Maintenance prediction is concerned with trying to assess the changes that may be required in a software system and with identifying those parts of the system that are likely to be the most expensive to change. If you understand this, you can design the software components that are most likely to change to make them more adaptable. You can also invest effort in improving those components to reduce their lifetime maintenance costs. By predicting changes, you can also assess the overall maintenance costs for a system in a given time period and so set a budget for maintaining the software. Figure 9.13 shows possible predictions and the questions that these predictions may answer.

Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. Some systems have a very complex relationship with their external environment, and changes to that environment inevitably result in changes to the system. To evaluate the relationships between a system and its environment, you should look at:

1. *The number and complexity of system interfaces* The larger the number of interfaces and the more complex these interfaces, the more likely it is that interface changes will be required as new requirements are proposed.

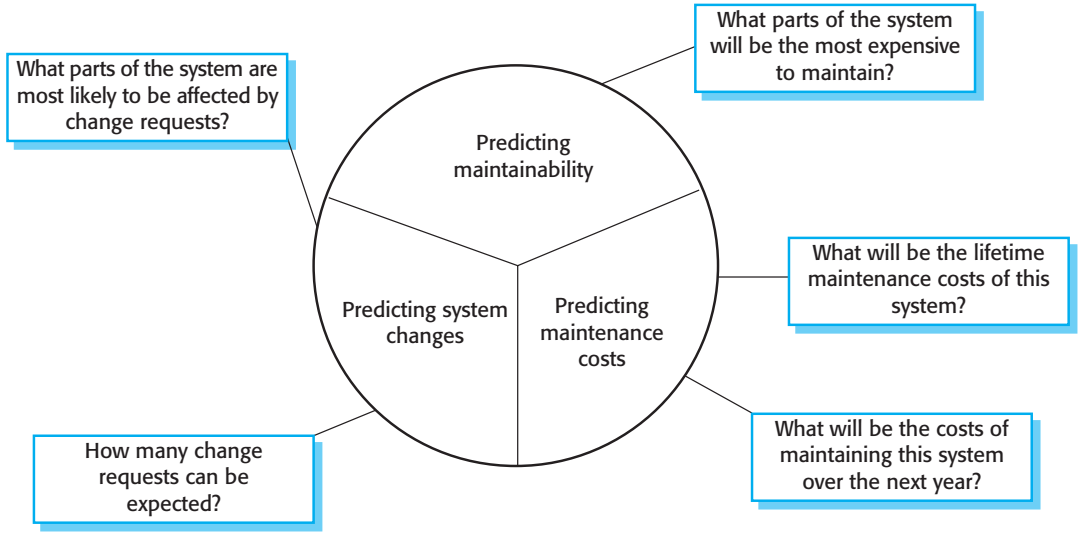

Figure 9.13
Maintenance prediction

2. *The number of inherently volatile system requirements* As I discussed in Chapter 4, requirements that reflect organizational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.

3. *The business processes in which the system is used* As business processes evolve, they generate system change requests. As a system is integrated with more and more business processes, there are increased demands for changes.

In early work on software maintenance, researchers looked at the relationships between program complexity and maintainability (Banker et al. 1993; Coleman et al. 1994; Kozlov et al. 2008). These studies found that the more complex a system or component, the more expensive it is to maintain. Complexity measurements are particularly useful in identifying program components that are likely to be expensive to maintain. Therefore, to reduce maintenance costs you should try to replace complex system components with simpler alternatives.

After a system has been put into service, you may be able to use process data to help predict maintainability. Examples of process metrics that can be used for assessing maintainability are:

1. *Number of requests for corrective maintenance* An increase in the number of bug and failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. This may indicate a decline in maintainability.

2. *Average time required for impact analysis* This is related to the number of program components that are affected by the change request. If the time required for impact analysis increases, it implies that more and more components are affected and maintainability is decreasing.

