from The Rational Edge: As a major update of IBM Rational's six best practices for software development, this paper articulates a new set of principles that characterize a mature approach to the creation, deployment, and evolution of software-intensive systems.

Since 1983, when the organization formerly known as Rational Software, now IBM Rational, was founded, our knowledge of the various processes and techniques for software development has increased through collaboration with a broad community of customers and partners. Over the years, many of them have joined our organization, and have continued to shape our understanding of software development practices as we learn what works, what doesn't, and why software development remains such a challenging endeavor.

As our customers, partners, and employees have heard us say many times, software development is a team sport. Ideally, the activity involves well-coordinated teams working within a variety of disciplines that span the software lifecycle. But it's not science, and it isn't exactly engineering, either -- at least not from the standpoint of quantifiable principles based on hard facts. Software development efforts that assume you can plan and create separate pieces and later assemble them, like you can in building bridges or spacecraft, constantly fail against deadlines, budgets, and user satisfaction.

So, in the absence of hard facts, the Rational organization has relied on software development techniques we call best practices, the value of which we have demonstrated repeatedly in our customer engagements. Rather than prescribing a plan-build-assemble sequence of activities for a software project, they describe an iterative, incremental process that steers development teams toward results.

Our six tried-and-true best practices have been the basis for the evolution of our tools and process offerings for more than a decade. Today, as we witness more companies pursuing software development as an essential business capability, we see these best practices maturing within the larger context of business-driven development. We think it's time to re-articulate our best practices for the broader lifecycle of continuously evolving systems, in which the primary evolving element is software.

This paper articulates a set of principles that we believe characterize the industry's best practices in the creation, deployment, and evolution of software-intensive systems:
• Adapt the process.
• Balance competing stakeholder priorities.
• Collaborate across teams.
• Demonstrate value iteratively.
• Elevate the level of abstraction.
• Focus continuously on quality.

We will explain each of these in order, describing the patterns of behavior that best embody each principle, as well as the most recognizable "anti-patterns" that can harm software development projects.

Adapt the process

**Benefits**: Lifecycle efficiency, open/honest communication of risks

**Pattern**: Precision and formality evolve from light to heavy over the project lifecycle as uncertainties are resolved. Adapt the process to the size and distribution of the project team, to the complexity of the application, and to the need for compliance.

Continuously improve your process.

**Anti-patterns**: Precise plans/estimates, track against static plan management style.

More process is better. Always use the same degree of process throughout the lifecycle.

More process, such as usage of more artifacts, production of more detailed documentation, development and maintenance of more models that need to be synchronized, and more formal reviews, is not necessarily better. Rather, you need to right-size the process to project needs. As a project grows in size, becomes more distributed, uses more complex technology, has a larger number of stakeholders, and needs to adhere to more stringent compliance standards, the process needs to become more disciplined. But, for smaller projects with co-located teams and known technology, the process should be more lightweight. These dependencies are illustrated in Figure 1.

Second, a project should adapt process ceremony to lifecycle phase. In the beginning of the project, you typically have a lot of uncertainty, and you want to encourage a lot of creativity to develop an application that addresses the business needs. More process typically leads to less creativity, not more, so you should use less process in the beginning of a project where uncertainty is an everyday factor. On the other side, late in the project, you often want to introduce more control, such as change control boards, to remove undesired creativity and associated risk for late introduction of defects. This translates to more process late in the project.

Third, an organization should strive to continuously improve the process. Do an assessment at the end of each iteration and each project to capture lessons learned, and leverage that knowledge to improve the process. Encourage all team members to continuously look for opportunities to improve.

Fourth, balance project plans and associated estimates with the uncertainty of a project. This means that early in projects when uncertainty typically is large, plans and associated estimates need to focus on big-picture planning and estimates, rather than aiming at providing 5-digit
levels of precision when there are none. Early development activities should aim at driving out uncertainty to gradually enable increased precision in planning.

