# 5

# LEGACY INFORMATION SYSTEMS

> When you have told anyone you have left him a legacy the only decent thing to do is to die at once.
>
> —Samuel Butler

## 5.1 GENERAL IDEA

In Oxford English Dictionary (OED) the word "legacy" is defined as "A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor." In software engineering, no standard definition of legacy exists; Bennett [1] used the following definition to define legacy systems: Legacy software systems are "large software systems that we don't know how to cope with but that are vital to our organization." This is the Phaseout stage of the *staged model* for CSS proposed by Rajlich and Bennett and discussed in Section 3.3 of Chapter 3. Similarly, Brodie and Stonebraker [2] define a legacy system as "any information system that significantly resists modification and evolution to meet new and constantly changing business requirements." Supporting both the definitions are a set of acceptable features of a legacy system:

- large with millions of lines of code;
- geriatric, often more than 10 years old;
- written in obsolete programming languages;

*Software Evolution and Maintenance: A Practitioner's Approach*, First Edition.
Priyadarshi Tripathy and Kshirasagar Naik.

- lack of consistent documentation;
- poor management of data, often based on flat-file structures;
- degraded structure following years of modifications;
- very difficult, if not impossible, to expand;
- runs on old processors which are generally much slower; and
- the support team is totally reliant on the expertise and knowledge of expert individuals.

In addition, these information systems are mission-critical—that is, essential to the organization's business—and must be operational at all times. In summary, due to the lack of available skills and flexibility of the system, there is a strong urge to get rid of the legacy software system as soon as possible and start with something modern.

There are several categories of solutions for legacy information system (LIS) or simply legacy system. These solutions generally fall into six categories as follows [3, 4]:

- *Freeze*. The organization decides to not carry out further work on an LIS.
- *Outsource*. An organization may decide that supporting legacy (or any) software is not its core business. The organization may outsource it to a specialist organization offering this service.
- *Carry on maintenance*. Despite all the problems associated with supporting a software system, the organization decides to carry on maintenance for another period.
- *Discard and redevelop*. The organization throws all the software away and redevelops the application once again from scratch, using a new hardware platform and modern architecture tools and database. This approach removes the legacy problem in one stroke, but also removes a useful source of information for defining the requirements of the new system. The existing system is discarded in spite of it being stable, thoroughly debugged, and being a useful asset. One real issue in this approach is that the business requirements and technology are undergoing frequent changes. Therefore, at the end of a long development process, there is this danger that an organization gets a new system based on obsolete technology that no longer satisfies its business requirements. If the LIS cannot keep pace with changing business needs and if migration is not cost effective, then it is more appropriate to redevelop the system [5].
- *Wrap*. It is a black-box-based modernization technique: surround the LIS with a new layer of software to hide the complexity of the existing interfaces, applications, and data, with the new interfaces. Wrapping gives existing components a modern and improved appearance.
- *Migrate*. By means of migration a legacy system is ported to a modern platform, while retaining most of the system's functionality and introducing minimal disturbance in the business environment. Migration avoids the expensive and long process that generally characterize new development. Rather, reuse of portions

of the LIS, namely, implementation, design, specification, and requirements, is maximized. In addition, the target system runs in a different computing environment. In Section 5.3 migration is discussed in detail.

## 5.2 WRAPPING

In 1988, Dietrich et al. [6] first introduced the concept of a "wrapper" at IBM. Wrapping is a straightforward way to modify and enhance a legacy component. A wrapper does not directly modify the source code, but, instead, it indirectly modifies and changes the software functionality of the legacy component. Wrapping means encapsulating the legacy component with a new software layer that provides a new interface and hides the complexity of the old component. The encapsulation layer can communicate with the legacy component through sockets, remote procedure calls (RPCs), or predefined application program interfaces (API). Wrapping is a "black-box" reengineering activity because the interface of the legacy system is analyzed but not its internal details [7]. The wrapped component is viewed similar to a remote server; it provides some service required by a client that does not know the implementation details of the server. By means of a message passing mechanism, a wrapper connects to the clients. On the input front, the wrapper accepts requests, restructures them, and invokes the target object with the restructured arguments. On the output front, the wrapper captures outputs from the wrapped entity, restructures the outputs, and pushes them to the requesting entity. Well-tested components are reused by organizations via wrappers so as to reduce the maintenance cost. Consequently, legacy components can be invoked from the new state-of-the-art applications, thereby extending and enhancing the life of the legacy components [8]. However, this technique does not solve the problems with legacy systems.

### 5.2.1 Types of Wrapping

Orfali et al. [9] classified wrappers into four categories depending upon what is being wrapped. The four categories of wrappers are explained in the following.

- *Database wrappers.* Database wrappers serve as entry points to existing databases. Those wrappers enable clients implemented in today's object-oriented languages to access legacy databases. Database wrappers can be further classified into *forward wrappers* (*f-wrappers* in Figure 5.1) and *backward wrappers* (*b-wrappers* in Figure 5.2) [10]. The forward wrappers' approach, depicted in Figure 5.1, shows the process of adding a new component to a legacy system. The new component is developed with modern practices, whereas the legacy database and the legacy code are not modified. Therefore, by means of translation service involving both legacy data and queries, the wrapper integrates the new component with the legacy system. Those are called *forward wrappers* because they emulate new technology. The backward wrappers' approach has been depicted in Figure 5.2. In this approach, data are migrated first following the *Database first* approach discussed in Section 5.5.2. Next, new components

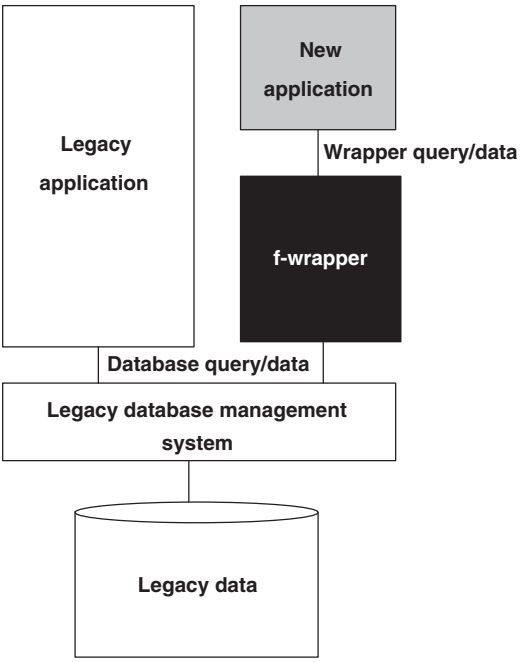


**FIGURE 5.1** Forward wrapper. From Reference 10. © 2006 ACM

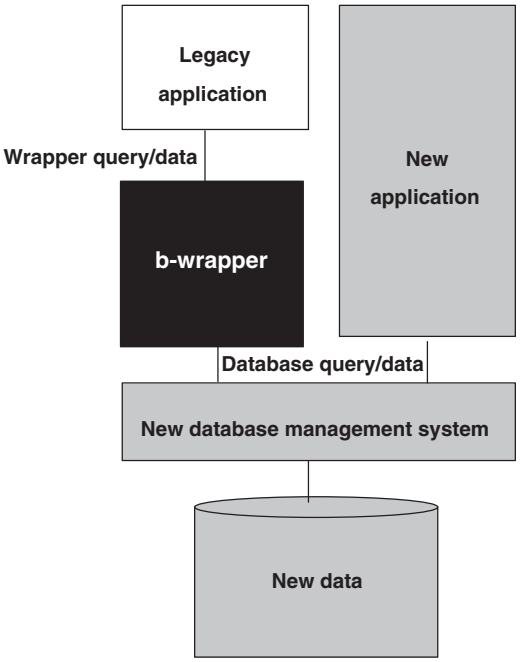


**FIGURE 5.2** Backward wrapper. From Reference 10. © 2006 ACM

are developed that use the new database; the legacy components access the new data via wrappers. Those are called *backward wrappers* because they emulate the legacy technology on top of the new one.

- *System service wrappers.* This kind of wrappers support customized access to commonly used system services, namely, routing, sorting, and printing. A client program may access those services without knowing their interfaces.
- *Application wrappers.* This kind of wrappers encapsulate online transactions or batch processes. These wrappers enable new clients to include legacy components as objects and invoke those objects to produce reports or update files.
- *Function wrappers.* This kind of wrappers provide an interface to call functions in a wrapped entity. In this mechanism, only certain parts of a program—and not the full program—are invoked from the client applications. Therefore, limited access is provided by function wrappers.

### 5.2.2 Levels of Encapsulation

Legacy software has many levels of granularity: procedures are at the lowest level and processes are at the highest level. It may be noted that an entire application is not included in the granularity hierarchy, because an application is just a logical collection of processes executed at different times. Therefore, processes are considered to be at the highest level of granularity. Sneed [11] classified five levels (see Figure 5.3) of granularity at which one can access legacy software applications, as explained in the following:

*Process level.* A job is started on the server which accepts input data, accesses databases, and produces output data. The input data and output data are contained in files. The input files are created by the client program and are

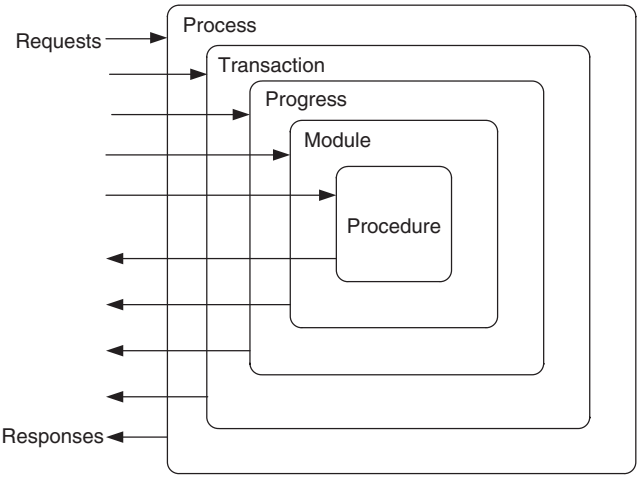


**FIGURE 5.3** Levels of encapsulation. From Reference 11. © 1996 IEEE

transferred to the server by the wrapper by means of a standard file transfer utility. Upon the completion of the job, the wrapper takes over the output files to be forwarded to the client.

*Transaction level.* On the server, a virtual terminal is created that sends the terminal input map and receives the terminal output map. The wrapper is a program which simulates the user terminal. This type transaction level wrapping has become simple because modern transaction processors (i) take care of host-to-client communication and (ii) perform restart and recovery task, exception handling, rollback, and commit.

