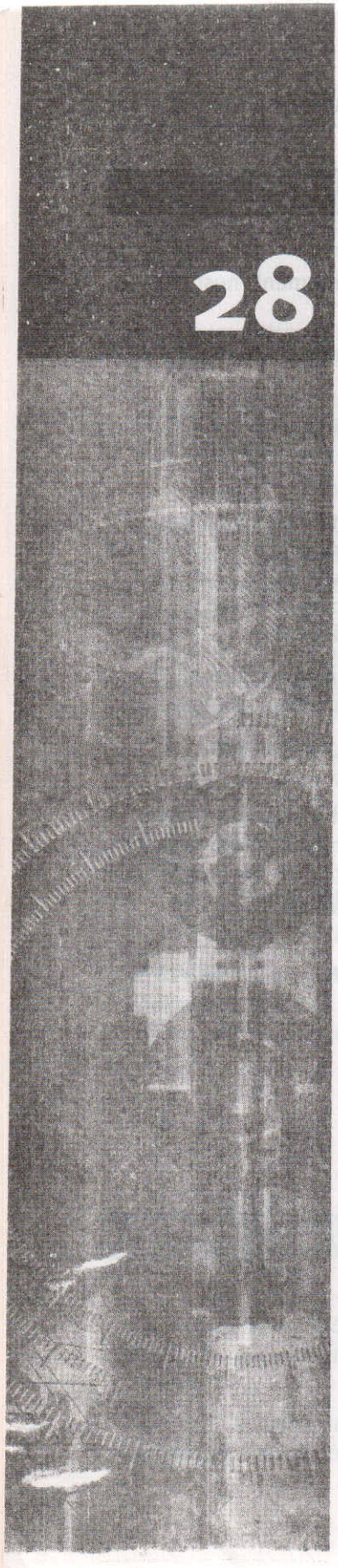


28 Software re-engineering

Objectives

The objective of this chapter is to explain the process of software re-engineering to improve the maintainability of a software system. When you have read this chapter, you will:

- understand why re-engineering is sometimes a cost-effective option for software system evolution;
- understand the activities such as reverse engineering and program restructuring which may be involved in the software re-engineering process;
- understand the differences between software and data re-engineering and understand why data re-engineering is an expensive and time-consuming process.

Contents

28.1 Source code translation
28.2 Reverse engineering
28.3 Program structure improvement
28.4 Program modularisation
28.5 Data re-engineering

In Chapters 26 and 27, I introduced legacy systems and different strategies for software evolution. Legacy systems are old software systems which are essential for business process support. Companies rely on these systems so they must keep them in operation. Software evolution strategies include maintenance, replacement, architectural evolution and, the topic of this chapter, software re-engineering.

Software re-engineering is concerned with reimplementing legacy systems to make them more maintainable. Re-engineering may involve redocumenting the system, organising and restructuring the system, translating the system to a more 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, the system architecture also remains the same.

From a technical perspective, software re-engineering may appear to be a second-class solution to the problems of system evolution. The software architecture is not updated so distributing centralised systems is difficult. It is not usually possible to radically change the system programming language, so old systems cannot be converted to object-oriented programming languages such as Java or C++. Inherent limitations in the system are maintained because the software functionality is unchanged.

However, from a business point of view, software re-engineering may be the only viable way to ensure that legacy systems can continue in service. It may be too expensive and too risky to adopt any other approach to system evolution. To understand the reasons for this, we must make a rough assessment of the legacy system problem.

The amount of code in legacy systems is immense. In 1990, it was estimated (Ulrich, 1990) that there were 120 *billion* lines of source code in existence. The majority of these systems have been written in COBOL, a programming language best suited to business data processing, or FORTRAN. FORTRAN is a language for scientific or mathematical programming. These languages have limited program structuring facilities and, in the case of FORTRAN, very limited support for data structuring.

Although many of these programs have now been replaced, most of them are probably still in service. Meanwhile, since 1990, there has been a huge increase in computer use for business process support. Therefore, I guess that there must now be roughly 250 billion lines of source code in existence which must be maintained. Most of this is not written in object-oriented languages and much of it still runs on mainframe computers.

There are so many systems in existence that complete replacement or radical restructuring is financially unthinkable for most organisations. Maintenance of old systems is increasingly expensive so re-engineering these systems extends their useful lifetime. As discussed in Chapter 26, re-engineering a system is cost-effective when it has a high business value but is expensive to maintain. Re-engineering improves the system structure, creates new system documentation and makes it easier to understand.

