2 An abstract model for system evolution

Objectives

- To provide the rationale behind the method for software evolution. An abstract model has been developed, which captures the rationale.
- To describe the abstract model. In particular, we present the model's domain of applicability, its structure, and its process model.
- To identify and classify a handful of useful evolution strategies.

Contents

2.1 The abstract model

2.2 Evolution strategies

In this short chapter, we focus on the model (framework) that has been developed to build the method (introduced in Chapter 3). The method may be viewed as an instance of the model. We begin by presenting the rationale for the system evolution method, and justify the existence of the framework. Section 2.1 introduces the model in terms of its structure and the process model it defines. A fundamental aim of the method is to identify an evolution strategy to be applied to a legacy system. An evolution strategy determines the evolution process. Section 2.2 describes a number of evolution strategies, which are referred to in the remainder of the book.

The state-of-the-art in software evolution is a collection of disparate techniques and concepts. Evolutionary systems and software reengineering, for example, help manage software evolution. They are however, relatively isolated. On their own, they do not address the complete problem of software evolution. What is needed is the glue, which can connect and unite these technologies in a manner that is coherent, and addresses the complete evolution phase of the software lifecycle. This chapter describes a framework for defining such a method.

The framework is abstract in that it identifies a generic model for software evolution. It does not contain the level of detail required by a practitioner to manage an evolution project. Rather, it specifies a high-level structure for viewing legacy systems, and identifies classes of responsibility-driven roles for performing a defined abstract process. The method is an instance of the abstract model, which is fleshed with sufficient detail to be useful to practitioners.

In Chapter 1, we characterised the develop-and-maintain model of software development. A fundamental feature of this model is that it separates development from evolution activities. The shift from system development to evolution generates many problems for those engaged in system evolution. The ultimate result is a significant number of legacy systems, which are difficult and expensive to evolve. This, coupled with the business criticality of many legacy systems provides substantial motivation for (more) effective management of software evolution. The effects on organisations that fail to adopt effective evolution management practice may be fatal.

The evolutionary system paradigm is an approach to software development that unifies development and evolution. An evolutionary system never ceases to evolve. It is designed explicitly for change. Chapter 1 identified elements of a process model and enabling technologies to support evolutionary systems development. The evolutionary model is one fundamental idea on which the abstract model described in this chapter is based.