*Program level.* Via APIs, batch programs are called in a wrapper. The wrapper substitutes program inputs with data inputs coming in from the client application. In addition, outputs from the wrapped program are captured, reformatted, and sent to the client.

*Module level.* Legacy modules are executed using their standard interfaces. A significant deviation from the usual calls is that parameters are passed by value—and not by references. Therefore, first, the wrapper buffers the received values in its own address space. Next, the buffered values are passed on to the invoked module. Finally, the output values received from the invoked module are passed on to the client.

*Procedure level.* A procedure internal to the system is invoked as if the procedure was compiled separately. As a result, this special treatment of an internal procedure requires (i) constructing a parameter interface and (ii) if needed, initializing global variables before calling the procedure.

### 5.2.3 Constructing a Wrapper

A legacy system is wrapped in three steps as follows [12]:

- A wrapper is constructed.
- The target program is adapted.
- The interactions between the target program and the wrapper are verified.

A wrapper, constructed in the first step, is a program which receives input messages from the client program, transforms the inputs into an internal representation, and invokes the target system with the newly formatted messages. The wrapper intercepts the outputs produced by the target system, transforms them into a format understood by the client, and transfers them to the client. Conceptually, a wrapper comprises two interfaces and three event-driven modules as shown in Figure 5.4. The two interfaces are an internal interface and an external interface, and the three modules are message handler, interface converter, and I/O-emulator.

*External interface.* The interface of the wrapper that is accessed by the clients is called the external interface. The wrapper and its client generally communicate by passing messages. Messages normally comprise a header and a body. The header of a message contains control information: (i) identity of the sender; (ii) a time stamp; (iii) the transaction code; (iv) target program type; and (v) the identity of

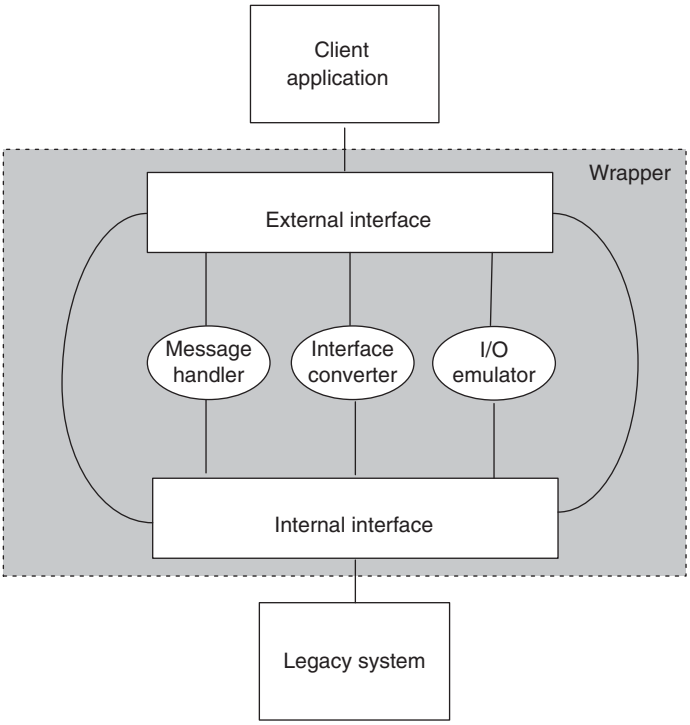


**FIGURE 5.4** Modules of a wrapping framework

the transaction, program, procedure, or module to be invoked. The message body contains the input arguments. Similarly, the message body contains the output values in an output message. The values contained in a message are normally ASCII strings separated by a delimiter, say, a back slash ("\").

*Internal interface.* A wrapper's internal interface is the interface visible to the server. The internal interface is dependent upon the language and type of the wrapped entity. For example, if the wrapped entity is a job, the internal interface of the wrapper is a job control procedure, and the job control procedure is interpreted to execute the job. On the other hand, if the encapsulated entity is a program, procedure, transaction, or a module, the internal interface is a list of parameters in the language of the encapsulated software. Therefore, the internal interface is extracted from the source code of the encapsulated software. The parameters of the internal interface are specified in terms of the target language, thereby necessitating a translation from the input ASCII strings to the target types.

*Message handler.* The message handler buffers input and output messages, because requests from the client may occasionally arrive at a faster rate than they can be consumed. Similarly, outputs from the wrapped program may be produced at a faster rate than they can be sent to the client.

*Interface converter.* This entity converts the internal interface into the external interface and vice versa. In general, between the parameters of the two interfaces,

there is a one-to-one mapping. In some cases, the mapping is one-to-many so the converter duplicates some parameters.

*I/O-emulator.* As the name suggests, the I/O-emulator intercepts the inputs to and outputs from the wrapped entity. Upon intercepting an input, the emulator fills the input buffer with the values of the parameters of the external interface. On the other hand, when it intercepts an output value, the emulator copies the contents of the output buffer into the parameter space of the external interface. Input/output functions of the original environment are emulated without affecting the encapsulated software.

### 5.2.4 Adapting a Program for Wrapper

A wrapped program is modified to some extent, and it is expected that the modified programs continue to operate in the normal mode. Therefore, programs are adapted with tools, rather than manually. Manual adaptations are prone to errors. Sneed [12] recommended four types of tools as discussed below.

*Transaction wrapper.* A transaction wrapper wraps a transaction processing program. If the transaction processing program supports panel input and output operations, then the wrapper converts the said input and output operations into calls to the wrapper. A panel is a particular arrangement of information grouped together for presentation to user.

*Program wrapper.* A batch processing program works on batches of file records. A batch processing program is handled by a wrapper by translating file input/output operations into wrapper calls. To access records on files, write and read statements in the batch processing programs are mapped to call statements in the wrapper. The program, instead of receiving a record from the file, receives the record from the wrapper. Similarly, output records are redirected to the wrapper.

*Module wrapper.* A module wrapper adapts the parameter interfaces of a subprogram that are called by reference. In the "called-by-reference" case, a parameter is an address in the address space of the calling program. If the inputs come from a program on another computer, a problem can arise in accessing the referenced location in the calling program. To solve the problem: (i) the interface of the wrapped module is modified so that values of the input parameters are put together in a single structure in a message buffer in the wrapper and (ii) a reference pointing to the message buffer in the wrapper is passed to the wrapped module.

*Procedure wrapper.* A procedure wrapper makes significant changes to the target program. New interfaces, not existing before, are created. A block of code that has been identified to be encapsulated as a method will get its separate invocation point with a parameter list and its separate exit point as well.

### 5.2.5 Screen Scraping

*Screen scraping* [13] is a common form of wrapping in which modern, say, graphical, interfaces replace text-based interfaces. The new interface can be a GUI (graphical user interface) or even a web-based HTML (hypertext markup language) client. Tools, such as Verastream from Attachmate [14], automatically generate the new screens.

Screen scraping is a short-term solution to a larger problem. Many serious issues are not addressed by simply mounting a GUI on a legacy system. For example, screen scraping (i) does not evolve to support new functions; (ii) incurs high maintenance cost; and (iii) ignores the problem of overloading. Overloading is simply defined as the ability of one function to perform different tasks. Screen scraping simply provides a straightforward way to continue to use a legacy system via a graphical user interface. At best it reduces the cost to train new employees. No new functionality is provided because not much code of the legacy system is overhauled.

## 5.3 MIGRATION

Migration of LIS is the best alternative, when wrapping is unsuitable and redevelopment is not acceptable due to substantial risk. However, it is a very complex process typically lasting 5–10 years [2]. If migration is successful, it gives long-term benefits. It offers better system understanding, easier maintenance, reduced cost, and more flexibility to meet future business requirements. Migration involves changes, often including restructuring the system, enhancing the functionality, or modifying the attributes. But, it retains the basic functionality of the existing system. In this section we discuss general guidelines for migrating a legacy system to a new target system. A brief overview of the migration steps is explained next.

**Migration steps:** LIS migration involves creation of a modern database from the legacy database and adaptation of the application program components accordingly [2, 15, 16]. A database has two main components: schema of the database and data stored in the database. Therefore, in general, migration comprises three main steps: (i) conversion of the existing schema to a target schema; (ii) conversion of data; and (iii) conversion of program.

*Schema conversion.* Schema conversion means translating the legacy database schema into an equivalent database structure expressed in the new technology. Both the schemas must convey the same semantics—that is, all the source data should be losslessly stored into the target database. The transformation of source schema to a target schema is made up of two processes. The first one is called DBRE and it aims to recover the conceptual schema that express the semantics of the source data structure, which is discussed in Section 4.8 of Chapter 4. The second process is straightforward and it derives the target physical schema from this conceptual schema: (i) an equivalent logical schema is obtained from the conceptual schema by means of transformations and (ii) a physical schema is obtained from the logical schema by means of transformations [15].

*Data conversion.* Data conversion means movement of the data instances from the legacy database to the target database. Data conversion requires three steps: extract, transform, and load (ETL) [17]. First, extract data from the legacy store. Second, transform the extracted data so that their structures match the format. In addition, perform data cleaning (a.k.a. *scrubbing or cleansing*) to fix or discard data that do not fit the target database [18]. Finally, load the transformed data in the target database.

*Program conversion.* In the context of LIS migration, program conversion means modifying a program to access the migrated database instead of the legacy data. The conversion process leaves the functionalities of the program unchanged. Program conversion depends upon the rules that are used in transforming the legacy schema into the target schema.

**Testing and functionality:** Migration engineers spent close to 80% of their time on testing the target system [19]. It is important to ensure that the outputs of the target system are consistent with those of the LIS. Therefore, new functionality is not to be introduced into the target system in a migration project. If both the LIS and the target system have the same functionality, it is easier to verify their outputs for compliance. However, to justify the project expense and the risk, in practice, migration projects often add new functionalities.

**Cut over, also referred to as roll over:** The last step of a migration project is the cut over from the LIS to the target system. The event of cutting over to the new system from the old one is required to cause minimal disruption to the business process. There are three kinds of transition strategies, as proposed by Simon [20]:

1. *Cut-and-run*. The simplest transition strategy is to switch off the legacy system and turn on the new system. Rolling over to the new system instantaneously in a single step is too risky because it would entail the entire information flow to be managed by a completely new system.
2. *Phased interoperability*. To reduce risks, cut over is gradually performed in incremental steps. In each step, replace a small number of legacy components—data or application—by their counterparts in the target system. For components to be independently migrated, this approach requires (i) the LIS to be partitioned into modules that perform different functions or (ii) the data into independent fragments.
3. *Parallel operation*. The target system and the LIS operate at the same time. During the period of simultaneous operation, tests are continuously performed on the target system; once the new system is considered to be reliable, the LIS is taken off service.

