27 Software change

Objectives

The objectives of this chapter are to introduce software change and to describe a number of ways of modifying software. When you have read this chapter, you will:

- understand three different strategies for changing software systems, namely software maintenance, architectural evolution and software re-engineering;
- understand the principles of software maintenance and why software is so expensive to maintain;
- understand how legacy systems may be transformed to distributed client–server systems to extend their life and to make effective use of modern hardware.

Contents

27.1 Program evolution dynamics
27.2 Software maintenance
27.3 Architectural evolution

It is impossible to produce systems of any size which do not need to be changed. Once software is put into use, new requirements emerge and existing requirements change as the business running that software changes. Parts of the software may have to be modified to correct errors that are found in operation, improve its performance or other non-functional characteristics. All of this means that, after delivery, software systems always evolve in response to demands for change.

Software change is very important because organisations are now completely dependent on their software systems and have invested millions of dollars in these systems. Their systems are critical business assets and they must invest in system change to maintain the value of these assets. As I suggested in Chapter 1, a key problem for organisations is implementing and managing change to their legacy systems so that they continue to support their business operations.

There are a number of different strategies for software change (Warren, 1998):

1. *Software maintenance* Changes to the software are made in response to changed requirements but the fundamental structure of the software remains stable. This is the most common approach used to system change. I discuss maintenance later in this chapter in section 27.2.
2. *Architectural transformation* This is a more radical approach to software change then maintenance as it involves making significant changes to the architecture of the software system. Most commonly, systems evolve from a centralised, data-centric architecture to a client–server architecture. I discuss architectural evolution in section 27.3.
3. *Software re-engineering* This is different from other strategies in that no new functionality is added to the system. Rather, the system is modified to make it easier to understand and change. System re-engineering may involve some structural modifications but does not usually involve major architectural change. I cover software re-engineering in Chapter 28.

The above strategies are not mutually exclusive. Re-engineering may be necessary to make the software easier to understand before its architecture is changed or components are reused. Some parts of a system may be replaced by off-the-shelf components while other, stable, parts are maintained. As I discussed in Chapter 26, the choice of the most appropriate strategy depends on both the technical quality of the system and its business value.

Equally, different strategies may be applied to different parts of a system or to different programs that make up a legacy information system. A program which is well structured and which does not need frequent maintenance may be maintained; another program which is used by many different people in different locations may have its architecture modified so that its user interface runs on a client computer, and a third program in the same system may be replaced by an off-the-shelf alternative. However, if the system data is re-engineered this will normally require changes to all of the programs in the system.

For some systems, none of these change strategies is appropriate and you have to replace the system. As I discussed in Chapter 26, it may be possible to replace part or all of the system with an off-the-shelf (COTS) system. However, there may be no suitable COTS alternative and you therefore have to develop a new customised system. In such a situation, it may be possible to reuse major parts of the original system in the development of the replacement.

Software change inevitably generates many different versions of a software systems and its components. It is critically important to keep track of these different versions and to ensure that the appropriate versions of components are used in each system version. The management of changing software products is called configuration management. I discuss this in Chapter 29.

27.1 Program evolution dynamics

Program evolution dynamics is the study of system change. The majority of work in this area has been carried out by Lehman and Belady (1985). From these studies, they proposed a set of 'laws' (Lehman's Laws) concerning system change. They claim these 'laws' are invariant and widely applicable. Lehman and Belady examined the growth and evolution of a number of large software systems. The proposed laws were derived from these measurements. The laws (hypotheses, really) are shown in Figure 27.1.

Figure 27.1
Lehman's Laws

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or 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 are approximately invariant for each system release.
Organisational 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.

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 into the environment, this promotes more environmental changes so the evolution process recycles.

The second law states that, as a system is changed, its structure is degraded. This is commonly observed in legacy systems as I discussed in Chapter 26. The only way to avoid this happening is to invest in preventative maintenance where 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.

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 result of fundamental structural and organisational factors.

Once a system exceeds some minimal size it acts in the same way as an inertial mass. Its size inhibits major change because changes introduce new faults that degrade the functionality of the system. If a large change increment is proposed, this will introduce many new faults that will limit the useful change delivered in the new version of the system.

Large systems are usually produced by large organisations. These have their own internal bureaucracies that set the change budget for each system and control the decision-making process. Major system changes require organisational decision making and changes to the project budget. Such decisions take time to make. During that time, other, higher-priority system changes may be proposed. It may be necessary to shelve the original changes to a later date. The rate of change of the system is therefore governed by the organisation's decision-making processes.

Lehman's fourth law suggests that most large programming projects work in what he terms a 'saturated' state. That is, a change to resources or staffing has imperceptible effects on the long-term evolution of the system. Of course, this is also suggested by the third law which suggests that program evolution is largely independent of management decisions. This law confirms that large software development teams are unproductive as the 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 where the new system faults are repaired. Relatively little new functionality will be included in this release. The law suggests that you should not budget for large functionality increments in each release without taking into account the need for fault repair.

