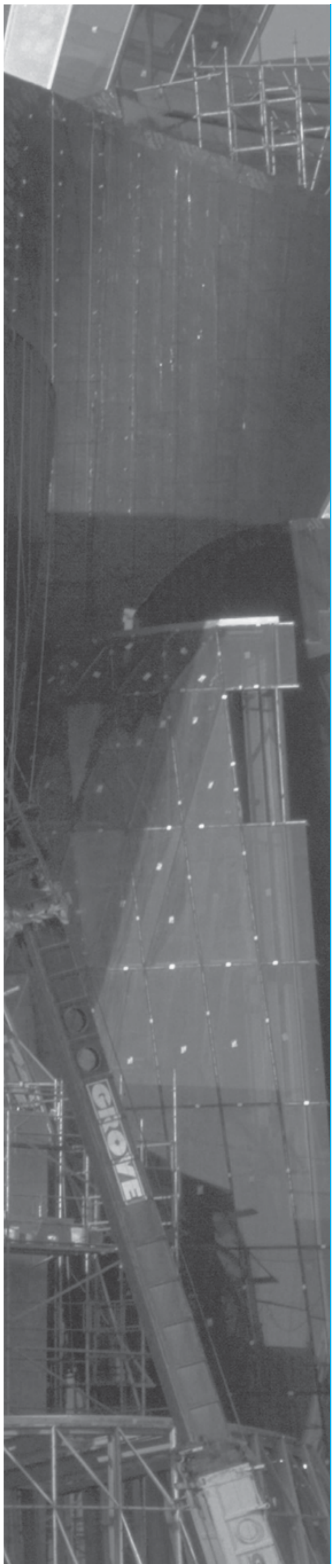

9

# Software evolution

## Objectives

The objectives of this chapter are to explain why software evolution is an important part of software engineering and to describe software evolution processes. When you have read this chapter, you will:

- understand that change is inevitable if software systems are to remain useful and that software development and evolution may be integrated in a spiral model;
- understand software evolution processes and influences on these processes;
- have learned about different types of software maintenance and the factors that affect maintenance costs; and
- understand how legacy systems can be assessed to decide whether they should be scrapped, maintained, reengineered, or replaced.

## Contents

Software development does not stop when a system is delivered but continues throughout the lifetime of the system. After a system has been deployed, it inevitably has to change if it is to remain useful. Business changes and changes to user expectations generate new requirements for the existing 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 evolution is important because organizations have invested large amounts of money in their software and are now completely dependent on these systems. Their systems are critical business assets and they have to invest in system change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development. Based on an informal industry poll, Erlikh (2000) suggests that 85–90% of organizational software costs are evolution costs. Other surveys suggest that about two-thirds of software costs are evolution costs. For sure, the costs of software change are a large part of the IT budget for all companies.

Software evolution may be triggered by changing business requirements, by reports of software defects, or by changes to other systems in a software system's environment. 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 many other software systems.

Therefore, the evolution of a system can rarely be considered in isolation. Changes to the environment lead to system change that may then trigger further environmental changes. Of course, the fact that systems have to evolve in a 'systems-rich' environment often increases the difficulties and costs of evolution. As well as understanding and analyzing an impact of a proposed change on the system itself, you may also have to assess how this may affect other systems in the operational environment.

Useful software systems often have a very long lifetime. For example, large 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. Software cost a lot of money so a company has to use a software system for many years to get a return on its investment. Obviously, the requirements of the installed systems change as the business and its environment change. Therefore, new releases of the systems, incorporating changes, and updates, are usually created at regular intervals.

You should, therefore, think of software engineering as 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 that later releases of the software may be under development before the current version has been released.

This model of software evolution implies that a single organization is responsible for both the initial software development and the evolution of the software. Most

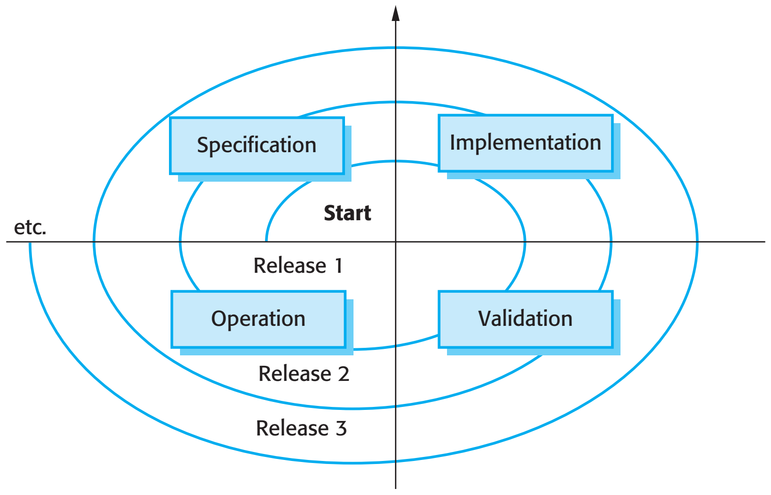

**Figure 9.1** A spiral model of development and evolution

packaged software products are developed using this approach. For custom software, a different approach is commonly used. A software company develops software for a customer and the customer's own development staff then take over the system. They are responsible for software evolution. Alternatively, the software customer might issue a separate contract to a different company for system support and evolution.

