CHAPTER

27 A STRATEGY FOR SOFTWARE SUPPORT

Regardless of its application domain, its size, or its complexity, computer software will evolve over time. Change drives this process. For computer software, change occurs when errors are corrected, when the software is adapted to a new environment, when customers request new features or functions, and when the application is reengineered to provide benefit in a modern context. Software support actually begins when the developers involve stakeholders in the requirements gathering and prototype evolution process (Figure 27.1). Software support ends with the decision to retire the system from active use.

KEY CONCEPTS

QUICK LOOK

What is it? Software support encompasses a set of activities that correct bugs, adapt the software to changes in its environment, enhance the software based on stakeholder requests, and reengineer the software to achieve better functionality and performance. During these activities, quality must be ensured and change must be controlled.

Who does it? At an organizational level, support staff from the software engineering organization perform all support activities. User training, bug report management, performing warranty repairs, and continuing to manage customer relations may be handled by other specialized teams.

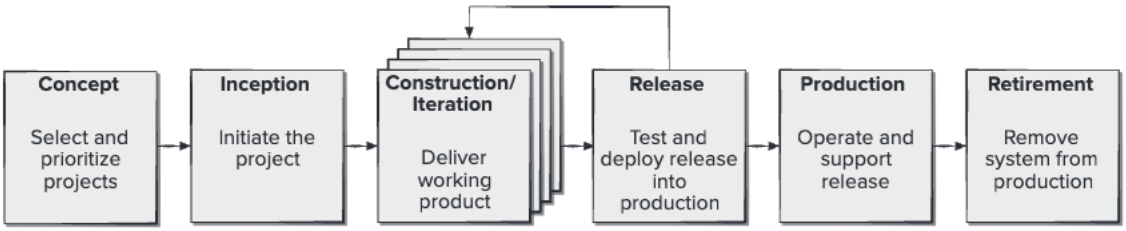
Why is it important? Software exists within a rapidly changing technology and business environment. That's why software must be maintained continually, and at the appropriate time, reengineered to keep pace.

What are the steps? Software support incorporates a maintenance function that corrects defects, adapts the software to meet a changing environment, and enhances functionality to meet the evolving needs of customers. At a strategic level, the support team works with stakeholders to examine the existing business goals for the software product and creates a revised software product to better meet the revised business goals. Software evolution using reengineering creates versions of existing programs that exhibit higher quality and better maintainability.

What is the work product? A variety of maintenance and reengineering work products (e.g., use cases, analysis and design models, test procedures) are produced. The final output is upgraded software that is easier to maintain and better meets the needs of its users.

How do I ensure that I've done it right? Use the same software quality and change management practices that are applied in every software engineering process.

FIGURE 27.1 Software prototype evolution process model



Over the past 40 years, Manny Lehman (e.g., [Leh97a]) and his colleagues have performed detailed analyses of industry-grade software and systems in an effort to develop a *unified theory for software evolution.* The details of this work are beyond the scope of this book, but a brief mention of some of these laws [Leh97b] is worthwhile:

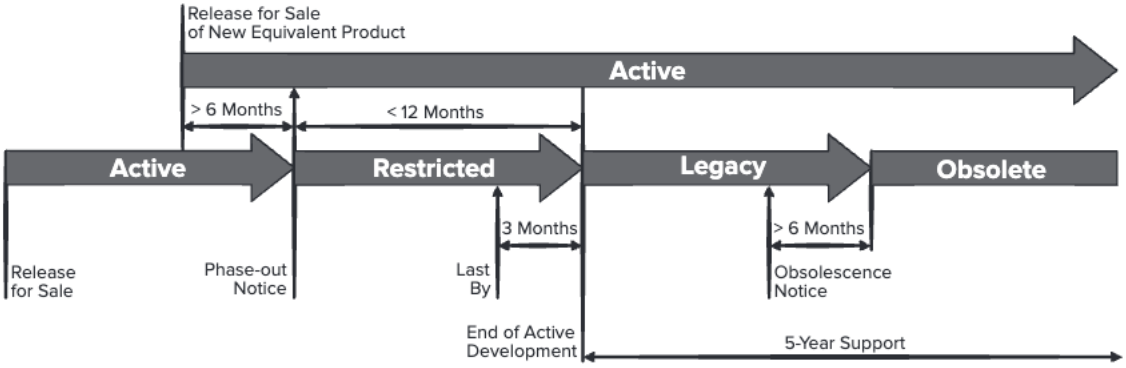
> **Law of continuing change (1974).** Software that has been implemented in a real-world computing context and will therefore evolve over time (called *E-type systems*) must be continually adapted else they become progressively less satisfactory.
>
> **Law of increasing complexity (1974).** As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
>
> **Law of conservation of familiarity (1980).** As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain knowledge of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that knowledge. Hence the average incremental growth remains invariant as the system evolves.
>
> **Law of continuing growth (1980).** The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
>
> **Law of declining quality (1996).** The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The laws that Lehman and his colleagues have defined are an inherent part of a software engineer's reality. In this chapter, we'll discuss the challenges of software support including maintenance and evolution activities that are required to extend the effective life of legacy systems.