Lehman's observations seem generally sensible. They should be taken into account when planning the maintenance process. It may be that business consider-

ations require them to be ignored at any one time. For example, for marketing reasons, it may 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.

It may appear that the radical differences which are obvious between releases of program products violate Lehman's Laws. For example, Microsoft Word has been transformed from a simple word processor which operated in 256K of memory to a gigantic, feature-laden system. It now needs many megabytes of memory and a fast processor to operate. Its evolution seems to contradict the fourth and fifth of Lehman's laws. However, I suspect that this program is not really a sequence of revisions. Rather, the same name has been retained for marketing reasons but the program itself has been largely rewritten between releases.

27.2 Software maintenance

Software maintenance is the general process of changing a system after it has been delivered. The changes 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. As I said in the introduction, software maintenance does not normally involve major architectural changes to the system. 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. *Maintenance to repair software faults* Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve the rewriting of several program components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.
2. *Maintenance to adapt the software to a different operating environment* 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. *Maintenance to add to or modify the system's functionality* This type of maintenance is necessary when the system requirements change in response to organisational 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 isn't a clear-cut distinction between these different types of maintenance. Software faults may be revealed because a system has been used

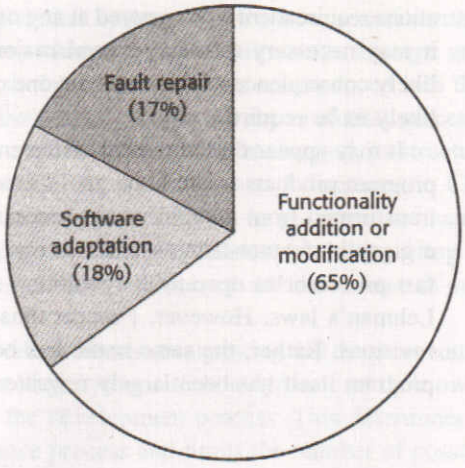


Figure 27.2
Maintenance effort
distribution

in an unanticipated way and the best way to repair these faults may be to add new functionality to help users with the system. When adapting the software to a new environment, functionality may be added to take advantage of new facilities supported by the environment. Adding new functionality to a system may be necessary because faults have changed the usage patterns of the system and a side-effect of the new functionality is to remove the faults from the software.

While these different types of maintenance are generally recognised, 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 and, in other cases, 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.

It is difficult to find up-to-date figures for the relative effort devoted to the different types of maintenance. A rather old survey by Lientz and Swanson (1980) discovered that about 65 per cent of maintenance was concerned with implementing new requirements, 18 per cent with changing the system to adapt it to a new operating environment and 17 per cent to correct system faults (Figure 27.2). Similar figures were reported by Nosek and Palvia (1990) 10 years later. For custom systems, I guess that this distribution of costs is still roughly correct.

From these figures we can see that repairing system faults is not the most expensive maintenance activity. Rather, evolving the system to cope with new environments and new or changed requirements consumes most maintenance effort. Maintenance is therefore a natural continuation of the system development process with associated specification, design, implementation and testing activities. A spiral model, such as that shown in Figure 27.3, is therefore a better representation of the

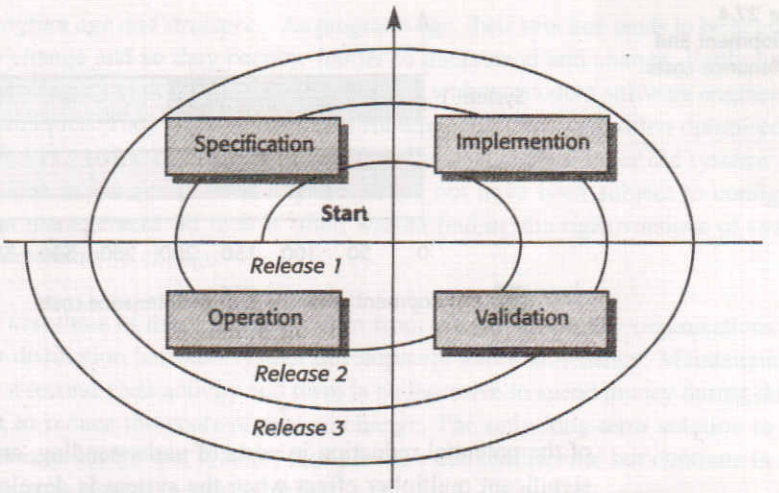


Figure 27.3
Spiral model of development

software process than representations such as the waterfall model (see Figure 3.1) where maintenance is represented as a separate process activity.

