The Spirit of the RUP

by **Per Kroll**
Director, Rational Unified Process Development
and Product Management Teams

At the core of the Rational Unified Process®, or RUP®, lie eight fundamental principles that represent the essential "Spirit of the RUP." These are the experiences gleaned from a huge number of successful projects distilled into a few simple guidelines. Although these principles represent neither a complete view of the RUP nor the full complexity of those many projects, understanding these principles will guide you in better applying the RUP to your own projects. These principles are:

1. *Attack major risks early and continuously... or they will attack you.*
2. *Ensure that you deliver value to your customer.*
3. *Stay focused on executable software.*
4. *Accommodate change early in the project.*
5. *Baseline an executable architecture early on.*
6. *Build your system with components.*
7. *Work together as one team.*
8. *Make quality a way of life, not an afterthought.*

You may find that one or several of these principles are incompatible with what you would like to apply to your project, and that's fine. Even at its essence, the RUP should be considered a smorgasbord from which you choose the dishes that fit your needs.

Introducing Iterative Development

The RUP proposes an iterative approach to software development, which means that a project is divided into several small projects that run sequentially, one directly after another. Each iteration has a well-defined set of objectives, and concludes by delivering an executable that is a step closer to the final product than the last one. Each iteration contains elements of requirements management; analysis and design; implementation; integration; and testing.

The RUP[1] provides a structured approach to iterative development that divides a *project* into four *phases*: Inception, Elaboration, Construction, and Transition. The objectives of the phases are:

- Inception: Understand what to build.
- Elaboration: Understand how to build it.
- Construction: Build a beta version of the product.
- Transition: Build the final version of the product.

Each *phase* contains one or more *iterations*, which focus on producing the technical deliverables necessary to achieve the business objectives of that phase. Each phase includes as many iterations as it takes to address the objectives of that phase, but no more.

This evolutionary approach to software development can be seen as an umbrella for all the principles of software development described in this article. Some of the principles can be applied outside the context of iterative development, and iterative development can be done without applying all of the principles. There is, however, a strong correlation between successful iterative development and the principles described in this article, and to optimize your implementation of an iterative approach, you should attempt to apply as many of the principles as is feasible for your project.

1. Attack Major Risks Early and Continuously...Or They Will Attack You

As Tom Gilb said, "If you don't actively attack the risks, they will actively attack you."[2] As Figure 1 shows, one of the prime benefits of the iterative approach is that it allows you to identify and address major risks *early* in the project.

Why address top risks early on? Unaddressed risks mean that you are potentially investing in a faulty architecture and/or a nonoptimal set of requirements. This is bad software economics. In addition, the amount of risk is directly correlated to the difference between the upper and lower estimates of how long it will take to complete a project. To come up with accurate estimations, you need to identify and address risks up front.

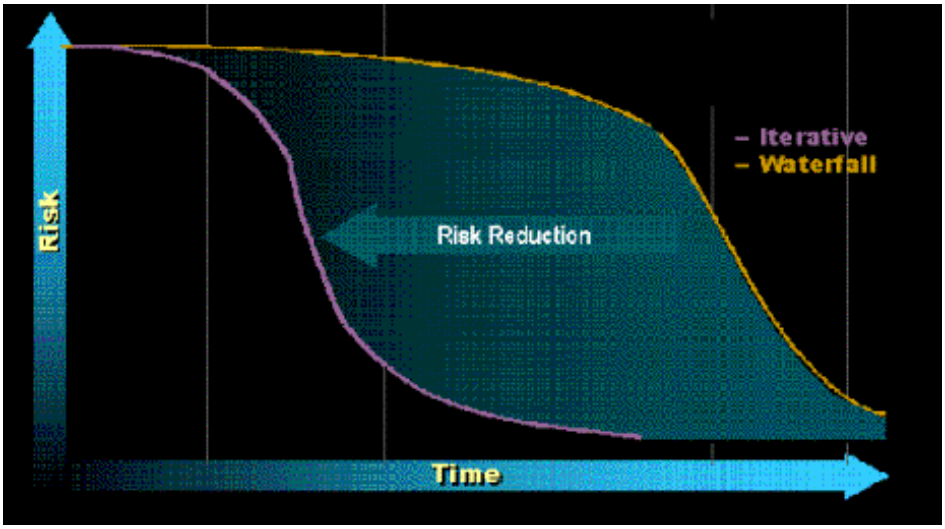

Figure 1: Risk Reduction Profiles for Waterfall and Iterative Developments. *A major goal with iterative development is to reduce risk early on. This is done by analyzing, prioritizing, and attacking top risks in each iteration.*

How do you deal with risks early on? At the beginning of each iteration, the RUP advises you to make, or revise, a list of top risks. Prioritize the risk list, then decide what you need to do to address, typically, the top three to five risks. For example, the risks may look as follows:

- **Risk 1:** We are concerned, based on past experience, that Department X will not understand what requirements we plan to provide, and as a result will request changes *after* beta software is delivered.
- **Risk 2:** We do not understand how we can integrate with legacy system Y.
- **Risk 3:** We have no experience in developing on the Microsoft .NET platform or in using Rational Rose.
- **Risk 4:** ...etc.