27.1 SOFTWARE SUPPORT

Software support can be considered an umbrella activity that includes many activities we have already discussed in this book: change management (Chapter 22), proactive risk management (Chapter 26), process management (Chapter 25), configuration management (Chapter 22), quality assurance (Chapter 17), and release management (Chapter 4). *Release management* is the process that brings high-quality code changes from a developer's workspace to the end user, encompassing code change integration, continuous integration, build system specifications, infrastructure-as-code, and

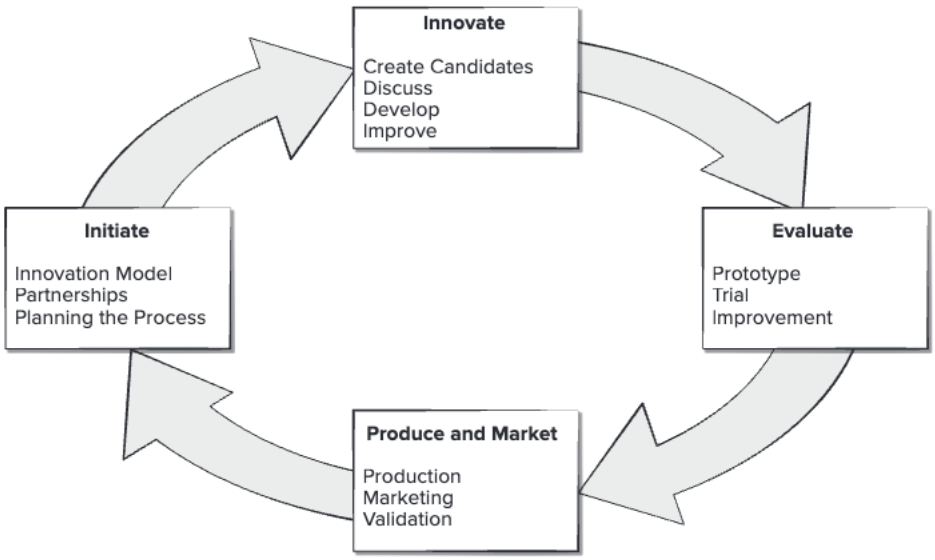
FIGURE 27.2 Software release and retirement example



deployment and release [Ada16]. Ultimately software needs to be retired. Figure 27.2 provides an example of a time line showing the release and retirement of a software product.

To effectively support industry-grade software, your organization (or its designee) must be capable of making the corrections, adaptations, and enhancements that are part of the maintenance activity. But in addition, the organization must provide other important support activities that include ongoing operational support, end-user support, and reengineering activities over the complete life of the software. Figure 27.3 shows one model for supporting software after it is released.

FIGURE 27.3
Iterative software support model



A reasonable definition of *software supportability* is

> . . . the capability of supporting a software system over its whole product life. This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, labor, or any other resource required to maintain the software operational and capable of satisfying its function. [SSO08]

In essence, supportability is one of many quality factors that should be considered during the analysis and design actions that are part of the software process. It should be addressed as part of the requirements model (or specification) and considered as the design evolves and construction commences. There should be some consideration for how long the software will be maintained before it is replaced by a new product.

For example, the need to "antibug" software at the component and code level has been discussed previously in this book. The software should contain facilities to assist support personnel when a defect is encountered in the operational environment (and make no mistake, defects *will* be encountered). In addition, support personnel should have access to a database that contains records of all defects that have already been encountered—their characteristics, cause, and cure. This will enable support personnel to examine "similar" defects and may provide a means for more rapid diagnosis and correction.

Although defects encountered in an application are a critical support issue, supportability also demands that resources be provided to support day-to-day end-user issues. The job of end-user support personnel is to answer user queries about the installation, operation, and use of the application.

27.2 Software Maintenance

Maintenance begins almost immediately. Software is released to end users, and within days, bug reports filter back to the software engineering organization. Within weeks, one class of users indicates that the software must be changed so that it can accommodate the special needs of their environment. And within months, another corporate group that wanted nothing to do with the software when it was released now recognizes that it may provide unexpected benefit. They'll need a few enhancements to make it work in their world.

The challenge of software maintenance has begun. You're faced with a growing queue of bug fixes, adaptation requests, and outright enhancements that must be planned, scheduled, and ultimately accomplished. Before long, the queue has grown long, and the work it implies threatens to overwhelm the available resources. As time passes, your organization finds that it's spending more money and time maintaining existing programs than it is engineering new applications. In fact, it's not unusual for a software organization to expend as much as 60 to 70 percent of all resources on software maintenance for products that have been in active use for several years.