The costs of system maintenance represent a large proportion of the budget of most organisations that use software systems. In the 1980s, Lientz and Swanson found that large organisations devoted at least 50 per cent of their total programming effort to evolving existing systems. McKee (1984) found a similar distribution of maintenance effort across the different types of maintenance but suggests that the amount of effort spent on maintenance is between 65 and 75 per cent of total available effort. As organisations have replaced old systems with off-the-shelf systems, such as enterprise resource planning systems, this figure may not have come down. Although the details may be uncertain, we do know that software change remains a major cost for all organisations.

Maintenance costs as a proportion of development costs vary from one application domain to another. For business application systems, a study by Guimaraes (1983) showed that maintenance costs were broadly comparable with system development costs. For embedded real-time systems, maintenance costs may be up to four times higher than development costs. The high reliability and performance requirements of these systems may require modules to be tightly linked and hence difficult to change.

It is usually cost-effective to invest effort when designing and implementing a system to reduce maintenance costs. It is more expensive to add functionality after delivery because of the need to understand the existing system and analyse the impact of system changes. Therefore, any work done during development to reduce the cost of this analysis is likely to reduce maintenance costs. Good software engineering techniques such as precise specification, the use of object-oriented development and configuration management all contribute to maintenance cost reduction.

Figure 27.4 shows how overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system. Because

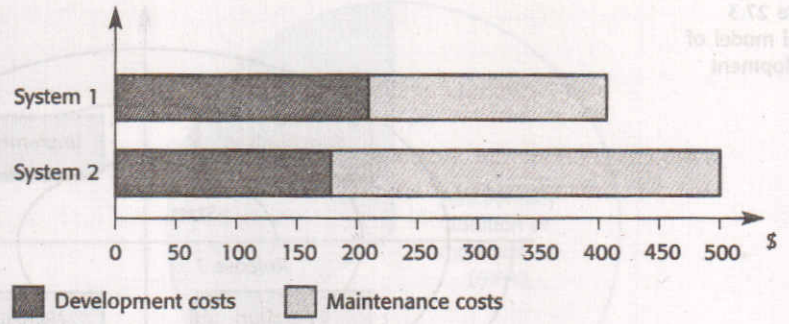


Figure 27.4 Development and maintenance costs

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 saving of $100,000 in maintenance costs over the lifetime of the system. This assumes that a percentage increase in development costs results in a comparable percentage decrease in overall system costs.

One important reason why maintenance costs are high is that it is more expensive to add functionality after a system is in operation than it is to implement the same functionality during development. The key factors that distinguish development and maintenance and which lead to higher maintenance costs are:

1. *Team stability* After a system has been delivered, it is normal for the development team to be broken up and people 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. A lot of the effort during the maintenance process is taken up with understanding the existing system before implementing changes to it.
2. *Contractual responsibility* 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 the software so that it is easy to change. If a development team can cut corners to save effort during development it is worthwhile for them to do so even if it means increasing maintenance costs.
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.

4. *Program age and structure* As programs age, their structure tends to be degraded by change and so they become harder to understand and change. Furthermore, many legacy systems have been developed without modern software engineering techniques. They were never well structured and they were often optimised for efficiency rather than understandability. The documentation for old systems may be lost or inconsistent. Old systems may not have been subject to 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 organisations still make a distinction between system development and maintenance. 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.

Rather than develop systems, maintain them until further maintenance is impossible and then replace them, we have to adopt the notion of evolutionary systems. Evolutionary systems are systems that are designed to evolve and change in response to new demands. They can be created from existing legacy systems by improving their structure through re-engineering (see Chapter 28) and by evolving the architecture of these systems as discussed in section 27.3.

The last issue in the list above, namely the problem of degraded system structure is, in some ways, the easiest problem to address. Re-engineering techniques may be applied to improve the system structure and understandability. If appropriate, architectural transformation (discussed later in this chapter) can adapt the system to new hardware. Preventative maintenance work (essentially incremental re-engineering) can be supported to improve the system and make it easier to change.

27.2.1 The maintenance process

Maintenance processes vary considerably depending on the type of software being maintained, the development processes used in an organisation and the people involved in the process. In some organisations, maintenance may be an informal process. Most maintenance requests come from conversations between the system users and developers. In other companies, it is a formalised process with structured documentation produced at each stage in the process. However, at an abstract level, all maintenance processes have the same fundamental activities of change analysis, release planning, system implementation and releasing a system to customers.

