Treating maintenance as a reuse-oriented development process provides a choice of maintenance approaches and improves the overall evolution process.

If you believe that software should be developed with the goal of maximizing the reuse of experience in the form of knowledge, processes, products, and tools, the maintenance process is logically and ideally suited to a reuse-oriented development process. There are many reuse models, but the key issue is which process model is best suited to the maintenance problem at hand.

In this article, I present a high-level organizational paradigm for development and maintenance in which an organization can learn from development and maintenance tasks and then apply that paradigm to several maintenance process models. Associated with the paradigm is a mechanism for setting measurable goals so you can evaluate the process and the product and learn from experience.

Maintenance models

Most software systems are complex, and modification requires a deep understanding of the functional and nonfunctional requirements, the mapping of functions to system components, and the interaction of components. Without good documentation of the requirements, design, and code with respect to function, traceability, and structure, maintenance becomes a difficult, expensive, and error-prone task. As early as 1976, Les Belady and Manny Lehman reported on the problems with the evolution of IBM OS/360. The literature is filled with similar examples.

Maintenance comprises several types of activities: correcting faults in the system, adapting the system to a changing operating environment (such as new terminals and operating-system modifications), and adapting the system to changes in the original requirements. The new system is...
scribing the existing system, from requirements through code. Although this may be a naïve assumption in practice, a side effect of this article’s presentation should be to motivate organizations to gain the benefits of having such documentation.

**Quick-fix model.** The quick-fix model represents an abstraction of the typical approach to software maintenance. In the quick-fix model, you take the existing system, usually just the source code, and make the necessary changes to the code and the accompanying documentation and recompile the system as a new version. This may be as straightforward as a change to some internal component, like an error correction involving a single component or a structural change or even some functional enhancement.

Figure 2 demonstrates the flow of change from the highest-level document affected by the change through the lowest-level document. This model supports the reuse orientation more explicitly. An environment that supports the iterative-enhancement model clearly supports the quick-fix model.

**Iterative-enhancement model.** Iterative enhancement is an evolutionary model proposed for development in environments where the complete set of requirements for a system was not fully understood or where the developer did not know how to build the full system. Although iterative enhancement was proposed as a development model, it is well suited to maintenance. It assumes a complete and consistent set of documents describing the system. The iterative-enhancement model starts with the existing system’s requirements, design, code, test, and analysis documents; modifies the set of documents, starting with the highest-level document affected by the changes, propagating the changes down through the full set of documents; and at each step of the evolutionary process, lets you redesign the system, based on analysis of the existing system.

The process assumes that the maintenance organization can analyze the existing product, characterize the proposed set of modifications, and redesign the current version where necessary for the new capabilities.

**Full-reuse model.** While iterative enhancement starts with evaluating the existing system for redesign and modification, a full-reuse process model starts with the requirements analysis and design of the new system and reuses the appropriate requirements, design, and code from any earlier versions of the old system. It assumes a repository of documents and components defining earlier versions of the current system and similar systems. The full-reuse model starts with the requirements for the new system, reusing as much of the old system as feasible, and builds a new system using documents and components from the old system and from other systems available in your repository; you develop new documents and components where necessary.

Here, reuse is explicit, packaging of existing components is necessary, and analysis is required to select the appropriate components.

Figure 3 demonstrates the flow of various documents into the various document repositories (which are all part of the larger repository) and how those repositories are accessed for documents for the new development. There is an assumption that the items in the repository are classified according to a variety of characteristics, some of which I describe later in the article.

This repository may contain more than just the documents from the earlier system — it may contain documents from earlier versions, documents from other products in the product line, and some
generic reusable forms of documents. An environment that supports the full-reuse model clearly supports the other two models.