Re-engineering a software system has two key advantages over more radical approaches to system evolution:

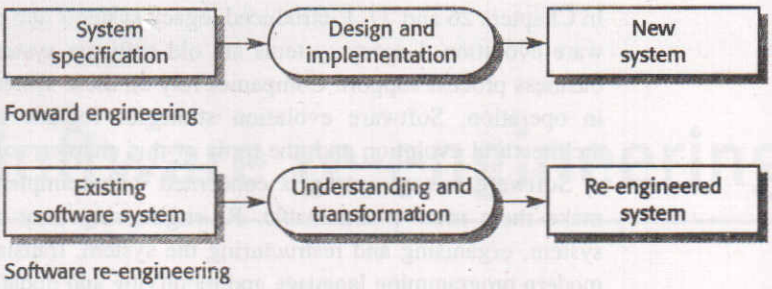


Figure 28.1 Forward engineering and re-engineering

1. *Reduced risk* There is a high risk in redeveloping software that is essential for an organisation. Errors may be made in the system specification, there may be development problems, etc.
2. *Reduced cost* The cost of re-engineering is significantly less than the costs of developing new software. Ulrich (1990) quotes an example of a commercial system where the reimplementation costs were estimated at $50 million. The system was successfully re-engineered for $12 million. If these figures are typical, it is about four times cheaper to re-engineer than to rewrite.

The term re-engineering is also associated with business process re-engineering (Hammer, 1990). Business process re-engineering is concerned with redesigning business processes to reduce the number of redundant activities and improve process efficiency. It is usually reliant on the introduction or the enhancement of computer-based support for the process. Process re-engineering is often a driver for software evolution as legacy systems may incorporate implicit dependencies on the existing processes. These have to be discovered and removed before process re-engineering is possible. Therefore, the need for software re-engineering may emerge in a company when it becomes clear that the scale of the changes required by the business process re-engineering cannot be accommodated through normal program maintenance.

The critical distinction between re-engineering and new software development is the starting point for the development. Rather than start with a written specification, the old system acts as a specification for the new system. Chikofsky and Cross (1990) call conventional development *forward engineering* to distinguish it from software re-engineering. This distinction is illustrated in Figure 28.1. Forward engineering starts with a system specification and involves the design and implementation of a new system. Re-engineering starts with an existing system and the development process for the replacement is based on understanding and transformation of the original system.

Figure 28.2 illustrates a possible re-engineering process. The input to the process is a legacy program and the output is a structured, modularised version of the same program. At the same time as program re-engineering, the data for the system may also be re-engineered. The activities in this re-engineering process are:

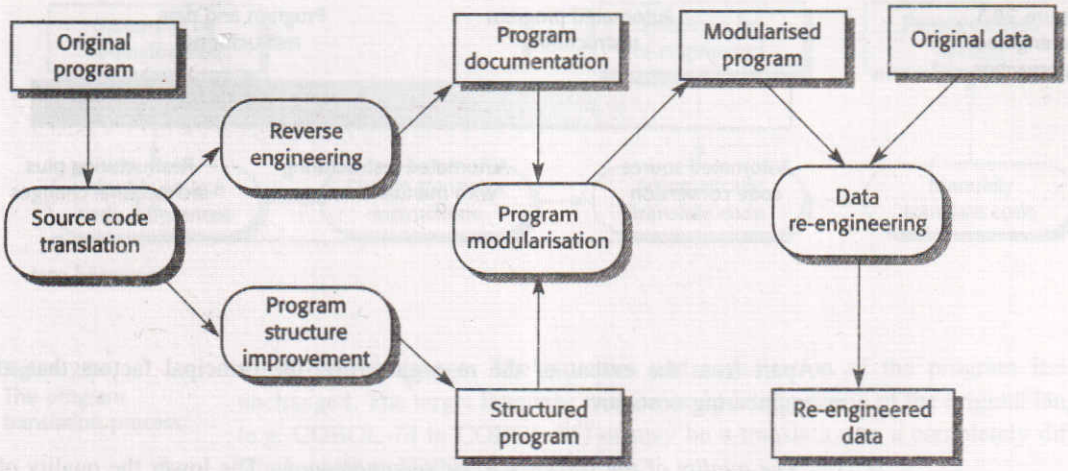


Figure 28.2
The re-engineering process

1. *Source code translation* 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 analysed and information extracted from it which helps to document its organisation and functionality.
3. *Program structure improvement* The control structure of the program is analysed and modified to make it easier to read and understand.
4. *Program modularisation* Related parts of the program are grouped together and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural transformation as discussed in Chapter 27.
5. *Data re-engineering* The data processed by the program is changed to reflect program changes.

Program re-engineering may not necessarily require all of the steps in Figure 28.2. Source code translation may not be needed if the programming language used to develop the system is still supported by the compiler supplier. If the re-engineering relies completely on automated tools then recovering documentation through reverse engineering may be unnecessary. Data re-engineering is only required if the data structures in the program change during system re-engineering. However, software re-engineering always involves some program restructuring.

The costs of re-engineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to re-engineering as shown in Figure 28.3. Costs increase from left to right so that source code translation is the cheapest option and re-engineering as part of architectural migration is the most expensive.

