A Software Development Process for Small Projects

The authors' development process integrates portions of an iterative, incremental process model with a quality assurance process and a measurement process used for process improvement. Their process aims to produce high quality and timely results with less overhead.

Melissa L. Russ and John D. McGregor,
Korson-McGregor, A Software Technology Company

A software development process can be just as critical to a small project's success as it is to that of a large one. A small project might appear to have less need for the coordination that a process provides. However, such projects often have a larger number of external dependencies per team member. For example, the small-project development team often has a closer association with its customers, requiring more team member interaction than does the large project development team. This is due to the nature of the customer being not generic or commercial, but rather being very specific stakeholders in the project. The small project often must achieve quality goals that are just as stringent as those of any large project, yet with fewer team members. Finally, the small-project team might include several people with only part-time participation on the project, such as domain experts, architects, and system test personnel, which requires more coordination and interaction to effectively utilize their skills. A process focuses the efforts of all the team members so that dependencies can be managed more efficiently to achieve the project's goals.

We have adapted portions of several standard process models to provide a software development process for small projects. The process integrates many activities that might appear in separate processes in a larger project. Its goal is to produce the high quality and timely results required for today's market without imposing a large overhead on a small project.

Small Projects and Process

A process can be important to small projects for reasons other than the large number of dependencies. Consider the following three scenarios.

My Favorite Activity Gets All My Attention

In this scenario, the development team does an outstanding job with whichever development phases its members care most about and does little or nothing about the other phases. For example, if the team consists of mostly domain-literate personnel, the domain model will be fully elaborated, but the code will not meet performance goals or might experience significant fail-

0740-7459/00/$10.00 © 2000 IEEE

ures. If the team favors code, the system will run but might not satisfy system requirements. A process provides a context that reminds team members of the steps necessary for producing a quality product.

Stuck in a Rut and Just Digging Deeper

This scenario includes the often-cited "paralysis by analysis" syndrome. In a rapidly changing domain, by the time a team completes a phase of the development process, some of its work is out of date. The team tries to repair this before moving to the next phase. The result: the team never moves on. You can overcome this quest for completeness by adopting a process that defines iterative passes through the phases. In this way, exit criteria ultimately require "absolute completeness" but accept partial results in initial iterations.

What Will I Do Today?

Here, the team makes little progress because it is unsure about what to do next. This scenario implies wasted time as team members try to complete activities for which they have insufficiently prepared. For example, attempting class implementation before properly specifying the methods usually results in incomplete handling of errors and special cases. So, routines for handling these special cases and error-handling must be added later in an ad hoc fashion. In this scenario, the team has no flow of activities to guide it. Grady Booch claims that every successful project has a rhythm.[1] Process establishes and guides that rhythm.

Our Development Process

Our experience and that of others in our company range from single-developer, in-house projects, to multiple small teams in a multinational corporation's business unit, to large projects with hundreds of developers and a full complement of support personnel. We have drawn on all those experiences to compare and contrast the process issues for large and small projects. (To see how we determine that a project is small, read the sidebar, "What Makes a Project Small?")

The software development process we describe here began as a process for our own in-house development projects, but we've also used elements of it in a variety of client projects. Our goal was to capture the activities essential for building a quality software product and to arrange the activities into a process that a small number of people can use effectively.

What Makes a Project Small?

Many environmental factors determine whether we should classify a project as small. If we understand these factors, we can define a process and tailor it to meet a small project's needs. Here are four factors and their potential impact on a project's dynamics.

The Development Organization's Size

A small project in an organization that runs hundreds of projects probably has access to an infrastructure of services and advice. In contrast, an organization with only one or a few projects probably cannot provide process writers, trained inspection moderators, and other supporting process and development services. For example, we were involved with a project, which was one of only a few projects, in an organization that mostly did system administration for third-party software. This organization did not perform or even understand standard process activities such as release scheduling and design reviews.

The Project's Complexity

One way of classifying a project's complexity is by examining the sophistication of the domain knowledge required for the project. For example, a classification might range from simple business processes to embedded real-time applications. The more complex the domain, the greater the need to formally structure the project activities. (We would probably all agree that following a checklist is more important for an airplane pilot than for a carpool driver.) The more complex a small project is, the more difficult it is to staff that project with a sufficient breadth of expertise. The development process should clearly define roles that allow persons with appropriate knowledge and skills to be assigned part-time responsibilities that the project team needs.