**Model differences.** The difference between the last two approaches is more one of perspective than style. The full-reuse model frees you to design the new system’s solution from the set of solutions of similar systems. The iterative-enhancement model takes the last version of the current system and enhances it. Both approaches encourage redesign, but the full-reuse model offers a broader set of items for reuse and can lead to the development of more reusable components for future systems. By contrast, the iterative-enhancement model encourages you to tailor existing systems to get the extensions for the new system.

**Reuse framework**

The existence of multiple maintenance models raises several questions. Which is the most appropriate model for a particular environment? a particular system? a particular set of changes? the task at hand? How do you improve each step in the process model you have chosen? How do you minimize overall cost and maximize overall quality?

To answer these questions, you need a model of the object of reuse, a model of the process that adapts that object to its target application, and a model of the reused object within its target application. Figure 4 shows a simple model for reuse. In this model, an object is any software process or product and a transformation is the set of activities performed when reusing that object.

The model steps are:
- identifying the candidate reusable pieces of the old object,
- understanding them,
- modifying them to your needs, and
- integrating them into the process.

To flesh out the model, you need a framework for categorizing objects, transformations, and their context. The framework should cover various categories. For example, is the object of reuse a process or a product? In each category, there are various classification schemes for the product (such as requirements documents, code module, and test plan) and for the process (such as cost estimation, risk analysis, and design).

**Framework dimensions.** There are a variety of approaches to classifying reusable objects, most notably the faceted scheme offered by Ruben Prieto-Díaz and Peter Freeman.

I offer here a scheme that categorizes three aspects of reuse: the reusable object, the reusable object’s context, and the process of transforming that object.

**I offer here a scheme that categorizes three aspects of reuse: the reusable object, the reusable object’s context, and the process of transforming that object.**
Comparing the models. When applying the reuse framework to maintenance, the set of reuse objects is a set of product documents. You compare the models to see which is appropriate for the current set of changes according to the framework’s three dimensions.

First consider the reuse-object dimension:

The objects of the quick-fix and iterative-enhancement models are the set of documents representing the old system. The object of the full-reuse model is any appropriate document in the repository.

For self-containedness, all the models depend on the unit of change. The quick-fix model depends on how much evolution has taken place, since the system may have lost structure over time as objects were added, modified, and deleted. In iterative enhancement, the evolved system’s structure and understandability should improve with respect to the application and the classes of changes made so far. In the full-reuse model, the evolved system’s structure, understandability, and generality should improve; the degree of improvement will depend on the quality and maturity of the repository.

For reuse-object quality, the quick-fix model offers little knowledge about the old object’s quality. In iterative enhancement, the analysis phase provides a fair assessment of the system’s quality. In full reuse, you have an assessment of the reuse object’s quality across several systems.

Now consider the context dimensions:

For the requirements domain, the quick-fix and iterative-enhancement models assume that you are reusing the same application — in fact, the same project. The full-reuse model allows manageable variation in the application domain, depending on what is available in the repository.

For the solution domain, the quick-fix model assumes the same solution structure exists during maintenance as during development. There is no change in the basic design or structure of the new system. In iterative enhancement, some modification to the solution structure is allowed because redesign is a part of the model. The full-reuse model allows major differences in the solution structure: You can completely redesign the system from a structure based on functional decomposition to one based on object-oriented design, for example.

For the knowledge-transfer mechanism, the quick-fix and iterative-enhancement models work best when the same people are developing and maintaining the system. The full-reuse model can compensate for having a different team, assuming that you have application specialists and a well-documented reuse-object repository.

The quick-fix model’s weaknesses are that the modification is usually a patch that is not well-documented, partly destroying the system structure and hindering future evolution.

Last, consider the transformation dimension:

For the transformation type, the quick-fix model typically uses activities like source-code lookup, reading for understanding, unconstrained modification, and recompilation. Iterative enhancement typically begins with a search through the highest-level (most abstract) document affected by the modification, changing it, and evolving the subsequent documents to be consistent, using several modification mechanisms. The full-reuse model uses a library search and several modification mechanisms; those selected depend on the type of change. In full reuse, modification is done offline.