Now, how will this risk list be used? Addressing risks should be a priority for everyone throughout the project. Look at what you would "normally" do for the iteration at hand, then slightly modify the plan to make sure that you deal with your risks. Typically, risks need to be addressed from multiple perspectives, such as requirements, design, and test. For each of these perspectives, start with a coarse solution, and successively detail it to diminish the risk. You may, for example, add the following actions to address the above-identified risks:

- **Risk 1:** As the use cases related to Department X are developed, complement them with a UI prototype. Set up a meeting with Department X, and walk them through each use case, using the UI prototype as a storyboard. Get a formal sign-off on the requirements. Throughout the project, keep Department X in the loop on progress, and provide them with early prototypes and/or alpha releases.
- **Risk 2:** Have a "tiger team" with one or two skilled developers build

an actual prototype that shows how to integrate with legacy system Y. The integration may be a throwaway, but the prototype should prove that you actually can integrate with the legacy system. Throughout the project, ensure appropriate testing of the integration with legacy system Y.

- An alternative would be to cut the scope of the project, so you do not have to integrate with legacy system Y. This strategy is called "risk avoidance" and is typically highly effective. Note that all other examples in this list are of the strategy type "risk mitigation."
- **Risk 3:** Send a couple of people for training on Microsoft .NET and Rational Rose, respectively, and find the budget to bring in a Rational Rose mentor two days per week for the first three weeks of the Elaboration phase. Recruit a team member with an understanding of the .NET platform.
- **Risk 4:**...etc.

Many project risks are associated with the architecture. This is why the RUP's primary objective in the Elaboration phase is getting the architecture right. To do this you not only design the architecture, but you also implement and test it. (Find out more about this in "Baseline an Executable Architecture Early On" below.)

One thing to keep in mind is that the risk list continuously changes. Attacking risk is a constant battle -- in each iteration you will be able to reduce or remove some risks, while others grow and new ones appear.

Summary

The RUP provides a structured approach to addressing top risks early on, which decreases overall costs, and allows you to earlier make realistic and accurate estimations of how much time it will take to complete the project. Remember that risk management is a dynamic and ongoing process.

2. Ensure that You Deliver Value to Your Customer

Delivering value to your customer is a pretty obvious goal, but how is it done? Our recommendation is closely related to iterative development and the "Use-Case-Driven Approach."[3] So, what are use cases? Use cases are a way of capturing functional requirements. Since they describe how a user will interact with the system, they are easy for a user to relate to. And since they describe the interaction in a time-sequential order, it is easy for both users and analysts to identify any holes in the use case. A use case can almost be considered a section in the future user manual for the system under development, but written with no knowledge about the specific user interface. You do not want to document the user interface in the use case; instead, you complement the use-case descriptions with a UI prototype, for example, in the form of screen shots.

Many people have learned to love use cases for their simplicity and their

ability to facilitate rapid agreement with stakeholders on what the system should do. But after ten years of use-case consulting, I feel that this is not their primary benefit. Rather, I think their major advantage is that they allow each team member to work very closely to the requirements when designing, implementing, testing, and finally writing user manuals. Use cases force you to stay externally focused on the user's perspective, and they allow you to validate the design and implementation with respect to the user requirements. They even allow you to carefully consider user needs when planning the project and managing scope (see Figure 2).

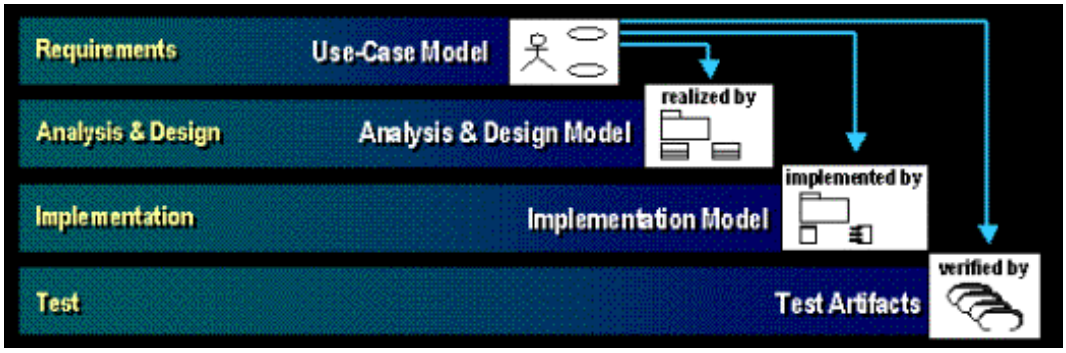

Figure 2: How Use Cases Relate to Other Software Engineering Models. *Use cases are a highly effective vehicle for capturing requirements. They also allow you to work closely to the requirements when doing design, implementation, and testing, ensuring that you deliver the requirements.*

Since a use case describes how a user will interact with the system, you can use UML[4] diagrams such as Sequence Diagrams or Collaboration Diagrams to show how this interaction will be implemented by your design elements. You can also identify test cases from a use case. This ensures that the services the users expect from the system really are provided. Since the use cases are "*user manuals without knowledge of the UI*," they make a solid starting point for writing user manuals. [**Editor's note**: for more on the relationship between use cases and the writing of user manuals, see Robert Pierce's article, "Keep Documentation Up to Date Throughout Development with Rational Rose" in this issue of *The Rational Edge*.] And when prioritizing which capabilities to deliver when managing scope, you choose which use cases to implement. And as we noted above, use cases allow you to work closely to the requirements throughout the development lifecycle. An expression that captures the essence of use cases is "*Documenting customer needs is great; implementing them is better*."