3. *Average time taken to implement a change request* This is not the same as the time for impact analysis although it may correlate with it. This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected. An increase in the time needed to implement a change may indicate a decline in maintainability.
4. *Number of outstanding change requests* An increase in this number over time may imply a decline in maintainability.

You use predicted information about change requests and predictions about system maintainability to predict maintenance costs. Most managers combine this information with intuition and experience to estimate costs. The COCOMO 2 model of cost estimation, discussed in Chapter 23, suggests that an estimate for software maintenance effort can be based on the effort to understand existing code and the effort to develop the new code.

9.3.2 Software reengineering

Software maintenance involves understanding the program that has to be changed and then implementing any required changes. However, many systems, especially older legacy systems, are difficult to understand and change. The programs may have been optimized for performance or space utilization at the expense of understandability, or, over time, the initial program structure may have been corrupted by a series of changes.

To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability. Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of the system's data. The functionality of the software is not changed, and, normally, you should try to avoid making major changes to the system architecture.

Reengineering has two important advantages over replacement:

1. *Reduced risk* There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.
2. *Reduced cost* The cost of reengineering may be significantly less than the cost of developing new software. Ulrich (Ulrich 1990) quotes an example of a commercial system for which the reimplementation costs were estimated at $50 million. The system was successfully reengineered for $12 million. I suspect that, with modern software technology, the relative cost of reimplementation is probably less than Ulrich's figure but will still be more than the costs of reengineering.

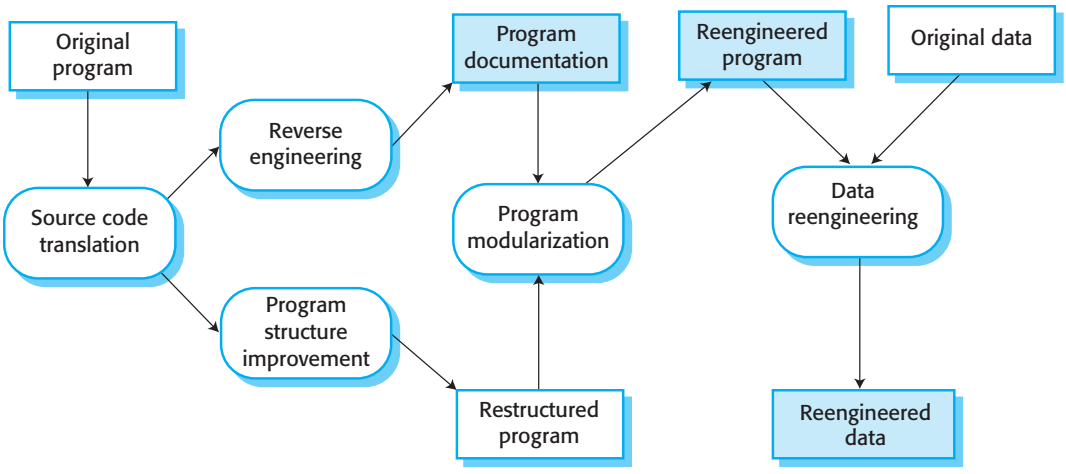

Figure 9.14 The reengineering process

Figure 9.14 is a general model of the reengineering process. The input to the process is a legacy program, and the output is an improved and restructured version of the same program. The activities in this reengineering process are:

1. *Source code translation* Using a translation tool, you can convert the program from an old programming language to a more modern version of the same language or to a different language.
2. *Reverse engineering* The program is analyzed and information extracted from it. This helps to document its organization and functionality. Again, this process is usually completely automated.
3. *Program structure improvement* The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated, but some manual intervention is usually required.
4. *Program modularization* Related parts of the program are grouped together, and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository). This is a manual process.
5. *Data reengineering* The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure. You should usually also clean up the data. This involves finding and correcting mistakes, removing duplicate records, and so on. This can be a very expensive and prolonged process.

Program reengineering may not necessarily require all of the steps in Figure 9.11. You don't need source code translation if you still use the application's programming language. If you can do all reengineering automatically, then recovering documentation through reverse engineering may be unnecessary. Data reengineering is required only if the data structures in the program change during system reengineering.