In practice, for a particular migration project, a transition strategy may be designed by combining all the three approaches, applied to different LIS components.

## 5.4 MIGRATION PLANNING

The success of any large project depends, to a large extent, upon good planning—and a migration project is not an exception. It is not easy to cut over and migrate several hundred databases and programs without interrupting the business of the organization. It is necessary to justify such a mission-critical project to the executive

management. Management must be convinced that the organization will achieve significant benefits without excessively using resources. In addition, with poor or no planning, a lower-quality, much delayed product may be delivered at a higher cost.

The activity of planning for migration comprises several tasks. Seacord et al. [21] suggested that the following 13 steps be considered when planning a migration project:

**Step 1:** Perform portfolio analysis.
**Step 2:** Identify the stakeholders.
**Step 3:** Understand the requirements.
**Step 4:** Create a business case.
**Step 5:** Make a go or no-go decision.
**Step 6:** Understand the LIS.
**Step 7:** Understand the target technology.
**Step 8:** Evaluate the available technologies.
**Step 9:** Define the target architecture.
**Step 10:** Define a strategy.
**Step 11:** Reconcile the strategy with the needs of the stakeholder.
**Step 12:** Determine the resources required.
**Step 13:** Evaluate the feasibility of the strategy.

The objective of this plan is to minimize the risk of modernization effort leading to the development of a migration plan for the LIS. The remainder of the section expands on the these 13 steps.

**Step 1: Perform portfolio analysis.** In the first step, measures are established to evaluate the business value and technical quality of software systems; this is performed by means of portfolio analysis. Portfolio analysis establishes measures of technical quality and business value for a set of software systems. It is represented on a chi-square chart like the one shown in Figure 5.5. The vertical and horizontal axes of the chi-square chart are technical quality and business value, respectively.

Technical quality is computed as a weighted mean of various product and process metrics chosen by the user. Some example criteria for technical quality are ease of making changes, frequency of release notes, accuracy, performance of the system, error rate, cyclomatic complexity, and availability of training. The coefficients are ratios represented on a scale of 0–1. To obtain the coefficient values, first we need to define upper and a lower bounds for each quality metric. The upper and lower bounds are the maximum and minimum acceptable measures, respectively. Then the ratio is defined as [5]:

$$r = 1 - \left( \frac{actual\,measure - lower\_bound}{upper\_bound - lower\_bound} \right).$$

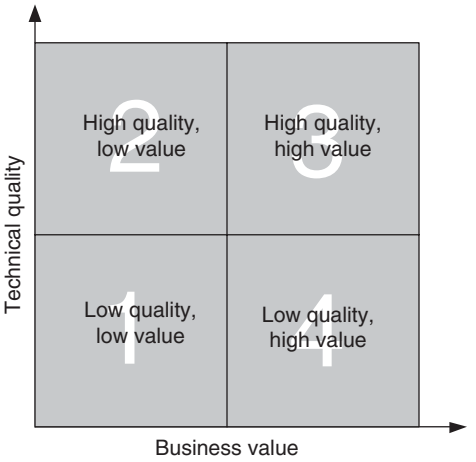


**FIGURE 5.5** Portfolio analysis chi-square chart

The real challenge is to set the upper and lower bounds. It must be defined in terms of current industry standard or in terms of what other practitioners are doing. For example, cyclomatic complexity of value 12 can be considered as the upper bound [22].

The importance of an application or a system to an organization is measured in terms of business value. Some examples of the criteria to establish business value are level of usage, number of business goal satisfied, contribution to profit, user satisfaction, and annual revenue. Business value can be represented as a coefficient in the range [0, 1]. For example, if 100 is the maximum value of a certain criteria, and an application scores 30, then the coefficient of the said criteria is 0.30, which is obtained by dividing 30 by 100.

- *Quadrant 1.* A system with low technical quality and low business value is a prime candidate for substitution with a commercial product if one is available. These systems might be in use in the noncore sector of the organization.
- *Quadrant 2.* A system with high technical quality but low business value need not be replaced, migrated, or restructured.
- *Quadrant 3.* A system with high technical quality and high business value should be actively evolved. The system is in the Evolution stage of the *staged model* for CSS proposed by Rajlich and Bennett and discussed in Section 3.3 of Chapter 3.
- *Quadrant 4.* A system with low technical quality but high business value is a good candidate for redevelopment or migration to a new system.

**Step 2: Identify the stakeholders.** Stakeholders are the people or organizations that influence a system's behavior, or those who are impacted by the system. They typically include architects, developers, maintainers, managers, customers, and end

users. The stakeholders ultimately judge the outcome and impact of the migration project. Each of the stakeholders bring their own perspectives to the table. Therefore, it is necessary to obtain their agreement and support.

**Step 3: Understand the requirements.** Requirements are a description of the needs and desires of stakeholders that a system is expected to implement. There are two challenges in defining requirements. First, ensure that the right requirements are captured. Second, express the requirements in such a way that the stakeholders can easily review and confirm their correctness. Therefore, it is essential to have an unambiguous representation of the requirements and have it made available in a centralized database so that all the stakeholders can review and validate the requirements [23]. Requirements can come from the LIS, business process reengineering, and stakeholders.

- *LIS.* Most requirements in a migration project are derived from the legacy system itself. The risk of deriving requirements from the subject system is that it may result in reimplementation of obsolete business practice.
- *Business process reengineering.* New requirements may come from business process reengineering (BPR). The objective of BPR is to improve the efficiency of business process. Business processes are embedded in existing information system, therefore it is logical to perform BPR before starting a legacy system migration project. Business processes need to be reevaluated, revalidated, and revisited before any migration works can begin.
- *Stakeholders.* Stakeholders generate new requirements, based on years of interactions with the LIS. These requirements are often recorded in outstanding change requests. Stakeholders often want to include technical advances as requirements.

**Step 4: Create a business case.** Based on a business case, executive management can decide whether or not the migration project will increase quality, reduce maintenance costs, and be financially viable. In general, a good business case provides the following information about the migration project:

- *Problem statement.* The problem statement describes the current LIS and highlights the inadequacies, inefficiencies, and weaknesses of the current situation.
- *Solution.* The business case must describe the solution at a high level. For example, the solution can be migrating the LIS to a new software/hardware architecture in an iterative manner. The solution should include an estimate of the cost, the effort needed, and a schedule. When calculating the effort in person-days, one can derive the minimum project duration using a variant of the COCOMO model [24], adjusted for the increased parallelism of the legacy migration project [5].
- *Risks.* Legacy migration projects benefit from identifying risk at an early stage. This allows to gain insight into and control of the project by enumerating things

**TABLE 5.1 Common Quantifiable Benefit Metrics**

| Objective | Sample Quantifiable Benefit Metrics |
|---|---|
| Lower maintenance cost | Average cycle time to close problem reports |
| | Average labor hours to close problem reports |
| | Total staff census |
| | Average problem-report backlog |
| | Post-release fix rework hours |
| Add new functionality | Count of new functions added to the product |
| | Value added or revenue generated by new functions |
| Increase performance | Number of delivered operations, such as transactions, per unit time |
| Replace old equipment | Net annualized cost of purchase and maintenance |
| Recode in different languages | Number of modules in each programming language |
| Reuse of existing artifacts | Number of artifacts used in other products |
| Data rationalization | Number of redundant database objects removed |
| Integrate disjoint applications | Number of unified applications accessible to users |
| | Measures of usability and training time required for application suite |

that can affect the effort and limit the resulting system. This will facilitate the management of the expectations of the stakeholders early, which is a crucial part of any successful migration effort. In addition, the business should also identify the assumptions in the migration project. Assumptions must be documented in case they later turn out to be incorrect or invalid. In either case, it may be necessary to reevaluate the business case.

- *Benefits.* This element of the business case identifies and, ideally, quantifies the benefits of the proposed solution, with an allowance for risks. Two ways to present benefits in a migration project are cost reductions and cost avoidance. Cost reduction refers to the actions taken immediately to reduce the costs. Cost avoidance relates to actions taken to reduce the cost in future. The initial cost of a migration project is high because of up-front investment in training, equipment, or redesign of the system. Therefore, it is helpful to include cost avoidance benefits by a *cost-benefit analysis*. In this analysis, the cost of migration effort is compared to the benefits of redeveloping and with the benefits of doing nothing at all. Moreover, one should find quantifiable benefits that can be measured. This requires the existence of a software metrics program to collect data. Table 5.1 lists sample quantifiable benefit metrics proposed by Tilley and Smith [25] that can serve as objective measures of migration project success.

**Step 5: Make a go or no-go decision.** Once a business case has been defined, it is reviewed by the stakeholders to reach an agreement. If the business case is unsatisfactory then the legacy migration project is terminated at this step.

**Step 6: Understand the LIS.** Understanding the LIS is essential to the success of any migration project. Techniques available to meet this challenge include program

comprehension and reverse engineering. Reverse engineering and program comprehension are discussed in detail in Chapters 4 and 8, respectively.

**Step 7: Understand the target technology.** This activity can proceed in parallel with the activity of Step 6. It is important to understand the technologies that can be used in the migration effort and the technologies that have been used in the legacy system. In general, four types of technologies are of interest in the migration effort:

1. Languages and DBMS available, including COBOL, Java, eXtensible Markup Language (XML), and modern DBMS.
2. Distributed transaction models, including distributed communication and transaction technologies such as RPC or message queues. Key attributes of distributed communication protocol include support for direct or indirect, connectionless or connection-oriented, and asynchronous or synchronous communication.
3. Middleware technologies and standards that may be used to develop an information system, including message-oriented middleware (MOM), XML Messaging, Java 2 Enterprise Edition (J2EE), and Enterprise JavaBeans (EJB).
4. Tools that are available to assist in migration of the LIS to the new information system.

**Step 8: Evaluate the available technologies.** One must compare and contrast all the available technologies to evaluate their capabilities. If the capabilities overlap, then we must appraise these technologies to understand the quality of service they will provide in the migration process. To formulate the eventual architecture and design of the system, those evaluations are performed.

**Step 9: Define the target architecture.** The target architecture is the desired architecture of the new system. It models the stakeholders' vision of the new system. This usually requires descriptions using different views with different levels of granularity. The target architecture is likely to evolve during the migration process. Therefore, the target architecture is continually reevaluated and updated during the migration process.