As we noted in Chapter 22, the ubiquitous nature of change underlies all software work. Change is inevitable when computer-based systems are built; therefore, you must develop mechanisms for evaluating, controlling, and making modifications.

Throughout this book, we've emphasized the importance of understanding the problem (analysis) and developing a well-structured solution (design). In fact, Part 2 of the book is dedicated to the mechanics of these software engineering actions, and Part 3 focuses on the techniques required to be sure you've done them correctly. Both analysis and design lead to an important software characteristic that we call maintainability. In essence, *maintainability* is a qualitative indication[1] of the ease with which existing software can be corrected, adapted, or enhanced. Much of what software engineering is about is building systems that exhibit high maintainability.

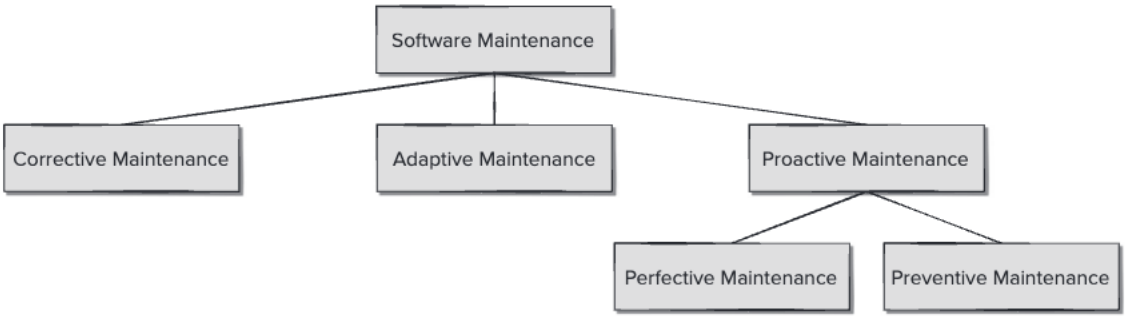
But what is maintainability? Maintainable software exhibits effective modularity (Chapter 9). It makes use of design patterns (Chapter 14) that allow ease of understanding. It has been constructed using well-defined coding standards and conventions, leading to source code that is self-documenting and understandable. It has undergone a variety of quality assurance techniques (Part 3 of this book) that have uncovered potential maintenance problems before the software is released. It has been created by software engineers who recognize that they may not be around when changes must be made. Therefore, the design and implementation of the software must "assist" the person who is making the change.

27.2.1 Maintenance Types

In Chapter 4 we discussed the four types of maintenance shown in Figure 27.4. It is clear that corrective and adaptive maintenance do not add new functionality. It is likely that new functionality will be added to the software during perfective maintenance and possibly during preventative maintenance as well.

In this chapter, we will discuss three broad classes of software maintenance that are relevant to the software support process: reverse engineering, software refactoring, and software evolution or reengineering. Reverse engineering is the process of analyzing a software system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction (Section 27.2.2). Often reverse engineering is used to rediscover system

FIGURE 27.4 Types of software maintenance



1 There are many quantitative measures that provide an indirect indication of maintainability (e.g., [Sch99], [SEI02]).

design elements and redocument them prior to modifying the system source code. Refactoring is the process of changing a software system in such a way that it does not alter its external behavior but improves its internal structure. Refactoring is often used to improve the quality of a software product and make it easier to understand and easier to maintain (Section 27.4). Reengineering (evolution) of software is the process of taking an existing software system and generating a new system from it that has the same quality as software created using modern software engineering practices [Osb90]. Reengineering and software evolution are discussed in Section 27.5.

27.2.2 Maintenance Tasks

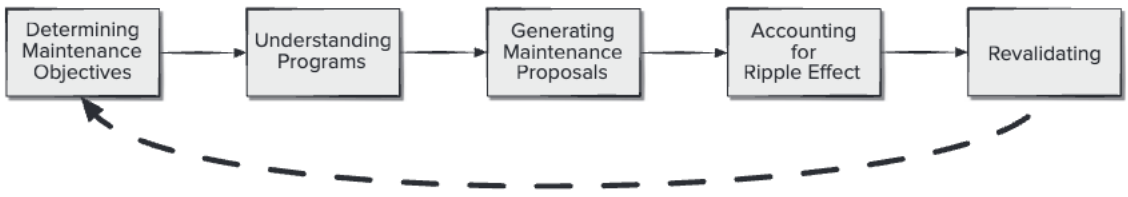
The scenario is all too common: An application has served the business needs of a company for 10 or 15 years. During this time period it has been corrected, adapted, and enhanced many times. People approached this work with the best intentions, but good software engineering practices were always shunted to the side (due to the urgency of other matters). Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur. Yet the application must continue to evolve. What to do?