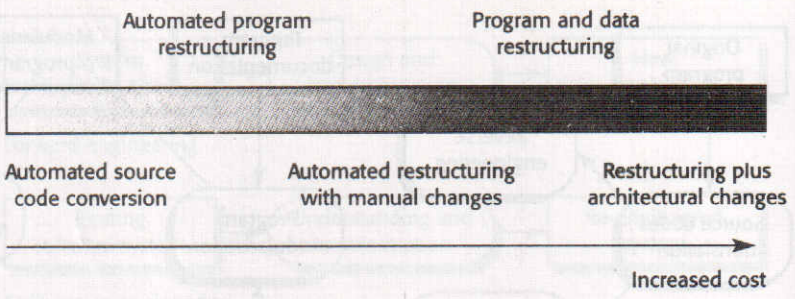


Figure 28.3 Re-engineering approaches

Apart from the extent of the re-engineering, the principal factors that affect re-engineering costs are:

1. *The quality of the software to be re-engineered* The lower the quality of the software and its associated documentation (if any), the higher the re-engineering costs.
2. *The tool support available for re-engineering* It is not normally cost-effective to re-engineer a software system unless you can use CASE tools to automate most of the program changes.
3. *The extent of data conversion required* If re-engineering requires large volumes of data to be converted, this significantly increases the process cost.
4. *The availability of expert staff* If the staff responsible for maintaining the system cannot be involved in the re-engineering process, this will increase the costs. System re-engineers will have to spend a great deal of time understanding the system.

The main disadvantage of software re-engineering is that there are practical limits to the extent that a system can be improved by re-engineering. 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 reorganising of the system data management cannot be carried out automatically, so involve high additional costs. Although re-engineering can improve maintainability, the re-engineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

28.1 Source code translation

The simplest form of software re-engineering is program translation where source code in one programming language is automatically translated to source code in

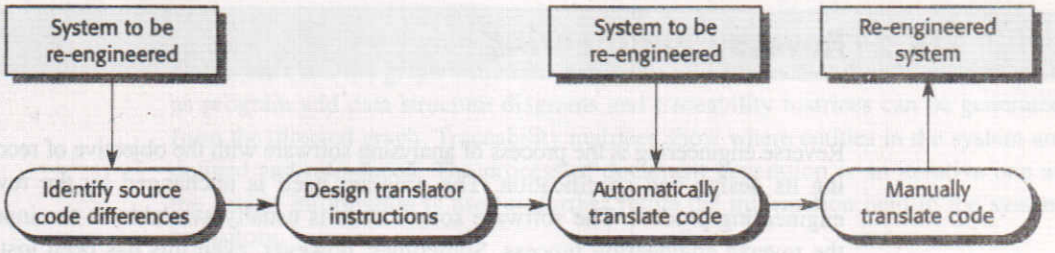


Figure 28.4 The program translation process

some other language. The structure and organisation of the program itself are unchanged. The target language may be an updated version of the original language (e.g. COBOL-74 to COBOL-85) or may be a translation to a completely different language (e.g. FORTRAN to C).

Source-level translation may be necessary for the following reasons:

1. *Hardware platform update* The organisation may wish to change its standard hardware platform. Compilers for the original language may not be available on the new hardware.
2. *Staff skill shortages* There may be a lack of trained maintenance staff for the original language. This is a particular problem where programs were written in a non-standard language that has now gone out of general use.
3. *Organisational policy changes* An organisation may decide to standardise on a particular language to minimise its support software costs. Maintaining many versions of old compilers can be very expensive.
4. *Lack of software support* The suppliers of the language compiler may have gone out of business or may discontinue support for their product.

Figure 28.4 illustrates the process of source code translation. There may be no need to understand the operation of the software in detail or to modify the system architecture. The analysis involved can focus on programming language considerations such as the equivalence of program control constructs.

Source code translation is only economically realistic if an automated translator is available to do the bulk of the translation. This may be a specially written program, a bought-in tool to convert from one language to another or a pattern matching system. In the latter case, a set of instructions how to make the translation from one representation to another has to be written. Parameterised patterns in the source language are defined and associated with equivalent patterns in the target language.

In many cases, completely automatic translation is impossible. Constructs in the source language may have no direct equivalent in the target language. There may be embedded conditional compilation instructions in the source code which are not supported in the target language. In these circumstances, you need to make changes manually to tune and improve the generated system.

28.2 Reverse engineering

Reverse engineering is the process of analysing software with the objective of recovering its design and specification. The program itself is unchanged by the reverse engineering process. The software source code is usually available as the input to the reverse engineering process. Sometimes, however, even this has been lost and the reverse engineering must start with the executable code.

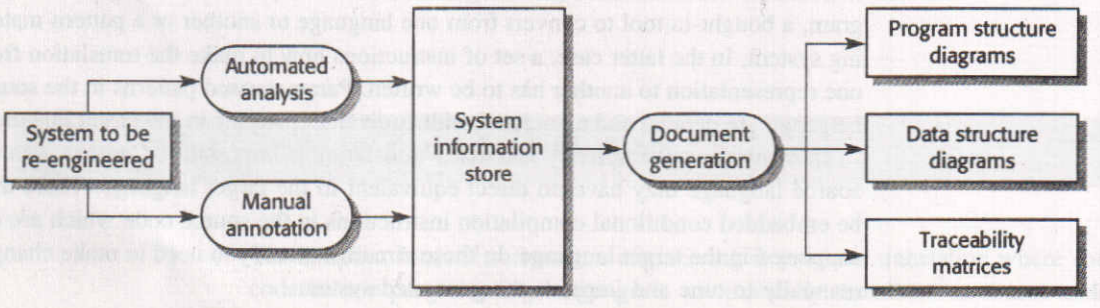
Reverse engineering is not the same thing as re-engineering. The objective of reverse engineering is to derive the design or specification of a system from its source code. The objective of re-engineering is to produce a new, more maintainable system. Of course, as we can see from Figure 28.2, reverse engineering to develop a better understanding of a system is often part of the re-engineering process.