**Step 10: Define a strategy.** A strategy defines the overall process of transforming the LIS to the new system. This includes migration methodology, that is schema conversion, data conversion, and program conversion, testing, and cut over. For a mission-critical legacy system, deploying the new system all at once is a risky procedure, therefore a legacy system is evolved incrementally to the new system. During the migration effort, many things can change: user requirements may change, additional knowledge about the system may be acquired, and the technology may change. Those changes must be accommodated by the migration effort. While accommodating those changes, a migration strategy needs to minimize risk, minimize development and deployment costs, support an aggressive but reliable schedule, and meet system quality expectations.

**Step 11: Reconcile the strategy with the needs of the stakeholder.** A consensus needs to be developed among stakeholders before implementing the migration plan. The migration strategy developed in the previous step must be reconciled with stakeholder needs. Therefore, this step includes briefing the stakeholders about the approach, reviewing the target architecture, and the strategy. The entire group evaluates the strategy and provides input for the final consensus profile.

**Step 12: Determine the resources required.** We estimate the resource need including cost of implementing the project. One can use the widely used cost estimation model called Constructive Cost Model II (COCOMO II). COCOMO II addresses nonsequential process models, reengineering work, and reuse-driven approach [26]. The COCOMO II model provides estimates of effort, schedule by phases, and staffing by phases and activities.

**Step 13: Evaluate the feasibility of the strategy.** After executing the first 12 steps, the management should have an understanding of the system under migration, the available technology options, a target architecture, migration strategy, cost of migration, and a schedule to effect migration. Based on available information, management determines whether or not the migration strategy is feasible. If the strategy is found to be viable, the migration plan is finalized. On the other hand, if it is unacceptable, a detailed report is produced. Based on the reasons stated in the report, one may revise the migration strategy until (i) a feasible approach can be identified or (ii) the migration strategy is determined to be infeasible and the project is terminated.

## 5.5 MIGRATION METHODS

In this section, seven approaches to migration have been explained [27–29]. No single approach can be applied to all kinds of legacy systems, because they vary in their scale, complexity, and risks of failure while migrating. The seven approaches are as follows:

- Cold turkey
- Database first
- Database last
- Composite database
- Chicken little
- Butterfly
- Iterative

### 5.5.1 Cold Turkey

The *Cold turkey* strategy [2] is also referred to as the Big Bang approach. It involves redesigning and recoding the LIS from the very beginning using a new execution

platform, modern software architecture, and new tools and databases. For a reasonable sized system, this approach takes much time. The risk of failure is high for this approach to be seriously considered. For this approach to be adopted, one must guarantee that the renovated system will include many new features in addition to the functionality provided by the original legacy system. The risk of failure is increased due to the complexity of migration. While the legacy system is under a spell of long redevelopment, new technologies emerge and an organization's business focus could shift. Thus, organizations may perceive the redeveloped system to be not meeting their business goals. In addition, the technology used might have been outdated. However, this approach can be adopted, if a legacy system has stable, well-defined functionality and is small in size.

### 5.5.2 Database First

The *Database first* approach [30] is also known as *forward migration method* [31]. This method first migrates the database, including the data, to a modern DBMS, and then gradually migrates the legacy application programs and interfaces. The LIS simultaneously operates with the new system via a *forward gateway* while interfaces and legacy applications are being reengineered. Implemented as a software module, a gateway mediates among operational software components. This enables the LIS applications to access the database on the target side as shown in Figure 5.6. The forward gateway translates legacy application (old technology) calls to target (new database service) calls. Similarly, outputs of the reengineered database are translated for use by the legacy system.

A key benefit of the database first approach is that upon the completion of migration of the legacy data, one can start receiving productivity benefits by accessing the data.

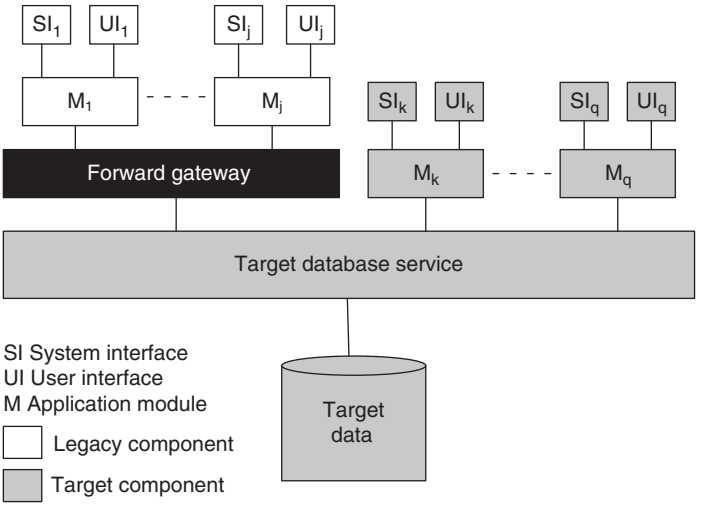


**FIGURE 5.6** Database first approach. From Reference 19. © 1999 IEEE

The legacy system can operate concurrently with those applications and interfaces that have been migrated to the new system. Upon the completion of migration, the gateway will no longer be required. However, this approach has two major drawbacks. First, the approach is only applicable to a completely decomposable legacy system, where a clean interface to the legacy database exists. Second, the new database structure must be defined before migration can begin. The structure of the new database may be negatively impacted by the legacy database. Consequently, it becomes difficult to construct the forward gateway because of the dissimilarities between the legacy system and the target system in terms of database structure and technology.

*Remark:* An information system is said to be fully decomposable if the user and system interfaces, applications, and databases are considered to be distinct components connected by clearly defined interfaces. In a semidecomposable information system, only the user and the system interfaces are separate components; the database service and applications are inseparable. A nondecomposable information system is one where no functional components can be separated [2].

### 5.5.3 Database Last

The *Database last* approach [30] is also known as the *reverse migration method* [31]. It is suitable only for a fully decomposable LIS. In this approach, legacy applications are incrementally migrated to the target platform, but the legacy database stays on the original platform. Migration of the database is done last. Similar to the database first approach, interoperability of both the information systems is supported by means of a gateway. Target applications access the LIS database via a *reverse gateway*. As illustrated in Figure 5.7, the reverse gateway translates calls from the new applications and redirects them to the legacy database.

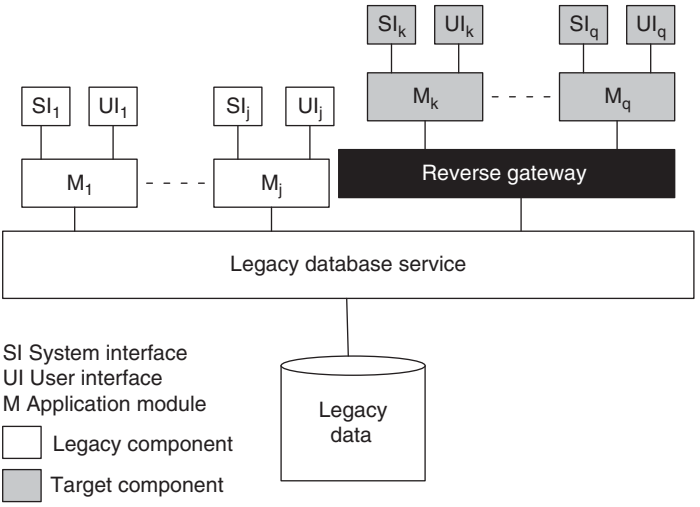


**FIGURE 5.7** Database last approach. From Reference 19. © 1999 IEEE

There are two main issues with the database last approach. First, mapping from the schema of the target database to the legacy database can be slow, thereby impacting new applications. Second, several useful features of relational databases, namely, triggers, integrity, and constraints may not be exploited by the new applications, because those features might not be present in the legacy database.

### 5.5.4 Composite Database

The *composite database* approach [31] is applicable to fully decomposable, semidecomposable, and nondecomposable LISs. However, in practice, a system may not strictly fit in one of those three categories. Most LISs have some decomposable components and some semidecomposable components, whereas the remaining components are nondecomposable.

In the composite database approach, the target information system is run in parallel with the legacy system during the migration process. In the beginning, the target system is a small one, but it grows in size as migration continues. Upon completion of the migration process, all the functionality of the old system are performed by the new system. While migration is continuing, as shown in Figure 5.8, the two systems form a composite information system, employing a combination of forward and reverse gateways. In this approach, data may be duplicated across both the databases: legacy and target. A transaction co-ordinators is employed to maintain data integrity. The co-ordinators intercepts update requests from both target and legacy applications and processes the requests to determine whether or not the requests refer to data replicated in both the databases. If replicated data are referred to, the update is propagated to both the databases using a two-phase commit protocol [32].

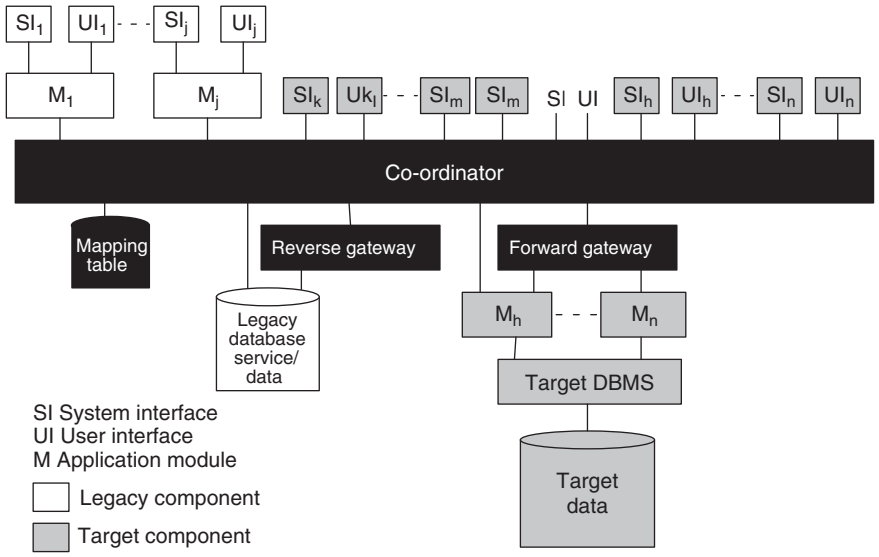


**FIGURE 5.8** Composite database approach. From Reference 19. © 1999 IEEE

The task of analyzing a nondecomposable legacy component is difficult. A good way to analyze such a component is to identify its functionality. Next, the component can be newly developed from a specification of the functionality. As done in the database first and database last approaches, the composite database approach does not require a one-shot large migration of legacy data.