The maintenance process is triggered by a set of change requests from system users, management or 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

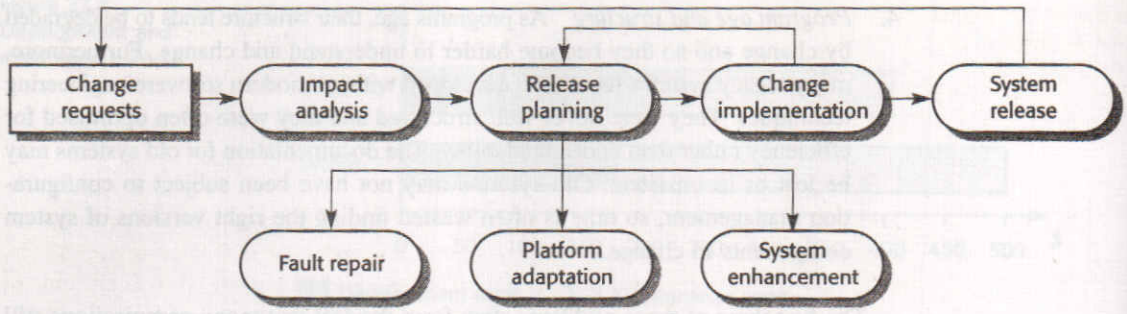


Figure 27.5 An overview of the maintenance process

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 new release. Figure 27.5, adapted from Arthur (1988), shows an overview of this process.

Ideally, the change implementation stage of this process should modify the system specification, design and implementation to reflect the changes to the system (Figure 27.6). New requirements that reflect the system changes are proposed, analysed and validated. System components are redesigned and implemented and the system is re-tested. If appropriate, prototyping of the proposed changes may be carried out as part of the change analysis process.

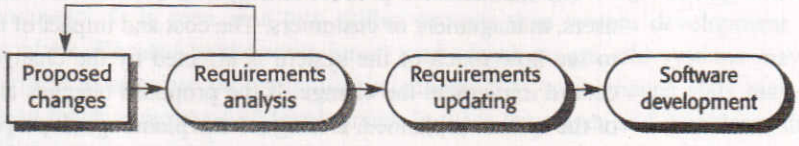
During this process, the requirements are analysed in detail and, frequently, implications of the changes emerge that were not apparent in the earlier change analysis process. This means that the proposed changes may be modified and further customer discussions may be required before they are implemented.

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

1. the emergence of a system fault which must be repaired to allow normal operation to continue;
2. environmental changes which have unexpected effects on the system;
3. unanticipated business changes which might be due to the emergence of new competitors or new legislation.

In these cases, it is usually more important to make the change quickly than to ensure that the formal change process is followed. Rather than modify the requirements

Figure 27.6 Change implementation



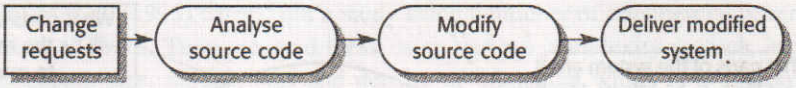


Figure 27.7 The emergency repair process

and design, an emergency fix is made to the code of the system to implement the change (Figure 27.7). However, the danger of this approach is that the requirements, the software design and the code gradually become inconsistent. It is difficult to avoid this happening because it may be difficult to implement the change quickly. Maintenance engineers may be told to deal with new emergency fixes to the software and the proper repair is delayed. If the engineer who made a code change then leaves a team before the design is updated, it is difficult for his or her replacement to retrofit the changes to the requirements and the design.

A further problem with emergency system repairs is that they have to be completed as quickly as possible. A workable solution rather than the best solution as far as system structure is concerned may be chosen. This accelerates the process of software ageing so that future changes become progressively more difficult and maintenance costs are increased.

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 changes will have a low priority and, after further system changes are made, it is unrealistic to redo the emergency repairs.

27.2.2 Maintenance prediction

Managers hate surprises, especially if these result in unexpectedly high costs. It therefore makes sense for them to try to predict what system changes are likely to be requested, what parts of the system are likely to cause the most difficulties for maintenance staff and the overall maintenance costs for a system in a given time period. Figure 27.8 illustrates these different predictions and associated questions.

These different predictions are obviously closely related:

1. Whether or not a system change should be accepted depends, to some extent, on the maintainability of the system components affected by that change.
2. Implementing system changes tends to degrade the system structure and hence reduce its maintainability.
3. Maintenance costs depend on the number of changes and the costs of change implementation depend on the maintainability of system components.