Quality Attributes

A system can have a number of specified quality attributes, such as reliability, security, and performance. The more exactly we can measure an attribute, the more we need specific process activities to achieve that attribute. Systems that must meet real-time performance requirements need more process support, such as detailed models and performance prototypes, than do those that have vaguely defined human-time performance criteria. Small projects usually involve less code, which means they often can achieve the required quality goals more easily. The exception to this might be quality attributes involved with the development of complex systems.

Personnel Interactions

Studies have shown that even small increases in the number of personnel can greatly increase the number of interactions necessary to make progress in a development project.[1] Small projects have an advantage in that the interactions are usually informal. Small projects also often benefit from a relatively shallow management structure. Occasionally the project staff will take an action that, after it percolates to higher management, must be reversed. The small project will be able to react more quickly because of the fewer layers of management involved.

Reference

1. J.O. Coplien, "A Development Process Generative Pattern Language," www.bell-labs.com/user/cope/Patterns/Process/index.html (current August 2000).

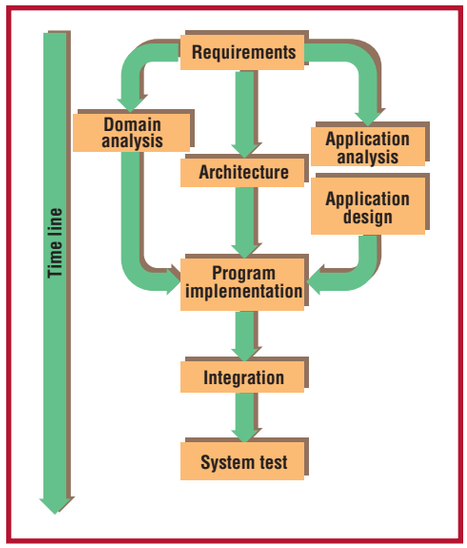

Figure 1. Temporal relationships between process phases.

An Integrated Approach

Most software development organizations use an iterative development approach to lower risk and improve quality.[2] These organizations also use an incremental approach to define customer releases and to divide a large problem into more manageable pieces. Small projects tend to be highly iterative both because synchronizing the developers requires less effort and because the management structure is sufficiently shallow, allowing quick feedback.

Our process integrates this basic iterative, incremental model with our *Guided Inspection* technique[3] and the measurement framework provided by Watts Humphrey's *Personal Software Process*.[4] Guided Inspections examine the products of software development for defects, using formal test cases based on specifications for the products. PSP defines tools and measures that help developers analyze and improve their personal productivity and quality. A number of organizations have successfully used it, and it has proven effective in facilitating individual improvement. However, PSP by itself does not fulfill the need for a project-level software development process that coordinates all the project's work.

Our process guides the developer on what steps to follow and what emphasis to place on activities in each development phase. The process uses Guided Inspection to ensure quality, and it uses a variation of PSP's measurement strategy to collect data for individual and project-level process improvement. To describe our process, we use a fairly typical set of attributes such as entry and exit criteria.

Merely combining several processes into one does not necessarily save time. So, we reduce the process overhead through several specific actions. First, management can require less formal communication among the project staff, resulting in developers creating fewer documents.

Second, by integrating processes we rearrange some of the tasks into more efficient configurations. This eliminates the startup overhead that would be incurred for a set of related tasks if those tasks were performed separately. For example, team members testing the integration of new software with the existing product are the same developers who created the new software. This eliminates the learning curve because the testers already understand the new software's functionality.

Finally, we eliminate some tasks or integrate them into other tasks. For example, integrating the Guided Inspection sessions into the development activities eliminates the need for formal design reviews.

Actuals and By-Products

Unlike PSP and other processes, our process defines two categories of artifacts: *actuals* and *by-products*. We base this on Walter Royce's advice[5] and on what we've learned about avoiding work-product-driven processes.

Actuals are those artifacts that are central to the product's successful development. For example, the actual output of the application analysis phase is "an understanding of the problem to be solved." By-products are the side-effects of the attempt to create the actuals—for example, diagrams, meetings, and prototypes. They can, and are intended to, help produce the actuals.

Unfortunately, some process writers believe that creating the by-products always results in achieving the actuals. Nothing could be further from the truth. By concentrating on completing each individual diagram because the process requires it, developers often miss important relationships among the diagrams that constitute the analysis model. They most often miss causal relationships in the domain that explain much of the problem. By-products are a necessary part of the process but are insufficient to reach the end goal: a quality, completed software application.

Roles

We assign roles for the different responsibilities required in the process; these roles guide interactions throughout the process cycle. One or more people can hold a particular role, and a person can simultaneously hold many roles. We determined the need for at least these roles:

- The *conceptualizer* has the original concept or idea for the project.
- The *customer* funds the project.
- The *user/domain expert* is knowledgeable about the domain and will use the system.

- The *manager* coordinates and facilitates the project.
- The *architect* creates and owns the architecture.
- The *developer* creates artifacts while performing some set of activities in the process.
- The *tester* performs some set of testing activities in the process.

Additional expertise is often needed—for example, an expert in foreign languages who will translate all messages presented to the user into a different language. This might require defining additional roles; however, most of this expertise is just a specialization of the developer role.

The Development Phases

Historically, people have solved problems using these steps: (1) determine the problem to be solved; (2) understand the problem to be solved; (3) develop a plan for solving the problem; (4) execute the plan; (5) assess whether the solution works.

Our software development process comprises eight phases that follow the problem-solving method:

1. *Requirements scoping*. Requirements definition captures from a variety of perspectives what the application is that we are trying to create. Requirements help define the scope of analysis activities and are the standards to which the final implementation is held accountable.
2. *Domain analysis*. This phase captures the concepts and relationships within the bodies of knowledge that underlie the basic problem to be solved by the application. Operating within the scope defined by requirements, domain analysis provides a superset of concepts and relationships needed for application analysis.
3. *Application analysis*. Using the captured concepts and relationships, this phase creates an understanding of what this specific application will become.
4. *Architecture*. This phase determines the basic structure of application components. This structure must be compatible with the structure of knowledge as revealed in domain analysis. It must also define sufficient functionality to enable the application to satisfy its requirements. The structure defines interfaces for the components.
5. *Application design*. This phase details the internals of each application component. The component designs must be compatible with the defined architectural interfaces and should be sufficiently detailed to support immediate implementation.
6. *Program implementation*. This phase translates information and concepts from domain analysis, architecture, and application design into a machine-executable form. The developers then run the executable machine code, using a variety of inputs to determine the correctness of their individual code.
7. *Integration*. This phase verifies program implementation in the context of domain analysis, architecture, and application design by clearly identifying interface ambiguities and implementation errors.
8. *System test*. This phase verifies that the program as assembled meets the requirements. If all the requirements are satisfied, the application is ready to place in service.

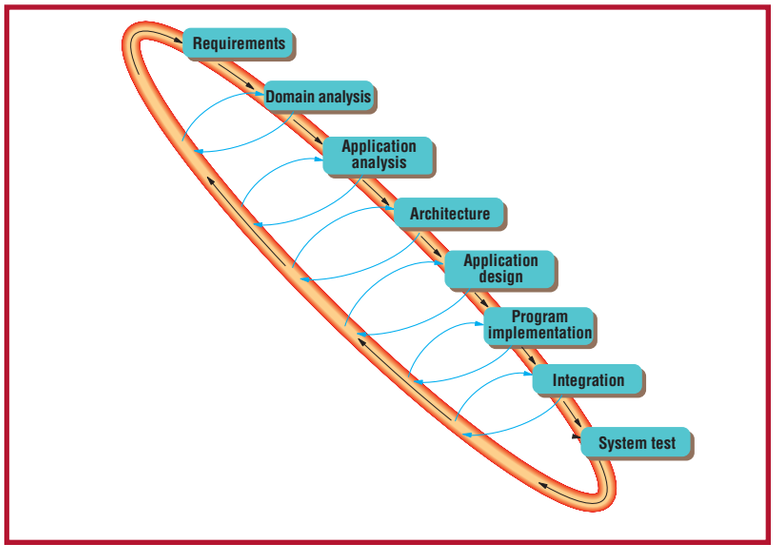

Figure 2. The iterative process model.

Figure 1 illustrates the temporal relationships between the process phases. The size of the boxes represents the relative amounts of time for each phase, and the concurrent threads show the relative starting times. This figure does not show the iterative nature of our process, which is shown in Figure 2.

Because of space limitations, we have only summarized the phases. To illustrate the process's characteristics and our process definition style, Figure 3 shows a complete outline of the domain analysis phase.

Planning a Project's Process

The first part of planning a specific project's process is laying out the increments.

Phase 2. **Domain Analysis**

2.1. **Description**: To capture the concepts and relationships within the bodies of knowledge that underlie the basic problem to be solved by the application.

2.2. **Responsibility**

- The conceptualizer helps determine the domain boundaries and serves as a domain expert.
- The user/domain expert provides concepts and connections between the concepts.
- The manager reviews this phase's output to produce schedules, resource allocations, and the software development plan.
- The architect gains an understanding of the domain to construct the appropriate architecture.
- The tester ensures that the domain analysis by-products are testable.

2.3. **Input**

2.3.1. Actuals: An understanding of what the system should be. The collective domain knowledge of those participating in the analysis.

2.3.2. By-products: A business plan describing the concept or idea and the requirements.

2.4. **Entry criteria:** Agreement has been reached that the requirements are sufficiently scoped to begin this phase.

2.5. **Activities**

2.5.1. Domain modeling using UML models.

2.5.2. Guided Inspection of the domain model.

2.6. **Output**

2.6.1. Actuals: An understanding of specific domains related to the problem being solved.

2.6.2. By-products: A UML model capturing the concepts and relationships.

2.7. **Exit criteria**

2.7.1. Go to the next phase if the Guided Inspection's results show the domain model is complete, correct, and consistent.

2.7.2. Iterate back to the requirements phase if the requirements are sufficiently incomplete that the domain analysis raises significant questions about scope.

2.8. **Metrics:** Quality measures applied to the UML models.

Figure 3. A detailed outline of the domain analysis phase.

Each increment is the complete development of an identified piece of system functionality. The initial increment is usually some level of architectural prototype and the infrastructure needed for much of the remainder of the system. In some cases, it is an opportunity for the team to investigate the new technologies to be used in the project. The next increments are the system's main functionality, followed finally by increments that add the system's unusual, seldom-used features. Increments are usually planned by reasonably partitioning the use cases or parts of use cases into sets so that the team members assigned to an increment can complete the work within one to two months. The smaller the team, the smaller each increment should be.

Each increment is constructed in a succession of iterations. Each iteration should move the functionality closer to its final maturity. Figure 2 illustrates the flow between the process phases. The team determines in which previous phase to begin an iteration, on the basis of the reason for halting forward progress and the entry criteria for previous phases. We usually handle the first and last iterations a bit differently from the others. The first is the "ramp-up" iteration, in which the team explores the work to be done in the increment. The final iteration is a "cooldown" iteration, which finalizes work for integration with previous increments.

As the team progresses through iterations, from ramp-up to cooldown, the relative amount of time spent in each development phase shifts. In the initial iterations, analysis receives the most attention, but we design and code enough to try out our understanding of the problem. In later iterations, with a reasonable understanding of the problem, we emphasize design, but only after reviewing the lessons learned in the previous iteration about the problem. The final iterations focus on developing code, modifying both the analysis and design models on the basis of concrete feedback from writing the code.

Evaluating a Project's Process

Small projects need process evaluation techniques that are easily applied; where possible, the techniques should contribute directly to producing quality actuals. To achieve this, each process phase specifies an assessment activity and a set of metrics.

Assessment Activities

An assessment activity's primary goal is to evaluate a particular phase's exit criteria. Also, assessment activities often guide the team to explore missed relationships.

For example, the architecture phase uses change cases[6] to guide the team to consider future requirements and industry trends and their impact on the architecture. Can the architecture accommodate those changes? Would modifying the architecture make it more amenable to those future changes? Guided Inspection modifies a typical design review process by providing a technique to select test cases. Using the change cases as the basis for test cases assesses the architecture's ability to accommodate certain types of changes.

Assessment activities produce as by-products information such as the quantity and types of defects found. The development team can use this information to evaluate the effectiveness of the phase's development activities. On the basis of this information, the team might revise the phase to require a different set of activities.

For example, in the domain analysis and

application design phases, Guided Inspection provides an objective testing technique that discovers defects in the design models. Over time, if these assessments continue to find the same types of defects in these models, the team should modify the process that produced the models.

Both having an assessment activity for each phase and the possibility of failing the exit criteria point toward the need to iterate on a variety of levels. We consider each development phase to be a self-contained process consisting of a set of activities. The assessment for a given activity or phase should provide information that helps determine whether to iterate back to a previous phase, iterate over the activities within the current phase, or move to the next phase.

Metrics

The metrics specified in a phase's description measure the effectiveness and quality of that phase's activities. Some of the metrics are product metrics; they measure attributes of the models. In some cases they can be computed automatically if the models were created using CASE tools. The data gathered from these measures should be used to improve the system's quality through either feedback in the current phase or feedback to a previous phase.

Some of the metrics are process metrics; they include measures such as defect detection rates, for which the basic data is collected during the development activities. The information obtained from analyzing the process metrics data crosses phase boundaries. The knowledge gained from these measures guides process modifications, including changing exit criteria (for example, raising the bar if quality is poor) or specifying additional activities needed to make a phase more effective. Once the project is delivering quality software on schedule and within budget, that indicates an effective process.

Outside Influences

A project that is behind schedule does not necessarily indicate process failure. Several problems that are external to the process can hinder the project:

- Are all personnel devoting their assigned percentage of work on the project? Most organizations assign personnel to several projects at the same time. On one project recently, one person was working approximately 5% of his time for that project, as opposed to the 80% he was assigned. In a small project, this decrease in effort, even if it is to help another project in the organization, will cause deviations from the schedule.
- Are the estimates based on project actuals being respected? Often, market conditions will be used to justify modifying the estimates. If the project personnel have been careful about collecting data and estimating accurately, market adjustments are probably doomed to failure.
- Are estimates based on a process different from the one being used? For small projects and immature organizations, this can be a particular problem. Changes to processes should be as evolutionary as possible so that you can anticipate how the changes affect estimates.

For a process to be successful, it must be followed. We have tried to ensure that our process is followed by making it useful. This means that the activities defined in a particular project's process clearly relate to achieving the project's goals. Small projects do not have spare cycles to waste on activities that satisfy either the egos of process writers or the misguided attempts of managers to increase organizational maturity. As more of the process monitoring and evaluation can be automated, this will further free team members to focus on the project's goal of producing a quality software system.

About the Authors

Melissa L. Russ is a software consultant and mentor in addition to being a project manager for Korson-McGregor, A Software Technology Company. During consulting work with some larger computer companies, she helped develop, implement, and improve software processes; taught basic object-oriented development techniques; and mentored staff concerning general software development. She has also worked with smaller companies providing consulting services, from domain analysis for project definition to use-case analysis for producing test plans. She is a highly rated instructor and often makes presentations at conferences regarding software engineering and object-oriented technology. She has a master's in computer science from Clemson University, where she has also been a PhD student and has taught computer courses. Contact her at Korson-McGregor, PO Box 3104, Collegedale, TN 37315; melissa.russ@korson-mcgregor.com; www.korson-mcgregor.com/~melissa.

John D. McGregor is a senior partner in Korson-McGregor, A Software Technology Company, specializing in object-oriented techniques and is an associate professor of computer science at Clemson University. He is also a visiting scientist at the Software Engineering Institute. He has developed testing techniques for object-oriented software and custom testing processes for a variety of companies. He is coauthor of *Object-Oriented Software Development: Engineering Software for Reuse* (Int'l Thompson Publishers, 1992) and of *A Practical Guide to Testing Object-Oriented Software*, to be published by Addison-Wesley. He also writes a monthly column on testing objects in the *Journal of Object-Oriented Programming*. Contact him at Korson-McGregor, PO Box 263, Clemson, SC 29633; john.mcgregor@korson-mcgregor.com; www.korson-mcgregor.com/~johnmc.

References

1. G. Booch, *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, Reading, Mass., 1995.
2. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, Vol. 21, No. 5, May 1988, pp. 61–72.
3. J.D. McGregor, "The Fifty Foot Look at Analysis and Design Models," *J. Object-Oriented Programming*, Vol. 11, No. 4, July/Aug. 1998, pp. 10–15.
4. W. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Reading, Mass., 1995.
5. W. Royce, *Software Project Management: A Unified Framework*, Addison-Wesley, Reading, Mass., 1998.
6. E.F. Ecklund Jr. and L.M.L. Delcambre, "Change Cases: Use Cases that Identify Future Requirements," *Proc. OOPSLA '96*, ACM Press, New York, 1996, pp. 342–358.