Unmaintainable software is not a new problem. In fact, the broadening emphasis on software evolution and reengineering (Section 27.5) has been spawned by software maintenance problems that have been building for almost half a century. Figure 27.5 shows a set of generic tasks that should be completed as part of a controlled software maintenance process.

Agile process models similar to the one we described in Chapter 4 deliver incremental prototypes in 4-week sprints. It can be argued that agile developers are in perpetual software support mode as they add new stakeholder requested features in every software increment. However, it is important to realize that software development is not maintenance. It is advisable for separate groups of engineers to handle these two tasks. Heeager and Rose [Hee15] suggest nine heuristics to help the maintenance become more agile.

1. Use sprints to organize the maintenance work. You should balance the goal of keeping customers happy with the technical needs of the developers.
2. Allow for urgent customer requests to interrupt scheduled maintenance sprints, by including time for them during maintenance sprint planning.
3. Facilitate team learning by ensuring that more experienced developers are able to mentor less experienced team members even when working on their own discrete tasks.

FIGURE 27.5 Software maintenance tasks



4. Allow multiple team members to accept customer requests as they arise and coordinate their processing with the other maintenance team members.
5. Balance the use of written documentation with face-to-face communication to ensure planning meeting time is used wisely.
6. Write informal use cases to supplement the other documentation being used for communications with stakeholders.
7. Have developers test each other's work (both defect repairs and new feature implementations). This allows for shared learning and improves the feelings of product ownership by the team members.
8. Make sure developers are empowered to share knowledge with one another. This can motivate people to improve the skills and knowledge (allows developers to learn new things, improves their professional skills, and distributes tasks more evenly).
9. Keep planning meetings short, frequent, and focused.

27.2.3 Reverse Engineering

The first task that needs to be completed by software engineers before responding to any maintenance request is to understand the system that needs to be modified. Sadly, the system being maintained often has low quality and lacks reasonable documentation. This is what technical debt is all about. Technical debt is often caused by developers adding features without documenting them or considering their impact on the larger software system.

Reverse engineering can be used to extract design information from source code, but the abstraction level of this information, the completeness of the documentation, and the degree to which human analysts work comfortably with the available tools, are highly variable. Reverse engineering conjures an image of the "magic slot." You feed a haphazardly designed, undocumented source file into the slot and out the other end comes a complete design description (and full documentation) for the computer program. Unfortunately, the magic slot doesn't exist.

Reverse engineering requires developers to evaluate the old software system by examining its (often undocumented) source code, developing a meaningful specification of the processing being performed, the user interface that was used, and the program data structures or associated database.

Reverse Engineering to Understand Data. This occurs at different levels of abstraction and is often the first reengineering task. In some cases, the first reverse engineering activity attempts to construct a UML class diagram. At the program level, internal program data structures must often be reverse engineered as part of an overall reengineering effort. At the system level, global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems). Reverse engineering of the current global data structures sets the stage for the introduction of a new systemwide database.

Internal Data Structures. Reverse engineering techniques for internal program data focus on the definition of object classes. This is accomplished by examining the

program code with the intent of grouping related program variables together. In many cases, the data organization within the code suggests several abstract data types. For example, record structures, files, lists, and other data structures often provide initial suggestions for possible classes.

Database Structure. Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects. Therefore, reengineering one database schema into another requires an understanding of existing objects and their relationships.

The following steps [Pre94] may be used to define the existing data model as a precursor to reengineering a new database model: (1) build an initial object model, (2) determine candidate keys (the attributes are examined to determine whether they are used to point to another record or table; those that serve as pointers become candidate keys), (3) refine the tentative classes, (4) define generalizations, and (5) discover associations using techniques that are analogous to the CRC approach. Once information defined in the preceding steps is known, a series of transformations [Pre94] can be applied to map the old database structure into a new database structure.

Reverse engineering to understand processing begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

The overall functionality of the entire application must be understood before more detailed reverse engineering work occurs. This establishes a context for further analysis and provides insight into interoperability issues among applications within a larger system. Each of the programs that make up the system represents a functional abstraction at a high level of detail. A block diagram, representing the interaction between these functional abstractions, is created. Each component performs some subfunction and represents a defined procedural abstraction. A processing narrative for each component is developed. In some situations, system, program, and component specifications already exist. When this is the case, the specifications are reviewed for conformance to existing code.[2]

Things become more complex when the code inside a component is considered. You should look for sections of code that represent generic procedural patterns. In almost every component, a section of code prepares data for processing (within the module), a different section of code does the processing, and another section of code prepares the results of processing for export from the component. Within each of these sections, you can encounter smaller patterns; for example, data validation and bounds checking often occur within the section of code that prepares data for processing.