### 5.5.5 Chicken Little

The *Chicken little* strategy [2] refines the composite database strategy, by proposing migration solutions for fully decomposable, semidecomposable, and nondecomposable legacy systems with different kinds of gateways. The differences between those gateways are based upon (i) the locations of the gateways in the system and (ii) the degree of functionality of the gateways. All the gateways have the common goal of mediating between operational software components. Placement of the gateways is a critical factor that affects the complexity of the migration architecture, gateway, and the migration method.

For a fully decomposed LIS, a *database gateway* is located between the database service and the application modules, as explained in Sections 5.5.2 and 5.5.3. The said database gateway can be either a *forward gateway* or a *reverse gateway*. Both the *forward gateway* and the *reverse gateway* are also known as *database gateways*, since those encapsulate the entire database service and the database from the perspective of application modules. An *application gateway* is used for a semidecomposable LIS. This gateway has been illustrated in Figure 5.9, and it is located between user and system interfaces and the legacy application. It is called *application gateway* because it encapsulates from the applications down, from the perspective of interfaces. For nondecomposable systems an *information systems gateway* is located between user and other information systems and LIS. This gateway has been illustrated in

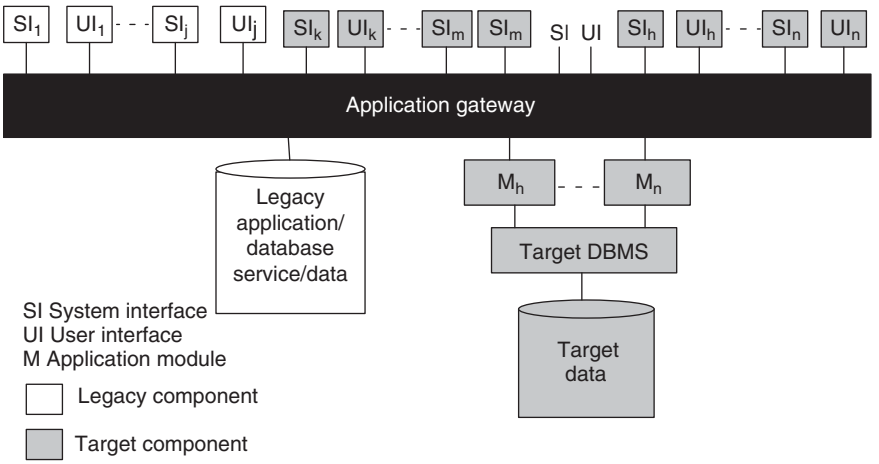


**FIGURE 5.9** Application gateway. From Reference 19. © 1999 IEEE

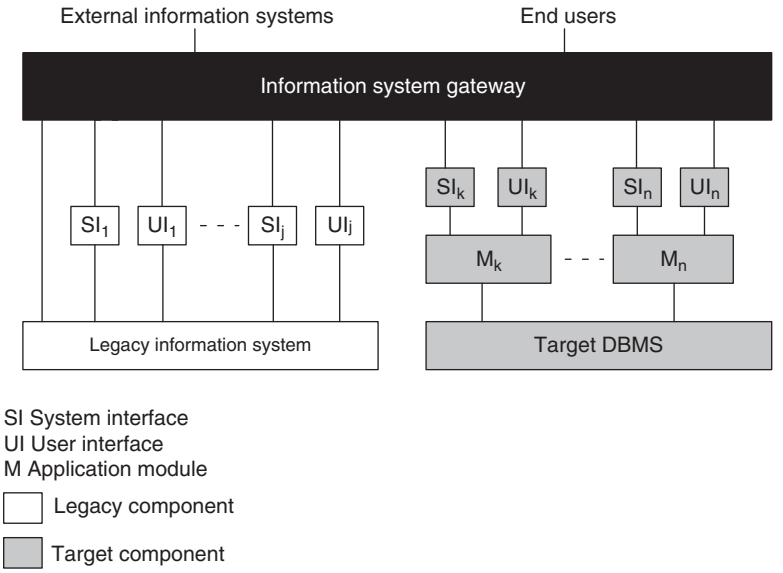


**FIGURE 5.10** Information system gateway. From Reference 19. © 1999 IEEE

Figure 5.10. The entire functionality of the legacy system is encapsulated in an information system gateway, whereas an application gateway encapsulates part of the legacy system, namely, only from application module down. An information system gateway is the primary means for dealing with the migration of user interface. It is recommended to use information gateway in all migration, due to the importance of user interface in migration of LIS. It may be noted that the strategy does not include a testing step. Executing a testing step, to ensure that the new system is equivalent to the legacy system, is useful before rolling over to the new system.

The Chicken little methodology proposes an 11-step generic strategy, as listed on Table 5.2, for migration projects. Each step handles a specific aspect of migration,

**TABLE 5.2 Chicken Little Migration Approach**

| | |
|---|---|
| **Step 1**: | Incrementally analyze the LIS |
| **Step 2**: | Incrementally decompose the structure of the LIS |
| **Step 3**: | Design the interfaces of the target system in an incremental manner |
| **Step 4**: | Build the target applications in an incremental manner |
| **Step 5**: | Design the database in an incremental manner |
| **Step 6**: | Install the target environment in an incremental manner |
| **Step 7**: | Create and install the necessary gateways in an incremental manner |
| **Step 8**: | Migrate the databases in an incremental manner |
| **Step 9**: | Migrate the legacy applications in an incremental manner |
| **Step 10**: | Migrate the legacy interfaces in an incremental manner |
| **Step 11**: | Cut over to the target system in an incremental manner |

and the methodology is designed to be incremental. For example, one step may handle database migration. The methodology can be fine tuned for individual legacy systems. The flow of steps is for each increment to be migrated, so the entire flow is repeated for each increment. Those 11 steps do not need to be performed in the given sequence, and some steps can be executed in parallel. In this methodology, using gateways, the legacy system and the target information system run concurrently during the entire process of migration. Initially, the size of the target system is small. However, with progress in migration, the target system becomes larger in size, and, finally, it is functionally indistinguishable from the old system. In the Chicken little methodology, data is stored in both the migrating legacy system and the evolving target system. Data consistency between the two systems is maintained by means of gateway co-ordinators.

### 5.5.6 Butterfly

The *Butterfly* methodology [33, 34] does not require simultaneous accesses of both the legacy database system and the target database system. Therefore, there is no need to make the two systems consistent. The target system is not operated in the production mode, at the time of reengineering the legacy system. In other words, the old system being reengineered remains operational during the migration stage. From the perspective of developing the target system, the semantics of data captured in schema are more important than the constantly changing data. Therefore, during the migration process, live data are not simultaneously stored in both the new system and the legacy system. For data migration, the Butterfly methodology introduces the following concepts [35]:

- Sample DataStore, Legacy SampleData, and Target SampleData
- TempStore
- Data-Access-Allocator
- Data-Transformer
- Termination-Condition and Threshold Value.

A representative subset of the legacy database is held in *Legacy SampleData*; the *Target SampleData* is derived from the *Legacy SampleData*. Based on the data model of the target system, *Sample DataStore* stores the *Target SampleData*. The *Sample DataStore* supports the initial development and testing of all components.

In order to migrate legacy data, the methodology uses a sequence of temporary datastores called *TempStores* (TS). During the migration process, the TempStores hold the results of manipulations on the legacy data. When migration of legacy data begins, the *Data-Access-Allocator* (DAA) directs those manipulations, and the results are stored by the DAA in the most recent TempStore. Similarly, the DAA retrieves the required data from the most recent TempStore.

To migrate the legacy data and the TempStores data to the target system, the methodology uses a *Data-Transformer* called *Chrysaliser*. Transformation of data

**TABLE 5.3 Phases of Butterfly Methodology**

| | |
|---|---|
| **Phase 1**: | Readiness for migration |
| **Phase 2**: | Comprehend the semantics of the system to be migrated and develop schema(s) for the target database |
| **Phase 3**: | Based upon the Target SampleData, construct a Sample DataStore |
| **Phase 4**: | Except the data, migrate the components of the legacy system |
| **Phase 5**: | Gradually migrate the legacy data |
| **Phase 6**: | Roll over to the new system |

from the legacy system and the TempStores is performed with a set of rules. Based on the earlier process of determining a target schema, the rules are designed. The schema of the target database is conceptually equivalent to the schema of the old database. To determine whether or not the new system is ready to be used, the concepts of *Termination-Condition* and *Threshold Value* are introduced. The *Threshold Value* is represented by a pre-determined value $\epsilon$. Basically, $\epsilon$ represents the acceptable quantity of data remaining in the current TS. The condition *size (TS)* $\leq \epsilon$ implies that the time required to migrate the data is negligibly small to shutdown the legacy system without inflicting much disturbance on the core business. The condition to end migration, denoted by Termination-Condition, is satisfied when $TS_n$ has been completely transformed and size $(TS_{n+1}) \leq \epsilon$ $(n \geq 0)$, where $TS_n$ is the $n$th TempStore.

The following are the key attributes of the methodology. During migration, live data are stored at the LIS. The target system is not rolled over to live operation before fully migrating the system. The legacy data store is "frozen" to be read-only when data migration begins.

The DAA redirects manipulations of legacy data, and the results are saved in a series of temporary data stores called *TempStore(s)* (TS). If a legacy application needs to access legacy data, the DAA extracts data from the correct entity: the current *TempStore* or the legacy database. As shown in Table 5.3, migration of a legacy system is organized into six major steps, also called phases [34].

**Phase 1: Readiness for migration.** Considered to be important issues in the Butterfly methodology are user's requirements and determination of the target system. Table 5.4 [34] lists the main activities in the preparation phase. Success of these activities depends upon much co-operation among users, migration engineers, and LIS experts.