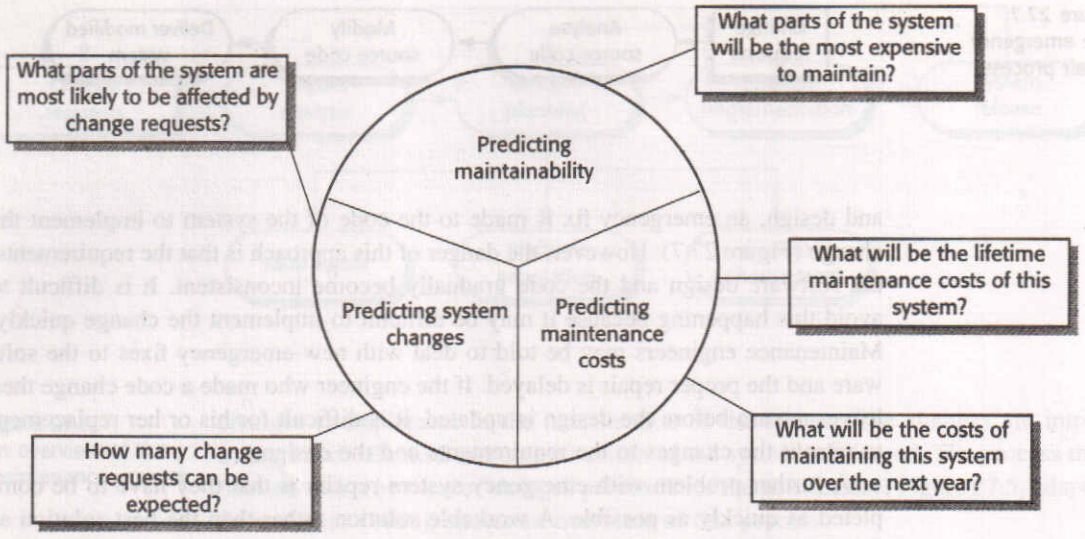


Figure 27.8 Maintenance prediction

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 make a judgement of 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 demands for change will be made.
2. *The number of inherently volatile system requirements* As I discussed in Chapter 6, requirements which reflect organisational policies and procedures are likely to be more volatile than requirements which 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.

To predict system maintainability you need to understand the number and the types of relationship between the different components of the system and the inherent complexity of these components. There have been various studies of the different types of complexity in a system (McCabe, 1976; Halstead, 1977) and of the relationships between complexity and maintainability (Kafura and Reddy, 1987; Banker *et al.*, 1993). It is not surprising that these studies have found that the more complex a system or component, the more expensive it is to maintain.

Banker *et al.* (1993) carried out a study using a number of commercial programs written in COBOL. They assessed these using various complexity metrics, including procedure size, module size and density of branching which is a measure of control complexity. By comparing the complexity of different parts of the program with the records of actual program maintenance they found that using good programming practice to reduce system complexity paid off in reduced maintenance costs.

Complexity measurements have been found to be particularly useful in identifying individual program components that are likely to be particularly expensive to maintain. In Kafura and Reddy's study (1987) they examined a number of system components and found that maintenance effort tended to be focused on a small number of complex components. The implications of this are that it may be cost-effective to replace particularly 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 which may be useful for assessing maintainability are:

1. *Number of requests for corrective maintenance* If the number of failure reports is increasing, this 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 that more and more components are affected and that 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. The activities involved are making changes to the system and its documentation rather than simply assessing what components are affected. This change time depends on the difficulty of programming so that non-functional requirements such as performance are met. If the time to implement a change increases, this may indicate a decline in maintainability.

4. *Number of outstanding change requests* If this number increases with time, it may imply a decline in maintainability.

You use predicted information about change requests and predictions about system maintainability to make predictions of maintenance costs. Most managers combine this information with intuition and experience to make an estimate of costs. The COCOMO 2 model of cost estimation (Boehm *et al.*, 1995) 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. Readers should look at Boehm's paper for details of this estimation technique.

27.3 Architectural evolution

During system maintenance most individual changes which are made are localised and do not affect the architecture of the system. However, since the 1980s, the economics of computer-based systems have changed radically, so that distributed rather than centralised systems are often the most cost-effective solution to business problems. Therefore, many companies are faced with the need to evolve their centralised mainframe systems to distributed client–server systems as discussed in Chapter 11.

There are a number of different drivers that contribute to this change:

1. *Hardware costs* The costs of buying and maintaining a distributed client–server system are usually much less than the costs of buying a mainframe computer of equivalent power.
2. *User interface expectations* Many legacy mainframe systems provide form-based, character interfaces. However, most users now expect graphical user interfaces and easier interaction with the system. These interfaces require much more local computation and can only be provided effectively in a client–server system.
3. *Distributed access to systems* Companies are, increasingly, physically distributing their organisation rather than maintaining all facilities on a single site. Their computer systems may have to be accessed from different locations and from different types of equipment. Customers and staff may access systems from their homes and this has to be supported.

By migrating to a distributed architecture, organisations can dramatically reduce hardware costs, can develop a system which has a more effective interface and a more modern 'look and feel' and can support distributed working. In the process of migration, there will inevitably be some conversion of the system to an object-oriented model and this is likely to reduce the costs of future system maintenance.