For large systems, reverse engineering is generally accomplished using a semi-automated approach. Automated tools can be used to help you understand the semantics of existing code. The output of this process is then passed to restructuring and forward engineering tools to complete the reengineering process.

2 Often, specifications written early in the life history of a program are never updated. As changes are made, the code no longer conforms to the specification.

Reverse engineering to understand user interfaces may need to be done as part of the maintenance task. Sophisticated GUIs have become *de rigueur* for computer-based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur.

To fully understand an existing user interface, the structure and behavior of the interface must be specified. Merlo and his colleagues [Mer93] suggest three basic questions that must be answered as reverse engineering of the UI commences:

- What are the basic actions (e.g., keystrokes and mouse clicks) that the interface must process?
- What is a compact description of the behavioral response of the system to these actions?
- What is meant by a "replacement," or more precisely, what concept of equivalence of interfaces is relevant here?

Behavioral modeling notation (Chapter 9) can provide a means for developing answers to the first two questions. Much of the information necessary to create a behavioral model can be obtained by observing the external manifestation of the existing interface. But additional information necessary to create the behavioral model must be extracted from the code.

It is important to note that a replacement GUI may not mirror the old interface exactly (in fact, it may be radically different). It is often worthwhile to develop a new interaction metaphor. For example, an old UI that prompts the user to provide a scale factor (ranging from 1 to 10) to shrink or magnify a graphical image. A reengineered GUI might use a touch-screen gesture to accomplish the same function.

27.3 PROACTIVE SOFTWARE SUPPORT

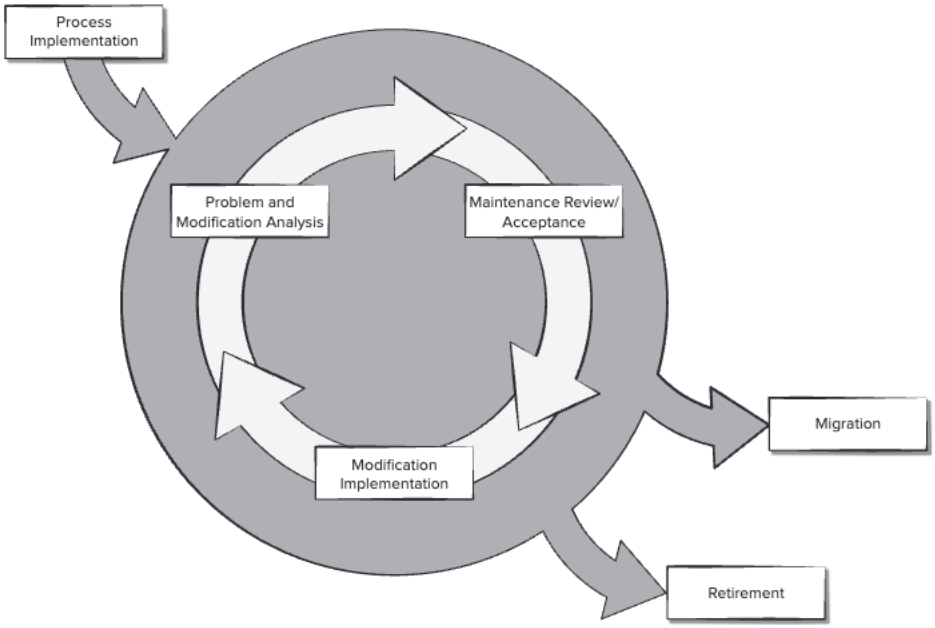
We described the differences between reactive and proactive risk management in Chapter 26. We also described preventative maintenance and perfective maintenance as being proactive maintenance activities (Section 27.2.1). If the goal of software engineering is to deliver high-quality products that meet customer needs in a timely and cost-effective manner, then software support, like other software engineering activities, requires the use of managed workflow processes so that unnecessary rework is avoided.

Supporting software means adapting it to meet changing customer demands and repairing defects reported by end users. This work may be required by law, by contractual agreement, or via product warranty. Repairing software can be a time-consuming and costly process, and it's important to anticipate problems and schedule the work required to respond to customer concerns before they become emergencies. *Proactive software support* requires software engineers to create tools and processes that can help them identify and resolve software issues before they become problems. A generic process for proactive software maintenance and support is shown in Figure 27.6.

The proactive support process is similar to risk monitoring and mitigation (Section 26.6). Developers need to search for indicators that suggest their software

FIGURE 27.6

Software maintenance and support process



product may have quality problems. Sometimes these problems can be addressed by evolving the product and migrating it to a newer version (Section 27.5). Sometimes the software can be restructured or refactored (Section 27.4) to improve its quality and make it easier to maintain. In some cases, the problems are so severe that the developer will need to make plans to retire the product and begin creating a replacement product before the customers abandon it.