**Figure 1: Factors driving the amount of process discipline**

**How Much Process is Necessary?**

<table>
<thead>
<tr>
<th>Simple upgrades</th>
<th>Dynamic web apps</th>
<th>Legacy upgrades</th>
</tr>
</thead>
<tbody>
<tr>
<td>R&amp;D Processes</td>
<td>Package applications</td>
<td>Systems of systems</td>
</tr>
<tr>
<td>Static web apps</td>
<td>Component based (C2, .Net)</td>
<td>Real-time, embedded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Certifiable quality</td>
</tr>
</tbody>
</table>

Many factors, including project size, team distributions, complexity of technology, number of stakeholders, compliance requirements, and where in the project lifecycle you are, steer how disciplined a process you need.

The anti-pattern to following this principle would be to always see more process and more detailed upfront planning as better. Force early estimates, and stick to those estimates.

**Balance competing stakeholder priorities**

**Benefit:** Align applications with business and user needs, reduce custom development, and optimize business value

**Pattern:** Define and understand business processes and user needs; prioritize projects and requirements and couple needs with software capabilities; understand what assets you can leverage; and balance user needs and reuse of assets.

**Anti-pattern:** Achieve precise and thorough requirements before any project work begins. Requirements focus the drive toward a custom solution.

This principle articulates the importance of balancing often conflicting business and stakeholder needs. As an example, most stakeholders would like to have an application that does exactly what they want it to do, while minimizing the application's development cost and schedule time. Yet these priorities are often in conflict. If you leverage a packaged application, for example, you may be able to deliver a solution faster and at a lower price, but you may have to trade off many of your requirements. On the other hand, if a business elects to build an application from scratch, it may be able to address every requirement on its wish list, but the budget and project completion date can both be pushed beyond their feasible limits.

Rather than sending our programming teams out to attack each element in a requirements list, the first thing we need to do is to understand and prioritize business and stakeholder needs. This means capturing business processes and linking them to projects and software capabilities, which
enables us to effectively prioritize the right projects and the right requirements, and to modify our prioritization as our understanding of the application and stakeholder needs evolve. It also means we need to involve the customer or customer representative in the project to ensure we understand what their needs are.

Second, we need to center development activities around stakeholder needs. For example, by leveraging use-case driven development and user-centered design, we can accept the fact that the stakeholder needs will evolve over the duration of the project, as the business is changing, and as we better understand which capabilities are the ones truly important to the business and end users. Our development process needs to accommodate these changes.

Third, we need to understand what assets are available, then balance asset reuse with stakeholder needs. Examples of assets include legacy applications, services, reusable components, and patterns. Reuse of assets can in many cases lead to reduced project cost; and for proven assets, their reuse often means higher quality in new applications. The drawback is that, in many cases, you need to trade off reuse of assets and perfectly addressing end-user needs. Reusing a component may lower development costs for a specific capability by 80 percent, but this may only address 75 percent of the requirements. So, effective reuse requires you to constantly balance the reuse of assets with evolving stakeholder needs.

**Figure 2: Balance the use of components with addressing requirements**

Using a component can radically reduce the cost and time to deliver a certain set of functionality. It may in many cases also require you to compromise on some functional or technical requirements, such as platform support, performance, or footprint (size of the application).

The anti-pattern to following this principle would be to thoroughly document precise requirements at the outset of the project, force stakeholder acceptance of requirements, and then negotiate any changes to the requirements, where each change may increase the cost or duration of the project. Since you lock down requirements up-front, you reduce the ability to leverage existing assets,
which in turn drives you toward primarily doing custom development. Another anti-pattern would be to architect a system only to meet the needs of the most vocal stakeholders.

**Collaborate across teams**

**Benefits:** Team productivity, better coupling between business needs, and the development and operations of software systems.

**Pattern:** Motivate people to perform at their best. Collaborate cross-functionally across analysts, developers, and testers. Manage evolving artifacts and tasks to enhance collaboration and progress/quality insight with integrated environments. Ensure that business, development, and operations teams work effectively as an integrated whole.