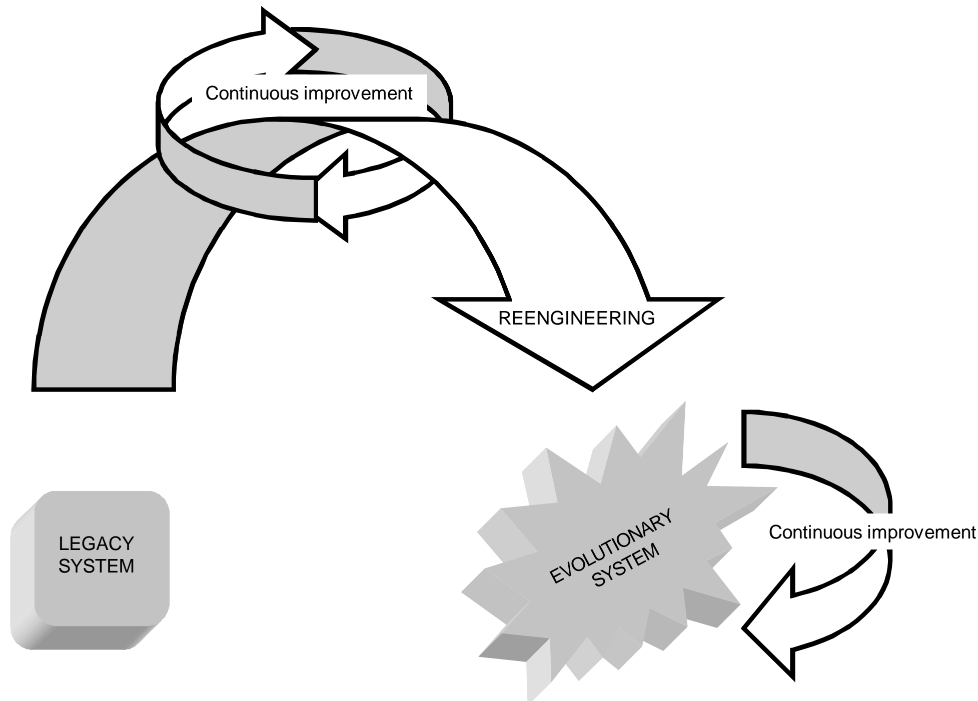

Figure 1 The domain and paradigm

The abstract model also incorporates the ideas of business process reengineering and Kaizen, both of which are from the business process management domain. However, the framework applies these ideas to software systems, and *not* to business processes.

BPR and Kaizen represent opposite approaches for improving business processes. BPR is a radical approach, which essentially involves rethinking a process from scratch. The are several success stories of BPR where efficiency of the new process is orders of magnitude greater than the old process. However, BPR is dangerous because of the risks inherent with radical change. Kaizen is a Japanese approach to business process change based on incremental change, where each increment involves small changes. With Kaizen, a process is continually improved by an on-going and slow-tuning procedure.

There are parallels between the business process change models and software engineering. Similarly to BPR, radical software changes, such as replacing an old system with a new system developed from scratch is associated with the significant risk of developing a system which does not satisfy its user's real needs. Software reengineering can be performed incrementally, and carries less risk than replacement because the starting point for development is an existing system, and not a new set of requirements.

The approach applied to *software evolution*, adopted by the abstract model, is to use both ideas: BPR and Kaizen. Continuous improvement is needed for software systems (essentially, evolutionary systems), but it is generally not realistic to satisfy expectations such as reduced evolution costs, simply by tuning a legacy system. It is not feasible to transform a legacy system into an evolutionary system without some form of reengineering. The framework thus proposes the following paradigm:

1. A stable basis is recovered from the existing software system through reengineering.
2. Kaizen is applied to the recovered system to continuously tune the system to its changing environment in an evolutionary fashion.

Software reengineering, by itself, is not the complete solution to the problem of evolving legacy systems. Where appropriate, reengineering provides a bridge to migrate a system from legacy to evolutionary form. In the cases where reengineering is inappropriate, a new system may be developed which exhibits evolutionary characteristics. The bias of the framework is however on reengineering software systems. Redevelopment is a drastic measure and should only be attempted when reengineering is not feasible.

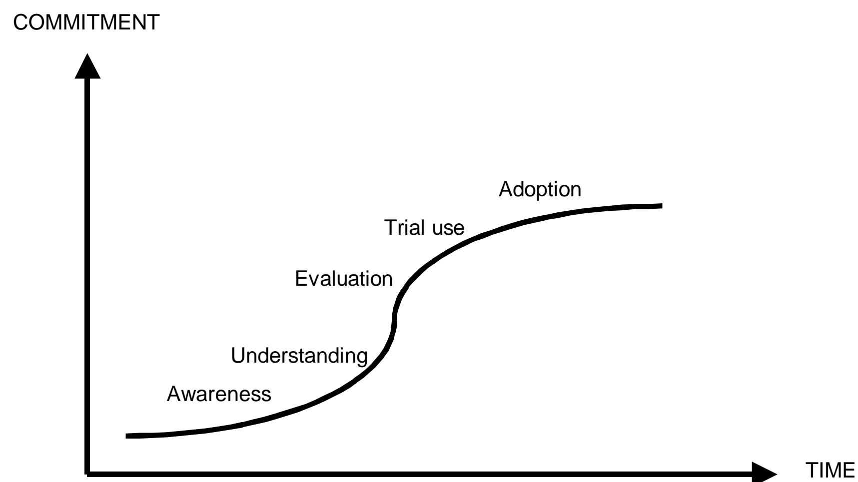

Figure 2 Conner and Patterson's Adoption curve for new technology

The framework not only advocates continuous improvement of software systems, but also of the method. As organisations gain experience with the method, they should be able to tailor it so it better supports their projects and company policies. This situation is shown in Figure 2. Conner and Patterson's curve illustrates how potential customers of some technology become aware of its applicability. If the technology appears promising, customers will invest more effort to understand it. These initial activities take a relatively long time. Promising technology is evaluated. In reality, to reduce risk, customers will only introduce new technology in small increments, or on pilot projects. At this point, lessons can be learnt, and feedback is

generated to tailor the technology to the customers needs. The framework from which the method is based provides this approach to introduce the method in an organisation.