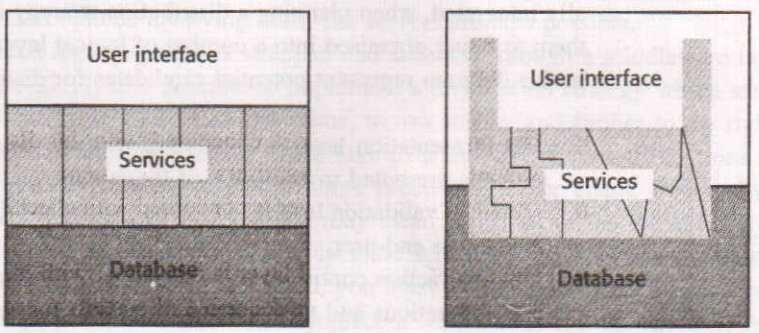
However, modifying the architecture of a legacy system is a major challenge and a very expensive process. Key factors which influence this decision are shown in Figure 27.9. Before embarking on architectural migration, organisations should make a careful assessment of their legacy systems to ensure that they will gain real business value from the architectural transformation.

A fundamental difficulty in migrating many centralised legacy systems to a distributed architecture is that the systems are not structured in such a way that the basic architectural components can be identified and separated from other components. Ideally, we would like legacy systems to have a structure as shown in the diagram on the left of Figure 27.10. In this case, the user interface, the services provided by the system and the database are clearly separated. Individual services are well defined. Within the service layer, it is possible to distinguish between the

Figure 27.9 Factors influencing system distribution decisions

Factor	Description
Business importance	Returns on the investment of distributing a legacy system depend on its importance to the business and how long it will remain important. If distribution provides more efficient support for stable business processes then it is more likely to be a cost-effective evolution strategy.
System age	The older the system, the more difficult it will be to modify its architecture because previous changes will have degraded the structure of the system.
System structure	The more modular the system, the easier it will be to change the architecture. If the application logic, the data management and the user interface of the system are closely intertwined, it will be difficult to separate functions for migration.
Hardware procurement policies	Application distribution may be necessary if there is a company policy to replace expensive mainframe computers with cheaper servers.

Figure 27.10 Ideal and realistic legacy system structures



Ideal model for distribution

Real legacy systems

different services. With this type of structure, the distributable elements can be identified in the system and can be rewritten to run on client computers.

In practice, most legacy systems are more like the right side of Figure 27.10 where user interface facilities, services and data access are intermingled. Services may overlap. Different parts of the service are implemented in different system components. User interface and service code are integrated in the same components and there may not be a clear distinction between the system services and the system database. In these cases, it may not be possible to identify the parts of the system which can be distributed.

In situations where it is impractical to separate the legacy system into distributable components and implement these components on a distributed system an

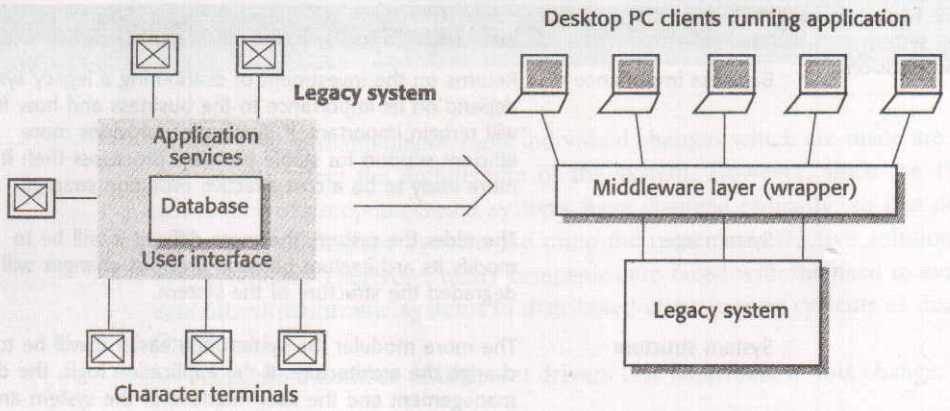


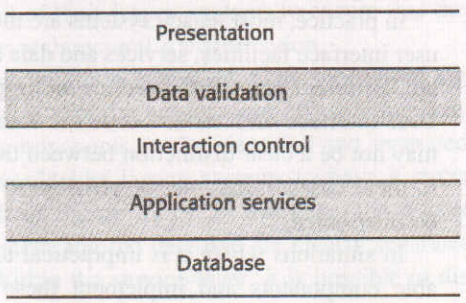
Figure 27.11 Legacy system distribution

alternative approach can be used. The legacy system may be frozen and the complete system packaged (wrapped) as a server. The user interface is reimplemented on the client and special-purpose middleware translates requests from the client into interactions with the unchanged legacy system. This situation is illustrated in Figure 27.11.

Although the user interface and services provided by a legacy system are usually integrated, when planning a distribution strategy it may be helpful to consider them as being organised into a number of logical layers (Figure 27.12). The layers in this diagram represent potential candidates for distribution.

1. The presentation layer is concerned with the display and organisation of the screens presented to end-users of the system.
2. The data validation layer is concerned with checking the data input by and output to the end-user.
3. The interaction control layer is concerned with managing the sequence of end-user operations and the sequence of screens presented to the user.
4. The application services layer is concerned with providing the basic computations provided by the application.
5. The database layer provides application data storage and management.