In this case, there are likely to be discontinuities in the spiral process. Requirements and design documents may not be passed from one company to another. Companies may merge or reorganize and 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 often 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 (2000) proposed an alternative view of the software evolution life cycle, as shown in Figure 9.2. In this model, they distinguish between evolution and servicing. Evolution is the phase in which significant changes to the software architecture and functionality may be made. During servicing, the only changes that are made are relatively small, essential changes.

During evolution, the software is used successfully and there is a constant stream of proposed requirements changes. However, as the software is modified, its structure tends to degrade and 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 often required. At some stage in the life cycle, the software reaches a transition point where significant changes, implementing new requirements, become less and less cost effective.

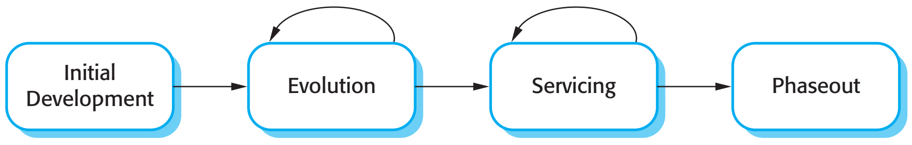

**Figure 9.2** Evolution and servicing

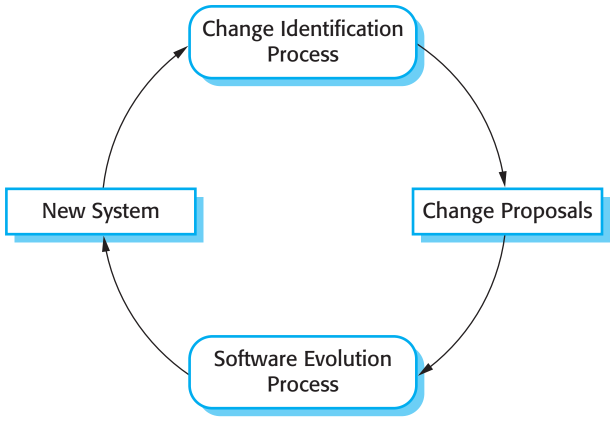

**Figure 9.3** Change identification and evolution processes

At that stage, the software moves from evolution to servicing. During the servicing phase, the software is still useful and used 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, phase-out, the software may still be used but no further changes are being implemented. Users have to work around any problems that they discover.

## 9.1 Evolution processes

Software evolution processes vary depending on the type of software being maintained, the development processes used in an organization and the skills of the people involved. In some organizations, evolution may be an informal process where change requests mostly come from conversations between the system users and developers. In other companies, it is a formalized process with structured documentation produced at each stage in the process.

System change proposals are the driver for system evolution in all organizations. Change proposals may come from 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 cyclic and continue throughout the lifetime of a system (Figure 9.3).

Change proposals should be linked to the components of the system that have to be modified to implement these proposals. This allows the cost and the impact of the change to be assessed. This is part of the general process of change management, which also should ensure that the correct versions of components are included in each system release. I cover change and configuration management in Chapter 25.

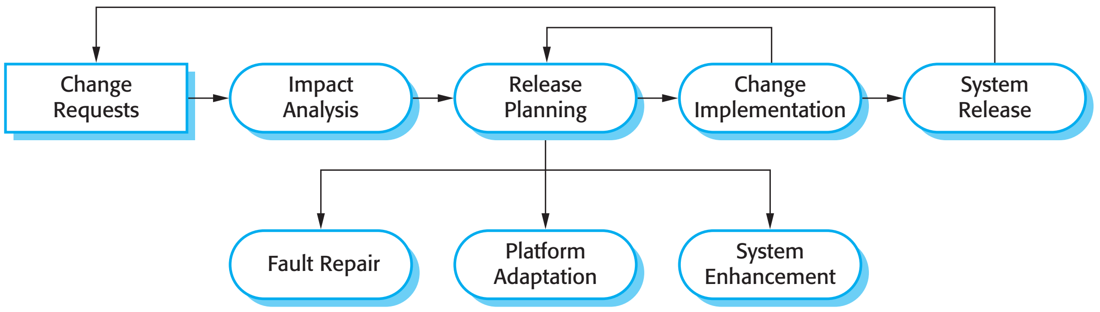

**Figure 9.4** The software evolution process

Figure 9.4, adapted from Arthur (1988), shows an overview of the evolution process. 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.