For activity integration, all activities are performed at the same time in the quick-fix model. Iterative enhancement associates the activities with all the normal development phases. In full reuse, you identify the candidate reusable pieces during project planning and perform the other activities during development.

For transformed quality, the quick-fix model usually works best on small, well-contained modifications because their effects on the system can be understood and verified in context. Iterative enhancement is more appropriate for larger changes where the analysis phase can provide better assessment of the full effects of changes. Full reuse is appropriate for large changes and major redesigns. Here, analysis and performance history of the reuse objects support quality.

Applying the models. Given these differences, you can analyze the maintenance process models and recommend where they might be most applicable.

But first, consider the relationship between the development and maintenance process models: You can consider development to be a subset of maintenance. Maintenance environments differ from development environments in the constraints on the solution, customer demand, timeliness of response, and organization.

Most maintenance organizations are set up for the quick-fix model but not for the iterative-enhancement or full-reuse models, since they are responding to timeliness — a system failure needs to be fixed immediately or a customer demands a modification of the system’s functionality. This is best used when there is little chance the system will be modified again.

Clearly, these are the quick-fix model’s strengths. But its weaknesses are that the modification is usually a patch that is not well-documented, the structure of the system has been partly destroyed, making future evolution of the system difficult and error-ridden, and the model is not compatible with development processes.

The iterative-enhancement model allows redesign that lets the system structure evolve, making future modifications easier. It focuses on making the system as good as possible. It is compatible with development process models. It is a good approach to use when the product will have a long life and evolve over time. In this case, if timeliness is also a constraint, you can use the quick-fix model for patches and the iterative-enhancement model for long-term change, replacing the patches. The drawbacks are that it is a more costly and possibly less timely approach (in the
short run) than the quick-fix model and provides little support for generic components or future, similar systems.

The full-reuse model gives the maintainer the greatest degree of freedom for change, focusing on long-range development for a set of products, which has the side effect of creating reusable components of all kinds for future developments. It is compatible with development process models and, in fact, is the way development models should evolve. It is best used when you have multiproduct environments or generic development where the product line has a long life. Its drawback is that it is more costly in the short run and is not appropriate for small modifications (although you can combine it with other models for such changes).

My assessment of when to apply these models is informal and intuitive, since it is a qualitative analysis. To do a quantitative analysis, you would need quantitative models of the reuse objects, transformations, and context. You would need a measurement framework to characterize (via classification), evaluate, predict, and motivate management and technical decisions. To do this, you would need to apply to the models a mechanism for generating and interpreting quantitative measurement, like the goal/question/metric paradigm.46 (See the box on p. 24 for a description of this paradigm and its application to choosing the appropriate maintenance process model.)

**Reuse enablers**

There are many support mechanisms necessary to achieve maximum reuse that have not been sufficiently emphasized in the literature. In this article, I have presented several: a set of maintenance models, a mechanism for choosing the appropriate models based on the goals and characteristics of the problem at hand, and a measurement and evaluation mechanism. To support these activities, there is a need for an improvement paradigm that helps organizations evaluate, learn, and enhance their software processes and products, a reuse-oriented evolution environment that encourages and supports reuse, and automated support for both the paradigm and environment as well as for measurement and evaluation.

**Improvement paradigm.** The improvement paradigm is a high-level organizational process model in which the organization learns how to improve its products and process. In this model, the organization should learn how to make better decisions on which process model to use for the maintenance of its future products based on past performance. The paradigm has three parts: planning, analysis, and learning and feedback.