**Anti-pattern:** Nurture heroic individuals and arm them with power tools.

Software is produced by talented and motivated people collaborating closely. Many complex systems require the activities of a number of stakeholders with varying skills, and the largest projects often span geographical and temporal boundaries, further adding complexity to the development process. This is why people issues and collaboration -- what some have referred to as the "soft" element of software development -- have been a primary focus in the agile development community. This principle addresses many questions, including: How do you motivate people to perform at their best? And how do you collaborate within a co-located software team, within a distributed team, and across teams responsible for the business, software development, and IT operations?

The first step in effective collaboration is to motivate individuals on the team to perform at their best. The notion of self-managed teams has gained popularity in the agile community; it is based on making a team commit to what they should deliver and then providing them with the authority to decide on all the issues directly influencing the result. When people feel that they are truly responsible for the end result, they are much more motivated to do a good job. As the agile manifesto states: "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

The second step is to encourage cross-functional collaboration. As we’ve mentioned for years, "software development is a team sport." An iterative approach increases the need for working closely as a team. We need to break down the walls that are often built up between analysts, developers, and testers, and broaden the responsibilities of these roles to ensure effective collaboration in an environment with fast churn. Each member needs to understand the mission and vision of the project.

As our teams grow, we need to provide effective collaborative environments. These environments facilitate and automate metrics collection and status reporting and automate build management and bookkeeping around configuration management. This efficiency allows fewer meetings, which frees team members to spend more time on more productive and creative activities. These environments should also enable more effective collaboration by simplifying communication, and bridging gaps in place and time between various team members. Examples of such an environment range from shared project rooms to networked or Web-based solutions, such as
Wikis or integrated development environments and configuration and change management environments.

Our ultimate goal under this principle is *integrated collaboration across business, software, and operation teams*. As software becomes increasingly critical to how we run our business, we need close collaboration between 1) the teams deciding how to run our current and future business, 2) the teams developing the supporting software systems, and 3) the teams running our IT operations.

**Figure 3: Collaborate across business, development, and operations teams**

As software becomes more critical to how we run our business, we need to collaborate more closely around teams responsible for how to run the business, how to develop applications, and how to run the applications. In most companies, these three groups have poor communication.

The anti-pattern to following this principle would be to nurture heroic developers willing to work extremely long hours, including weekends. A related anti-pattern is to have highly specialized people equipped with powerful tools for doing their jobs, with limited collaboration between different team members, and limited integration between different tools. The assumption is that if just everybody does his or her job, the end result will be good.

**Demonstrate value iteratively**

**Benefits**: Early risk reduction, higher predictability, trust among stakeholders

**Pattern**: Adaptive management using an iterative process. Attack major technical, business, and programmatic risks first. Enable feedback by delivering incremental user value in each iteration.

**Anti-pattern**: Plan the whole lifecycle in detail, track variances against plan. Detailed plans are better plans. Assess status by reviewing specifications.

There are several imperatives underlying this principle. The first one is that you want to deliver *incremental value to enable early and continuous feedback*. This is done by dividing our project...
into a set of iterations. In each iteration, you do some requirements, design, implementation, and testing of your application, thus producing a deliverable that is one step closer to the final solution. This allows you to demonstrate the application to end users and other stakeholders, or have them use the application directly, enabling them to provide rapid feedback on how you are doing. Are you moving in the right direction? Are stakeholders satisfied with what you have done so far? Do you need to change the features implemented so far, and what additional features need to be implemented to add business value? By being able to satisfactorily answer these questions, you are more likely to build trust among stakeholders by delivering a system that will address their needs. You are also less likely to over-engineer your approach, adding capabilities not useful to the end user.3

The second imperative is to leverage demonstrations and feedback to adapt your plans. Rather than relying on assessing specifications, such as requirements specifications, design models, or plans, you instead need to assess how well the code that's been developed thus far actually works. This means you focus on test results and demonstrate working code to various stakeholders to assess how well you are doing. This provides you with a good understanding of where you are, how fast the team can make progress within your development environment, and whether you need to make course corrections to successfully complete the project. You use this information to update the overall plan for the project and to develop detailed plans for the next iteration.