You can think of change implementation as an iteration of the development process, where the revisions to the system are designed, implemented, and tested. However, a critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for change implementation. During this program understanding phase, you have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. You need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

Ideally, the change implementation stage of this process should modify the system specification, design, and implementation to reflect the changes to the system (Figure 9.5). New requirements that reflect the system changes are proposed, analyzed, and validated. System components are redesigned and implemented and the system is retested. If appropriate, prototyping of the proposed changes may be carried out as part of the change analysis process.

During the evolution process, the requirements are analyzed in detail and implications of the changes emerge that were not apparent in the earlier change analysis process. This means that the proposed changes may be modified and further customer discussions may be required before they are implemented.

Change requests sometimes relate to system problems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault occurs that has to be repaired to allow normal operation to continue.

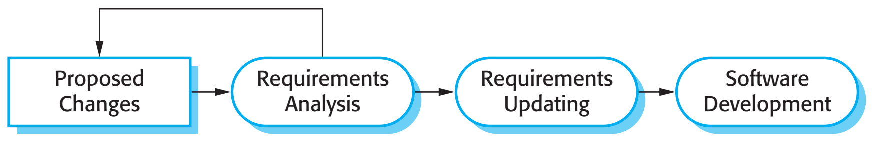

**Figure 9.5** Change implementation

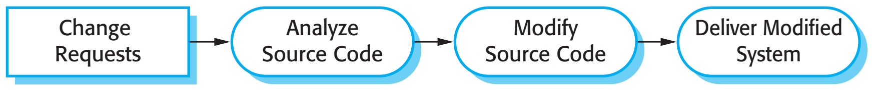

**Figure 9.6** The emergency repair process

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 follow the formal change analysis process. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 9.6). However, the danger is that the requirements, the software design, and the code become inconsistent. Although 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.

Emergency system repairs usually have to be completed as quickly as possible. You chose a quick and workable solution rather than the best solution as far as system structure is concerned. This accelerates the process of software ageing so that future changes become progressively more difficult and maintenance costs increase.

Ideally, when emergency code repairs are made the change request should remain outstanding after the code faults have been fixed. It can then be reimplemented more carefully after further analysis. Of course, the code of the repair may be reused. An alternative, better solution to the problem may be discovered when more time is available for analysis. In practice, however, it is almost inevitable that these improvements will have a low priority. They are often forgotten and, if further system changes are made, it then becomes unrealistic to redo the emergency repairs.

Agile methods and processes, discussed in Chapter 3, may be used for program evolution as well as program development. In fact, because these methods are based on incremental development, making the transition from agile development to post-delivery evolution should be seamless. Techniques such as automated regression testing are useful when system changes are made. Changes may be expressed as user stories and customer involvement can prioritize changes that are required in an operational system. In short, evolution simply involves continuing the agile development process.

However, problems may arise in situations in which there is a handover from a development team to a separate team responsible for evolution. There are two potentially problematic situations:

1. Where the development team has used an agile approach but the evolution team is unfamiliar with agile methods and 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 and the code in the system may not have been refactored and simplified as is expected in agile development. In this case, some reengineering may be required to improve the code before it can be used in an agile development process.

Poole and Huisman (2001) report on their experiences in using Extreme Programming for maintaining a large system that was originally developed using a plan-based approach. After reengineering the system to improve its structure, XP was used very successfully in the maintenance process.

## 9.2 Program evolution dynamics

Program evolution dynamics is the study of system change. In the 1970s and 1980s, Lehman and Belady (1985) carried out several empirical studies of system change with a view to understanding more about characteristics of software evolution. The work continued in the 1990s as Lehman and others investigated the significance of feedback in evolution processes (Lehman, 1996; Lehman et al., 1998; Lehman et al., 2001). From these studies, they proposed ‘Lehman’s laws’ concerning system change (Figure 9.7).

Lehman and Belady claim these laws are likely to be true for all types of large organizational software systems (what they call E-type systems). These are systems in which the requirements are changing to reflect changing business needs. New releases of the system are essential for the system to provide business value.

The first law states that system maintenance is an inevitable process. As the system’s environment changes, new requirements emerge and the system must be modified. When the modified system is reintroduced to the environment, this promotes more environmental changes, so the evolution process starts again.