Summary

Use cases make it easy to document user requirements and to help all stakeholders understand the capabilities that are to be delivered. More essentially, use cases allow you to work closely to the requirements when doing design, implementation, and testing, thereby ensuring that you not only document, but also deliver, the user requirements.

3. Stay Focused on Executable Software

This third essential RUP principle has several facets. First, you should, to

the extent possible, measure progress by measuring your *executable* software. It is great to know that ten out of twenty use cases have been described, but that does not necessarily mean that 50 percent of the requirements are completed. What if you later find that ten of those use cases require major rewrites because you did not properly understand the user requirements? That could mean that you were only 25 percent complete, right? So what you can really say when you have completed half of the use cases is that you are probably not more than 50 percent done with the requirements.

The best way of measuring progress is by measuring what software is up and running. This allows you to do testing and, based on testing and defect rates, assess the true progress that has been made. When the typical developer states, "I am 90 percent done," you can ask, "Great, but can you please demo what is up and running?" and then get a solid idea of what has actually been accomplished. As an architect/team leader/manager, you should always strive to have working software demonstrated and to look at test coverage and test results, rather than be fooled by the often-false reality of completed documents. This does not mean that you should disregard the information in completed documents, but when considered in isolation they provide a poor measure of true progress.

Second, a clear focus on executable software also promotes right thinking among your team; you run less risk of overanalyzing and theorizing, and instead get down to work to prove whether solution A or B is the better. Forcing closure by producing executable software is often the fastest way of mitigating risk.

A third attribute of this focus on executable software is that *artifacts other than the actual software are viewed as supporting artifacts*. They are there to allow you to produce better software. By staying focused on executable software, you are better prepared to assess whether producing other artifacts -- such as requirement management plans, configuration management plans, use cases, test plans, and so on -- will really lead to software that works better and/or is easier to maintain. In many cases the answer is yes, but not always. You need to weigh the cost of producing and maintaining an artifact against the benefit of producing it. The benefit of producing many artifacts typically increases as your project grows larger, as you have more complicated stakeholder relations, as your team becomes distributed, as the cost of quality issues increases, and as the software is more critical to the business. All these factors drive toward producing more artifacts, and treating them more formally. But for every project, you should strive to minimize the number of artifacts produced, to reduce overhead.

A good guideline is that if you are in doubt as to whether or not to produce an artifact, don't produce it. But do not use this guideline as an excuse to skip essential activities such as setting a vision, documenting requirements, having a design, and planning the test effort. Each of these activities produces artifacts of obvious value. If the cost of producing an artifact is going to be higher than the return on investment, however, then you should skip it.

One of the most common mistakes RUP users make is to produce artifacts just because the RUP describes how to produce them. Remember, the RUP is a smorgasbord, and it is typically unwise to eat every dish at a table like the one in Figure 3.

Figure 3: Consider the RUP as a Smorgasbord. *You can think of the RUP as a smorgasbord of best practices. Rather than eat everything, eat only your favorite dishes -- the ones that make sense for your specific project.*

Summary

Working software is the best indicator of true progress. When assessing progress, as much as possible look at what code is up and running, and which test cases have been properly executed. A strong focus on working software also enables you to minimize overhead by producing only those artifacts that add more value to your project than they cost to produce.

4. Accommodate Change Early in the Project

Change is good. Actually, change is great. Why? Because most modern systems are too complex to allow you to get the requirements and the design right the first time. Change allows you to improve upon a solution. If there is no change, then you will deliver a defective solution -- possibly so defective that the application has no business value. That is why you should welcome and encourage change. And the iterative approach has been optimized to do exactly that.

But change can also have severe consequences. Constant change will prevent project completion. Certain types of change late in the project typically mean a lot of rework, increased cost, reduced quality, and probable delays -- all things you want to avoid. To optimize your change management strategy, you need to understand the relative cost of introducing different types of changes at different stages of the project lifecycle (see Figure 4). For simplicity, I group changes into four categories.