2.1 The abstract model

The abstract model (framework), shown in Figure 3, is the unifying structure that embodies the full spectrum of activities needed to transform a legacy system to its evolutionary complement. The abstract model provides support for identifying, defining, and assessing a host of factors that characterise an evolution project.

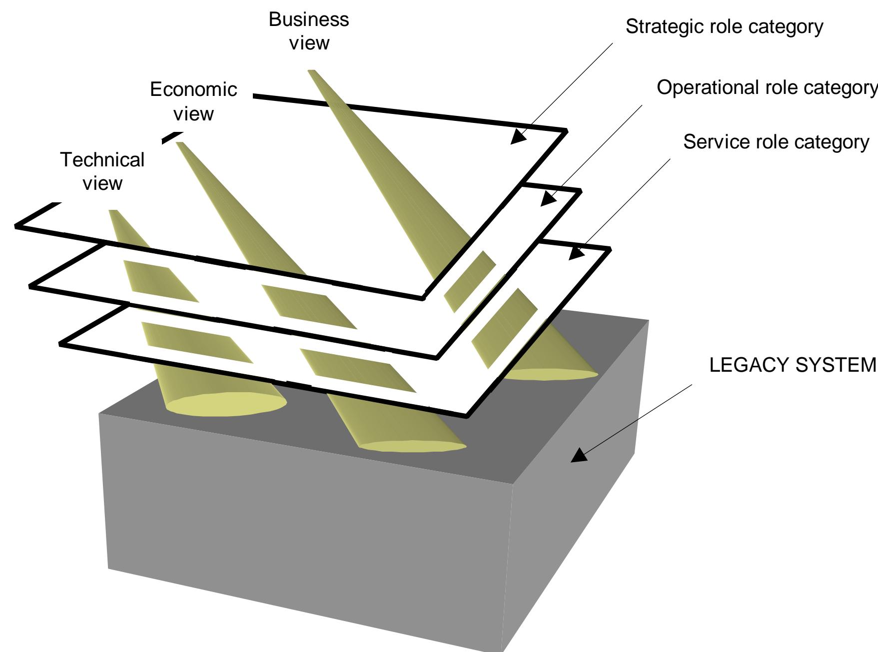

Figure 3 Abstract model structure

The model uses three viewpoints for studying legacy systems:

- *Technical*. This view represents the technical perspective of the system. Particular information that is visible through this view includes the technical condition of the legacy system, documentation, the development and maintenance processes, and the core technologies used.
- *Economic*. The economic perspective is concerned with understanding the business value of the system, and with preparing options for decision making. This view exposes techniques and models for calculating system attributes such as business value in addition to cost and risk estimates for a particular evolution strategy.
- *Managerial*. The decision making process is the focus of this view. Decisions are made after consideration of the technical and economic views.

Each view acts as a filtering mechanism, which focuses on a particular set of related factors. In reality, the system is not represented by a concatenation of the views, but by amalgamation of them. This is because there will be some overlap of the views. The amalgamation of the three views is performed by activities involving roles. Views typically intersect with roles. The framework identifies three role categories:

- *Strategic*. This role is concerned with defining market strategies and identifying future needs of existing systems. In addition, continuously striving for quality improvement and reducing costs form part of the responsibility of this category. Agents who belong to this category are primarily

involved in activities that focus on the managerial and economic views. Strategic management is geared towards long-term decision making which spans several projects.

- *Operational.* The operational role category is concerned with providing effective control over evolution projects. This involves negotiating project contracts with customers and adhering to the organisational strategy. Members of this role category participate in activities that are visible through the technical and economic views. Operational management typically addresses short to medium term needs for a single project.
- *Service.* The main objective of this role category is to provide services, which are required to satisfy the objectives of the strategic and operational roles. Particular service roles work in the technical view of the system.