Reverse engineering is used during the software re-engineering process to recover the program design which engineers use to help them understand a program before reorganising its structure. However, reverse engineering need not always be followed by re-engineering:

1. The design and specification of an existing system may be reverse engineered so that they can serve as an input to the requirements specification for that program's replacement.
2. Alternatively, the design and specification may be reverse engineered so that they are available to help program maintenance. With this additional information, it may not be necessary to re-engineer the system source code.

The reverse engineering process is illustrated in Figure 28.5. The process starts with an analysis phase. During this phase, the system is analysed using automated tools to discover its structure. In itself, this is not enough to re-create the system design. Engineers then work with the system source code and its structural model. They add information to this which they have collected by understanding the system. This information is maintained as a directed graph that is linked to the program source code.

Figure 28.5 The reverse engineering process



Information store browsers are used to compare the graph structure and the code and to annotate the graph with extra information. Documents of various types such as program and data structure diagrams and traceability matrices can be generated from the directed graph. Traceability matrices show where entities in the system are defined and referenced. The process of document generation is an iterative one as the design information is used to further refine the information held in the system repository.

Tools for program understanding may be used to support the reverse engineering process. These usually present different system views and allow easy navigation through the source code. For example, they allow users to select a data definition, then move through the code to where that data item is used. Examples of such program browsers are discussed by Cleveland (1989), Oman and Cook (1990) and Ning *et al.* (1994).

After the system design documentation has been generated, further information may be added to the information store to help re-create the system specification. This usually involves further manual annotation of the system structure. The specification cannot be deduced automatically from the system model.

28.3 Program structure improvement

The need to optimise memory use and the lack of understanding of software engineering by many programmers have meant that many legacy systems are not well structured. Their control structure is tangled with many unconditional branches and unintuitive control logic. This structure may also have been degraded by regular maintenance. Changes to the program may have made some code unreachable but this can only be discovered after extensive analysis. Maintenance programmers often dare not remove code in case it may be accessed indirectly.

Figure 28.6 illustrates how complex control logic can make a relatively simple program difficult to understand. The program is written in a notation similar to FORTRAN which was often used to write this type of program. However, I have not made the program even more difficult to understand by using cryptic variable names. The example in Figure 28.6 is a controller for a heating system. A panel switch may be set to **On**, **Off** or **Controlled**. If the system is controlled, then it is switched on and off depending on a timer setting and a thermostat. If the heating is on, **Switch-heating** turns it off and vice versa.

Typically, programs develop this complex logic structure as they are modified during maintenance. New conditions and associated actions are added without changing the existing control structure. In the short term, this is a quicker and less risky solution as it reduces the chances of introducing faults in the system. In the long term, however, it leads to incomprehensible code. Complex code structures can also arise when programmers tried to avoid duplicating code. This was sometimes necessary when programs were constrained by limited memory.

Figure 28.6 A control program with spaghetti logic

```
Start:   Get (Time-on, Time-off, Time, Setting, Temp, Switch)
         if Switch = off goto off
         if Switch = on goto on
         goto Cntrld
off:     if Heating-status = on goto Sw-off
         goto loop
on:      if Heating-status = off goto Sw-on
         goto loop
Cntrld:  if Time = Time-on goto on
         if Time = Time-off goto off
         if Time < Time-on goto Start
         if Time > Time-off goto Start
         if Temp > Setting then goto off
         if Temp < Setting then goto on
Sw-off:  Heating-status := off
         goto Switch
Sw-on:   Heating-status := on
Switch:  Switch-heating
loop:    goto Start
```

Figure 28.7 A structured control program

```
loop
  -- The Get statement finds values for the given variables from the system's
  -- environment.
  Get (Time-on, Time-off, Time, Setting, Temp, Switch) ;
  case Switch of
      when On => if Heating-status = off then
                      Switch-heating ; Heating-status := on ;
                 end if ;
      when Off => if Heating-status = on then
                      Switch-heating ; Heating-status := off ;
                  end if ;
      when Controlled =>
          if Time >= Time-on and Time < = Time-off then
              if Temp > Setting and Heating-status = on then
                  Switch-heating; Heating-status = off ;
              elsif Temp < Setting and Heating-status = off then
                  Switch-heating; Heating-status := on ;
              end if ;
          end if ;
  end case ;
end loop ;
```