**Phase 2: Comprehend the semantics of the system to be migrated and develop new schema(s) for the target database.** Table 5.5 shows the activities performed

**TABLE 5.4 Migration Activities in Phase 1**

| | |
|---|---|
| 1.1 | Identify the basic requirements of migration |
| 1.1.1 | Identify the user requirements |
| 1.1.2 | Identify the criteria to measure the success of the migration process |
| 1.2 | Design the architecture of the target system |
| 1.3 | Set up the target hardware platform |

**TABLE 5.5 Migration Activities in Phase 2**

| | |
|---|---|
| 2.1 | Comprehend the existing interfaces, determine redundancies in the existing interface, and determine the new interfaces |
| 2.2 | Comprehend the legacy applications, determine redundancies in the legacy applications, and identify the functional requirements of the new applications |
| 2.3 | Comprehend the existing legacy data, determine redundancies in the legacy data, and determine what legacy data to migrate |
| 2.4 | Identify and comprehend the interactions of the legacy system with its environment |
| 2.5 | Identify the requirements of migration |
| 2.6 | Design and develop the data redirector tool called DAA |
| 2.7 | Design the schema for the new data and identify the rules for mapping of data |

**TABLE 5.6 Migration Activities in Phase 3**

| | |
|---|---|
| 3.1 | Construct the Legacy SampleData. |
| 3.2 | Develop the Chrysaliser. |
| 3.3 | Construct the Sample DataStore by using the Target SampleData derived from the Legacy SampleData. |

in this phase. The migration requirements are finalized in activity 2.5. However, not all requirements can be identified until the legacy system is fully understood. In this phase, the DAA tool is developed to redirect all manipulations of legacy data and data stored in TempStores.

**Phase 3: Based upon the Target SampleData, construct a Sample DataStore.** The activities of Phase 3 are listed on Table 5.6. Developing the Chrysaliser and determining the legacy SampleData are the main tasks in Phase 3. The Chrysaliser derives the Sample DataStore from the SampleData, and the Sample DataStore is used to test and develop the target system.

**Phase 4: Except the data, migrate the components of the legacy system.** The activities in Phase 4 are listed on Table 5.7. In this phase, forward software engineering principles and methods are used in the migration process. Constructed in Phase 2, the

**TABLE 5.7 Migration Activities in Phase 4**

| | |
|---|---|
| 4.1 | Migrate legacy interface |
| | 4.1.1 Migrate/develop a portion of the interface of the target system |
| | 4.1.2 For correctness, test the target interface against Sample DataStore |
| | 4.1.3 Validate the target interface against user's requirements |
| 4.2 | Migrate legacy applications |
| | 4.2.1 Migrate/develop a target application |
| | 4.2.2 Test the target application against Sample DataStore for corrections |
| | 4.2.3 Validate the target application against the requirements |
| 4.3 | Migrate the reusable components of the legacy system |
| 4.4 | Integrate target components/system |
| 4.5 | Test target components/system |
| 4.6 | Validate target components/system against the user's requirements |
| 4.7 | Train the users on the new system |

**TABLE 5.8 Migration Activities in Phase 5**

| | |
|---|---|
| 5.1 | Integrate the Data-Access-Allocator into the legacy system. |
| 5.2 | Create TempStore $TS_1$ and set the access mode of the legacy datastore ($TS_0$) to read-only. |
| 5.3 | The contents of $TS_0$ are migrated into the new data store by means of the Chrysaliser. Accesses to the legacy data store are redirected by the DAA while migration is continuing, and results of the manipulations are stored in $TS_1$. |
| 5.4 | Create TempStore $TS_2$; then set $TS_1$ to read-only. |
| 5.5 | Through the Chrysaliser, migrate $TS_1$ into the target datastore(s). The DAA redirects all accesses to the legacy data, and all manipulation results are stored in $TS_2$. |
| 5.6 | Repeat Steps 5.4 and 5.5 for $TS_{n+1}$ and $TS_n$ until the *Termination-Condition* is satisfied. |
| 5.7 | Do not make changes to the legacy system. Transform $TS_0$ into the new, target datastore(s) by means of the Chrysaliser. |
| 5.8 | Train the users about the target system. |

Sample DataStore is used in supporting the "design-develop-test" cycle for new target components. The interactions among the target system's components are verified in activity 4.4.

**Phase 5: Gradually migrate the legacy data.** Table 5.8 [35] lists the activities of Phase 5. Migration of legacy data is performed in Phase 5, and it is central to the Butterfly methodology. Legacy data are incrementally migrated by using TempStores, the Chrysaliser, and the DAA. It may be noted that the Chrysaliser element serves as a data transformer.

Once migration of legacy data is started, no changes are allowed to be performed on the legacy data store. As indicated in Figure 5.11, the DAA redirects manipulation

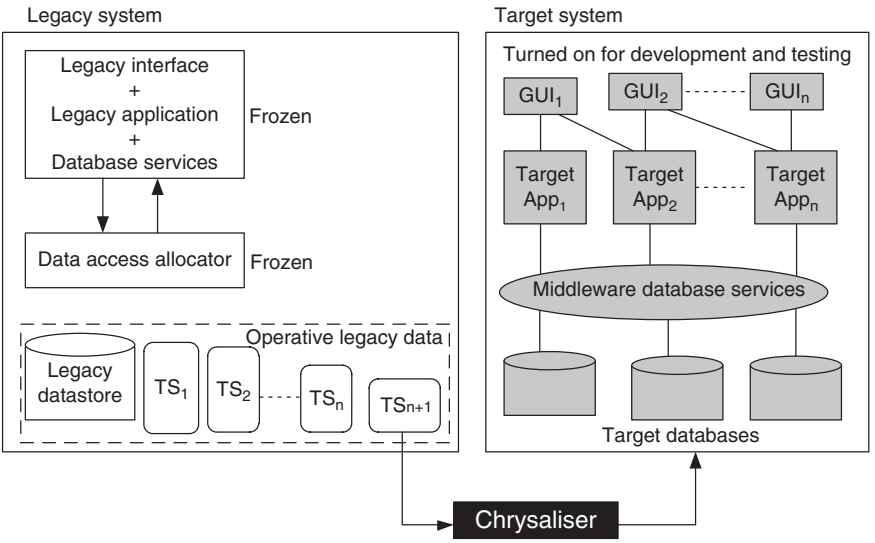


**FIGURE 5.11** Migrating TempStore in Butterfly methodology. From Reference 19. © 1999 IEEE

operations of legacy data, and the results are saved in a series of TempStore(s) (TS). When a legacy application needs to access data, the DAA correctly decides which source—the correct TempStore or the legacy data—to retrieve the data from.

By means of a Data-Transformer, called Chrysaliser, data are migrated to the new system from the legacy store and the series of TempStores. A series of temporary stores (TS's) are used by the Chrysaliser in the data migration process. For example, all manipulations are stored in $TS_1$ when migrating the legacy data $TS_0$. Similarly, all manipulations are stored in $TS_2$ when migrating $TS_1$, and so on.

To determine when one can roll over to the new system, the Butterfly methodology introduces two concepts: a *Termination-Condition* and a *Threshold Value* (represented by $\epsilon$). The allowable amount of data in the final TempStore $TS_N$ is denoted by the pre-determined value $\epsilon$. If size $(TS_N) \leq \epsilon$, where $TS_N$ denotes the final TempStore, the time required to migrate the remaining data in $TS_N$ is sufficiently small to allow the legacy system to be shutdown without creating any problem to the environment. Thus, during the migration process in this methodology, the legacy system is not inaccessible for a significant amount of time. Figure 5.11 depicts a data migration scenario, where it is shown that the Chrysaliser and DAA working together serve as a data migration engine.

**Phase 6: Roll over to the new system.** Roll over to the new system is the final phase of the Butterfly methodology. After the new system is built and the legacy data are migrated, the new system is ready for operation.

The main drawback of the Butterfly methodology is that the legacy database is used for reading only, whereas modifications are placed in a separate, temporary database. Consequently, when there is a need to access some data, the system reads both the databases—the old database and the target database—plus the temporary database. As a result, there is an increase in the time to access the required data. For data-oriented systems, an increase in access time is likely to degrade system performance. In addition, the legacy functions do not simultaneously operate with the newly added functions and the reengineered system.

### 5.5.7 Iterative

The iterative method implies that one component at a time is reengineered. Thus, the legacy system gradually evolves over a period of time. A component represents a physical piece of implementation of a system, including software [29]. The methodology enables simultaneous operations of the reengineered system and the components of the legacy system. The legacy database is gradually moved into a new database, rather than being frozen or duplicated. The components of the reengineered system access either the legacy database or the new database, based upon the location of the actual data to be accessed. This enables a gradual reengineering of the legacy system and the coexistence of the reengineered components with the legacy components. An iterative methodology partitions legacy data into two categories:

1. *Primary data.* Primary data are those data that are essential to an application's business functions.

2. *Residual data.* These data are not required to carry out the application's business functions, but are used by the legacy system. The legacy database holds the residual data until the code that needs to access them is reengineered.

Next, the primary and residual categories of data are explained in detail. Primary data comprise two kinds of data: *conceptual* and *structural* as follows:

- *Conceptual data.* These data describe concepts specific to the application's domain.
- *Structural data.* These data are needed to organize and support data structures that are used to correctly access the conceptual data.

Data in the *residual* category comprise those data in the legacy store which will eventually be discarded. The residual data are grouped into four categories: *control data*, *redundant structural data*, *semantically redundant data*, and *computationally redundant data*. By means of control data one procedure communicates to another procedure that a certain event has occurred. Some data are used to support the organization of the legacy system, but those data are not necessarily needed; those nonessential data are called redundant structural data, and those can be removed with an improved database design. The definition domain of some data is the same as the definition domain of other data, and two identical values in the two domains have the same meanings. Therefore, one of those two definition domains is semantically redundant. Some data is called computationally redundant if it can be derived from another data set in the same database.

Each component may be in one of the following states during the reengineering process:

- *Legacy.* This state means that the component has not been reengineered, which implies that it continues to be a legacy component.
- *Restored.* The structure of the component remains the same and it performs the same functions, but the component accesses data through the new Data Banker.
- *Reengineered.* The component has already been modified, and a desired level of quality has been achieved.
- *New.* The component was not part of the legacy system and has been newly added in order to introduce new functions.