In planning, there are three integrated activities that are iteratively applied:

- Characterize the current project environment to provide a quantitative analysis of the environment and a model of the project in the context of that environment. For maintenance, the characterization provides product-dimension data, change and defect data, cost data and customer-context data for earlier versions of the system, information about the classes of candidate components available in the repository for the new system, and any feedback from previous projects with experience with different models for the types of modifications required.
- Set up goals and refine them into quantifiable questions and metrics using the goal/question/metric paradigm to get performance that has improved compared to previous projects. This consists of a top-down analysis of goals that iteratively decomposes high-level goals into detailed subgoals. The iteration terminates with subgoals that you can measure directly.
- Choose and tailor the appropriate construction model for this project and the supporting methods and tools to satisfy the project goals. Understanding the environment quantitatively lets you choose the appropriate process model and fine-tune the methods and tools needed to be most effective. For example, knowing the effect of earlier applications of the maintenance models and methods in creating new projects from old systems lets you choose and fine-tune the appropriate process model and methods that have been most effective in creating new systems of the type required from older versions and component parts in the repository.

In analysis, you evaluate the current practices, determine problems, record the findings, and make recommendations for improvement. You must conduct data analysis during and after the project. The goal/question/metric paradigm lets you trace from goals to metrics and back, which lets you interpret the measurement in context to ensure a focused, simpler analysis. The goal-driven operational measures provide a framework for the kind of analysis you need.

In learning and feedback, you organize and encode the quantitative and qualitative experience gained from the current project into a corporate information base to help improve planning, development, and assessment for future projects. You can feed the results of the analysis and interpretation phase back to the organization to change how it does business based on explicitly determined successes and failures.

In this way, you can learn how to improve quality and productivity and how to improve goal definition and assessment. You can start the next project with the experience gained from this and previous projects. For example, understanding the problems associated with each new version of a system provides insights into the need for redesign and redevelopment.

**Reuse-oriented environment.** Reuse can be more effectively achieved in an environment that supports reuse. (See the article by Ted Biggerstaff and Charles Richter7 for a set of reusability technologies and the article by myself and Dietter Rombach8 for a set of environment
Goal/question/metric paradigm

The goal/question/metric paradigm represents a systematic approach for setting project goals (tailored to the needs of an organization) and defining them in an operational, tractable way. Goals are associated with a set of quantifiable questions and models that specify metrics and data for collection. The tractability of this software-engineering process supports the analysis of the collected data and computed metrics in the appropriate context of the questions, models and goals, feedback (by integrating constructive and analytic activities), and learning (by defining the appropriate synthesis procedure for lower level into higher level pieces of experience).

The goals are defined in terms of purpose (why the project is being analyzed), perspective (the models of interest and the point of view of the analysis), and the environment (the context of the project). When measuring a product or process, you ask questions in three general categories:

- product or process definition,
- definition of the quality perspectives of interest, and
- feedback.

Product definition includes physical attributes of the product, cost, changes and defects, and the context in which the product will be used. Process definition includes a model of the process, an evaluation of conformance to the model, and an assessment of the project-specific documents and experience with the application.

Definition of the quality perspectives of interest includes the quality models used (such as reliability and user friendliness) and the interpretation of the data collected relative to the models.

Feedback involves the return of information for improving the product and process based on the quality perspective of interest.

The following is an informal application of the goal/question/metric paradigm to a particular maintenance problem. The answers to some of the questions are obvious. The answers to others assume a database of experience that management must estimate if it is not available.

Goals. The goal-definition phase has three parts:

- Purpose: Analyze the new product requirements to determine the appropriate evolution model.
- Perspective: Examine the cost of the current enhancement and future evolution of the system from the organization's point of view.
- Environment: Along with the standard environmental factors, like resource and problem factors, you would like to pay special attention to the context dimensions in the reuse framework.

In the requirements domain, you typically use product objects from the same application domain, although you can choose objects from other domains in the repository, if they are generally applicable.

The solution domain defines the process models, methods, and tools used in the development of the old product. If you plan to use the same processes for the evolving product, there is no problem with reuse. If future evolution dictates changes to the solution domain, the full-reuse model lets you make these changes, but at the cost of reusing less of the old product.

For knowledge-transfer mechanism, you must determine what form of documentation is needed to transfer the required application, process, and product knowledge to the maintainers. If the maintenance group is the same as the development group, the major transfer mechanism is the people.