The second law states that, as a system is changed, its structure is degraded. The only way to avoid this happening is to invest in preventative maintenance. You spend time improving the software structure without adding to its functionality. Obviously, this means additional costs, over and above those of implementing required system changes.

| Law | Description |
|---|---|
| Continuing change | A program that is used in a real-world environment must necessarily change, or else become progressively less useful in that environment. |
| Increasing complexity | As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure. |
| Large program evolution | Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release. |
| Organizational stability | Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development. |
| Conservation of familiarity | Over the lifetime of a system, the incremental change in each release is approximately constant. |
| Continuing growth | The functionality offered by systems has to continually increase to maintain user satisfaction. |
| Declining quality | The quality of systems will decline unless they are modified to reflect changes in their operational environment. |
| Feedback system | Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement. |

**Figure 9.7** Lehman's laws

The third law is, perhaps, the most interesting and the most contentious of Lehman's laws. It suggests that large systems have a dynamic of their own that is established at an early stage in the development process. This determines the gross trends of the system maintenance process and limits the number of possible system changes. Lehman and Belady suggest that this law is a consequence of structural factors that influence and constrain system change, and organizational factors that affect the evolution process.

The structural factors that affect the third law come from the complexity of large systems. As you change and extend a program, its structure tends to degrade. This is true of all types of system (not just software) and it occurs because you are adapting a structure intended for one purpose for a different purpose. This degradation, if unchecked, makes it more and more difficult to make further changes to the program. Making small changes reduces the extent of structural degradation and so lessens the risks of causing serious system dependability problems. If you try and make large changes, there is a high probability that these will introduce new faults. These then inhibit further program changes.

The organizational factors that affect the third law reflect the fact that large systems are usually produced by large organizations. These companies have internal bureaucracies that set the change budget for each system and control the decision-making process. Companies have to make decisions on the risks and value of the changes and the costs involved. Such decisions take time to make and, sometimes, it takes longer to decide on the changes to be made than change implementation. The speed of the organization's decision-making processes therefore governs the rate of change of the system.

Lehman's fourth law suggests that most large programming projects work in a 'saturated' state. That is, a change to resources or staffing has imperceptible effects on the long-term evolution of the system. This is consistent with the third law, which suggests that program evolution is largely independent of management decisions. This law confirms that large software development teams are often unproductive because communication overheads dominate the work of the team.

Lehman's fifth law is concerned with the change increments in each system release. Adding new functionality to a system inevitably introduces new system faults. The more functionality added in each release, the more faults there will be. Therefore, a large increment in functionality in one system release means that this will have to be followed by a further release in which the new system faults are repaired. Relatively little new functionality should be included in this release. This law suggests that you should not budget for large functionality increments in each release without taking into account the need for fault repair.

The first five laws were in Lehman's initial proposals; the remaining laws were added after further work. The sixth and seventh laws are similar and essentially say that users of software will become increasingly unhappy with it unless it is maintained and new functionality is added to it. The final law reflects the most recent work on feedback processes, although it is not yet clear how this can be applied in practical software development.

Lehman's observations seem generally sensible. They should be taken into account when planning the maintenance process. It may be that business considerations require them to be ignored at any one time. For example, for marketing reasons, it may be necessary to make several major system changes in a single release. The likely consequences of this are that one or more releases devoted to error repair are likely to be required. You often see this in personal computer software when a major new release of an application is often quickly followed by a bug repair update.

## 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 in which 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 accommodate new requirements. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system.

There are three different types of software maintenance:

1. *Fault repairs* Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.

2. *Environmental adaptation* This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system, or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.

3. *Functionality addition* This type of maintenance is necessary when the 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 not a clear-cut distinction between these types of maintenance. When you adapt the 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 all of these terms in this chapter.

There have been several studies of software maintenance which have looked at the relationships between maintenance and development and between different maintenance activities (Krogstie et al., 2005; Lientz and Swanson, 1980; Nosek and Palvia, 1990; Sousa, 1998). Because of differences in terminology, the details of these studies cannot be compared. In spite of changes in technology and different application domains, it seems that there has been remarkably little change in the distribution of evolution effort since the 1980s.

The surveys broadly agree that software maintenance takes up a higher proportion of IT budgets than new development (roughly two-thirds maintenance, one-third development). They also agree that more of the maintenance budget is spent on implementing new requirements than on fixing bugs. Figure 9.8 shows an approximate distribution of maintenance costs. The specific percentages will obviously vary from one organization to another but, universally, repairing system faults is not the most expensive maintenance activity. Evolving the system to cope with new environments and new or changed requirements consumes most maintenance effort.

The relative costs of maintenance and new development vary from one application domain to another. Guimaraes (1983) found that the maintenance costs for business application systems are broadly comparable with system development costs.