*System architecture.* While reengineering is in progress, Figure 5.12 shows the system architecture of gradual evolution of a legacy system. It is illustrated that legacy, restored (temporary), and reengineered components are enclosed within a unique architecture. The original legacy components have been represented without shading, and those will gradually disappear as reengineering continues. The temporary components are shown with dotted shading, and in this case reengineering of procedures follow data reengineering. The components with dark shade will continue to

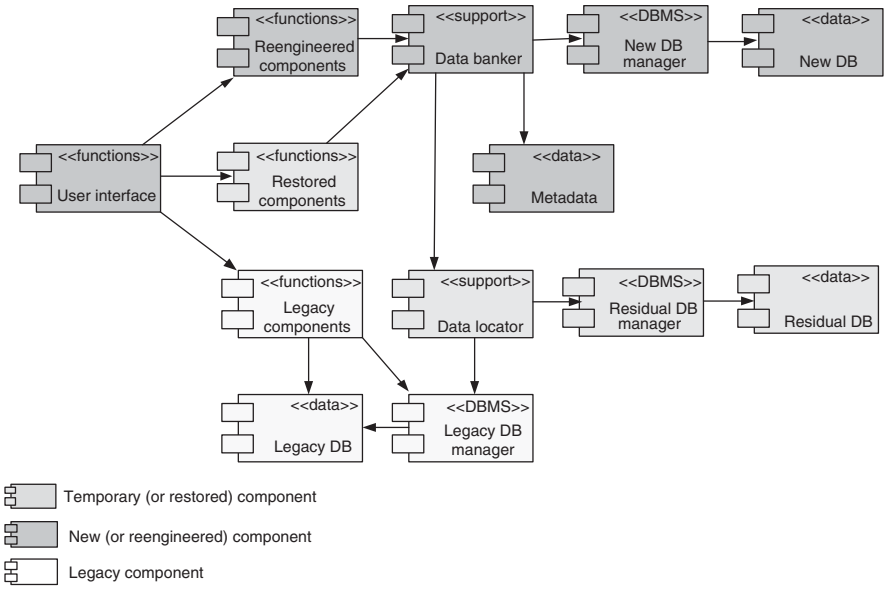


**FIGURE 5.12** The iterative system architecture methodology during reengineering. From Reference 29. © 2003 IEEE

exist to make up the reengineered system. The architecture lets the system to be normally used, while its components undergo modifications. In other words, the entity "User Interface" intercepts the external inputs and invokes the desired components which may be in various states: reengineered, restored, or legacy.

In Figure 5.12, the "Legacy DB manager" manages all accesses of the "Legacy DB" by the legacy components. The database with the new structure is represented by "New DB." "New DB" includes (i) all primary data for those functions which have been added to the new system in the process of reengineering and (ii) all primary data which have been migrated from the legacy DB. "Residual DB" stores the residual data from the legacy database, and "Residual DB Manager" is the corresponding data manager. The "Residual DB Manager" may be the DBMS of the legacy software system or the new database. The "Reengineered Components" and the "Restored Components" get data from "Data Banker." By means of "Metadata," "Data Banker" knows whether or not the required data is available in "New DB," "Residual DB," or "Legacy DB." If the "Data Banker" needs to access the "New DB," metadata are used to retrieve their physical locations. Next, "Data Banker" uses the knowledge of the physical data locations to obtain data from the "New DB Manager." In the aforementioned two cases, "Data Banker" routes the requests for data to the "Data Locator." The "Data Locator" interprets the information that the "Data Banker" has extracted from metadata and gets the required data from the "Residual DB Manager" or the "Legacy DB Manager."

Both the "Data Banker" and "Metadata" will continue to exist beyond the reengineering effort. The structure of "New DB" is known to "Metadata" and the services of

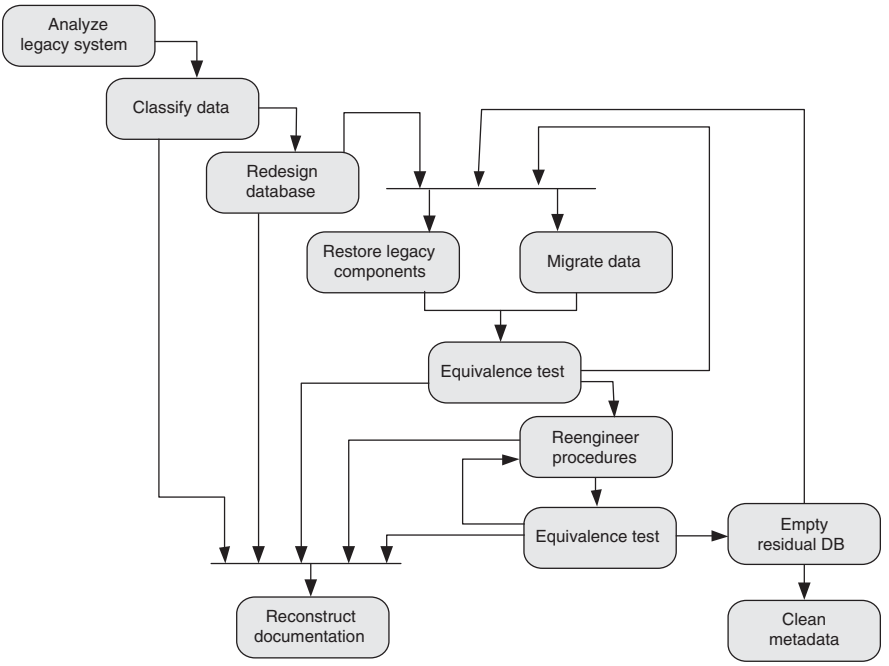


**FIGURE 5.13** The iterative migration process. From Reference 29. © 2003 IEEE

the "New DB Manager" are known to "Data Banker." Therefore, if there are changes to the physical structure of the "New DB", only the contents of "Metadata" need to be modified. On the other hand, changes to the "New DB Manager" require changes to the "Data Banker" alone.

*Migration process.* The migration process diagram is shown in Figure 5.13 [29]. In the following, all the phases are discussed one by one.

**Analyze legacy system:** All the change requests having an influence on a component to be reengineered are put aside until the component is reengineered. Therefore, the system is partitioned into components so as to minimize their client–supplier relationships for two reasons: (i) to minimize the number of change requests (CRs) to be frozen and (ii) to minimize the time durations for which CRs are frozen. Hence, in the partitioning process, identify the legacy system's components which minimize the impact of reengineering the system.

For the $i$th component, $n_i$ denotes the total number of change requests made so far, and $\Delta t_{j,i} = t_{j,i} - t_{j-1,i}$ denotes the time gap between two consecutive change requests. The $j$ index captures successive change requests. $\text{MTMR}_i$ denotes the *mean time to maintenance request* for the $i$th component, and it is expressed as $\text{MTMR}_i = \sum_{j=1}^{n_i} \frac{\Delta t_{j,i}}{n_i}$. For all $i$, $\text{MTMR}_i$ values are used to partition the set of components that need to be reengineered. The historical archive of the system can be used to generate the values of $n_i$ and $\Delta t_{j,i}$. When the expected time for reengineering the $i$th component,

$RT_i$ is less than $MTMR_i$, then it is reasonable to assume that there are unlikely to be requests for change of *i*th component during the time it is being reengineered. Conversely, if $RT_i$ is larger than $MTMR_i$, divide the *i*th component into smaller subcomponents, to prevent the requests for change that are likely to be made at this time during the process.

**Classify data:** Legacy data are classified as *primary* or *residual data*. All the data in the Legacy DB, which are nonduplicated, are recorded in a table produced by means of data classification.

**Redesign database:** The data classified before are restructured in this phase: (i) to reorganize them in the new database for more effective and efficient access; (ii) to eliminate all the defects in the legacy database; and (iii) to eliminate redundant data. While describing the mode of access for the new database, define the characteristics of the data to be stored in the database for Metadata.

**Restore legacy components:** After having reengineered the database, in this phase, the legacy system program is made compatible with the new database. To make the program compatible, the codes involved in accessing the data are identified. The identified code are replaced with new ones that call the Data Banker, rather than access data directly. Depending upon the client's request, after the adaptation, the system acts as follows:

1. If reengineered data are not involved in the request, invoke the procedures of the legacy system that access the legacy data.
2. Otherwise, invoke the procedures of the restored components that access the reengineered data.

The aforementioned behavior is not visible to users, who will continue to use the system as they did before with the legacy system due to the Data Banker component. On the other hand, if users access the reengineered data, the New DB or the Residual DB, will be physically accessed, instead of the Legacy DB. The appropriate choice is executed by the User Interface in Figure 5.12.

**Migrate data:** The legacy database is incrementally migrated to the new database, and nonessential data are put in the residual database. A data migration tool can be developed for this purpose. For each data file to be reengineered, the tool must read all the data it contains, copies them into the Residual DB or New DB based on the information contained in the Metadata.

**Reengineer procedures:** Functions are reengineered in this phase, thereby evolving them from the *restored* state to the *reengineered* state. Quality deficiencies of the procedures are analyzed and suitable remedies are introduced. Specifically, to conduct analysis, one performs the following tasks on individual components:

- Restructure the components to improve their maintainability. For example, application of the principle of information hiding improves maintainability.

- Identify the procedures that are clones of reengineered procedures, and replace those procedures with the reengineered procedures.
- Improve the algorithm used, for better maintainability and performance.
- Update the user interface.
- Update the module interfaces.
- Recall that some maintenance operations had been put on hold during reengineering. Execute those operations.
- Use a modern programming language in the project.

While executing the above tasks, original components of the legacy system are reused as much as possible for two reasons: (i) reduce the cost of reengineering and (ii) preserve the skills that the maintainers have developed. In addition, documentation of the system design is performed in this phase.

**Equivalence tests:** Equivalence testing ensures that, after migration, the system continues to execute the same way as it did before. Therefore, tests are conducted after procedure restoration and data migration. In addition, this phase allows the test plan documentation to be updated.

**Empty residual DB:** In iterative reengineering, some parts of the legacy database are migrated into the New DB, whereas the remaining parts move into the Residual DB. While performing reengineering in an iterative manner, data to be no longer used are removed from the Residual DB. Therefore, by the end of the reengineering process, the Residual DB will be completely empty. Consequently, the Residual Data Manager and the Data Locator components can be withdrawn from the system.

**Iteration:** The components of the legacy system are reengineered one at a time. After reengineering a selected component, the process is repeated for another component and so on, until the entire system has been reengineered.

**Clean metadata:** Upon completion of reengineering, Metadata only reflects the contents of the New DB. Therefore, metadata concerning Residual DB are no more needed; hence, those are eliminated from Metadata. In addition, all procedures in the Data Banker, which communicate with the Data Locator are removed.

**Reconstruct documentation:** Documentation reengineering proceeds concurrently with the actual migration. Up-to-date documentation can be performed throughout the reengineering process.

## 5.6 SUMMARY

This chapter began by identifying the problem an organization faces in dealing with LISs. Next, six viable solutions to the problem were discussed: freeze, outsourcing, carry on maintenance, discard and redevelop, wrap, and migrate.

Next, we studied wrapping techniques in detail. We explained four types of wrappers that practitioners use: database wrapper, system service wrapper, application wrapper, and function wrapper. In addition, we discussed five different levels of encapsulations: process level, transaction level, program level, module level, and procedural level. We introduced a three-step detailed procedure to construct wrappers:

- first, the wrapper is constructed;
- second, the target program is adapted; and
- finally, the interactions between the target programs and the wrapper are tested.