- **Cost of change to the business solution.** Change to the business solution involves major rework of requirements to address a different set of users or user needs. There is a fair amount of flexibility in making this type of modification during the Inception phase, but costs escalate as you move into the Elaboration phase. This is why you force an agreement on the vision for the system in Inception.
- **Cost of change to the architecture.** When following the RUP, you can make fairly significant architectural changes until the end of Elaboration at low cost. After that, significant architectural changes become increasingly costly, which is why the architecture must be baselined at the end of Elaboration.
- **Cost of change to the design and implementation.** Due to the component-based approach, these types of changes are typically localized, and can hence be made at fairly low cost through the Construction phase. These types of changes are, however, increasingly expensive in the Transition phase, which is why you typically introduce feature freeze at the end of Construction.
- **Cost of change to the scope.** The cost of cutting scope -- and hence postponing features to the next release -- is relatively inexpensive throughout the project, if done within limits. Scope cutting is a key tool project managers should use to ensure on-time project delivery.

The above cost considerations mean that you need to manage change. You need to:

- Have procedures in place for approving whether to introduce a change.
- Be able to assess the impact of change.
- Minimize the cost of change.

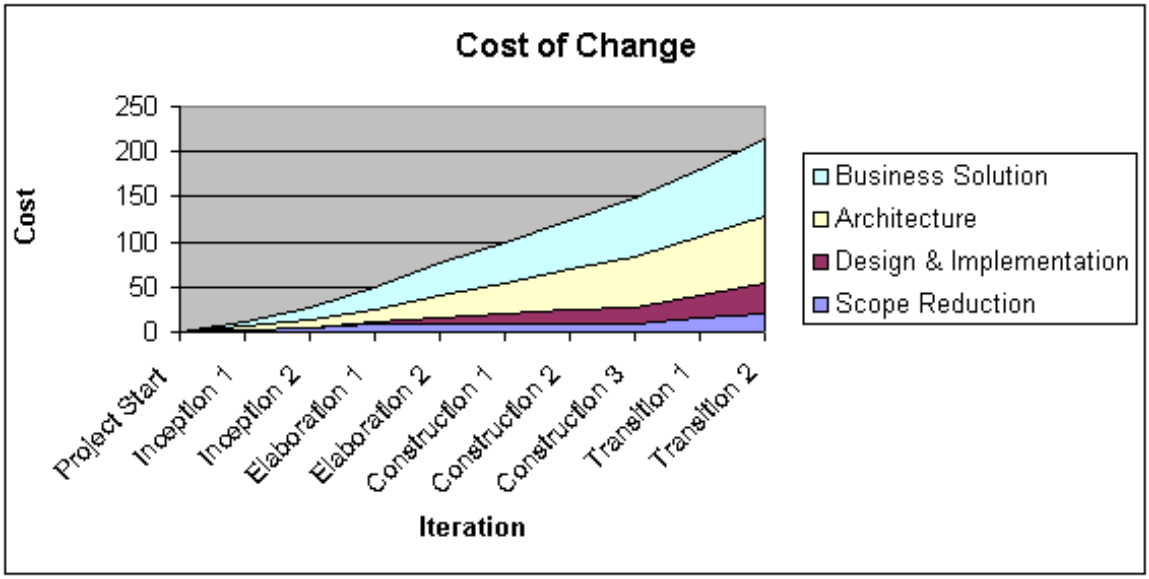

Figure 4: The Cost of Introducing Change Varies Through the Lifecycle. *The cost of*

introducing a change varies according to the lifecycle phase, and each type of change has its own cost profile. The RUP milestones at the end of each phase are optimized to minimize overall cost of change, while maximizing freedom to make changes. In general, as the project progresses, you should be more careful about introducing change, especially to the overall business solution and the architecture.

Change approvals are done through a combination of manual procedures (for example, through Change Control Boards), and through tools such as Configuration and Change Management software. Assessing the impact of change is primarily done through traceability between tools for requirements, design, code, and test. When changing a class or a requirement, you should be able to understand what other things are potentially affected; this is where traceability saves a lot of grief.

The cost of change is minimized by having procedures in place for managing changes and also by assessing their impact. Again, the right tool support can have a tremendous effect. Round-trip engineering between requirements and design, and between design and code, are examples of automation reducing the cost of change.

Summary

The cost of change increases the further you are into a project, and different types of changes have different cost profiles. The RUP's phases have been optimized to minimize overall cost of change, while maximizing the ability to allow for change. This is why the RUP forces agreement on the overall vision at the end of the Inception phase, a baselined architecture at the end of the Elaboration phase, and feature freeze at the end of the Construction phase. Using the right tools can play a powerful role in managing change and minimizing its cost.

5. Baseline an Executable Architecture Early On

A lot of project risks are associated with the architecture. That is why you want to get the architecture right. In fact, the ability to baseline a functioning architecture -- that is, to design, implement, and test the architecture -- early in the project is considered so essential to a successful project that the RUP treats this as the primary objective of the Elaboration phase, which is phase two of this four-phase process.

First, what do we mean by architecture?[5] The architecture comprises the software system's most important building blocks and their interfaces -- that is, the subsystem, the interfaces of the subsystems, and the most important components and their interfaces (see Figure 5). The architecture provides a skeleton structure of the system, comprising perhaps 10 to 20 percent of the final amount of code. The architecture also consists of so-called "architectural mechanisms." These are common solutions to common problems, such as how to deal with persistency or garbage collection. Getting the architecture right is difficult, which is why you typically use your most experienced people for this task.