The third underlying imperative is to embrace and manage change. Today's applications are too complex to allow you to perfectly align the requirements, design, implementation, and test the first time through. Instead, the most effective application development methods embrace the inevitability of change. Through early and continuous feedback, we learn how to improve the application, and the iterative approach provides us with the opportunity to implement those changes incrementally. All this change needs to be managed by having the processes and tools in place so we can effectively manage change without hindering creativity.

The fourth imperative underlying this principle is the need to drive out key risks early in the lifecycle, as illustrated in Figure 1. You must address the major technical, business, and programmatic risks as early as possible, rather than postponing risk resolution toward the end of the project. This is done by continuously assessing what risks you are facing, and addressing the top remaining risks in the next iteration. In successful projects, early iterations involve stakeholder buy-in on a vision and high-level requirements, including architectural design, implementation, and testing to mitigate technical risks. It is also important to retain information required to force decisions around what major reusable assets or commercial-off-the-shelf (COTS) software to use.
A major goal with iterative development is to reduce risk early on. This is done by analyzing, prioritizing, and attacking top risks in each iteration.

A typical anti-pattern (i.e., former software development practices that actually contribute to project failure) is to plan the whole lifecycle in detail upfront, and then track variances against plan. Another anti-pattern is to assess status in the first two thirds of the project by relying on reviews of specifications, rather than assessing status of test results and demonstrations of working software.

Elevate the level of abstraction

**Benefits**: Productivity, reduced complexity  
**Pattern**: Reuse existing assets, reduce the amount of human-generated stuff through higher-level tools and languages, and architect for resilience, quality, understandability, and complexity control.  
**Anti-pattern**: Go directly from vague high-level requirements to custom-crafted code.

One of the main problems we face in software development is complexity. We know that reducing complexity has a major impact on productivity. Working at a higher level of abstraction reduces complexity and facilitates communication.

One effective approach to reducing complexity is **reusing existing assets**, such as reusable components, legacy systems, existing business processes, patterns, or open source software. Two great examples of reuse that have had a major impact on the software industry over the last decade is the reuse of middleware, such as databases, Web servers and portals, and, more recently, open source software that provides many smaller and larger components that can be leveraged. Moving forward, Web services will likely have a major impact on reuse, since they provide simple ways of reusing major chunks of functionality across disparate platforms and with loose coupling between the consumer and provider of a service, as illustrated in Figure 6. This means that you can more easily leverage different combinations of services to address business needs. Reuse is also facilitated by open standards, such as RAS, UDDI, SOAP, WSDL, XML, and UML.
Figure 5: Reuse existing assets through service-oriented architectures

One of the problems with reuse is that two components need to know about each other's existence at development time. Service-oriented architectures alleviate that problem by providing what is called "loose coupling," shown here via the double-headed black arrows. A consumer of a service can dynamically find a provider of a service. You can hence wrap existing components or legacy systems, allowing other components or applications to dynamically access their capabilities through a standards-based interface, independent of platform and implementation technology.

Another approach to reducing complexity and improving communication is to leverage higher-level tools, frameworks, and languages. Standard languages, such as Unified Modeling Language (UML), and rapid application languages such as EGL\(^6\) provide the ability to express high-level constructs, such as business processes and service components, to facilitate collaboration around high-level constructs while hiding unnecessary details. Design and construction tools can automate moving from high-level constructs to working code by providing wizards to automate design, construction, and test tasks by generating code and enabling usage of code snippets, and by converting integration and testing into seamless development tasks through integrated development, build, and test environments. Another example is project and portfolio management tools, which allow you to manage financial and other aspects of multiple projects as one entity versus as a set of separate entities.

A third approach to managing complexity is to focus on architecture, no matter whether you are trying to define a business or develop a system or an application. In software development, we aim at getting the architecture designed, implemented, and tested early in the project. This means that early in the project we define the high-level building blocks and the most important components, their responsibilities, and their interfaces. We define and implement the architectural mechanisms, that is, ready-made solutions to common problems, such as how to deal with persistency or...
garbage collection. By getting the architecture right early on, we define a skeleton structure for our system, making it easier to manage complexity as we add more people, components, capabilities, and code to the project. We also understand what reusable assets we can leverage, and what aspects of the system need to be custom built.