27.3.1 Use of Software Analytics

There are currently three dominant uses for artificial intelligence methods in software engineering work [Har12b]: probabilistic reasoning, machine learning and prediction, and search-based software engineering. Probabilistic reasoning techniques can be used to model software reliability (Section 17.7.2). Machine learning can be used to automate the process of discovering root causes of software failures before they occur by predicting the presence of defects likely to cause these failures (Section 15.4.3). Search-based software engineering may be used to assist developers in identifying useful test cases to make regression testing more effective (Section 20.3). All of these AI applications make use of software analytics similar to those we discussed in Chapter 23.

For analytics to be useful they must be actionable, which means expending the effort to determine which measures are worth collecting because of their predictive value and which are not. Port and Tabor [Por18] suggest that analytics can be used to estimate: defect discovery rates based on estimates of yet undiscovered defects present in the product, time between defect discoveries during operation, and effort required to repair defects. Having an understanding of these can allow for better

planning in terms of the cost and time that should be allocated for maintaining the system once it is released to the end users for active use. It is important to keep in mind that even the best estimates contain elements of guessing, so unanticipated failures may still occur.

Zhang and her colleagues [Zha13] report several lessons learned when using software analytics for proactive maintenance tasks.

1. Be sure you are using analytics to identify meaningful development problems, or you will get no buy-in from the software engineers.
2. The analytics must make use of application domain knowledge to be useful to developers (this implies the use of experts to validate the analytics).
3. Developing analytics requires iterative and timely feedback from the intended users.
4. Make sure the analytics are scalable to larger problems and customizable to incorporate new discoveries made over time.
5. Evaluation criteria used needs to be correlated to real software engineering practices.

Mining of the historical information housed in software repositories is a popular way of obtaining the training information needed for the AI techniques mentioned earlier [Sun15]. Using this discovered knowledge helps developers to target their software support actions. Additional discussion of the use of analytics and data science appears in Appendix 2 to this book.

27.3.2 Role of Social Media

Many online stores such as Google Play or Apple's App Store allow users to provide feedback on the apps by posting ratings or comments. The feedback found in these reviews may contain usage scenarios, bug reports, or feature requests. Mining these reports using natural language processing and machine learning techniques can help developers identify potential maintenance and software evolution tasks [Sun15]. However, much of this information is unstructured, and there is so much of it that it is hard to make sense of it without the use of automated statistical tools to reduce the sheer volume of information and create actionable analytics to guide support decisions.

Many companies maintain Facebook pages or Twitter feeds to support their user communities. Some companies encourage their software product users to send program crash information for analysis by the support team members. Still other companies pursue the questionable practice of tracking how and where their products are being used by their customers without their knowledge. It is very easy to collect a lot of user information automatically. Software engineers must resist the temptation to make use of this information in unethical ways.

27.3.3 Cost of Support

In a perfect world, every unmaintainable program would be retired immediately, to be replaced by high-quality, reengineered applications developed using modern software engineering practices. But we live in a world of limited resources. Software evolution and maintenance tasks drain resources that can be used for other business

purposes. Therefore, before an organization attempts to modify or replace an existing application, it should perform a cost-benefit analysis.

A cost-benefit analysis model for reengineering has been proposed by Sneed [Sne95]. Nine parameters are defined:

P_1 = current annual maintenance cost for an application
P_2 = current annual operations cost for an application
P_3 = current annual business value of an application
P_4 = predicted annual maintenance cost after reengineering
P_5 = predicted annual operations cost after reengineering
P_6 = predicted annual business value after reengineering
P_7 = estimated reengineering costs
P_8 = estimated reengineering calendar time
P_9 = reengineering risk factor ($P_9 = 1.0$ is nominal)
L = expected life of the system

The cost associated with continuing maintenance of a candidate application (i.e., reengineering is not performed) can be defined as

$$C_{\text{maint}} = [P_3 - (P_1 + P_2)] \times L \tag{27.1}$$

The costs associated with reengineering are defined using the following relationship:

$$C_{\text{reeng}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \tag{27.2}$$

Using the costs presented in Equations (27.1) and (27.2), the overall benefit of reengineering can be computed as

$$\text{Cost benefit} = C_{\text{reeng}} - C_{\text{maint}} \tag{27.3}$$

The cost-benefit analysis presented in these equations can be performed for all high-priority applications identified as candidates to evolve or retire (Section 27.5). Those applications that show the highest cost-benefit can be targeted for proactive maintenance or evolution, while work on other applications can be postponed until resources are available.

27.4 REFACTORING

Software refactoring (also known as restructuring) modifies source code and/or data in an effort to make it amenable to future changes. In general, refactoring does not modify the overall program architecture. It tends to focus on the design details of individual modules and on local data structures defined within modules. If the refactoring effort extends beyond module boundaries and encompasses the software architecture, restructuring becomes forward engineering (Section 27.5).