Having the skeleton structure in place provides a sound understanding of

the building blocks or components needed for the final product. And, having followed the RUP's iterative process, your team will already have gained some valuable experience in doing analysis, design, implementation, and testing, so you will usually have a firm grasp of what it will take to complete the system. Baselining an executable architecture also lays the groundwork for accurate assessments of how many resources are needed, and how long it will take to complete the project. Early understanding of this enables you to optimize your resource profile and manage scope to best address your business needs.

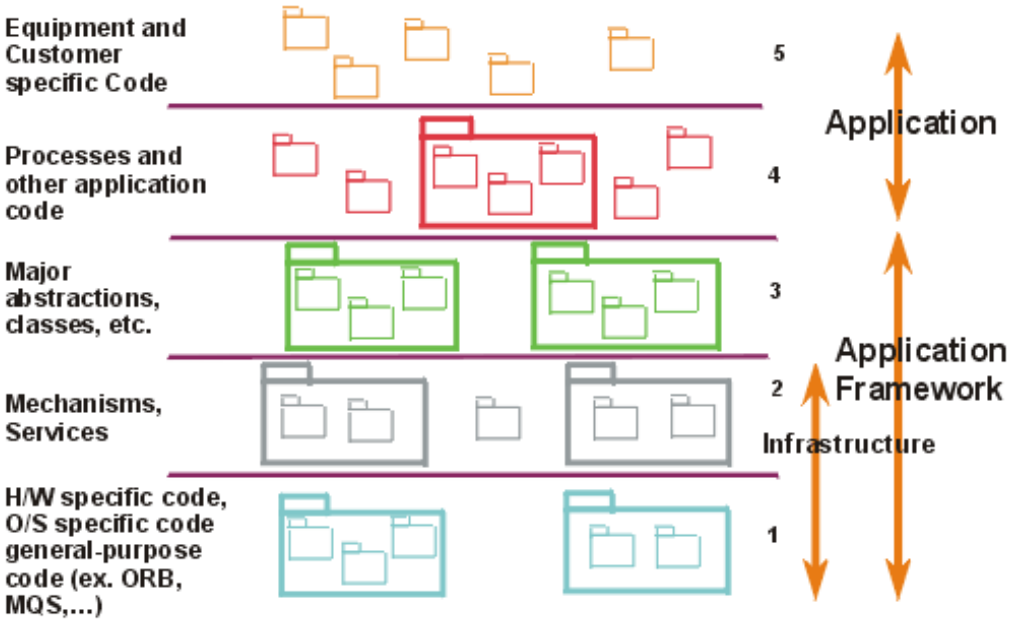

Figure 5: A System Architecture. *The architecture provides an understanding of the overall system and a skeleton structure for the application. It includes, among other things, the subsystems and their interfaces, and the most common components and their interfaces.*

When the architecture is in place, you have addressed many of the most difficult parts of building the system. It is now much easier to introduce new members to the project; boundaries are provided for additional code through the definition of key components and baselining of interfaces, and the architectural mechanisms are ready to be used, providing ready-made solutions to common problems.

Summary

The architecture is the system's skeleton structure. By designing, implementing, and testing the architecture early in the project, you address major risks and make it easier to scale up the team and to introduce less-experienced team members. Finally, since the architecture defines the system's building blocks or components, it enables you to accurately assess the effort needed to complete the project.

6. Build Your System with Components

One aspect of functional decomposition[6] is that it separates data from functions. One of the drawbacks with this separation is that it becomes expensive to maintain or modify a system. For example, a change to how

data is stored may impact any number of functions, and it is generally hard to know which functions throughout a given system may be affected (see Figure 6). This is the major reason the Y2K issue was so difficult to address.