A role category is an abstraction of some related group of roles. By its abstract nature, the framework does not identify particular roles, such as system operator or reverse engineer. Rather, the framework defines more general role categories, which identify responsibilities that should be fulfilled by individuals involved in an evolution project. In Chapter 3, we offer a set of default roles for each role category. The selection of actual roles is however, project and company specific.

2.1.1 Process model

The structure of the abstract model, described above, represents the static part of the model. Individuals will be assigned particular roles to satisfy the responsibilities of the role categories. These individuals will work within the views of the system defined by the model. The dynamic part of the model is the abstract process model. This is necessary to describe the activities that individuals must perform. Table 1 introduces the six activities of the model.

Activity	Description
Trade-off analysis	A trade-off analysis is the starting point for an evolution project. A thorough investigation of open technical issues, technical market trends and business goals must be made to realise pre-planned product improvement (introduced in Chapter 1).
Issue assessment	The scope and direction of the evolution project are established in this phase. Part of this activity involves assessing the legacy system for fitness for evolution.
Decision analysis	The first two activities feed into decision analysis. This activity involves choosing the most appropriate evolution strategy and developing a plan to implement the selected strategy.
Solution implementation	On the basis of the evolution plan developed in the previous activity, some design effort must be deployed to design a solution that satisfies the requirements identified during analysis. The resulting system should exhibit evolutionary properties.
Solution deployment	The evolutionary system is validated, and if accepted, it is deployed.
Kaizen improvement	Both the evolutionary system and the process of system evolution must be continuously refined. The latter means that the method, defined on the basis of this abstract model should also be subject to evolution.

Table 1 Process model activities

Figure 4 shows the logical representation of the process model. The *what-to-do* activities are concerned with the decision-making process associated with the legacy system. The *how-to-do* phases refer to implementing the evolution strategy, which is derived from the what-to-do activities. In many cases, a business process reengineering exercise may preclude software evolution. Business process reengineering is however out of scope of the framework. This framework deals exclusively with software systems.

The link between the solution deployment and issue assessment activities in

Figure 4 reflects the evolutionary paradigm adopted by the model. An evolutionary system is in a continual state of evolution. The link also captures the continuous improvement property of the method used to mange system evolution.

Continuous improvement of both the software system and the evolution method is promoted by use of a system repository. The repository is used to manage all information concerning the system being evolved. This ranges from system assessment results, through candidate evolution strategies and solution designs, to testing, acceptance and deployment specifications. The repository supports cooperation among activities for a subject legacy system, and exchanges of information between individuals who play the roles involved.