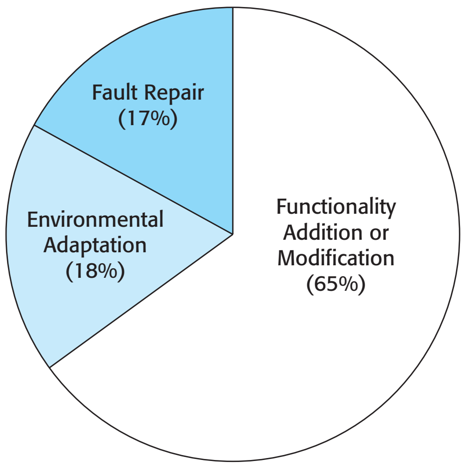

**Figure 9.8**
Maintenance effort distribution

For embedded real-time systems, maintenance costs were up to four times more than development costs. The high reliability and performance requirements of these systems mean that modules have to be tightly linked and hence difficult to change. Although these estimates are more than 25 years old, it is unlikely that the cost distributions for different types of system have significantly changed.

It is usually 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. Therefore, work done during development to make the software easier to understand and change is likely to reduce evolution costs. Good software engineering techniques, such as precise specification, the use of object-oriented development, and configuration management, contribute to maintenance cost reduction.

Figure 9.9 shows how overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system. Because of the potential reduction in costs of understanding, analysis, and testing, there is a significant multiplier effect when the system is developed for maintainability. For System 1, extra development costs of $25,000 are invested in making the system more maintainable. This results in a savings of $100,000 in maintenance costs over

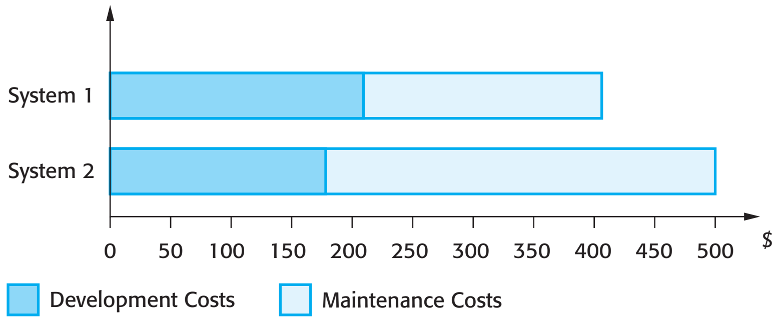

**Figure 9.9**
Development and maintenance costs

**Legacy systems**

Legacy systems are old systems that are still useful and are sometimes critical to business operation. They may be implemented using outdated languages and technology or may use other systems that are expensive to maintain. Often their structure has been degraded by change and documentation is missing or out of date. Nevertheless, it may not be cost effective to replace these systems. They may only be used at certain times of the year or it may be too risky to replace them because the specification has been lost.

**http://www.SoftwareEngineering-9.com/Web/LegacySys/**

the lifetime of the system. This assumes that a percentage increase in development costs results in a comparable percentage decrease in overall system costs.

These estimates are hypothetical but there is no doubt that developing software to make it more maintainable is cost effective, when the whole life costs of the software are taken into account. This is the rationale for refactoring in agile development. Without refactoring, the code becomes more and more difficult and expensive to change. However, in plan-based development, the reality is that additional investment in code improvement is rarely made during development. This is mostly due to the ways most organizations run their budgets. Investing in maintainability leads to short-term cost increases, which are measurable. Unfortunately, the long-term gains can't be measured at the same time so companies are reluctant to spend money for an unknown future return.

It is usually more expensive to add functionality after a system is in operation than it is to implement the same functionality during development. The reasons for this are:

1. *Team stability* 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 implementing changes to it.

2. *Poor development practice* The contract to maintain a system is usually separate from the system development contract. The maintenance contract may be given to a different company rather than the original system developer. This factor, along with the lack of team stability, means that there is no incentive for a development team to write maintainable software. 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 the future.

3. *Staff skills* Maintenance staff are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers. It is seen as a less-skilled process than system development and is often allocated to the most junior staff. Furthermore, old systems may be written in obsolete programming languages. The maintenance staff may not have much experience of development in these languages and must learn these languages to maintain the system.

**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. Although proponents of agile approaches such as XP 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 the code.

I have written a separate chapter on documentation that you can download.

**http://www.SoftwareEngineering-9.com/Web/ExtraChaps/Documentation.pdf**

4. *Program age and structure* As changes are made to programs, their structure tends to degrade. Consequently, as programs age, 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 time is often wasted finding the right versions of system components to change.

The first three of these problems stem from the fact that many organizations still consider 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 accept that systems rarely have a defined lifetime but continue in use, in some form, for an indefinite period. As I suggested in the introduction, you should think of systems as evolving throughout their lifetime through a continual development process.

The fourth issue, the problem of degraded system structure, is 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.

### 9.3.1 Maintenance prediction

Managers hate surprises, especially if these result in unexpectedly high costs. You should therefore try to predict what system changes might be proposed and what parts of the system are likely to be the most difficult to maintain. You should also try to estimate the overall maintenance costs for a system in a given time period. Figure 9.10 shows these predictions and associated questions.