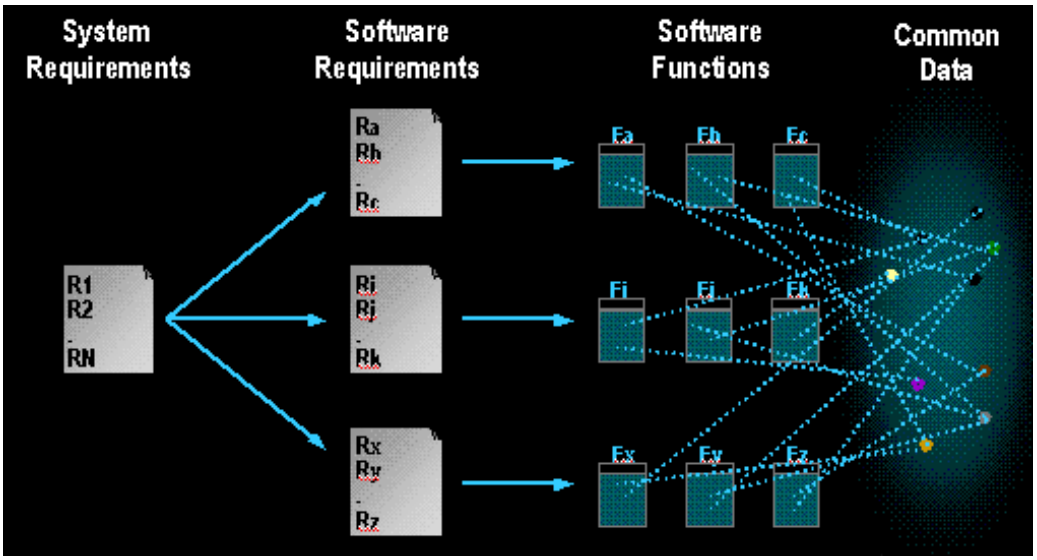

Figure 6: A Functional Decomposition Architecture. *Functional decomposition has the disadvantage that changes (to how data is stored, for example) may impact many functions, leading to systems that are both highly difficult and expensive to maintain.*

On the other hand, component-based development encapsulates data and the functionality operating upon that data into a component. When you need to change what data is stored, or how the data can be manipulated, those changes can be isolated to one component. This makes the system much more resilient to change (see Figure 7).

To communicate with a component, and hence take advantage of all its capabilities and code, you only need to know the component's *interface*. You don't need to worry about its internal workings. Even better, a component can be completely rewritten without impacting the system or system code, as long as its interface does not change. This is an important feature of component-based development called encapsulation, which makes components easy to reuse.

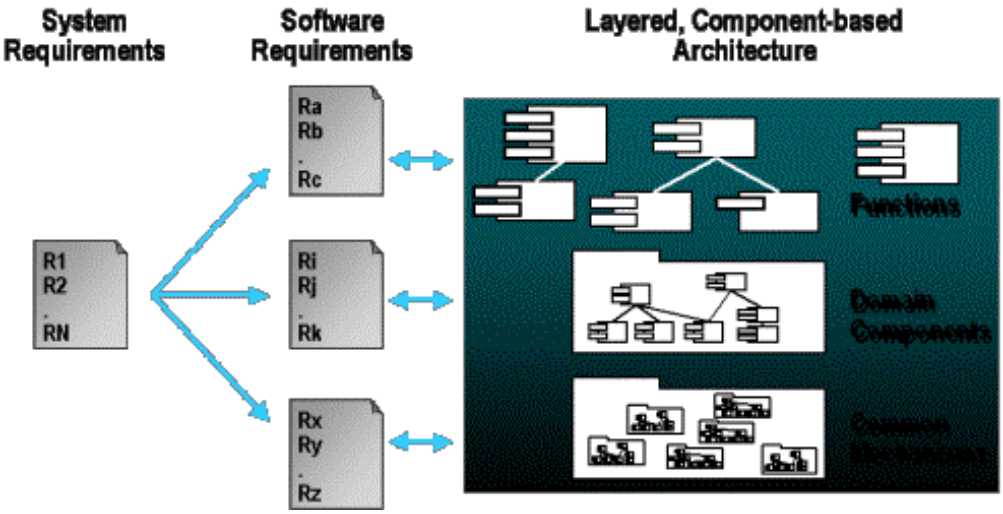

Figure 7: A Component-based Architecture. *Component-based development leads to systems that are more resilient to changes in requirements, technology, and data.*

A component can also be assembled by other components, thereby allowing it to provide many advanced capabilities. The combination of encapsulation and the availability of large components radically increases the productivity associated with reuse when developing applications.

Component technology is also the basis for the Web services initiatives recently launched by all the major platform vendors on both J2EE and .NET platforms, and it is why Web services may well be "the next big thing" in software development. In short, Web services are "Internet-enabled" components. Think of them as components on steroids. Like all components, they have a well-defined interface, and you can take advantage of all their capabilities simply by knowing that interface. As long as the interface does not change, you are unaffected by changes to a Web service. The major difference is that while a normal component typically limits you to communicating with components developed on the same platform, and sometimes only with components compiled on the same system, a Web services architecture allows you to communicate independently of the platform by exposing component interfaces over the Internet.[7]

Summary

Component-based development relies on the object-oriented principle of *encapsulation*, and enables you to build applications that are more resilient to change. Components also enable a higher degree of reuse, allowing you to build higher quality applications faster. This can radically decrease system maintenance costs. Component-based technology is the basis for Web services offered on J2EE and .NET platforms.

7. Work Together as One Team

People are the project's most important asset. Software development has become a team sport, and an iterative approach to software development impacts the ways in which you organize your team, the tools your team needs, and the values of each team member.

Traditionally, many companies have had a functional organization: All the analysts are in one group, the designers are in another group, and testers are in yet another group -- maybe even in another building. Although this organizational structure builds competency centers, the drawback is that effective communication among the three groups becomes compromised. Requirements specifications produced by analysts are not used as input by developers or testers, for example. This leads to miscommunication, extra work, and missed deadlines.

Functional organizations may be acceptable for long-term waterfall projects, perhaps as long as eighteen months or more. But as you move toward iterative development and shorter projects of nine months, or even

two to three months, you need a much higher bandwidth of communication between teams. To achieve this, you must:

- Organize your projects around cross-functional teams containing analysts, developers, and testers.
- Provide teams with the tools required for effective collaboration across functions.
- Ensure that team members have the attitude of "I'll do what it takes to get high-quality working software," rather than "I'll do my small part of the lifecycle."