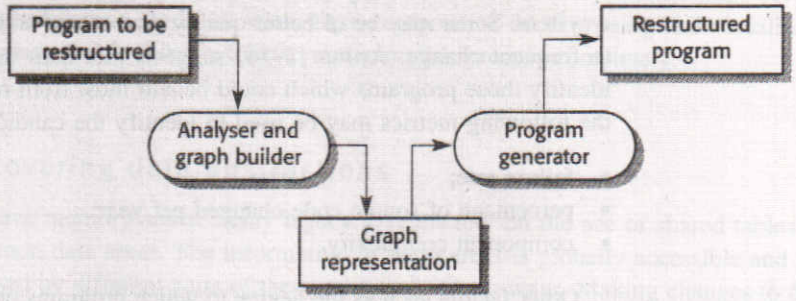
Figure 28.7 shows the same control system which I have rewritten using structured control statements. The program may be read sequentially from top to bottom so it is much easier to understand. The three switch positions, on, off and controlled, are clearly identified and linked to their associated code. I have not used Java here as the original program is not object-oriented.

Figure 28.8
Condition simplification

```
-- Complex condition
if not (A > B and (C < D or not ( E > F) ) )...

-- Simplified condition
if A <= B and (C>= D or E > F)...
```

Figure 28.9
Automated program restructuring



As well as unstructured control, complex conditions can also be simplified as part of the program restructuring process. Figure 28.8 shows how a conditional statement including 'not' logic may be made more understandable.

Bohm and Jacopini (1966) proved that any program may be rewritten in terms of simple if-then-else conditionals and while-loops and that unconditional goto statements were not required. This theorem is the basis for automatic program restructuring. Figure 28.9 shows the stages in the automatic restructuring of a program. It is first converted to a directed graph, then a structured equivalent program, without goto statements, is generated.

The directed graph that is generated is a program flow graph which shows how control moves through the program. Simplification and transformation techniques can be applied to this graph without changing its semantics. These detect and remove unreachable parts of the code. Once simplification has been completed, a new program is generated. While-loops and simple conditional statements are substituted for goto-based control. This program may be in the original language or in a different language (e.g. FORTRAN may be converted to C).

Problems with automatic program restructuring include:

1. *Loss of comments* If the program has in-line comments, these are invariably lost as part of the restructuring process.
2. *Loss of documentation* Similarly, the correspondence between external program documentation and the program is also lost. In many cases, however, both the comments and the documentation of a program are out of date, so this is not an important factor.
3. *Heavy computational demands* The algorithms embedded in restructuring tools are complex. Even with fast, modern hardware it can take a long time to complete the restructuring process for large programs.

If the program is data-driven with components tightly coupled through shared data structures, restructuring the code may not lead to a significant improvement in understandability. Program modularisation, as discussed in the following section, may also be necessary. If the program is written in a non-standard language dialect, standard restructuring tools may not work properly and significant manual intervention may be required.

In some cases, it may not be cost-effective to restructure all of the programs in a system. Some may be of better quality than others and some may not be subject to frequent change. Arthur (1988) suggests that data should be collected to help identify those programs which could benefit most from restructuring. For example, the following metrics may be used to identify the candidates for restructuring:

- failure rate;
- percentage of source code changed per year;
- component complexity.

Other factors such as the degree to which programs or components meet current standards might also be taken into account in making restructuring decisions.

28.4 Program modularisation

Program modularisation is the process of reorganising a program so that related program parts are collected together and considered as a single module. Once this has been done, it becomes easier to remove redundancies in these related components, to optimise their interactions and to simplify their interface with the rest of the program. For example, in a program that processes seismographic data, all operations associated with graphical presentation of the data may be collected together into a single module. If the system is to be distributed, the modules created can be encapsulated as objects and accessed through a common interface.

Several different types of module may be created during the program modularisation process. These include:

1. *Data abstractions* These are abstract data types that are created by associating data with processing components. I discuss this in section 28.4.1.
2. *Hardware modules* These are closely related to data abstractions and collect together all of the functions which are used to control a particular hardware device.
3. *Functional modules* These are modules which collect together functions that carry out similar or closely related tasks. For example, all of the functions concerned with input and input validation may be incorporated in a single module. This type of modularisation should be considered where it is impractical to recover program data abstractions.

4. *Process support modules* These are modules where all of the functions and the specific data items required to support a particular business process are grouped. For example, in a library system, a process support module may include all of the functionality required to support the issue and return of books.

Program modularisation is usually carried out manually by inspecting and editing the code. To modularise a program, you have to identify relationships between components and work out what these components do. Browsing and visualisation tools help but it is impossible to automate this process completely.

28.4.1 Recovering data abstractions

To save memory space, many legacy systems rely on the use of shared tables and common data areas. The information in these areas is globally accessible and may be used by different parts of the system in different ways. Making changes to these global data areas is expensive because of the costs of analysing change impacts across all uses of the data.