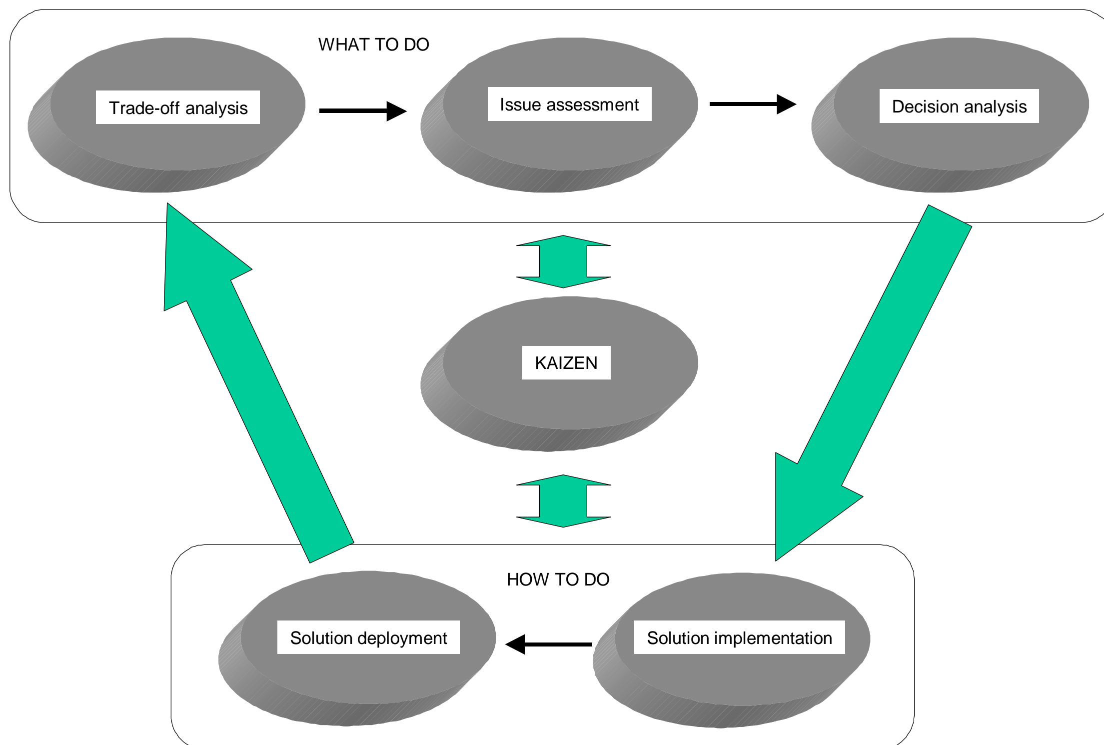

Figure 4 Abstract process model

Trade-off analysis views a legacy system as a black box. The system's internal characteristics are irrelevant to this activity. Rather, the system is analysed as a business asset. The business focus of this activity enables individuals to determine whether the system is actually needed by an organisation. Where a system is necessary to the continued operation of an organisation, it may not actually be required for much longer. Identification of business goals, will in part, determine the required lifetime of a system. Business goals also generate new requirements. These must be extracted in the interests of pre-planned product improvement.

Trade-off analysis is also concerned with understanding new technology and the interdependencies between technologies. Investing in new technology incurs risk. Buying into technology that is inappropriate, or is likely to become obsolete in the near future must clearly be avoided. Trade-off analysis thus prepares individuals for making informed decisions regarding business and technology issues.

Issue assessment involves a white-box assessment of the legacy system. A system is assessed for its fitness for evolution based on its technical, business and organisational composition. The resulting assessment, combined with the knowledge accrued from the trade-off analysis phase, can be used to choose an appropriate long-term strategy. The principal result of this activity is a set of candidate evolution strategies.

Decision analysis involves a robust decision making process, based on the results of the trade-off analysis and issue assessment activities. The decision-making process is thus fed with accurate information. For each of the possible evolution strategies, a cost/benefit/risk analysis exercise is performed. The result of this activity is a decision as to which evolution strategy is most appropriate, and a project plan for implementing the strategy. At this point, it may be decided that the best course of action is to do nothing, but to continue to maintain the system. This would be sensible where the required life of the system is very short. In most cases however, some means of transforming the system towards an evolutionary system will be pragmatic.

Where some form of reengineering is chosen as the evolution strategy, it is vital to the success of the evolution project that the legacy system is sufficiently understood. For reengineering projects, the legacy system is discarded, but transformed towards an evolutionary system. During *solution implementation*, the system must be understood, so that the effects and extent of change can be determined. Depending on the form of reengineering, an adequate level of understanding may not require that the entire system be understood. System models offer the means to promote system understanding.

In addition to understanding the legacy system, solution implementation involves the creation of a validation program, and designing and implementing the evolved system. The validation program will be used in the next phase for assessing the evolved system.