Let's take a closer look at each of these points.

The project team should consist of analysts, developers, testers, a project manager, architects, and so on. You might say that this works for small projects, but what happens when projects become bigger with, say, fifty people involved? The answer is to organize around the architecture,[8] to group what we call "teams of teams." (See Figure 8.) Have a team of architects that own the architecture; they decide on the subsystems and the interfaces between them. Then, for each subsystem, have a cross-functional team consisting of analysts, developers, and testers who work closely to ensure high-bandwidth communication and fast decisions. They communicate with other teams primarily through the architecture and the architecture team.

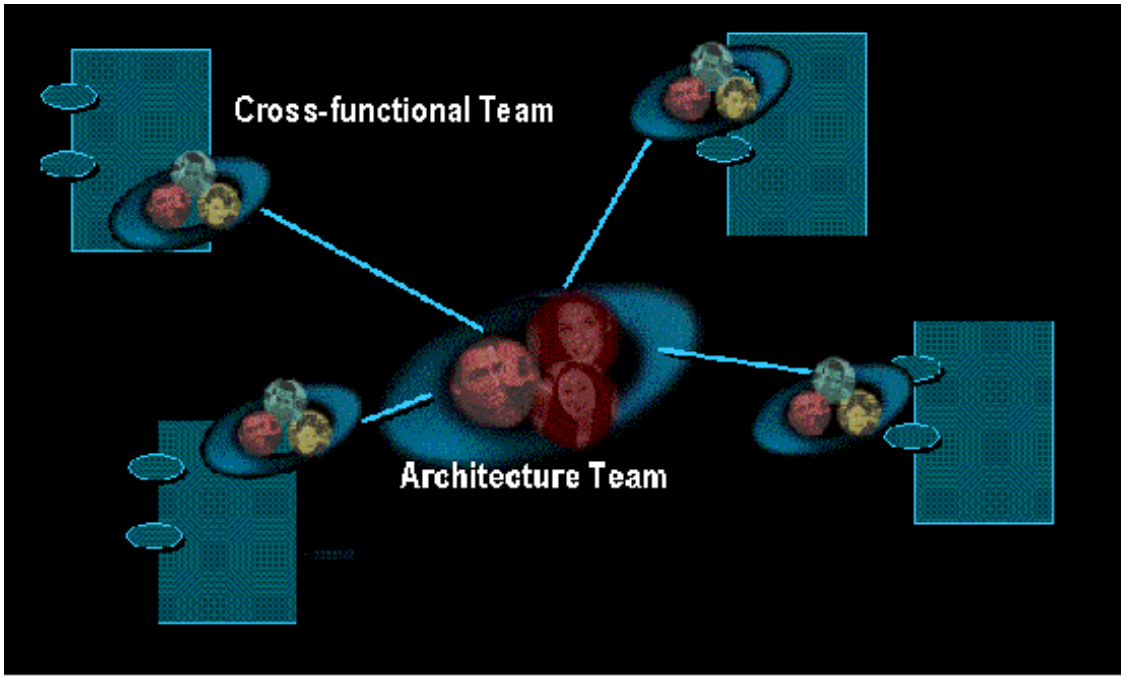

Figure 8: Teams Organized Around Architecture. *If the project is too big to have everyone on one team, organize teams around the architecture in "teams of teams." An architecture team owns the subsystems and their interfaces, and a cross-functional team is responsible for each of the subsystems.*

For a team of analysts, developers, and testers to work closely together, they need the right infrastructure. You need to ensure that all team members have access to the requirements, defects, test status, and so on.

This, in turn, puts requirements on tooling. Communication between the different team members must be facilitated through integration between their tools. Among other things, this increases the ROI on tools, allowing round-trip engineering and synchronization of requirements, design elements, and test artifacts.

Working together as a team also enforces joint ownership of the final product. It eliminates finger pointing and assertions such as "*Your* requirements were incomplete" or "*My* code has no bugs."[9] Everyone shares project responsibility and should work together to solve issues as they arise.

Summary

An iterative approach increases the need for working closely as a team. Avoid functional organizations and instead use cross-functional teams of generalists, analysts, developers, and testers. Ensure that the tool infrastructure provides each team member with the right information, and promotes synchronization and round-trip engineering of artifacts across disciplines. Finally, make sure that team members take joint ownership of the project results.

8. Make Quality a Way Of Life, Not an Afterthought

One of the major benefits of iterative development is that it allows you to initiate testing much earlier than is possible in waterfall development. Already in the second phase, Elaboration, executable software is up and running, implementing the architecture (see Figure 9). This means you can start testing to verify that the architecture really works. You can, for example, do some simple load and performance testing of the architecture. Gaining early feedback on this (perhaps one third of the way into the project) may result in significant time and cost savings down the road.