Figure 27.12 Layered distribution model



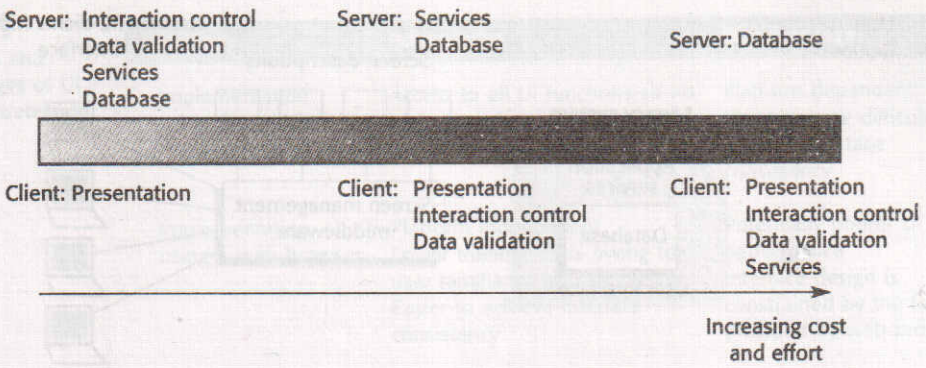


Figure 27.13 Spectrum of distribution options

It is impractical to distribute the database for most legacy systems but there is a spectrum of alternative distribution options as shown in Figure 27.13. In the simplest option, the client computer is concerned only with the presentation of the user interface and all other functions are retained on the server. In the most radical distribution option, the server only manages the system data and all other functionality is distributed to the client. Of course, these are not exclusive options. You may decide to start with presentation distribution and distribute other logical layers when time and resources are available. Furthermore, as I discuss in Chapter 11, other distribution options involving multi-tier servers are also possible.

When a legacy system is wrapped and accessed through a middleware layer as in Figure 27.11, it is possible to implement a distribution strategy which starts on the left of Figure 27.13 and, over time, moves further and further to the right. As new services are implemented, these take over the legacy system functions in the server, thus transferring more and more processing to the client. Eventually, this gradual distribution of functionality may mean that most of the initial legacy system is unused and it acts only as a database server for the distributed system.

Once this stage has been reached, you must then decide if it is worth retaining the legacy system or if you should replace it with a database management system. Factors that you should consider include the system hardware, the available expertise, whether or not a DBMS is already in use, whether it can cope with the amount of data to be managed and costs of data re-engineering.

27.3.1 User interface distribution

Many legacy systems were designed before graphical user interfaces were available. They use forms-based interfaces which run on specialised terminals that can only display characters. These terminals have limited processing power and display characteristics, so all display and associated computation functions are handled by a central mainframe system. Even when these terminals have been replaced by PCs, the character-based interface may still be maintained through a terminal emulation program running on the PC.

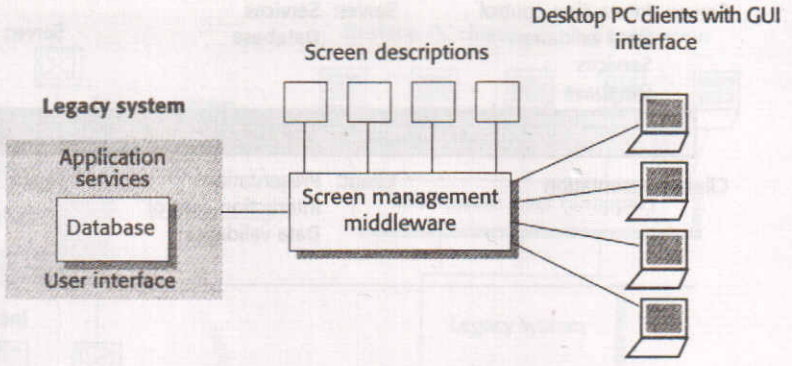


Figure 27.14 User interface distribution

User interface distribution takes advantage of the local processing power available on desktop PCs to provide a more responsive graphical interface to system users. User interface services (presentation, interaction control and validation) are moved to the desktop machine and the character-based forms interface is replaced by a GUI interface where the user can point at menus, fields, etc. The data being processed and most or all of the application services remain on the server system.

If the legacy system is structured so that the user interface services are clearly identifiable then the legacy system can be modified to implement user interface distribution. To distribute the user interface, those parts of the system which deal with user interaction are reimplemented on the client computer and communicate with the application through the same interface as the character-based UI.

In many cases, however, user interface code and the application logic are tightly integrated and it is very difficult to separate the user interface code from other code in the legacy system. In this case, the distributed user interface can be implemented as illustrated in Figure 27.14.