To reduce the costs of change to these shared data areas, the program modularisation process may focus on the identification of data abstractions. Data abstractions or abstract data types collect together data and associated processing and are resilient to change. Data abstractions hide the data representation and provide constructor and access functions to modify and inspect the data. So long as the interface is maintained, changes in the data type should not affect other parts of the program.

The steps involved in converting shared global data areas to objects or abstract data types are:

1. Analyse common data areas to identify logical data abstractions. It will often be the case that several abstractions are combined in a single shared data area. These should be identified and logically restructured.
2. Create an abstract data type or object for each of these abstractions. If the programming language does not have data hiding facilities, simulate an abstract data type by providing functions to update and access all fields of the data.
3. Use a program browsing system or cross-reference generator to find all references to the data. Replace these with calls to the appropriate functions.

This process seems to be time-consuming but relatively straightforward. In practice, however, it can be very difficult because of the ways in which shared data areas are used. In older versions of languages like FORTRAN which have limited data structuring facilities, programmers may have designed complex data management strategies which they have implemented using shared arrays. The array therefore may actually be used as a different kind of data structure. Further problems are caused by indirect addressing of shared structures and addressing by offsets from some other structure.

If the target machine for the original program had a limited memory, this causes other problems. The programmers may have used knowledge about data lifetimes and embedded this in the program. To avoid allocating extra space, they use the same data area to store different abstractions at different points in the program. These can only be discovered after a detailed static and dynamic analysis of the program.

28.5 Data re-engineering

So far, most of the discussion of software evolution has focused on the problems of changing programs and software systems. However, in many cases, there are associated problems of data evolution. The storage, organisation and format of the data processed by legacy programs may have to evolve to reflect changes to the software. The process of analysing and reorganising the data structures and, sometimes, the data values in a system to make it more understandable is called *data re-engineering*.

In principle, data re-engineering should not be necessary if the functionality of a system is unchanged. In practice, however, there are a number of reasons why you may have to modify the data as well as the programs in a legacy system:

1. *Data degradation* Over time, the quality of data tends to decline. Changes to the data introduce errors, duplicate values may have been created and changes to the external environment may not be reflected in the data. This is inevitable because data lifetimes are often very long. For example, personal banking data comes into existence when an account is opened and may have to persist for at least the lifetime of the customer. As the customer's circumstances change, these changes may not be properly included in the bank's data. Program re-engineering can bring data quality problems to light and thus highlight the need for associated data re-engineering.
2. *Inherent limits that are built into the program* When originally designed, developers of many programs included built-in constraints on the amount of data which could be processed. However, programs are now often required to process much more data than was originally envisaged by their developers. Data re-engineering may be required to remove the limitations. For example, Rochester and Douglass (1993) describe a funds management system that was originally designed to handle up to 99 funds. The company running the system was managing more than 2000 funds and had to run 23 separate copies of the system. They therefore decided to re-engineer the system and its associated data.
3. *Architectural evolution* If a centralised system is migrated to a distributed architecture it is essential that the core of that architecture should be a data management system that can be accessed from remote clients. This may require a large data re-engineering effort to move data from separate files into the server

database management system. The move to a distributed program architecture may be initiated when an organisation decides to move from file-based data management to a database management system.

As with program re-engineering, there are a spectrum of approaches to data re-engineering which reflect the reasons why data re-engineering may be required. These are shown in Figure 28.10.

Rickets *et al.* (1993) describe some of the problems with data which can arise in legacy systems made up of several cooperating programs:

1. *Data naming problems* Names may be cryptic and difficult to understand. Different names (synonyms) may be given to the same logical entity in different programs in the system. The same name may be used in different programs to mean different things.
2. *Field length problems* This is a problem when field lengths in records are explicitly assigned in the program. The same item may be assigned different lengths in different programs or the field length may be too short to represent current data. To solve this problem, other fields may be reused in some cases so that usage of a named data field across the programs in a system is inconsistent.
3. *Record organisation problems* Records representing the same entity may be organised differently in different programs. This is a problem in languages like COBOL where the physical organisation of records is set by the programmer and reflected in files. It is not a problem in languages like C++ or Java where the physical organisation of a record is the compiler's responsibility.
4. *Hard-coded literals* Literal (absolute) values, such as tax rates, are included directly in the program rather than referenced using some symbolic name.
5. *No data dictionary* There may be no data dictionary defining the names used, their representation and their use.

Figure 28.10 Approaches to data re-engineering

Approach	Description
Data cleanup	The data records and values are analysed to improve their quality. Duplicates are removed, redundant information is deleted and a consistent format applied to all records. This should not normally require any associated program changes.
Data extension	In this case, the data and associated programs are re-engineered to remove limits on the data processing. This may require changes to programs to increase field lengths, modify upper limits on the tables, etc. The data itself may then have to be rewritten and cleaned up to reflect the program changes.
Data migration	In this case, data is moved into the control of a modern database management system. The data may be stored in separate files or may be managed by an older type of DBMS. This situation is illustrated in Figure 28.11.