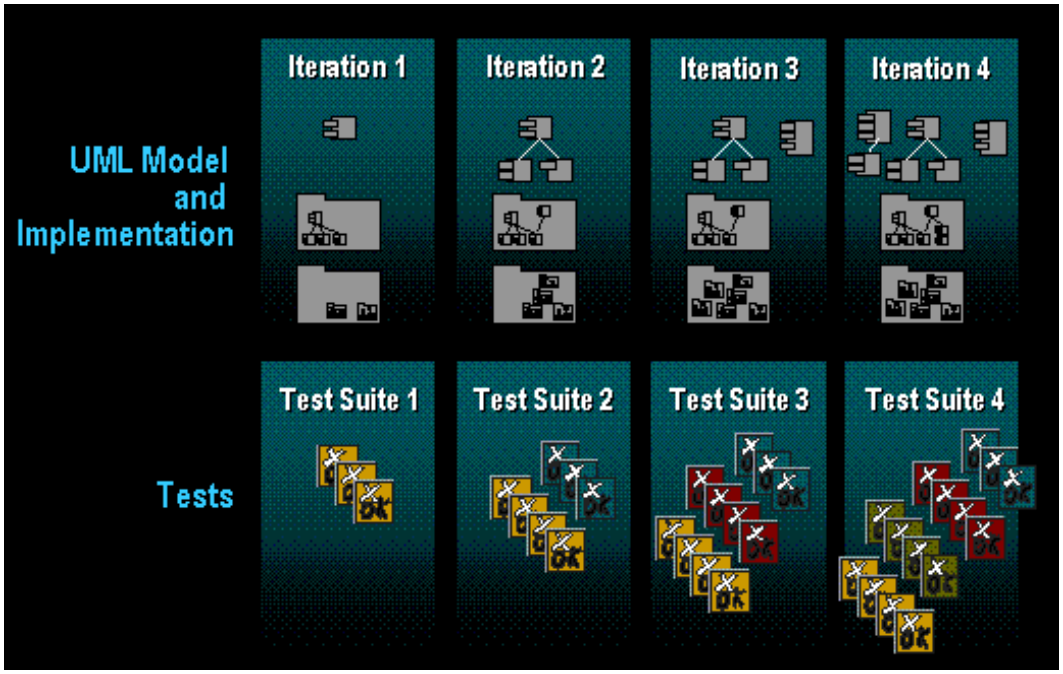

Figure 9: Testing Is Initiated Early and Expanded Upon in Each Iteration. *The RUP*

promotes 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.

In general, the RUP requires you to test capabilities as you implement them. 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 is likely to have been tested for months. It is not a surprise that most projects adopting the RUP claim that an increase in quality is the first tangible result of the improved process.

Another enabler is what we call "Quality By Design." This means coupling testing more closely with design. Considering how the system should be tested when you design it can lead to greatly improved test automation, because test-code can be generated 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. (After all, test scripts are software and typically contain a lot of defects, just like any other software.)

Iterative development not only allows earlier testing; it also forces you to test more often. On one hand, this is good because you keep testing existing software (so-called regression testing) to ensure that new errors are not introduced. On the other hand, the drawback is that regression testing may become expensive. To minimize costs, try to automate as much testing as possible. This often radically reduces costs.

Finally, quality is something that concerns every team member, and it needs to be built into all parts of the process. You need to review artifacts as they are produced, think of how to test requirements when you identify them, design for testability, and so on.

Summary

Ensuring high quality requires more than just the participation of the testing team. It involves *all* team members and *all* parts of the lifecycle. By using an iterative approach you can do earlier testing, and the quality-by-design concept allows software designers to automate test code generation for earlier and higher quality testing, thus reducing the number of defects in the test code.

Conclusion

We have examined the fundamental principles that represent the "Spirit of the RUP." Understanding these principles will make it easier for you to properly adopt the RUP. Don't get lost in the wide set of available activities and artifacts that the RUP offers; instead, let the "spirit" guide you, and you will more quickly find which activities and artifacts are appropriate for your specific project.

Notes

[1] Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. Addison-Wesley, 2000.

[2] Tom Gilb, *Principles of Software Engineering Management*. Addison-Wesley, 1988.

[3] For more information, see Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley, 1992.

[4] The UML, or Unified Modeling Language, is a standard managed by the Object Management Group. Rational Software developed the initial proposal for this standard.

[5] For a good discussion on this topic, see Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed. Addison-Wesley, 2000.

[6] In functional decomposition, complex functions are recursively broken down into smaller and simpler functions, until you have functions that can easily be implemented. You also separate the data from the functions, which gives you a many-to-many dependency mapping between functions and data.

[7] For more information on Web Services for .NET platform, see Thuan Thai & Hoang Q. Lam, *.NET Framework Essentials*. O'Reilly, 2001.

[8] For more information on how to organize your team, and other management issues, see Walker Royce, *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

[9] For more on this topic, see Philippe Kruchten, "From Waterfall to Iterative Development: A Challenging Transition for Project Managers." *The Rational Edge*, December 2000. http://www.therationaledge.com/content/dec_00/m_iterative.html

---

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!