This chapter surveys best practices for software development and establishes a context for the Rational Unified Process.

The Value of Software

Software is the fuel on which modern businesses are run, governments rule, and societies become better connected. Software has helped us create, access, and visualize information in previously inconceivable ways and forms. Globally, the breathtaking pace of progress in software has helped drive the growth of the world’s economy. On a more human scale, software-intensive products have helped cure the sick and have given voice to the speechless, mobility to the impaired, and opportunity to the less able. From all these perspectives, software is an indispensable part of our modern world.¹

The good news for software professionals is that worldwide economies are becoming increasingly dependent on software. The

kinds of software-intensive systems that technology makes possible and society demands are expanding in size, complexity, distribution, and importance.

The bad news is that the expansion of these systems in size, complexity, distribution, and importance pushes the limits of what we in the software industry know how to develop. Trying to advance legacy systems to more modern technology brings its own set of technical and organizational problems. Compounding the problem is that businesses continue to demand increased productivity and improved quality with faster development and deployment. Additionally, the supply of qualified development personnel is not keeping pace with the demand.

The net result is that building and maintaining software is hard and getting harder; building quality software in a repeatable and predictable fashion is harder still.

SYMPTOMS AND ROOT CAUSES OF SOFTWARE DEVELOPMENT PROBLEMS

Different software development projects fail in different ways—and, unfortunately, too many of them fail—but it is possible to identify a number of common symptoms that characterize these kinds of projects:\textsuperscript{2,3}

\begin{itemize}
  \item Inaccurate understanding of end-user needs
  \item Inability to deal with changing requirements
  \item Modules that don’t fit together
  \item Software that’s hard to maintain or extend
  \item Late discovery of serious project flaws
  \item Poor software quality
  \item Unacceptable software performance
\end{itemize}

---


Team members in each other’s way, making it impossible to reconstruct who changed what, when, where, and why

An untrustworthy build-and-release process

Unfortunately, treating these symptoms does not treat the disease. For example, the late discovery of serious project flaws is only a symptom of larger problems, namely, subjective project status assessment and undetected inconsistencies in the project’s requirements, designs, and implementations.

Although different projects fail in different ways, it appears that most of them fail because of a combination of the following root causes:

- Ad hoc requirements management
- Ambiguous and imprecise communication
- Brittle architectures
- Overwhelming complexity
- Undetected inconsistencies in requirements, designs, and implementations
- Insufficient testing
- Subjective project status assessment
- Failure to attack risk
- Uncontrolled change propagation
- Insufficient automation

SOFTWARE BEST PRACTICES

If you treat these root causes, not only will you eliminate the symptoms, but you’ll also be in a much better position to develop and maintain quality software in a repeatable and predictable fashion.

That’s what software best practices are all about: commercially proven approaches to software development that, when used in combination, strike at the root causes of software development problems. They are called “best practices” not so much because

you can precisely quantify their value but rather because they are observed to be commonly used in industry by successful organizations. These best practices are as follows.

1. Develop software iteratively.
2. Manage requirements.
3. Use component-based architectures.
4. Visually model software.
5. Verify software quality.
6. Control changes to software.

DEVELOP SOFTWARE ITERATIVELY

Classic software development processes follow the waterfall life cycle, as illustrated in Figure 1-1. In this approach, development proceeds linearly from requirements analysis through design, code and unit testing, subsystem testing, and system testing.

The fundamental problem of this approach is that it pushes risk forward in time so that it’s costly to undo mistakes from earlier phases. An initial design will likely be flawed with respect to its key requirements, and, furthermore, the late discovery of design de-

FIGURE 1-1  The waterfall life cycle
ffects tends to result in costly overruns or project cancellation. As Tom Gilb aptly said, "If you do not actively attack the risks in your project, they will actively attack you." The waterfall approach tends to mask the real risks to a project until it is too late to do anything meaningful about them.

An alternative to the waterfall approach is the iterative and incremental process, as shown in Figure 1-2.

In this approach, building on the work of Barry Boehm’s spiral model, the identification of risks to a project is forced early in the life cycle, when it’s possible to attack and react to them in a timely and efficient manner.

This approach is one of continuous discovery, invention, and implementation, with each iteration forcing the development team

---


to drive the project’s artifacts to closure in a predictable and repeatable way.

Developing software iteratively offers a number of solutions to the root causes of software development problems.