Refactoring occurs when the basic architecture of an application is solid, even though technical internals need work. It is initiated when major parts of the software are serviceable and only a subset of all modules and data need extensive modification.[3]

3 It is sometimes difficult to make a distinction between extensive refactoring and evolution. Both are reengineering.

27.4.1 Data Refactoring

Before data refactoring can begin, a reverse engineering activity called *source code analysis* should be conducted. All programming language statements that contain data definitions, file descriptions, I/O, and interface descriptions are evaluated. The intent is to extract data items and objects, to get information on data flow, and to understand the existing data structures that have been implemented. This activity is sometimes called *data analysis*.

Once data analysis has been completed, *data redesign* commences. In its simplest form, a *data record standardization* step clarifies data definitions to achieve consistency among data item names or physical record formats within an existing data structure or file format. Another form of redesign, called *data name rationalization,* ensures that all data naming conventions conform to local standards and that aliases are eliminated as data flow through the system.

When refactoring moves beyond standardization and rationalization, physical modifications to existing data structures are made to make the data design more effective. This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.

27.4.2 Code Refactoring

Code refactoring is performed to yield a design that produces the same function but with higher quality than the original program. The objective is to take "spaghetti-bowl" code and derive a design that conforms to the quality factors discussed in Chapters 15 and 17.

Other restructuring techniques have also been proposed for use with refactoring tools. One approach might rely on the use of anti-patterns (Section 14.5), both to identify bad code design practices and suggest possible solutions to reduce coupling and improve cohesion [Bro98]. Although code refactoring can alleviate immediate problems associated with debugging or small changes, it is not reengineering. Real benefit is achieved only when data and architecture are refactored as well.

27.4.3 Architecture Refactoring

We made the point in Chapter 10 that making architectural changes to a software product already in production can be a costly and time-consuming process. However, when a program with control flow that looks like the graphic equivalent of a bowl of spaghetti, with "modules" that are 2,000 statements long, with few meaningful comment lines in 290,000 source statements and no other documentation must be modified to accommodate changing user requirements, it may be desirable to consider architectural refactoring as one of the design trade-offs. In general, for a messy program like this you have the following options:

1. You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes.
2. You can attempt to understand the broader inner workings of the program in an effort to make modifications more effectively.
3. You can revise (redesign, recode, and test) those portions of the software that require modification, applying a meaningful software engineering approach to all revised segments.

4. You can completely redo (redesign, recode, and test) the complete program, using reengineering tools to assist in understanding the current design.

There is no single "correct" option. Circumstances may dictate the first option even if the others are more desirable.

Rather than waiting until a maintenance request is received, the development or support organization uses the results of inventory analysis to select a program that (1) will remain in use for a preselected number of years, (2) is currently being used successfully, and (3) is likely to undergo major modification or enhancement in the near future. Then, option 2, 3, or 4 is applied. We will discuss software evolution and reengineering in Section 27.5.

27.5 SOFTWARE EVOLUTION

At first glance, the suggestion that you redevelop a large program when a working version already exists may seem quite extravagant. Reengineering takes time, it has significant cost, and it absorbs resources that might be otherwise occupied on immediate concerns. For all these reasons, reengineering is not accomplished in a few months or even a few years.

Reengineering of software systems is an activity that will absorb software engineering resources for many years. That's why every organization needs a pragmatic strategy for software reengineering. If time and resources are in short supply, you might consider applying the Pareto principle to the software that is to be reengineered and apply the reengineering process to the 20 percent of the software that accounts for 80 percent of the problems.

Before passing judgment, consider the following arguments. The cost to maintain one line of source code may be 20 to 40 times the cost of initial development of that line. In addition, redesign of the software architecture (program and/or data structure), using modern design concepts, can greatly facilitate future maintenance. Automated tools for reengineering or software evolution will make some part of the job easier. Because a prototype of the software already exists, development productivity should be much higher than average. The user now has experience with the software. Therefore, new requirements and the direction of change can be ascertained with greater ease. At the end of this evolutionary preventive maintenance, the developers will end up with a complete software configuration (documents, programs, and data).

Reengineering is a rebuilding activity. To better understand it, consider an analogous activity: the rebuilding of a house. Consider the following situation. You've purchased a house in another state. You've never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt. How would you proceed?

- Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.
- Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).

- Before you start rebuilding, be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.
- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.

Although these principles focus on the rebuilding of a house, they apply equally well to the evolving computer-based systems and applications.