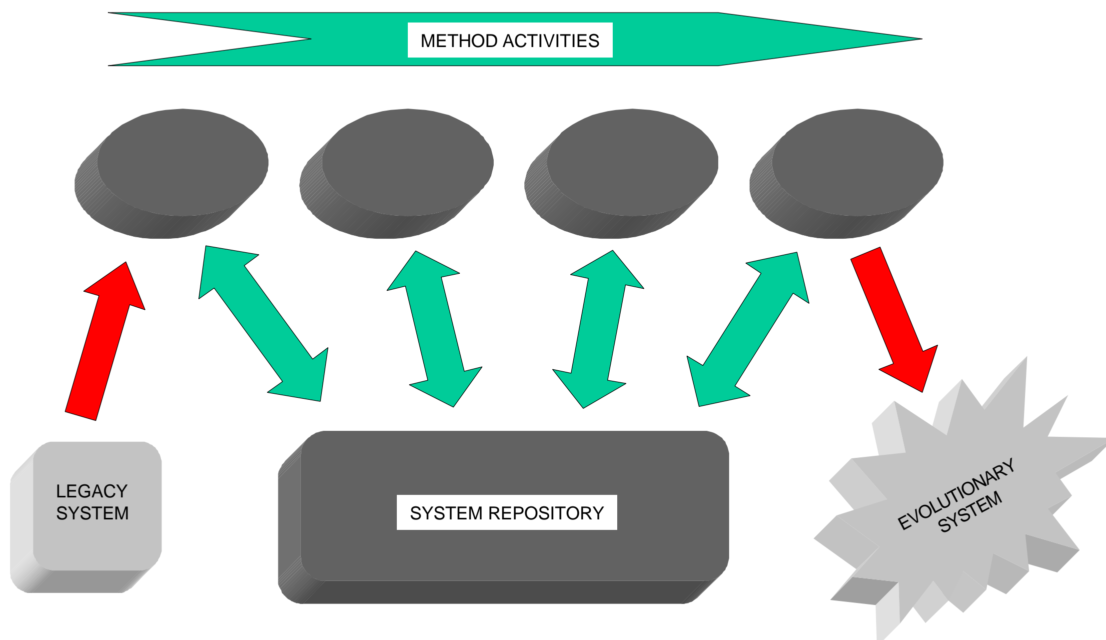

Figure 5 Process model activities and the system repository

Solution deployment involves validation and acceptance testing of the evolved system, prior to deploying it in its operational environment. Users of the system many require training prior to deployment of the

system. Depending on the evolution plan, the evolved system may be introduced incrementally, or as a completely new system. For the former case, parts of the evolutionary system will coexist for some period with the original system. In the latter case, the legacy system will be completely halted before the new system is made operational. This decision is largely affected by the organisation in which the system operates.

Kaizen improvement is a special activity, which is used to enforce the evolutionary system concept for both the subject software system, and the method for system evolution. In Chapter 1, we discussed that a process model supporting evolutionary system development should be based on proven engineering principles. The use of process metrics, process evaluation techniques, and tailoring the method to company-specific characteristics all support Kaizen improvement. The continuous evolution property of evolutionary systems is further supported by use of enabling technology (introduced in Chapter 1) used to implement these systems.

2.2 Evolution strategies

The aim of applying the method to a legacy system is to provide guidance on selecting the most appropriate evolution strategy for the system, and then, to provide advice on implementing the selected strategy. An evolution strategy determines the evolution process. There are three broad classes of evolution strategy (Table 2):

1. Continued maintenance.
2. Reengineering.
3. Replacement.

Strategy	Reengineering	Characteristics
Continued maintenance		Bug fixes and small changes which do not change the structure or architecture of the software.
Revamp	√	The user-interface of a system will be updated to a more modern technology. For example, migration from character mode to GUI. The general structure of the software will remain unchanged.
Restructure	√	The structure of the system will be changed, but the underlying hardware will remain unaltered.
Rearchitecture	√	The structure of the system will be changed in addition to the hardware architecture.
Redesign with reuse	√	A new system will be created from reusable assets of the existing system.
Replacement		The existing system will be replaced by a new developed system without regard for the existing one.

Table 2 Evolution strategies defined by the model

Reengineering, is itself, a classification term for a host of forms of improvement for legacy systems. Chapter 1 introduced a not exhaustive list of possible reengineering forms. The framework abstracts four reengineering strategies. Revamping, restructuring, rearchitecturing, and redesign with reuse provide a workable and useful subset of reengineering approaches. Chapter 4, Evolution planning, advises on the use of the strategies defined by the framework.

Continued maintenance may appear as the obvious strategy for many legacy systems. However, chapter 1 explained the many reasons for aging systems becoming less able to accommodate change. For many legacy systems, the continued maintenance strategy incurs excessive expense because of the effort required to understand contrived software structures, and make changes to them. However, when reengineering assessments are insecure, or when a system is well documented and is maintained by a body of experienced personnel who are familiar with the system, continued maintenance may be appropriate.