We concluded the wrapper discussion with an introduction to a specific wrapper known as screen scrapper. We then focused our attention on migration of LIS, starting with migration issues and a 13-step migration plan. Next, we discussed seven available migration approaches in detail.

## LITERATURE REVIEW

Narsim Ganti and William Brayman (*The Transition of Legacy Systems to a Distributed Architecture*, John Wiley & Son Inc., Somerset, NJ, 1995) propose guidelines for migrating legacy systems to a distributed environment. LISs are analyzed to identify the systems having data and business logic of value in a distributed environment.

Harry Sneed [5] suggested to consider five steps when planning reengineering of legacy projects: (i) project justification, which analyzes the existing products, the maintenance process, and the business value of the applications; (ii) portfolio analysis, which prioritizes the applications to be reengineered based on their business value and technical quality; (iii) cost estimation, which calculates the cost of executing the project; (iv) cost-benefit analysis, which compares the costs and expected returns; and (v) contracting, which identifies the tasks and the distribution of efforts.

For researchers, we recommend part II of the edited book by Tom Mens and Serge Demeyer (*Software Evolution*, Springer, Berlin, Heidelberg, 2008). Part II of the book has three chapters focusing on migration or reengineering of a legacy software system that is no longer outdated and more easy to maintain and adapt. Ideally, data migration is the process of moving an organizations data from one device to another, without disabling or disrupting the running applications. An organization goes for data migration for a variety of reasons:

- upgradation or replacement of server or storage technology;
- consolidation of servers or storage devices;
- relocation of the data center; and
- optimization of server or storage, including load balancing.

Data upgrade, consolidation, migration, and integration differ in two ways:

*Connectivity between data sources and targets*. The movement of data from their sources to their targets can take three forms: (i) many-to-one; (ii) one-to-one; and (iii) many-to-many. The many-to-one scenario is called consolidation, the one-to-one scenario is called migration or upgrade, and the many-to-many scenario is called integration.

*Diversity of source and target data models*. Data movement is achieved by developing transforms and mappings between source data models and target data models. Therefore, data movements with more diversity will need more number of data models, and, hence, more development time.

## REFERENCES

[1] K. H. Bennett. 1995. Legacy systems: coping with success. *IEEE Software*, January, pp. 19–23.

[2] M. Brodie and M. Stonebraker. 1995. *Migrating Legacy Systems*. Morgan Kaufmann, San Mateo, CA.

[3] A. Cimitile, H. Mủller, and R. Klosch (Eds.). 1997. *Pulling Together*. Proceedings of the International Conference on Software Engineering, Workshop on Migration Strategies for Legacy Systems. Available as Technical Report TUV-1841-97-06 from Technical University University of Vienna, A-1040 Vienna, Austria.

[4] K. Bennett, M. Ramage, and M. Munro. 1999. Decision model for legacy systems. *IEE Proceedings on Software*, June, 153–159.

[5] H. M. Sneed. 1995. Planning the reengineering of legacy systems. *IEEE Software*, January, 24–34.

[6] W. C. Dietrich Jr., L. R. Nackman, and F. Gracer. 1989. *Saving a Legacy with Objects*. Proceedings of the 1989 ACM OOPSLA Conference on Object-Oriented Programming. ACM SIGPLAN Notices, ACM, New York, NY, Vol. 24, No. 10, pp. 77–83.

[7] S. Comella-Dorda, K. Wallnau, R. C. Seacord, and J. Robert. 2000. *A Survey of Black-box Modernization Approaches for Information Systems*. Proceedings of the International Conference on Software Maintenance, October 2000, San Jose, CA. IEEE Computer Society Press, Los Alamitos, CA. pp. 173–183.

[8] F. P. Coyle. 2000. Legacy integration—changing perpectives. *IEEE Software*, March/April, 37–41.

[9] R. Orfali, D. Harkey, and J. Edwards. 1995. *The Essential Distributed Objects Survival Guide*. John-Wiley & Sons, Hoboken, NJ.

[10] P. Thiran, J. Hainaut, G. Houben, and D. Benslimane. 2006. Wrapper-based evolution of legacy information systems. *ACM Transactions on Software Engineering and Methodology*, 15(4), 329–359.

[11] H. M. Sneed. 1996. *Encapsulating Legacy Software for Use in Client/Server Systems.* 3rd Working Conference on Reverse Engineering, Washington, DC. IEEE Computer Society Press, Los Alamitos, CA. pp. 104–119.

[12] H. M. Sneed. 2000. Encapsulation of legacy software: a technique for reusing legacy software components. *Annals of Software Engineering*, 9, 293–313.

[13] D. F. Carr. 1998. Web-enabling legacy data when resources are tight. *Internet World*, August.

[14] Attachmate. 2009. *Application Integration*. Available at http://www.attachmate.com/products/products.html. (accessed February, 2009).

[15] J. Hainaut, A. Cleve, J. Henrard, and J. Hick. 2008. Migration of legacy information systems. In: *Software Evolution* (Eds T. Mens and S. Demeyer), pp. 105–138. Springer-Verlag, Berlin.

[16] J. Henrard, J. Hick, P. Thiran, and J. Hainaut. 2002. *Strategies for Data Engineering*. 9th Working Conference on Reverse Engineering, Washington, DC. IEEE Computer Society Press, Los Alamitos, CA. pp. 211–220.

[17] R. Kimball and J. Caserta. 2004. *The Data Warehouse ETL Toolkit*. John Wiley & Sons, Hoboken, NJ.

[18] E. Rahm and H. Do. 2000. Data cleaning: problems and current approaches. *Data Engineering Bulletin*, 23(4), 3–13.

[19] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. 1999. Legacy information systems: issue and directions. *IEEE Software*, September/October, 103–111.

[20] A. R. Simon. 1992. *System Migration—A Complete Reference*. Van Nonstrand Reinhold, New York, NY.

[21] R. C. Seacord, D. Plakosh, and G. Lewis. 2003. *Modernizing Legacy Systems*. Addison Wesley, Boston, MA.

[22] T. J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, December, 308–320.

[23] K. Naik and P. Tripathy. 2008. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, Hoboken, NJ.

[24] B. Boehm. 1983. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

[25] S. Tilley and D. Smith. 1996. *Perspective on Legacy System Reengineering*. Software Engineering Institute, Pittsburg, PA. p. 146.

[26] B. Boehm, C. Abts, A. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. 2000. *Software Cost Estimation with COCOMO II*. Prentice Hall, Englewood Cliffs, NJ.

[27] J. Bisbal, D. Lawless, B. Wu, J. Grimson, V. Wade, R. Richardson, and D. O'Sullivan. 1997. A survey of research into legacy system migration. *Technical Report TCD-CS-1997-01, Computer Science Department, Trinity College, Dublin*, January, 39.

[28] M. Battaglia, G. Savoia, and J. Favaro. 1998. *Renaissance: A Method to Migrate from Legacy to Immortal Software Systems*. Proceedings of Second Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 197–200.

[29] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio. 2003. Iterative reengineering of legacy systems. *IEEE Transactions on Software Engineering*, March, 225–241.

[30] A. Bateman and J. Murphy. 1994. Migration of legacy system. *Working Paper CA-2894, School of Computer Applications, Dublin City University,* http://www.compapp.dcu.ie/CA_Working_Papers.

[31] M. Brodie and M. Stonebraker. 1993. Darwin: On the incremental migration of legacy information systems. *TR-022-10-92-165*, GTE Labs Inc. Available at http://info.gte.com/ftp/doc/tech-reports/tech-reports.html (accessed March 1993).

[32] D. Bell and J. Grimson. 1992. *Distributed Database Systems*. Addison-Wesely Longman Publishing Co., Boston, MA.

[33] B. Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade, and D. O'Sullivan. 1997. *The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems*. Proceedings of the International Conference on Engineering of Complex Computer Systems, September 1997, Como, Italy. IEEE Computer Society Press, Los Alamitos, CA. pp. 200–205.

[34] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wade, D. O'Sullivan, and R. Richardson. 1997. *Legacy Systems Migration: A Method and its Tool-kit Framework*. Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference, December 1997, Hong Kong, China. IEEE Computer Society Press, Los Alamitos, CA. pp. 312–320.

[35] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wade, R. Richardson, and D. O'Sullivan. 1997. *Legacy Systems Migration: A Legacy Data Migrating Engine*. Proceedings of the 17th International Database Conference, October 1997, Brno, Czech Republic. IEEE Computer Society Press, Los Alamitos, CA. pp. 129–138.

## EXERCISES

**1.** What is a legacy system? Discuss different types of solutions to the legacy problem.

**2.** Which of the following characteristics are part of the definition of a legacy system? Give reasons.

**(a)** The system is very difficult to maintain.

**(b)** The system has a great many users.

**(c)** The system is performing a useful function for its organization.

**(d)** The system has not been modified since it was installed.

**(e)** The system has an old-fashioned terminal-style interface.

**3.** Explain the difference between migration and reengineering.

**4.** Explain the concept of wrapping. Why do you think it is not useful to modernize the LIS in long term?

**5.** What is a database wrapper? Discuss the difference between *b-wrapper* and *f-wrapper* with examples. How these two concepts can be used in migration?

**6.** Why is a screen scrapping technique not the solution to the real issue?

**7.** Calculate coefficient ratio for the following technical quality criteria:

**(a)** Actual error rate is 3 per 1000 lines of code and the maximum and minimum error rates are 7 and 0 per 1000 lines of code, respectively.

**(b)** Actual Cyclomatic complexity is 22 per procedure and the maximum and minimum cyclomatic complexities are 50 and 10 per procedure, respectively.

**8.** Discuss the similarity and difference between Database first and Database last approach.

**9.** Why is the risk of failure for Cold turkey migration strategy high?

**10.** Which of the following is *not one* of the expected benefits of the Chicken little migration approach?

**(a)** Some improvement or new functionality can be delivered to the client in a relatively short timescale.

**(b)** If the current increment fails, then only a limited amount of rework needs to be done.

**(c)** The needs of all users can be satisfied in a short timescale.

**(d)** Cut over of each individual step is less risky.

**11.** Modify the Chicken little steps (see Table 5.2) for (i) forward migration method and (ii) reverse migration method.

**12.** What are the drawbacks of the Butterfly approach?

**13.** Describe the features of Iterative reengineering method that are similar to both Chicken little methodology and Butterfly methodology. What are the advantages of Iterative reengineering method?