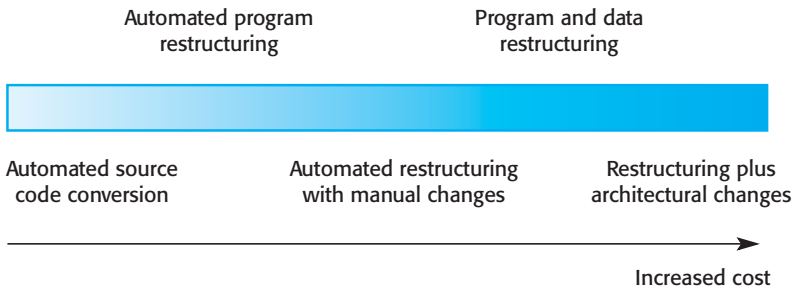

Figure 9.15 Reengineering approaches

To make the reengineered system interoperate with the new software, you may have to develop adaptor services, as discussed in Chapter 18. These hide the original interfaces of the software system and present new, better-structured interfaces that can be used by other components. This process of legacy system wrapping is an important technique for developing large-scale reusable services.

The costs of reengineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to reengineering, as shown in Figure 9.15. Costs increase from left to right so that source code translation is the cheapest option, and reengineering, as part of architectural migration, is the most expensive.

The problem with software reengineering is that there are practical limits to how much you can improve a system by reengineering. It isn't possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes or radical reorganizing of the system data management cannot be carried out automatically, so they are very expensive. Although reengineering can improve maintainability, the reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

9.3.3 Refactoring

Refactoring is the process of making improvements to a program to slow down degradation through change. It means modifying a program to improve its structure, reduce its complexity, or make it easier to understand. Refactoring is sometimes considered to be limited to object-oriented development, but the principles can in fact be applied to any development approach. When you refactor a program, you should not add functionality but rather should concentrate on program improvement. You can therefore think of refactoring as "preventative maintenance" that reduces the problems of future change.

Refactoring is an inherent part of agile methods because these methods are based on change. Program quality is liable to degrade quickly, so agile developers frequently refactor their programs to avoid this degradation. The emphasis on regression testing in agile methods lowers the risk of introducing new errors through refactoring. Any errors that are introduced should be detectable, as previously successful tests should then fail. However, refactoring is not dependent on other "agile activities."

Although reengineering and refactoring are both intended to make software easier to understand and change, they are not the same thing. Reengineering takes place after a system has been maintained for some time, and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Fowler et al. (Fowler et al. 1999) suggest that there are stereotypical situations (Fowler calls them "bad smells") where the code of a program can be improved. Examples of bad smells that can be improved through refactoring include:

1. *Duplicate code* The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
2. *Long methods* If a method is too long, it should be redesigned as a number of shorter methods.
3. *Switch (case) statements* These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
4. *Data clumping* Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program. These can often be replaced with an object that encapsulates all of the data.
5. *Speculative generality* This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Fowler, in both his book and website, also suggests some primitive refactoring transformations that can be used singly or together to deal with bad smells. Examples of these transformations include Extract method, where you remove duplication and create a new method; Consolidate conditional expression, where you replace a sequence of tests with a single test; and Pull up method, where you replace similar methods in subclasses with a single method in a superclass. Interactive development environments, such as Eclipse, usually include refactoring support in their editors. This makes it easier to find dependent parts of a program that have to be changed to implement the refactoring.

Refactoring, carried out during program development, is an effective way to reduce the long-term maintenance costs of a program. However, if you take over a program for maintenance whose structure has been significantly degraded, then it may be practically impossible to refactor the code alone. You may also have to think about design refactoring, which is likely to be a more expensive and difficult problem. Design refactoring involves identifying relevant design patterns (discussed in Chapter 7) and replacing existing code with code that implements these design patterns (Kerievsky 2004).