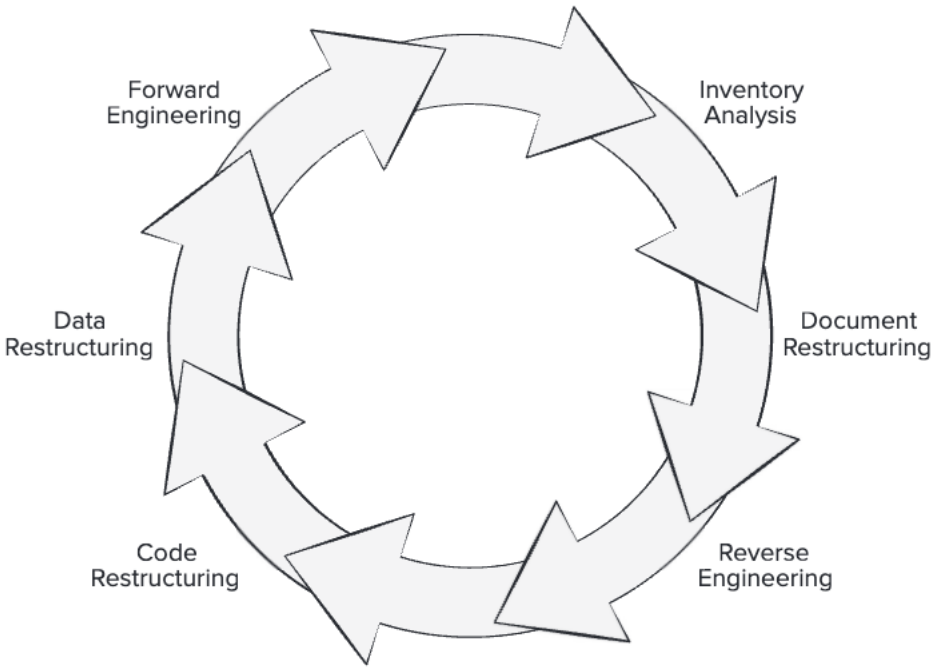
To implement these principles, you can use a cyclical process model for reengineering like the one shown in Figure 27.7. This model defines six activities. Because it is cyclical, each of the activities presented may be revisited as often as needed. For any particular cycle, the process can terminate after any one of these activities.

27.5.1 Inventory Analysis

Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity,

FIGURE 27.7

A software reengineering process model



current maintainability and supportability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.

It is important to note that the inventory should be revisited on a regular basis. The status of applications (e.g., business criticality) can change as a function of time, and as a result, priorities for reengineering will shift.

27.5.2 Document Restructuring

Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options? In some cases, creating documentation when none exists is simply too costly. If the software works, let it be! In other cases, some documentation must be created, but only when changes are made. If a modification occurs, document it. Finally, there are situations in which a critical system must be fully documented, but even here, documents should achieve an essential minimum. Your software organization must choose the documentation option that is most appropriate for each case.

27.5.3 Reverse Engineering

Reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is a process of *design recovery*. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

27.5.4 Code Refactoring

The most common type of reengineering (actually, the use of the term *reengineering* is questionable in this case) is *code refactoring*. Some legacy systems have a solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of good design practices are noted, and code is then refactored or even rewritten in a more modern programming language. The resultant refactored code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

27.5.5 Data Refactoring

A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, information architecture has more to do with the long-term viability of a program than the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data restructuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak (e.g., flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.

Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

27.5.6 Forward Engineering

In an ideal world, applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an "engine" will appear, but vendors have introduced tools that provide a limited subset of these capabilities that addresses specific application domains (e.g., applications that are implemented using a specific database system). More important, these reengineering tools are becoming increasingly more sophisticated.

Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software re-creates the function of the existing system and also adds new functions and/or improves overall performance. In most cases, forward engineering does not simply create a modern equivalent of an older program. Rather, new user and technology requirements are integrated into the reengineering effort. The redeveloped program extends the capabilities of the older application.

27.6 SUMMARY

Software support is an ongoing activity that occurs throughout the life cycle of an application. During support, maintenance actions are initiated, defects are corrected, applications are adapted to a changing operational or business environment, enhancements are implemented at the request of stakeholders. In addition, users are supported as they integrate an application into their personal or business work flow.

Software maintenance and support activities need to be proactive in their nature. It is better to anticipate problems and remove their root causes before the customers find them and become dissatisfied with the software product. The use of software analytics may help software developers identify potential defects and maintenance issues before they become problematic.

At the software level, reengineering examines information systems and applications with the intent of restructuring or reconstructing them so that they exhibit higher quality. Software evolution or reengineering encompasses a series of activities that include inventory analysis, document restructuring, reverse engineering, program and data restructuring, and forward engineering. The intent of these activities is to create versions of existing programs that exhibit higher quality and better maintainability—programs that will be viable well into the twenty-first century.

The cost-benefit of reengineering can be determined quantitatively. The cost of the status quo, that is, the cost associated with ongoing support and maintenance of an existing application, is compared to the projected costs of reengineering and the resultant reduction in maintenance and support costs. In almost every case in which a program has a long life and currently exhibits poor maintainability or supportability, reengineering represents a cost-effective business strategy.