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

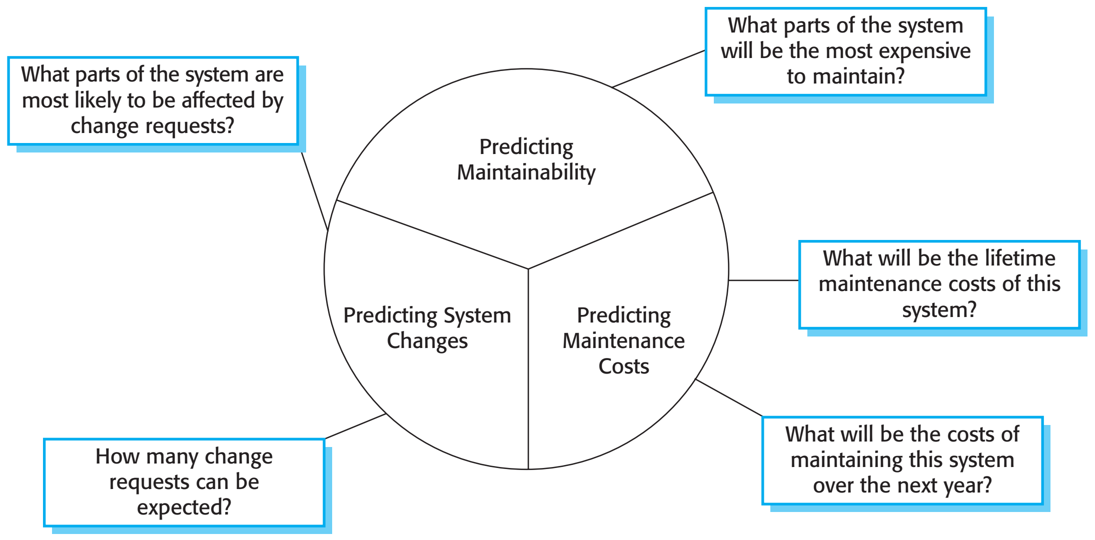
What parts of the system are most likely to be affected by change requests?
What parts of the system will be the most expensive to maintain?
Predicting Maintainability
What will be the lifetime maintenance costs of this system?
Predicting System Changes
Predicting Maintenance Costs
What will be the costs of maintaining this system over the next year?
How many change requests can be expected?

**Figure 9.10**
Maintenance prediction

environment inevitably result in changes to the system. To evaluate the relationships between a system and its environment, you should assess:

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.

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. The more business processes that use a system, the more the demands for system change.

For many years, researchers have looked at the relationships between program complexity, as measured by metrics such as cyclomatic complexity (McCabe, 1976), and maintainability (Banker et al., 1993; Coleman et al., 1994; Kafura and Reddy, 1987; Kozlov et al., 2008). It is not surprising that these studies have 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. Kafura and Reddy (1987) examined a number of system components and found that maintenance effort tended to be focused on a small number of complex components. To reduce maintenance costs, therefore, 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 as follows:

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 reflects the number of program components that are affected by the change request. If this time increases, it implies 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 (Boehm et al., 2000), discussed in Chapter 24, 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

As I discussed in the previous section, the process of system evolution involves understanding the program that has to be changed and then implementing these 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, and 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.

There are two important benefits from reengineering rather than 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.

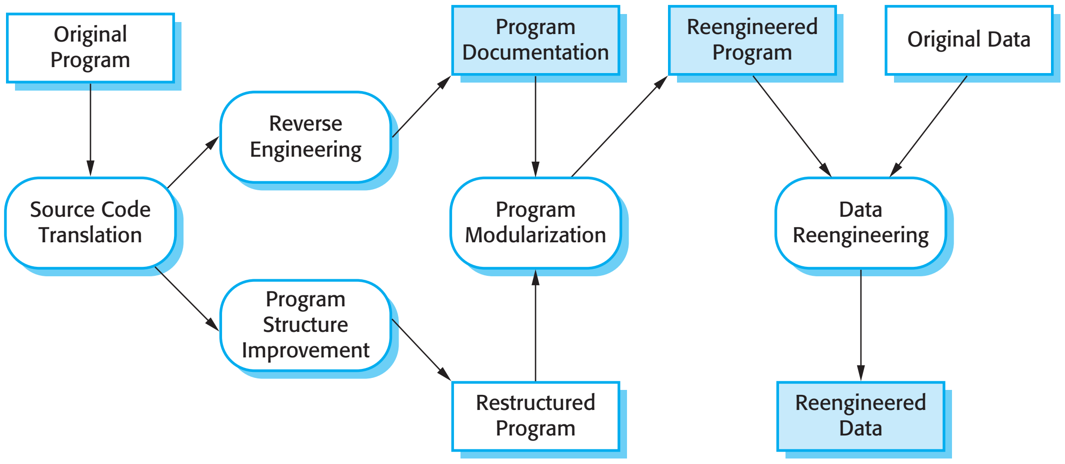

**Figure 9.11** The reengineering process

2. *Reduced cost* The cost of reengineering may be significantly less than the cost of developing new software. 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 this but will still considerably exceed the costs of reengineering.