The screen management middleware in Figure 27.14 communicates with the application and behaves in exactly the same way as a user terminal. The screen management system uses a description of each screen to interpret the data on the display. It then sends this to the client where the user interface software presents this within a graphical interface. This can now often be implemented fairly easily using XML (St Laurent and Cerami, 1999) to describe the interface structure. There is therefore no need to change the legacy system itself. All that is required is to write the screen management middleware and the user interface software for the client computer.

There are two implementation strategies for user interface distribution:

1. Implement the interface using the window management system which is native to the client PC and implement communications with the server.
2. Implement the user interface using a WWW browser.

Figure 27.15 Advantages and disadvantages of UI migration strategies

Strategy	Advantages	Disadvantages
Implementation using the window management system	Access to all UI functions so no real restrictions on UI design Better UI performance	Platform dependent May be more difficult to achieve interface consistency
Implementation using a web browser	Platform independent Lower training costs owing to user familiarity with the WWW Easier to achieve interface consistency	Potentially poorer UI performance Interface design is constrained by the facilities provided by web browsers

In the first case, the user interface is programmed in a conventional programming language such as Java or in a scripting language such as Visual Basic. Calls are made to functions in the client operating system to implement the user interface. In the second case, implementation involves using the facilities in HTML and in web browsers to construct a user interface based on WWW pages. Computations which are required may be implemented on the client using Java or on the web server using a CGI interface or servlets. Each of these approaches has different advantages and disadvantages as described in Figure 27.15.

Converting interfaces to web-based interfaces is attractive because of the platform independence and widespread availability of web browsers. Java applets can be used to provide local client-side computation, so these interfaces can be comparable to interfaces developed using the window management system. However, some users may still use versions of browsers which are either incapable of running Java or the latest versions of HTML or which are not properly configured. Interface designers often have to design to these lower-capability systems and this restricts the functionality of web-based interfaces.

A basic difficulty which arises in converting forms-based interfaces into graphical user interfaces comes from the different approaches which each of these take to interaction control and validation. In a forms-based system, the computer system controls the interaction and applies data validation rules as soon as the data is entered. Most applications require form fields to be completed in a particular sequence and display forms only when other forms have been completely and correctly filled in.

In a GUI, the user is in control and uses a model of interaction where he or she points and clicks to select the fields. It is unnatural for the order of this to be controlled by the machine. If control is implemented, this can result in additional network traffic between the client and the server. Data validation may only be possible after a form has been completely filled in or the system may be slowed down by frequent communications between the client and the server.

KEY POINTS

- Software change strategies include software maintenance, architectural evolution and software re-engineering.
- There appear to be a number of invariant relationships (Lehman's Laws) which affect the evolution of a software system. These have been derived from empirical observations and show that maintenance costs are inevitable. They provide guidelines on how to manage the maintenance process.
- There are three principal types of software maintenance. These are maintenance to repair defects in the software, maintenance to adapt the software to a different operating environment and maintenance to add to or modify the functionality of the system.
- The cost of software change usually exceeds the cost of software development. As companies manage an increasing number of legacy systems, more of their software budget is taken up in legacy system maintenance.
- High maintenance costs are due to lack of staff stability, development contracts that don't encourage the production of maintainable code, shortages of the skills required to maintain a system and system structures that have degraded owing to age and regular system change.
- Architectural evolution involves modifying the architecture of a system from a centralised, data-centric architecture to a distributed architecture. Both the user interface and system functionality may be distributed.
- A common architectural evolution strategy for legacy systems is to encapsulate the legacy system as a server and to implement a distributed user interface which accesses the system functionality through special-purpose middleware.

FURTHER READING

The Renaissance of Legacy Systems. This book includes a chapter which discusses various strategies for migrating centralised systems to client–server systems. (I. Warren, 1998, Springer.)

Software Evolution and Reuse. Discusses a number of industrial case studies where research techniques have been applied. (S. Hallsteinsen and M. Paci, 1997, Springer.)

EXERCISES

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

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

27.3 Explain the difficulties of measuring program maintainability. Suggest why you should use several different complexity metrics when trying to assess the maintainability of a program.

27.4 As a software project manager in a company that specialises in the development of software for the offshore oil industry, you have been given the task of discovering those factors which affect the maintainability of the particular systems which are developed by your company. Suggest how you might set up a programme to analyse the maintenance process to discover appropriate maintainability metrics for your company.

27.5 Explain why encapsulating a mainframe legacy system and using it as a server should only be considered as a short-term solution to the problems of architectural evolution.

27.6 Discuss the advantages and disadvantages of distributing each of the layers in Figure 27.12.

27.7 Two major international banks with different customer information databases merge and decide that they need to provide access to all customer information from all bank branches. Giving reasons for your answer, suggest the most appropriate strategy for providing access to these systems and briefly discuss how the solution might be implemented.

27.8 Do software engineers have a professional responsibility to produce maintainable code even if this is not explicitly requested by their employer?