1. Serious misunderstandings are made evident early in the life cycle, when it’s possible to react to them.
2. This approach enables and encourages user feedback so as to elicit the system’s real requirements.
3. The development team is forced to focus on those issues that are most critical to the project and are shielded from those issues that distract them from the project’s real risks.
4. Continuous, iterative testing enables an objective assessment of the project’s status.
5. Inconsistencies among requirements, designs, and implementations are detected early.
6. The workload of the team, especially the testing team, is spread out more evenly throughout the life cycle.
7. The team can leverage lessons learned and therefore can continuously improve the process.
8. Stakeholders in the project can be given concrete evidence of the project’s status throughout the life cycle.

MANAGE REQUIREMENTS

The challenge of managing the requirements of a software-intensive system is that they are dynamic: you must expect them to change during the life of a software project. Furthermore, identifying a system’s true requirements—those that weigh most heavily on the system’s economic and technical goals—is a continuous process. Except for the most trivial system, it is impossible to completely and exhaustively state a system’s requirements before the start of development. Indeed, the presence of a new or evolving system changes a user’s understanding of the system’s requirements.

A requirement is a condition or capability a system must meet. The active management of requirements encompasses three activities: eliciting, organizing, and documenting the system’s required functionality and constraints; evaluating changes to these require-
ments and assessing their impact; and tracking and documenting trade-offs and decisions.

Managing the requirements of your project offers a number of solutions to the root causes of software development problems.

1. A disciplined approach is built into requirements management.
2. Communications are based on defined requirements.
3. Requirements can be prioritized, filtered, and traced.
4. An objective assessment of functionality and performance is possible.
5. Inconsistencies are more easily detected.
6. With suitable tool support, it is possible to provide a repository for a system’s requirements, attributes, and traces, with automatic links to external documents.

USE COMPONENT-BASED ARCHITECTURES

Visualizing, specifying, constructing, and documenting a software-intensive system demand that the system be viewed from a number of different perspectives. Each of the different stakeholders—end users, analysts, developers, system integrators, testers, technical writers, and project managers—brings a different agenda to a project, and each of them looks at that system in a different way at different times over the project’s life. A system’s architecture is perhaps the most important deliverable that can be used to manage these different viewpoints and thereby controls the iterative and incremental development of a system throughout its life cycle.

A system’s architecture encompasses the set of significant decisions about

- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified by the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems
The architectural style that guides this organization: these elements and their interfaces, their collaborations, and their composition

Software architecture is concerned not only with structure and behavior but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

Building resilient architectures is important because they enable economically significant degrees of reuse, offer a clear division of work among teams of developers, isolate hardware and software dependencies that may be subject to change, and improve maintainability.

Component-based development (CBD) is an important approach to software architecture because it enables the reuse or customization of existing components from thousands of commercially available sources. Microsoft’s component object model (COM), the Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA), and Sun’s Enterprise JavaBeans (EJB) offer pervasive and widely supported platforms on which component-based architectures are made possible. As Figure 1-3 indicates, components make reuse possible on a larger scale, enabling systems to be composed from existing parts, off-the-shelf third-party parts, and a few new parts that address the specific domain and glue the other parts together.

Coupled with the practice of developing software iteratively, using component-based architectures involves the continuous evolution of a system’s architecture. Each iteration produces an executable architecture that can be measured, tested, and evaluated against the system’s requirements. This approach permits the team to continuously attack the most important risks to the project.

Using component-based architectures offers a number of solutions to the root causes of software development problems.

1. Components facilitate resilient architectures.
2. Modularity enables a clear separation of concerns among elements of a system that are subject to change.
3. Reuse is facilitated by leveraging standardized frameworks (such as COM+, CORBA, and EJB) and commercially available components.
4. Components provide a natural basis for configuration management.
5. Visual modeling tools provide automation for component-based development.

VISUALLY MODEL SOFTWARE

A model is a simplification of reality that completely describes a system from a particular perspective, as shown in Figure 1-4. We build models so that we can better understand the system we are modeling; we build models of complex systems because we cannot comprehend such systems in their entirety.

Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of a system’s architecture. Using a standard modeling language
such as the UML (Unified Modeling Language), different members of the development team can unambiguously communicate their decisions to one another.

Visual modeling tools facilitate the management of these models, letting you hide or expose details as necessary. Visual modeling also helps to maintain consistency between a system’s artifacts: its requirements, designs, and implementations. In short, visual modeling helps improve a team’s ability to manage software complexity.

When coupled with the practice of developing software iteratively, visual modeling helps you expose and assess architectural changes and communicate those changes to the entire development team. With the right kind of tools, you can then synchronize your models and source code during each iteration.