Figure 9.11 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 as follows:

1. *Source code translation* Using a translation tool, the program is converted 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

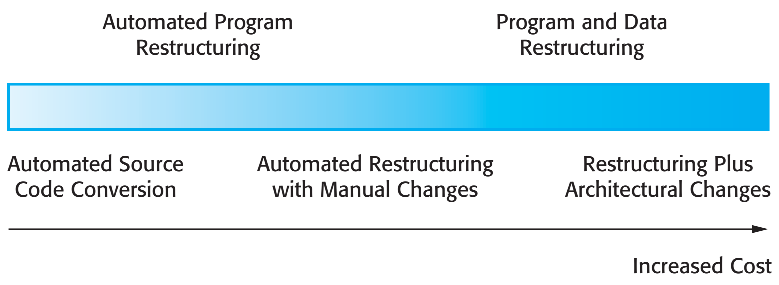
Automated Program Restructuring
Program and Data Restructuring
Automated Source Code Conversion
Automated Restructuring with Manual Changes
Restructuring Plus Architectural Changes
Increased Cost

**Figure 9.12** Reengineering approaches

data. This involves finding and correcting mistakes, removing duplicate records, etc. Tools are available to support data reengineering.

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 only required if the data structures in the program change during system reengineering.

To make the reengineered system interoperate with the new software, you may have to develop adaptor services, as discussed in Chapter 19. 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 9.12. Costs increase from left to right so that source code translation is the cheapest option. 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 Preventative maintenance by refactoring

Refactoring is the process of making improvements to a program to slow down degradation through change (Opdyke and Johnson, 1990). It means modifying a program to improve its structure, to reduce its complexity, or to make it easier to understand. Refactoring is sometimes considered to be limited to object-oriented development but the principles can be applied to any development approach. When you refactor a program, you should not add functionality but should concentrate on program improvement. You can therefore think of refactoring as 'preventative maintenance' that reduces the problems of future change.

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.

Refactoring is an inherent part of agile methods such as extreme programming because these methods are based around change. Program quality is therefore 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' and can be used with any approach to development.

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

1. *Duplicate code* The same of 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 some 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) reoccur in several places in a program. These can often be replaced with an object encapsulating all of the data.

5. *Speculative generality* This occurs when developers include generality in a program in case it is required in future. This can often simply be removed.

Fowler, in his book and website, also suggests some primitive refactoring transformations that can be used singly or together to deal with the 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 super class. Interactive development environments, such as Eclipse, 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). I don't have space to discuss this here.

## 9.4 Legacy system management

For new software systems developed using modern software engineering processes, such as incremental development and CBSE, it is possible to plan how to integrate system development and evolution. More and more companies are starting to understand that the system development process is a whole life-cycle process and that an artificial separation between software development and software maintenance is unhelpful. However, there are still many legacy systems that are critical business systems. These have to be extended and adapted to changing e-business practices.

Most organizations usually have a portfolio of legacy systems that they use, with a limited budget for maintaining and upgrading these 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 commonly 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 a 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 in which major system components are replaced by off-the-shelf systems with other components reused wherever possible.

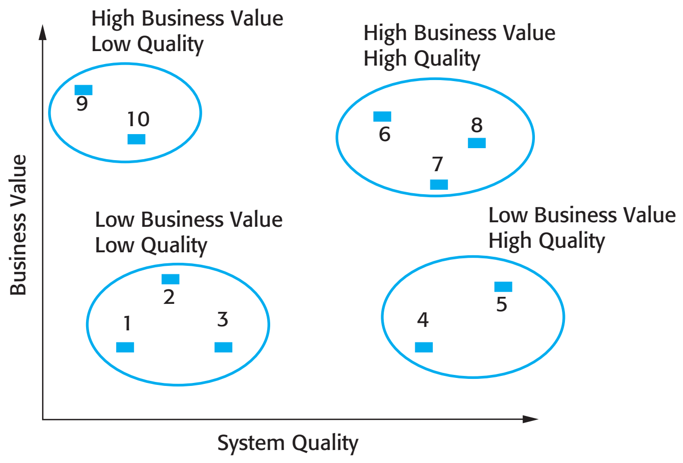

**Figure 9.13** An example of a legacy system assessment

Naturally, these options are not exclusive. When a system is composed of several programs, different options may be applied to each program.

When you are assessing a legacy system, you have to look at it from 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. This is shown in Figure 9.13.

From Figure 9.13, 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 it is expensive to maintain them. These systems should be reengineered to improve their quality. They may be replaced, if a suitable off-the-shelf system is available.