In Chapter 1, we provided the following definition for reengineering:

> *Reengineering is the systematic transformation of an existing system into a new form to realise quality improvements in operation, system capability, functionality and performance, or evolvability at lower cost, schedule, or risk to the customer.*

Revamping involves replacing or modifying user interfaces of a legacy system. The internal workings of the system are not improved. Revamping is a useful reengineering strategy for organisation wishing to adopt graphical user interfaces (GUIs). Migrating to GUI technology can often be accomplished without changing any existing source code. This is made possible by middleware products, which sit between the legacy system and user interface. The general aims of middleware are to promote interoperability and portability.

Revamping allows for the coexistence of legacy (typically text-based) interfaces with GUIs. This is desirable to support incremental improvement, and in some cases, it may be that only a subset of user interfaces should be upgraded to GUI. For example, data entry clerks may work better with traditional form-based interfaces, but senior staff may warrant the benefits of GUI technology. Revamping gives the impression of a new system, but the underlying problems of legacy software remain.

Restructuring is the inverse of revamping. It is the transformation of a system's internal structure, without changing any external interfaces. Yourdon explains that "restructuring a program means changing 'spaghetti bowl' programs – programs whose flowcharts resemble the New York City subway map – into programs that are easy to read and comprehend." Restructuring can also be applied to data, otherwise known as data reengineering. Restructuring does not change the semantics of the legacy system. Rather, it elevates the evolvability of a system.

Rearchitecturing is the transformation of a system by reengineering it on a different architecture. Rearchitecturing involves changes to both the hardware and software architectures. For the former, legacy systems are typically centralised mainframe systems, which are migrated to distributed client/server technology. Software rearchitecturing usually means adoption of the Object paradigm. The benefits of distributed client/server systems include openness, fault-tolerance, and scalability. Rearchitecturing involves a design recovery exercise, building models of the system's functionality, and creating object models of the target system. Similarly to restructuring, rearchitecturing yields an evolvable system, but the result of rearchitecturing is the use of modern implementation technology.

Redesign with reuse is the transformation of a system by redeveloping it utilising some parts of the legacy system. This approach to reengineering reduces the effort that would be required for replacement with a new system, because legacy components are salvages and incorporated in the transformed system. A further benefit of this approach is the potential for developing reusable components, which can be used in other systems. This represents a step towards componentware and system families (introduced in Chapter 1).

In reality, a combination of reengineering strategies will typically be applied to a legacy system. For example, a system may be subjected to some revamping to provide senior staff with GUIs, and some restructuring – of data for example to achieve Year 2000 compliance. In another scenario, where redesign with reuse is the dominant strategy, it may first be required to restructure the system. A poorly structured system does not lend itself to identifying reusable components.

System replacement is a total redevelopment of the legacy system from scratch. It represents a drastic strategy, and is only recommended when other strategies are not feasible. For example, where a business process is radically changed, and the legacy system that supports the old process cannot adapt to the new process, replacement may be the only alternative. In other cases, where source code and documentation is nonexistent, reverse engineering may be too difficult and expensive. Organisational factors, such as

company policy to introduce a prescribed system (hardware and software package typically), may also rule out reengineering of the existing system.

Key points

- To provide the rationale behind the method for software evolution. An abstract model has been developed, which captures the rationale.
- The method for system evolution is an instance of the abstract model defined in this chapter.
- The abstract model is a generic framework for evolution. Continuous improvement is fundamental to the framework, and thus to the method.
- The framework provides a high-level structure for studying legacy systems from technical, business, and economic views.
- The framework defines a number of responsibility-driven role categories. These are designed to be instanciated with particular roles in the method. A role identifies the responsibilities of individual(s) involved in the evolution project.
- An abstract process model is described, which is designed to manage the complete evolution phase of the software lifecycle.
- The framework introduces six evolution strategies. An evolution strategy characterises the evolution process for a particular legacy system. In reality, a combination of strategies will be applied to a system.