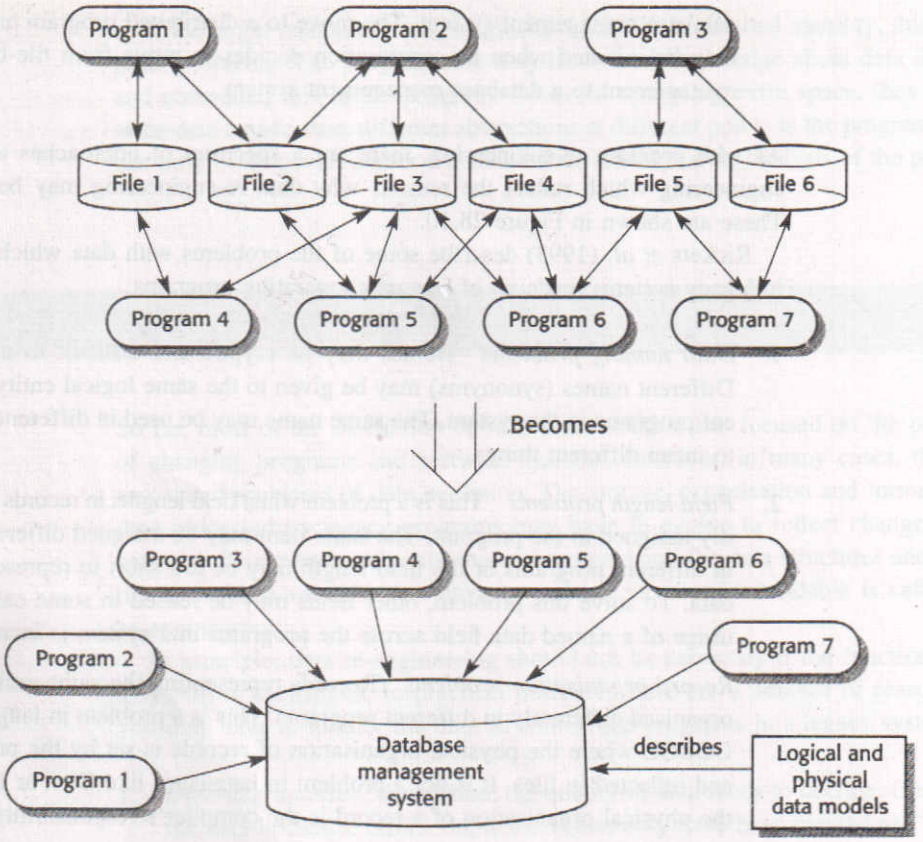


Figure 28.11
Data migration

As well as inconsistent data definitions, data values may also be stored in an inconsistent way. After the data definitions have been re-engineered, the data values must also be converted to conform to the new structure. Rickets *et al.* also describe some possible data value inconsistencies. These are shown in Figure 28.12.

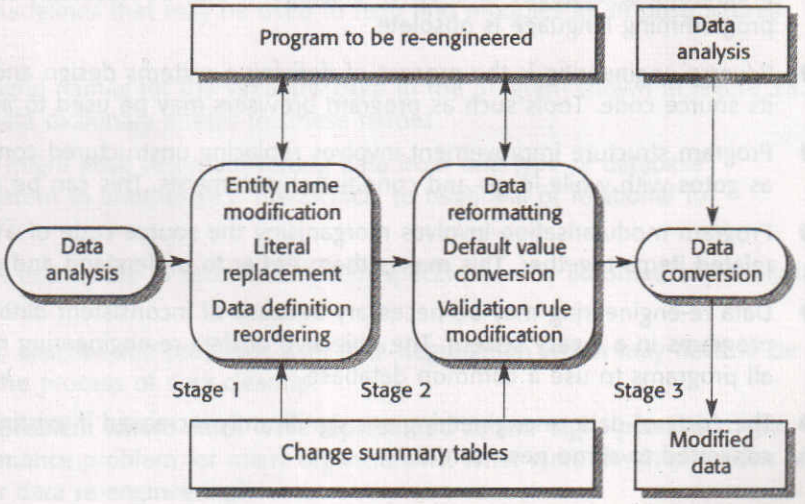
Detailed analysis of the programs that use the data is essential before data re-engineering. This analysis should be aimed at discovering the function of identifiers in the program, finding the literal values which should be replaced with named constants, discovering embedded data validation rules and data representation conversions. Tools such as cross-reference analysers and pattern matchers may be used to help with this analysis. A set of tables should be created which show where data items are referenced and the changes to be made to each of these references.

Figure 28.13 illustrates the process of data re-engineering, assuming that data definitions are modified, literal values named, data formats reorganised and the data values converted. The change summary tables hold details of all the changes to be made. They are therefore used at all stages of the data re-engineering process.