3. *High quality, low business value* These are systems that don't contribute much to the business but which 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.

To assess the business value of a system, you have to identify system stakeholders, such as end-users of the 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 systems are only used occasionally or by a small number of people, they may have a low business value. A legacy system may have been developed to meet a business need that has either changed or that can now be met more effectively in other ways. You have to be careful, however, about occasional but important use of systems. For example, in a university, a student registration system may only be used at the beginning of each academic year. However, 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 designed 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. *The 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 the 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 easily generated in some other way or if the system produces outputs that are rarely used, then its business value may be low.

For example, let's 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 these. 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

| 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 there are high costs 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? Is hardware emulation required? |

**Figure 9.14** Factors used in environment assessment

because they feel that their supplier is aware of their needs. The outputs from such a system are very important to the business and this system therefore 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 (compilers, development environments, etc.) that are required to maintain the system. The environment is important because many system changes result from changes to the environment, such as upgrades to the hardware or operating system.

If possible, in the process of environmental assessment, you should make measurements of the system and its maintenance processes. 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.

Factors that you should consider during the environment assessment are shown in Figure 9.14. 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 these suppliers are no longer in business, there may not be support for their systems.

To assess the technical quality of an application system, you have to assess a range of factors (Figure 9.15) that are primarily related to the system dependability,

| 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 9.15** Factors used in application assessment

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. Data that may be useful in quality assessment are:

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 that there will be inconsistencies and redundancies in these interfaces.

3. *The volume of data used by the system* The higher the volume of data (number of files, size of database, etc.), the more likely that it is that there will be data inconsistencies that reduce the system quality.

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 systems. If senior management in an organization decide to move to a new hardware platform, then this may require applications to be replaced. If there is no budget available for system transformation in a particular year, then system maintenance may be continued, even though this will result in higher long-term costs.

## KEY POINTS

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning, and change implementation.
- Lehman's laws, such as the notion that change is continuous, describe a number of insights derived from long-term studies of system evolution.
- There are three types of software maintenance, namely bug fixing, modifying the software to work in a new environment, and implementing new or changed requirements.
- Software reengineering is concerned with restructuring and redocumenting software to make it easier to understand and change.
- Refactoring, making small program changes that preserve functionality, can be thought of as preventative maintenance.
- The business value of a legacy system and the quality of the application software and its environment should be assessed to determine whether the system should be replaced, transformed, or maintained.

## FURTHER READING

'Software Maintenance and Evolution: A Roadmap'. As well as discussing research challenges, this paper is a good, short overview of software maintenance and evolution by leading researchers in this area. The research problems that they identify have not yet been solved. (V. Rajlich and K.H. Bennett, *Proc. 20th Int. Conf. on Software Engineering*, IEEE Press, 2000.) http://doi.acm.org/10.1145/336512.336534.

*Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. This excellent book covers general issues of software maintenance and evolution as well as legacy system migration. The book is based on a large case study of the transformation of a COBOL system to a Java-based client-server system. (R. C. Seacord, D. Plakosh and G. A. Lewis, Addison-Wesley, 2003.)

*Working Effectively with Legacy Code*. Solid practical advice on the problems and difficulties of dealing with legacy systems. (M. Feathers, John Wiley & Sons, 2004.)

## EXERCISES

**9.1.** Explain why a software system that is used in a real-world environment must change or become progressively less useful.

**9.2.** Explain the rationale underlying Lehman's laws. Under what circumstances might the laws break down?

**9.3.** From Figure 9.4, you can see that impact analysis is an important subprocess in the software evolution process. Using a diagram, suggest what activities might be involved in change impact analysis.

**9.4.** As a software project manager in a company that specializes in the development of software for the offshore oil industry, you have been given the task of discovering the factors that affect the maintainability of the systems developed by your company. Suggest how you might set up a program to analyze the maintenance process and discover appropriate maintainability metrics for your company.

**9.5.** Briefly describe the three main types of software maintenance. Why is it sometimes difficult to distinguish between them?

**9.6.** What are the principal factors that affect the costs of system reengineering?

**9.7.** Under what circumstances might an organization decide to scrap a system when the system assessment suggests that it is of high quality and of high business value.

**9.8.** What are the strategic options for legacy system evolution? When would you normally replace all or part of a system rather than continue maintenance of the software?

**9.9.** Explain why problems with support software might mean that an organization has to replace its legacy systems.

**9.10.** Do software engineers have a professional responsibility to produce code that can be maintained and changed even if this is not explicitly requested by their employer?

## REFERENCES

Arthur, L. J. (1988). *Software Evolution*. New York: John Wiley & Sons.

Banker, R. D., Datar, S. M., Kemerer, C. F. and Zweig, D. (1993). 'Software Complexity and Maintenance Costs'. *Comm. ACM*, **36** (11), 81–94.

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall.