

▶ **The Seven Habits of Effective Iterative Development**

by [Eric Lopes Cardozo](#)
Director, Empulsys

In his book The Seven Habits of Highly Effective People,¹ Stephen Covey describes seven related principles that can make a person more effective in his or her personal relations, interpersonal relations, and management efforts. At the time I read this book, I was mentoring the implementation of the Rational Unified Process® (RUP®) into an organization. After reading three or four chapters, it came to me that the "seven habits" could serve as a framework to highlight the essentials of managing iterative software projects.



Based on my experience, many project managers have problems with planning and controlling iterative projects, determining iteration content, selling the iterative approach to customers and sponsors, and establishing effective communication, both among project team members and with others in the project environment.

Because of its collaborative, problem-solving character,² iterative software development is similar to a multidisciplinary project or parallel development, which places a high demand on communication between the project team and project stakeholders, and among team members themselves. Covey places great emphasis on communication, so his seven habits seem to fit iterative development perfectly.

In this article, we will first have a look at five common causes of iterative project failure that managers must avoid. Then, based on Covey's work, we will define seven habits project managers can use to help their iterative projects succeed.

[▶ subscribe](#)[▶ contact us](#)[▶ submit an article](#)[▶ rational.com](#)[▶ issue contents](#)[▶ archives](#)[▶ mission statement](#)[▶ editorial staff](#)

Five Top Reasons for Iterative Project Failure

An iterative approach to software development goes a long way toward addressing some of the fundamental reasons why projects fail, but it is still up to the project manager to implement the approach effectively. We will discuss the five top reasons for failure below.

1. Lack of Effective Communication

This is one of the most common sources of failure on development projects. Considerable research has been done on the need for effective communication, and the conclusions are clearly captured by the following definition:

"Communication is a prerequisite for effective coordination, as it is the vehicle through which personnel from multiple functional areas share information critical to the successful implementation of projects."³

A well-performed project start-up can lay a foundation for effective communication. The goal of a project start-up is to establish a credible basis for the project that is acceptable for *all* stakeholders. However, project managers too often fail to seek answers to some fundamental questions:

- What business objectives/benefits is the project intended to achieve?
- What level of quality is expected for the end product(s)?
- What risks did the customer consider in deciding to set up this project?

Getting the answers to these questions requires effective communication with the customer; building a project around these answers requires effective communication among project team members.

2. Lack of Buy-In for an Iterative Development Approach

A common fail factor on iterative projects is lack of buy-in for the process from both the customer and the project team.

Customer buy-in on the development process, particularly end-user involvement, is crucial. *Insufficient* end-user involvement is the number one reason why projects fail. Unfortunately, some customers resist getting involved. "*Why should I spend time thinking about what I want?*" they complain. "*I'm paying you a lot -- you go figure it out!*" In fact, in such cases, the customer is delegating the definition of the problem rather than the task of solving it. That is why it is important to get their buy-in on the development process during project start-up.

The same is true for project team buy-in. In contrast to a waterfall approach, iterative software development is a team effort. Activities such

as requirements capture, analysis and design, and testing are performed in parallel rather than sequentially, and this requires more coordination. Wars over methodology, languages, and tools will spell disaster for any iterative project.

3. Faulty Methodology

Of course, a major source of failure is the methodology itself. Everyone knows the big complaint about the waterfall approach: It doesn't provide a complete picture of project progress until near the end of the project, at the system integration phase, so you may not be able to detect serious problems until that point. This phenomenon, known as "late design breakage," can result in unnecessary rework and a lot of stress on people and budget.

But the waterfall methodology is not the only one that fails. Many projects using Rapid Application Development (RAD) or the Dynamic Systems Development Method (DSDM) have failed because they followed an incremental approach instead of an iterative one. The incremental approach looks like a sequence of small, but perfect, waterfall-like projects. It tends to deliver systems with a brittle or "stovepipe" software architecture, because the initial architecture is based on only a part of the problem space. The result is that the system's architecture never stabilizes, and the project team spends many hours doing unnecessary rework to the architecture instead of adding functionality with each iteration.

4. Poor Requirements Gathering and Documentation

Another fail factor lies in the way requirements are treated and documented. In some projects, requirements engineers seem to focus mainly on the customer, disregarding other stakeholders. They do not recognize that requirements must be unambiguous for a whole range of stakeholders, including developers and testers. In addition, traditional software development approaches typically document requirements as functions, which are not directly related to business value. This makes them hard to prioritize, thus compromising scope management and project steering. Another problem with functions is that they make it hard to develop test cases. Test designers are forced to guess at scenarios that will cover the real usage of the system from the user's point of view, when in fact these *should* have been captured by the requirements.

5. Lack of Unified Tool Use

Effective iterative development depends on the effective use of software development tools. However, unless the project managers choose the right tools and train their teams to use them correctly, then the tools may absorb more time and attention than the processes they are supposed to support. Many teams waste precious time and resources trying to use unsuitable tools that are forced upon them by organizational policies. Also, because iterative development depends on team effort, projects can get derailed if members are not using the same toolset or using it according to project guidelines. When a project member becomes ill and no one knows

where his or her code resides on the network, or a new team member comes on board who is unfamiliar with the practices of other developers on the team, then the project is at risk. Every developer brings his or her experience and habits to the project, and project managers should leverage this experience to improve project performance -- but only if it fits the project.

Seven Habits That Foster Success

Now that we have seen what often causes projects to fail, let's look at some of the habits a project manager can adopt to make their projects succeed.

1. Be Proactive

In the context of a software development project, Covey's "Be Proactive" habit means "Actively attack risks and seek out the project's stakeholders if they don't come to you."

The main benefit of working iteratively is risk mitigation. Every project is confronted with risks that may include design flaws, ambiguous requirements, lack of experience, new technology, inadequate user involvement, an inadequate development environment, and so forth. It is almost impossible to recognize them all on day one. At project start-up and near project closeout, most risks are associated with the project's environment: Do we really have users? Are the users trained and ready for deployment? When the project team starts to analyze requirements, however, risks are increasingly associated with technical issues. The risk associated with a "big-bang" integration strategy is that the design may not reflect the (real) requirements. Late discovery of design defects can then cause budget and schedule over-runs, which may eventually kill the project.⁴

Develop the "Delivery Habit"

In iterative development, risks are mitigated by developing parts in sequence, so that the system evolves instead of being constructed and integrated all at once, near the end of a project. From a cultural perspective, this means that the project team must adopt a "delivery habit" that ensures progress (i.e., a demonstration-based approach). With each iteration, they will mitigate more risks and deliver more function to the user. Progress will be measured by the results of system tests and user feedback that indicate which requirements are now specified, designed, incorporated, tested, or deployed. If delivery stops, there will be no visible progress, and the project will be in danger. One important practice in iterative development is the delivery of *an executable system* at the end of each iteration (except for the first or second one). The motto is: Make sure you're making *progress, even imperfect progress*.

The "delivery habit" reflects a proactive attitude. When they develop iteratively, project members work cooperatively on more artifacts within a tighter timeframe; they do not sit around waiting for someone else to finish an activity before they get started. They realize that they can get a

lot done, even if the other person is only halfway through.

Take Action in the Face of Risk

Adopting a proactive attitude also means that one must act when confronted with risks.⁵ For example:

- When the scope of the iteration turns out to be too large to deliver on schedule, reduce the scope and deliver a smaller solution.
- If the customer is not able to visit the development site, try to go to them.
- If the customer cannot seem to express his or her ideas about the user interface, then develop a prototype they can react to.
- When a project depends on services to be developed by another project, create stubs in case the other project does not deliver on schedule.

Build a Partnership with Stakeholders

A proactive attitude also helps in developing a partnership with project stakeholders and establishing effective communications. Demonstrate that you understand the business needs the project is designed to meet and that the project team is committed to building the right solution for the right problem. Also explain the development process to stakeholders and show how it supports building the right solution.

To be proactive, it is essential to determine who the stakeholders are: Who influences and who makes decisions? This activity is known as *stakeholder analysis*. Once you know who these people are, you can think of ways to get them on board (or deliberately *not* bring them in) and keep it that way. In general, there are four types of stakeholders:

- **End users.** People who will use or buy the product.
- **System users.** People who will keep the product "alive" during its post-deployment lifecycle (i.e., maintenance and support personnel, suppliers).
- **Temporary users.** People who develop the product or are involved with the product roll-out (e.g., the project team, engineers, marketing people, trainers).
- **Other Stakeholders.** People who are not directly involved in the project but have the power to either make or break it (e.g., laws and regulations, other projects, environmental movements).

2. Begin with the End in Mind

For our purposes, we can interpret Covey's "Begin with the End in Mind" as "Structure project iterations according to project lifecycle goals."

One of the pitfalls of iterative development is that you can get yourself

into a never-ending sequence of iterations. Obviously, when this happens, your project will be late and over budget. In most cases the project team begins enthusiastically but then gets lost somewhere near the "end." One way to avoid this is to begin with the end in mind. Iterative projects should be structured around goals. In the Rational Unified Process (RUP), these goals take the form of phases and milestones.

Lifecycle Objectives Milestone

The Lifecycle Objectives milestone marks the end of the Inception phase. The most important goal of this phase is to achieve concurrence among stakeholders on the project's scope and vision. According to the RUP, the Vision artifact, along with the Use-Case Model, is used to document the project's scope. The Vision artifact captures the problem statement, business needs (problem space), and traceable high-level requirements or features. The Vision describes the "end" of the project, and project closeout is achieved when the Vision is successfully delivered to the stakeholders. After the Vision is stabilized, all subsequent iterations must deliver a part of that Vision.

Lifecycle Architecture Milestone

The Lifecycle Architecture milestone marks the end of the Elaboration phase. In this phase, the most important goal is to develop an architectural baseline that demonstrates that the software architecture is capable of delivering the Vision. Technical risks are mitigated by developing and testing the architecturally significant part of the Use-Case Model. Iteration goals could also incorporate development of non-architectural parts of the Use-Case Model in order to mitigate non-technical risks such as scope risks. Also, a system must satisfy non-functional requirements such as performance and availability. In order to provide the requested performance (the end), the project team must first determine whether the software architecture is capable of satisfying the requested performance requirements. Furthermore, the team must determine whether the requested performance requirements are both realistic and capable of satisfying business needs.

Initial Operational Capability Milestone

The Initial Operational Capability milestone marks the end of the Construction phase. The goal of this phase is to produce a first product release (beta), which will be used for product acceptance in the next and last phase. Once the software architecture is stabilized (Elaboration), the project team is capable of developing the remaining (*big*) part of the Vision safely (and with more people) in two to four iterations. In this stage, "speed and quality" is the motto.

Product Release Milestone

The Product Release milestone marks the end of the Transition phase. In this phase, the goals are to deploy the system in the user environment and to achieve project closeout.

Keep in mind that each phase mentioned in these milestone descriptions

actually consists of one or more iterations. Phase goals are used to structure the sequence of iterations. Iteration goals are refinements of these high-level phase goals, and the instruction to "begin with the end in mind" applies to each phase *and* each iteration. For example, we noted that the goal of the Elaboration phase is to provide an architectural baseline capable of delivering the project's Vision. From an activity-driven point of view, this involves analysis, design, coding, integration, and test activities. From a goal-driven point of view, it involves addressing technical and scope risks. Each iteration should address one or more risks by developing related parts of the Use-Case Model.

From a project management point of view, there is a part of the project plan where *being proactive* intersects with *beginning with the end in mind*. Early in the project (e.g., during Inception) the project manager, together with the stakeholders, should develop a Product Acceptance Plan. This plan helps to clearly define the end of the project and will also confront stakeholders with conditions for project closeout early in the lifecycle. A Product Acceptance Plan describes *who* accepts *what* and *when*, against *which* criteria. In most cases it identifies new stakeholders (e.g., maintenance and support) and new requirements. It also provides important information about the contents of the Transition phase (e.g., how many iterations it will entail). Often the Transition phase is limited to one iteration because of cost constraints. However, the number of Transition iterations and the length of each iteration are defined by the stakeholders who are responsible for product acceptance.

3. Put First Things First

We can translate Covey's "Put First Things First" as "Organize and perform activities according to their priority."

Within the context of iterative development, this habit is where "*Be Proactive*" (be risk-driven) meets "*Begin with the End in Mind*" (be goal-driven). We have already discussed the importance of a *demonstration-based* approach, which means that intermediate artifacts (user interface prototypes, product baselines) are assessed in an executable environment rather than by reviewing reports. Transitioning intermediate artifacts into an executable demonstration of relevant scenarios stimulates earlier convergence on integration, a more tangible understanding of design trade-offs, and earlier elimination of architectural defects.⁶ The demonstration-based approach ensures that progress is measured against tangible results instead of placing blind faith in projections on paper.

What is to be demonstrated in which iteration very much depends on the project's state with respect to its lifecycle and risks inherent in the project. These risks can vary over the project lifecycle, as it is defined by the RUP.

Risks During Inception

During Inception, most risks fall within the project environment, which includes the organization, funding, people, tight schedules, and expected benefits of new technology. During this phase, developing a business case is a crucial step. A business case promotes understanding of the business

problem and buy-in from the project sponsors. It also helps to explain the project's business drivers to other stakeholders. Furthermore, a business case is the most powerful weapon against feature creep. Developing one must be a joint effort by the project team -- which is responsible for determining development costs and schedule -- and the customer, which is responsible for defining the benefits.

Risks During Elaboration

During Elaboration, the focus shifts to technology. The goal of the Elaboration phase is to achieve an architectural baseline by mitigating (mostly) technical risks. Mitigation is achieved by developing and testing critical parts of the system (i.e., through an architecture-first approach). Therefore, be very careful with the functional load of elaboration iterations. Developing and testing the tricky parts of the system is hard enough. Remember that each iteration must result in executable software. Although tight schedules are necessary to keep the project team going, unrealistic ones can wear out project teams. In addition, the customer will not be pleased if the goals of iterations are not met.

Which part of the system must be developed during Elaboration? The trick is to match technical risks with use cases or scenarios that:

- Represent crucial business functions (without these there is not much of a system).
- Interact with external systems.
- Incorporate critical non-functional requirements such as availability and performance.
- Represent parts of the system whose scope isn't clear (so that delivering a small solution might *clear the fog*.⁷)

Another risk in the Elaboration phase is that the user will see an early version of the system for the first time, and typically only about 20 percent of it will work. To maintain buy-in from the user group, it's critical to manage expectations (i.e., to *be proactive*).

Risks During Construction

In the Construction phase, the focus is on completing the system, and, as I noted earlier, the motto is "speed and quality." This can only be achieved by adding functionality to a stable architecture and having an effective build/release process. Both should be established in the Elaboration phase. So what can go wrong? Well first, when customers see the system in Elaboration, they might come up with new requirements. In addition, because more developers are brought in during Construction, communications can begin to break down within the project team. In this phase, whatever happens, the team must be focused on adding use cases to the system. To overcome feature creep, they must negotiate any new requirements (by sticking to the business case) and ensure that an effective change process is in place.

At the end of the Construction phase, the project team hands over a preliminary product release to the customer for acceptance testing. A *proactive attitude* is necessary to ensure that the customer is ready for this. The team should be thinking through all concerns about hardware, software, manuals, test data, maintenance and support personnel, and end-users (or representatives). As mentioned earlier, developing a product acceptance plan during Inception is a start to mitigating any risks associated with the hand-over. The team should also start thinking about deployment during the Elaboration phase, when the Deployment View of the software architecture (mapping of software to hardware) stabilizes (*an instance of "Begin with the end in mind"*).

Risks During Transition

In the Transition phase, the focus is on acceptance testing and repairing defects. The project team must facilitate the acceptance process. Therefore, the release and change process must be quick and reliable. The deployment of new releases or fixes must be rapid to keep the acceptance-testing going. When repairing defects, the team must ensure that the integrity of the system is not jeopardized and that old defects do not return. *"First things first"* applies to the order in which defects should be addressed. The chances of introducing *new* defects when fixing known defects is high, and in this phase, the focus must be on repairing *must-fix* defects -- those that impact primary use cases (i.e., that crash the system or make it unreliable or inaccurate). Think twice before attempting to add missing functionality, and don't even think about adding "nice-to-have" features and functions. Whatever you do, ensure you have a change control board in place that includes the customer. This board should classify all defects and authorize repair only of those they identify as *must-fix*.

4. Think Win/Win

For a software project, we can interpret Covey's "Think Win/Win" to mean: "Satisfy a maximum number of business needs with a minimum of effort."

In iterative development, *win/win* relates to scope management. The return on investment (ROI) for software projects can be improved by reducing the product size. This can mean reducing the amount of code required for the product to fulfill the needs of the business, and/or reducing the number of system features. In most systems, 20 percent of the features solve 80 percent of the business needs. Ever see a remote control for a stereo set with thirty buttons? Chances are that only six of those will ever get used.

One way to manage scope effectively is to follow the RUP's recommendation for adopting a *use-case-driven* approach to iterative development. This means that use cases are the basis for the entire development process.⁸ Traditional software development approaches use functions rather than use cases. The trouble with functions is that, unlike use-cases, they are not directly related to business value. Use cases more or less tie system functions together. They describe what the system must do in the language of the customer, so they are understandable by a wide range of stakeholders. They also form the basis for estimating, planning,

defining test cases and test procedures, and traceability.

Delivering software that is valuable to the business is as essential to the iterative approach as risk management, and with use cases, projects are driven by business value rather than by time or money. Customers do not like spending more money than was budgeted, nor do they like late delivery, or poor quality software. To help overcome these classic problems, the RUP offers the discipline of

- Prioritizing business needs and their use cases.
- Ensuring that use cases are traceable to business needs.

In combination with an architecture-first approach, this leads to a traceable development order based on risk and customer priority. Use cases should be developed in the following order:

1. High Risk, High Priority.
2. Low Risk, High Priority.
3. Low Risk, Low Priority.
4. High Risk, Low Priority.

Murray Cantor describes this way of ordering use cases in his book *Object-Oriented Project Management with UML*.⁹ High Risk, High Priority use cases should typically be developed in the Elaboration phase. High Risk, Low Priority use cases should be eliminated altogether because they add unnecessary risk to the project at a time when ROI is low. A better option is to *be proactive* and negotiate for less risky alternatives.

Thinking win/win gives the customer as well as the project team an edge when time-to-market is important. Ordering the use cases by technical risk and business priority and steering the project with use cases makes it very easy to reduce the scope. Removing low priority use cases should lead to fewer iterations during Construction and enable the team to tighten up the project schedule.

5. Seek First to Understand, Then to Be Understood

Covey urges us to "Seek First to Understand, Then to Be Understood." In a software development context, we can interpret this as "Understand the business objectives before thinking of solutions to realize them."

The initial iterations of a project must focus on *what* the system should do and *why* it should be built in the first place instead of *how* to build it. *Seek first to understand, then to be understood* has to do with understanding the *real* needs of the business and stakeholders before plunging into prototyping, detailing requirements, and writing code. As one ancient saying goes, "Once we know the problem, we can solve it many ways."

With the possible exception of very technical projects, at least some effort must be put into business modeling, especially when building information

systems. And when the customer is creating a new business process to implement along with the system, you are likely to fail unless you do business modeling, because chances are high that you will create a system that does not support the process. You may also increase development time and expense because end users and support and maintenance departments are not prepared to use the product. Again, it's critical to *begin with the end in mind*.

Another instance in which the *Seek first...* habit can serve you well is in dealing with organizational technology decisions. Most of the time business people have some idea of how the system should be built, but in some cases they have misguided notions about technology and insist upon using a certain brand of database or middleware for the wrong reasons. If this becomes a big source of friction, then you must be capable of explaining the negative impact of their decisions on the business. Therefore, it is necessary to understand the business needs first. This will also help you establish a strong partnership with the customer.

Finally, this habit will also help you set the focus for early iterations (*put first things first*). During the initial iterations (during Inception), the team should take great care not to get bogged down with developing user-interface prototypes and haggling with customers over details such as the size of buttons and colors. Establishing a user-interface style guide can be very important but there will be enough time to think about these details after your team has a better grasp of the problem.

6. Synergize

Covey's "Synergize" habit translates to "The value of a well-functioning software team is greater than the sum of the value of all individuals working on the team."

Synergize relates to the team aspect of software development. From one perspective, all team activities should contribute to writing code. In truth, however, writing code is usually the easiest part of a software project. As Rational's Grady Booch likes to say: "Software development is a team sport," and like all team sports, each person must have a clear role with clear responsibilities. Furthermore, team members must understand one another's strengths and weaknesses and be willing to compensate for them.

There are two basic prerequisites for a high-functioning team:

1. A well-chosen group of talented, highly skilled *team* players.
2. An environment within which these team players can operate synergistically.

Some people just can't get along with each other because their characters are either too similar or too different. According to team-building guru Dr. R. Meredith Belbin,¹⁰ each team member plays a double role. The first is a clear, functional role: A Java developer knows how to write Java code, for example. The second, or *team role*, characterizes a team member's

behavior in relation to other members and to facilitating team progress. Belbin recognizes nine team roles, as shown in Table 1.

Table 1: Belbin's Nine Team Roles

| Role | S/W | Description |
|-------------------------|------------|---|
| Innovator ¹¹ | Strength: | People who fall into this role are creative, imaginative, unorthodox, and capable of solving difficult problems. |
| | Weakness: | These innovators tend to ignore incidentals and can be too preoccupied to communicate effectively. |
| Coordinator | Strength: | A coordinator is mature, confident, and a good chairperson. Also clarifies goals, promotes decision-making, and delegates well. |
| | Weakness: | Coordinators can be manipulative. They also tend to off-load personal work. |
| Monitor | Strength: | A monitor is sober, strategic, and discerning. Sees all options and judges accurately. |
| | Weakness: | Monitors lack drive and ability to inspire others. |
| Implementer | Strength: | An implementer is disciplined, reliable, conservative, efficient, and capable of turning ideas into practical actions. |
| | Weakness: | Implementers are somewhat inflexible and slow to respond to new possibilities. |
| Completer | Strength: | A completer is painstaking, conscientious, and anxious. Excels at searching out errors and omissions and delivers on time. |
| | Weakness: | Completers tend to worry unduly and are reluctant to delegate. |
| Resource Investigator | Strength: | A resource investigator is extroverted, enthusiastic, and communicative. Good at exploring opportunities and developing contacts. |
| | Weakness: | Resource investigators are overly optimistic and tend to lose interest once initial enthusiasm has passed. |

| | | |
|------------|-----------|---|
| Shaper | Strength: | A shaper is challenging and dynamic, and thrives on pressure and courage to overcome obstacles. |
| | Weakness: | Shapers are prone to provocation and tend to offend people. |
| Teamworker | Strength: | A teamworker is co-operative, mild, perceptive, and diplomatic. Also listens, builds, and averts friction. |
| | Weakness: | Teamworkers are indecisive in crunch situations. |
| Specialist | Strength: | A specialist is single-minded, self-starting, and dedicated. Also provides knowledge and skills in rare supply. |
| | Weakness: | Specialists contribute only on a narrow front and may dwell on technicalities. |

Ideal teams comprise all of these team roles. Belbin's descriptions can help project managers assemble a compatible team, and help team members identify each person's role. Understanding someone's habitual way of communicating or interacting often makes his or her annoying habits easier to tolerate. Of course, if resources are scarce, then it can be hard to incorporate each of these roles into the project team, and team members may have to either play multiple roles or adopt a role that may not be his or her first preference. Belbin calls this shift in preferred behavior a "team-role sacrifice."

Once the project manager has assembled a team, the next thing is to establish an environment or structure within which the team can operate. Effective teamwork depends on sufficient communication. Communication is often insufficient when teams are structured according to their specialties or phase deliverables. Ever heard people complaining about bureaucracy and a "*throw it over the wall*" mentality? On the other hand, lack of structure can lead to *too much* communication, because everyone has to talk to everyone else to figure out how to get the job done.

One way to ensure the right amount of communication is through an Integrated Product Team (IPT) approach. This involves forming teams that are responsible for the major activities of the software development process; most project members belong to more than one team. This approach optimizes communication because of the overlap in team membership. It also brings down the walls because the IPTs consist of stakeholders who have an interest in software process activities rather than in the ownership of deliverables. For example, the project manager and the software architect together define the content of an iteration (i.e., they *think win/win*). The project manager must ensure that the iteration adds business value, and the software architect must ensure that technical risks are mitigated and the integrity of the architecture is guaranteed. When the content of the iteration has been defined, the project manager, software architect, implementer, and tester together review the iteration plan and define the number of builds within it, as well as the content and

delivery date for each build. They then document the results in an Integration Build plan. In this approach, planning is a team effort that involves members of several teams.

7. Sharpen the Saw

We can translate Covey's "Sharpen the Saw" into "Learn and improve during the project."

According to Stephen Covey, *"Improvement is the principle and the process on which one gains strength to grow in an upward spiral."* Software projects offer many opportunities for team members to learn and improve their skills, but this habit also applies to activities and artifacts. Improvements can be made in plans, requirements, designs, and code as well as in the use of tools, processes, guidelines, standards, and so on.

Iterations also provide an excellent mechanism for learning and improvement that goes beyond the process of making mistakes and then resolving them. The key to "sharpening the saw" lies in *artifact evolution*. Traditional software development emphasizes the completion of artifacts. In other words, it is activity-driven. Iterative software development, in contrast, is goal-driven (as in *begin with the end in mind*). For example, a project does not need a fully specified requirements document to enter the Elaboration phase. In general only 20 percent of the requirements set is required. The remaining part can be developed during the Elaboration phase and maybe the first iteration of the Construction phase. In other words, the requirements set *evolves*.

The concept of artifact evolution applies to all disciplines of a software project. During the Inception phase, one or more candidate software architectures are selected, and those architectures are refined during the Elaboration phase by designing, coding, integrating, and testing the critical part of the requirements set. The same concept applies to project management. Plans and estimates grow more and more detailed as the project progresses. The state of these artifacts should be in line with phase and iteration goals.

Even the project team evolves. It grows during the period from Inception to Transition and shrinks at the beginning of the Transition phase.

Another important mechanism for learning and improvement is the use of metrics. The amount of elements to measure depends on the size, complexity, and risks of a project. However, metrics must serve a clear purpose. Tracking the state of a use-case reflects progress over time. Measuring the number of subsystem defects over time can help detect specific risk areas in the software architecture. Measuring the baselined number of source-lines-of-code can be used to evaluate estimates. Measuring the number of open and closed defects over time can be used to assess whether the software architecture is stabilizing and to ensure that the team isn't on overload.

Above all keep in mind that the project is finished only after the customer formally accepts ownership of the software product. Until that point, all

team members should strive to "sharpen the saw by: being proactive; begin each phase and iteration with the end in mind; put first things first; think win/win; seek first to understand and then to be understood, and synergize with other team members.

Acknowledgments

I would like to thank Mark Tolsma for convincing me to read *The Seven Principles of Highly Effective People*. I would also like to thank Jakob Moll and Alex Krouwel for their support. Finally, I would like to thank my associate DJ de Villiers for his support and encouragement.

References

Stephen R. Covey, *The Seven Habits of Highly Effective People*. Simon & Schuster, 1989.

Walker Royce, *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

Alistair Cockburn, *Surviving Object-Oriented Projects*. Addison-Wesley, 1998.

Murray Cantor, *Object-Oriented Project Management with UML*. Wiley Computer Publishing, 1998.

For more information on team roles, see Dr. Meredith R. Belbin's Web site: <http://www.belbin.com>

M.B. Pinto and J.K Pinto, "Project Team Communication and Cross-Functional Cooperation in New Program Development." *Journal of Product Innovation Management*, No. 7, pp. 200-12, 1990.

Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd Edition. Addison-Wesley, 2000.

Notes

¹ Stephen Covey, *The Seven Habits of Highly Effective People*. Simon & Schuster, 1989.

² Murray Cantor, *Object-Oriented Project Management with UML*. Wiley Computer Publishing, 1998.

³ M.B. Pinto and J.K Pinto, "Project Team Communication and Cross-Functional Cooperation in New Program Development." *Journal of Product Innovation Management*, No. 7, pp. 200-12, 1990.

⁴ It is worth mentioning that project cancellation in itself is not necessarily a bad thing. Some projects are just not worth the effort because they will never lead to the expected benefits (business case). These projects should be cancelled as early as possible.

⁵ Alistair Cockburn describes a number of risk reduction strategy patterns related to the

subject in his book *Surviving Object-Oriented Projects*, Addison-Wesley, 1998.

- Prototype
- Clear the Fog
- Gold Rush
- Someone Always Makes Progress
- Team per Task
- Sacrifice one Person

6 Walker Royce, *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.

7 Alistair Cockburn, *Op.Cit.*

8 Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd Edition. Addison-Wesley, 2000.

9 Murray Cantor, *Object-Oriented Project Management with UML*. Wiley Computer Publishing, 1998.

10 See <http://www.belbin.com>

11 Editor's note: In Belbin's list, this role is actually called "plant" -- so named because some teams in Belbin's original experiments included individuals who were considered innovative brainchildren, and the organizers wanted to see what effect "implanting" these individuals on teams might have on team behavior and performance. For the purposes of this article, we've renamed it along the lines of the other role names.



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