Modeling your software visually offers a number of solutions to the root causes of software development problems.

1. Use cases and scenarios unambiguously specify behavior.
2. Models unambiguously capture software design.
3. Nonmodular and inflexible architectures are exposed.
4. Detail can be hidden when necessary.
5. Unambiguous designs reveal their inconsistencies more readily.
6. Application quality starts with good design.
7. Visual modeling tools provide support for UML modeling.

VERIFY SOFTWARE QUALITY

As Figure 1-5 illustrates, software problems are 100 to 1,000 times more expensive to find and repair after deployment. For this reason, it’s important to continuously assess the quality of a system with respect to its functionality, reliability, application performance, and system performance.

Verifying a system’s functionality—the bulk of the testing activity—involves creating tests for each key scenario, each of which represents some aspect of the system’s desired behavior. You can assess a system’s functionality by asking which scenarios failed and where, as well as which scenarios and corresponding code have not yet been exercised. As you are developing your software iteratively, you test at every iteration, a process of continuous, quantitative assessment.

Verifying software quality offers a number of solutions (see next page) to the root causes of software development problems.

FIGURE 1-5 The cost of fixing problems
1. Project status assessment is made objective, and not subjective, because test results, and not paper documents, are evaluated.
2. This objective assessment exposes inconsistencies in requirements, designs, and implementations.
3. Testing and verification are focused on areas of highest risk, thereby increasing their quality and effectiveness.
4. Defects are identified earlier, radically reducing the cost of fixing them.
5. Automated testing tools provide testing for functionality, reliability, and performance.

**CONTROL CHANGES TO SOFTWARE**

A key challenge when you’re developing software-intensive systems is that you must cope with multiple developers organized into different teams, possibly at different sites, working together on multiple iterations, releases, products, and platforms. In the absence of disciplined control, the development process rapidly degenerates into chaos.

Coordinating the activities and the artifacts of developers and teams involves establishing repeatable workflows for managing changes to software and other development artifacts. This coordination allows a better allocation of resources based on the project’s priorities and risks, and it actively manages the work on those changes across iterations. Coupled with developing your software iteratively, this practice lets you continuously monitor changes so that you can actively discover and then react to problems.

Coordinating iterations and releases involves establishing and releasing a tested baseline at the completion of each iteration. Maintaining traceability among the elements of each release and among elements across multiple, parallel releases is essential for assessing and actively managing the impact of change.

Controlling changes to software offers a number of solutions to the root causes of software development problems.

1. The workflow of requirements change is defined and repeatable.
2. Change requests facilitate clear communications.
3. Isolated workspaces reduce interference among team members working in parallel.
4. Change rate statistics provide good metrics for objectively assessing project status.
5. Workspaces contain all artifacts, facilitating consistency.
6. Change propagation is assessable and controlled.
7. Changes can be maintained in a robust, customizable system.

THE RATIONAL UNIFIED PROCESS

A software development process has four roles.\(^7\)

1. Provide guidance as to the order of a team’s activities.
2. Specify which artifacts should be developed and when they should be developed.
3. Direct the tasks of individual developers and the team as a whole.
4. Offer criteria for monitoring and measuring the project’s products and activities.

Without a well-defined process, your development team will develop in an ad hoc manner, with success relying on the heroic efforts of a few dedicated individual contributors. This is not a sustainable condition.

By contrast, mature organizations employing a well-defined process can develop complex systems in a repeatable and predictable way. Not only is that a sustainable business, but also it’s one that can improve with each new project, thereby increasing the efficiency and productivity of the organization as a whole.

Such a well-defined process enables and encourages all of the best practices described earlier. When you codify these practices into a process, your development team can build on the collective experience of thousands of successful projects.

The Rational Unified Process, as described in the following chapters, builds on these six commercial best practices to deliver a well-defined process. This is the context for the Rational Unified Process, a software development process focused on ensuring the production of quality systems in a repeatable and predictable fashion.

SUMMARY

- Building quality software in a repeatable and predictable fashion is hard.
- There are a number of symptoms of common software development problems, and these symptoms are the observable results of deeper root causes.
- Six commercial best practices strike at the root causes of these software development problems.
  - Develop software iteratively.
  - Manage requirements.
  - Use component-based architectures.
  - Visually model software.
  - Verify software quality.
  - Control changes to software.
- The Rational Unified Process brings these best practices together in a form that is suitable for a wide range of projects and organizations.