Product definition. With the goal defined, you then define your product. In this example, there are several products: the new product to be built (the new version of the system), the old versions, and any other relevant objects in the repository that may be reused.

For the category of physical attributes, sample questions are:

- How many requirements are there for the new system? What is the mapping of the requirements to system components in the old system? How independent are the components to be modified in the old system? What is the complexity of the old system and its individual components? What candidate objects are available in the repository and what are their object, context, and transformation classifications? How many new requirements, categorized by class (such as size, type, and whether it is a new, modified, or deleted requirement) are there that not in the old system? How many components, categorized by class (such as size and type of change) in the old system must be changed, added, and deleted?

- For the category of changes and defects, sample questions are:
  - How many errors, faults, and failures (categorized by class) are there associated with the requirements and components that need to be changed? What is the profile of past and future changes to the system, categorized by class (such as cost and number of times a component has been and must be changed)?

- For the category of cost, sample questions are:
  - What was the cost of the original system? What was the cost of each prior version? What is the cost of each prior requirement change by class? What is the estimated cost of modifying the old system to meet the new requirements? What is the estimated cost of building a new system, reusing the experience and parts of the old system and the repository?

- For the category of customer context, sample questions are:
  - What are the various customer classes and how are they using the system? What are the estimated future enhancements based on your analysis of customer profiles, past modifications, and the state of technology in the application domain?

Quality perspective of interest. With the product defined, you now define the perspectives for the qualities that you are trying to achieve.

You should make a model of the system's evolution, along with its associated costs. Based on the data from the evolution of this system and other systems, as well as on the characteristics of the set of new requirements, the model should let you estimate the cost and benefits associated with each of the three process models and let you choose the appropriate one. Parameters for the model will include such items as the projected system lifetime, the number of future related systems, and the projected cost of changes for various classes of requirements.

Feedback. With the quality perspectives defined, you can now get the information needed to improve the product or process. The feedback should provide with deeper insights into the model and our environment.

Sample questions include:
- Is the model appropriate? How can the model be improved? How can the classifications be improved?

Other goals. There are many relevant goals. Consider the following examples:

- Evaluate the modification activities in the reuse model to improve them. Examine the cost and correctness of the resulting objects from the customer's point of view.
- Evaluate the components of the existing system to determine whether to reuse them. Examine their independence and functional appropriateness from the viewpoint of reuse in future systems.
- Predict the ability of a set of code components to be integrated into the current system from the developer's point of view.
- Encourage the reuse of a set of repository components built for reuse. Examine the reward structure from the manager's and developer's points of view.
The approach you take to maintenance depends on the nature of the problem and the size and complexity of the modification. Viewing maintenance as a reuse-oriented process in the context of the improvement paradigm gives you a choice of maintenance models and a measurement framework. You can evaluate the strengths and weaknesses of the different maintenance approaches, learn how to refine the various process models, and create an experience base from which to support further management and technical decisions.

If you do not adapt the maintenance approach, you will find it difficult to know which process model to use for a particular project, whether you are evolving the system appropriately, and whether you are maximizing quality and minimizing cost over the system lifetime.

References

Victor R. Basili is a professor at the University of Maryland at College Park's Institute for Advanced Computer Studies and Computer Science Dept. His research interests include measuring and evaluating software development in industrial and government settings. He is a founder and principal of the Software Engineering Laboratory, a joint venture between the National Aeronautics and Space Administration, the University of Maryland, and Computer Science Corp.

Basili received a BS in mathematics from Fordham College, an MS in mathematics from Syracuse University, and a PhD in computer science from the University of Texas at Austin. He is a member of the IEEE Computer Society and is editor-in-chief of IEEE Transactions on Software Engineering.

Address questions about this article to Basili at Computer Science Dept., A.V. Williams Bldg., Rm. 4187, University of Maryland, College Park, MD 20742.