The anti-pattern to following this principle would be to go directly from vague, high-level requirements to custom-crafted code. Since few abstractions are used, a lot of the discussions are made at the code level versus a more conceptual level, which misses many opportunities for reuse, among other things. Informally captured requirements and other information require many decisions and specifications to be revisited over and over, and limited emphasis on architecture causes major rework late in the project.

**Focus continuously on quality**

**Benefits**: Higher quality and earlier progress/quality insight  
**Pattern**: Team responsibility for end product. Testing becomes a high priority given continuous integration of demonstrable capabilities. Incrementally build test automation.  
**Anti-pattern**: Postpone integration testing until all code has been completed and unit-tested. Peer-review all artifacts, rather than also driving partial implementation and testing, to discover issues.

Improving quality is not simply "meeting requirements," or producing a product that meets user needs and expectations. Rather, quality also includes identifying the measures and criteria to demonstrate its achievement, as well as the implementation of a process to ensure that the product created by the team has achieved the desired degree of quality, which can be repeated and managed.

Ensuring high quality requires more than the participation of the testing team; it requires that the entire team owns quality. It involves all team members and all parts of the lifecycle. Analysts are responsible for making sure that requirements are testable, and that we specify clear requirements for the tests to be performed. Developers need to design applications with testing in mind, and must be responsible for testing their code. Management needs to ensure that the right test plans are in place, and that the right resources are in place for building the testware and performing required tests. Testers are the quality experts. They guide the rest of the team in understanding software quality issues, and they are responsible for functional-, system-, and performance-level testing, among other things. When we experience a quality issue, every team member should be willing to chip in to address the issue.

One of the major benefits of iterative development is that it enables a test early and continuously approach, as illustrated in Figure 2. Since the most important capabilities are implemented early in the project, by the time you get to the end, the most essential software may have been up and running for months, and it is likely to have been tested for months. It is not a surprise that most projects adopting iterative development claim that an increase in quality is a primary tangible result of the improved process.
As we incrementally build our application, we should also incrementally build test automation to detect defects early, while minimizing up-front investments. As you design your system, consider how it should be tested. Making the right design decisions can greatly improve your ability to automate testing. You may also be able to generate test code directly from the design models. This saves time, provides incentives for early testing, and increases the quality of testing by minimizing the number of bugs in the test software. Automated testing has been a key area of focus for, among others, the agile community, where the aim is to automate testing of all code, and where tests are written before the code is written (so called test-first design.)

**Figure 6: Testing Is initiated early and expanded upon in each iteration**

Iterative development enables early testing. Software is built in every iteration and tested as it is built. Regression testing ensures that new defects are not introduced as new iterations add functionality.

The anti-pattern for this principle involves conducting in-depth peer-review of all intermediate artifacts, which is counter productive since it delays application testing, and hence, identification of major issues. Another anti-pattern is to complete all unit testing before doing integration testing, again delaying identification of major issues.

**Summary**

As we noted at the beginning, these principles have evolved from their origins over a decade ago, when software developers generally worked in a somewhat more limited context. Our knowledge about software development, along with the overall conditions for project success, has matured and expanded. Today's business-driven software development organizations need guideposts that map a broader landscape, which includes geographically distributed development, IT governance and regulatory compliance needs, service oriented-architecture, and more.

Just as software development is not hard science, we believe that these principles should not be taken as statements of absolute, everlasting truth, but rather as guides to improved software
project results. These principles will continue to evolve -- not at a breathtaking pace, but gradually, as we acquire more experience about what works, what doesn't, etc. The one thing that won't change is the long-standing Rational commitment to helping the success of customers whose businesses depend on developing and deploying software.

Notes

1 See http://www.agilemanifesto.org/principles.html


3 According to the Standish Group's Chaos Report, 2002, 45 percent of features implemented on the average project are never used.


© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)