Figure 28.12 Data value inconsistencies

Data inconsistency	Description
Inconsistent default values	Different programs assign different default values to the same logical data items. This causes problems for programs other than those that created the data. The problem is compounded when missing values are assigned a default value that is valid. The missing data cannot then be discovered.
Inconsistent units	The same information is represented in different units in different programs. For example, in the US or the UK, weight data may be represented in pounds in older programs but in kilograms in more recent systems. A major problem of this type has arisen in Europe with the introduction of a single European currency. Legacy systems have been written to deal with national currency units and data has to be converted to euros.
Inconsistent validation rules	Different programs apply different data validation rules. Data written by one program may be rejected by another. This is a particular problem for archival data which may not have been updated in line with changes to data validation rules.
Inconsistent representation semantics	Programs assume some meaning in the way items are represented. For example, some programs may assume that upper-case text means an address. Programs may use different conventions and may therefore reject data which is semantically valid.
Inconsistent handling of negative values	Some programs reject negative values for entities which must always be positive. Others, however, may accept these as negative values or fail to recognise them as negative and convert them to a positive value.

Figure 28.13 The data re-engineering process



In Stage 1 of this process, the data definitions in the program are modified to improve understandability. The data itself is not affected by these modifications. It is possible to automate this process to some extent using pattern matching systems such as awk (Aho *et al.*, 1988) to find and replace definitions or to develop XML descriptions of the data (St Laurent and Cerami, 1999) and use these to drive data conversion tools. However, some manual work is almost always necessary to complete the process. The data re-engineering process may stop at this stage if the goal is simply to improve the understandability of the data structure definitions in a program. If, however, there are data value problems as discussed above, Stage 2 of the process may then be entered.

If an organisation decides to continue to Stage 2 of the process, it is then committed to Stage 3, data conversion. This is usually a very expensive process. Programs have to be written which embed knowledge of the old and the new organisation. These process the old data and output the converted information. Again, pattern matching systems may be used to implement this conversion.

KEY POINTS

- The objective of system re-engineering is to improve the system structure and make it easier to understand. The cost of future system maintenance should therefore be reduced.
- The re-engineering process includes source code translation, reverse engineering, program structure improvement, program modularisation and data re-engineering.
- Source code translation is the automatic conversion of a program written in one programming language to another language. It may be necessary when the original programming language is obsolete.
- Reverse engineering is the process of deriving a systems design and specification from its source code. Tools such as program browsers may be used to assist this process.
- Program structure improvement involves replacing unstructured control constructs such as gotos with while-loops and conditional statements. This can be automated.
- Program modularisation involves reorganising the source code of a program to group related items together. This makes them easier to understand and change.
- Data re-engineering may be necessary because of inconsistent data management by the programs in a legacy system. The objective of data re-engineering may be to re-engineer all programs to use a common database.
- The costs of data re-engineering are significantly increased if existing data has to be converted to some new format.

FURTHER READING

'Examining data quality'. This special section includes a number of papers which discuss data quality issues and the impact of poor data quality. (G. K. Tayi and D. P. Ballou, *Comm. ACM*, 41(2), Feb. 1998.)

Software Re-engineering. This is an IEEE tutorial that includes most of the important papers on re-engineering which were published before 1992. Many of the papers referenced in this chapter are reprinted in it. (R. S. Arnold, 1994, IEEE Press.)

'DoD legacy systems: Reverse engineering data requirements'. This is a good description of the practical problems which arise with legacy systems. The paper focuses on data re-engineering where systems managing similar but incompatible data were combined. Other papers in this special issue on reverse engineering are also relevant. (P. Aiken, A. Muntz, R. Richards, *Comm. ACM*, 37(5), May 1994.)

EXERCISES

28.1 Under what circumstances do you think that software should be scrapped and rewritten rather than re-engineered?

28.2 Compare the control constructs (loops and conditionals) in any two programming languages which you know. Write a short description of how to translate the control constructs in one language to the equivalent constructs in the other.

28.3 Translate the unstructured routine shown in Figure 28.14 into its structured equivalent and work out what it is supposed to do.

28.4 Write a set of guidelines that may be used to help find modules in an unstructured program.

28.5 Suggest meaningful names for the variables used in the program shown in Figure 28.14 and construct data dictionary entries for these names.

28.6 What problems might arise when converting data from one type of database management system to another (e.g. hierarchical to relational or relational to object-oriented)?

28.7 Explain why it is impossible to recover a system specification by automatically analysing system source code.

28.8 Using examples, describe the problems with data degradation which may have to be tackled during the process of data cleanup.

28.9 The Year 2000 problem where dates were represented as two digits posed a major program maintenance problem for many organisations. What were the implications of this problem for data re-engineering?

Figure 28.14
An unstructured program

```
routine BS (K, T, S, L)
B:= 1
NXT:  if S >= B goto CON
L = -1
goto STP
CON:  L := INTEGER (B / S)
L := INTEGER ((B+S) / 2)
if T (L) = K then return
if T(L) > K then goto GRT
B := L+1
goto NXT
GRT:  S := L-1
goto NXT
STP: end
```

28.10 A company routinely places contractual conditions on freelance programmers working on re-engineering their applications which prevents them from taking on contracts with similar companies. The reason for this is that re-engineering inevitably reveals business information. Is this a reasonable position for a company to take given that they have no obligations to contractors after their contract has finished?