Feature

The following article is the introductory chapter from The Unified Development Process *by Ivar Jacobson, Grady Booch, and James Rumbaugh. These "three amigos" have been influential in creating a standardized object-oriented analysis and design notation, UML. This offering describes the three amigos' vision of a standardized software development process.*

—Steve McConnell, editor-in-chief

The Unified Process

Ivar Jacobson, Grady Booch, and James Rumbaugh, Rational Software

Today, the trend in software is toward bigger, more complex systems. This is due in part to the fact that computers become more powerful every year, leading users to expect more from them. This trend has also been influenced by the expanding use of the Internet for exchanging all kinds of information—from plain text to formatted text to pictures to diagrams to multimedia. Our appetite for ever-more sophisticated software grows as we learn from one product release to the next how the product could be improved. We want software that is better adapted to our needs, but that, in turn, merely makes the software more complex. In short, we want more.

We also want it faster. Time to market is another important driver.

Getting there, however, is difficult. Our demands for powerful, complex software have not been matched with how software is developed. Today, most people develop software using the same methods that were used as long as 25 years ago. This is a problem. Unless we update our methods, we will not be able to accomplish our goal of developing the complex software needed today.

The software problem boils down to the difficulty developers face in pulling together the many strands of a large software undertaking. The software development community needs a controlled way of working. It needs a process that integrates the many facets of software development. It needs a common approach, a process that

- provides guidance to the order of a team's activities,
- directs the tasks of individual developers and the team as a whole,
- specifies what artifacts should be developed, and
- offers criteria for monitoring and measuring a project's products and activities.

The presence of a well-defined and well-managed process is a key discriminator between hyperproductive projects and unsuccessful ones. The Unified Software Development Process—the outcome of more than 30 years of experience—is a solution to the software problem.

The Unified Process in a Nutshell

First and foremost the Unified Process is a software development process. A software development process is the set of activities needed to transform a user's requirements into a software system (see Figure 1). However, the Unified Process is more than a single process; it is a generic process framework that can be specialized for a very large class of software systems, for different application areas, different types of organizations, different competence levels, and different project sizes.

The Unified Process is component-based, which means that the software system being built is made up of software components interconnected via well-defined interfaces.

The Unified Process uses the Unified Modeling Language when preparing all blueprints of the software system. In fact, UML is an integral part of the Unified Process—they were developed hand in hand.

However, the real distinguishing aspects of the Unified Process are captured in the three key words—use-case driven, architecture-centric, and iterative and incremental. This is what makes the Unified Process unique.

The Unified Process Is Use-Case Driven

A software system is brought into existence to serve its users. Therefore, to build a successful system we must know what its prospective users want and need.

Figure 1. A software development process.

The term *user* refers not only to human users but to other systems. In this sense, the term user represents someone or something (such as another system outside the proposed system) that interacts with the system being developed. An example of an interaction is a human who uses an automatic teller machine. He or she inserts the plastic card, replies to questions called up by the machine on its viewing screen, and receives a sum of cash. In response to the user's card and answers, the system performs a sequence of actions that provide the user with a result of value, namely the cash withdrawal.

An interaction of this sort is a *use case*. A use case is a piece of functionality in the system that gives a user a result of value. Use cases capture functional requirements. All the use cases together make up the *use-case model*, which describes the complete functionality of the system. This model replaces the traditional functional specification of the system. A functional specification can be said to answer the question, What is the system supposed to do? The use case strategy can be characterized by adding three words to the end of this question: for each user? These three words have a very important implication. They force us to think in terms of value to users and not just in terms of functions that might be good to have.

However, use cases are not just a tool for specifying the requirements of a system. They also drive its design, implementation, and test; that is, they drive the development process. Based on the use-case model, developers create a series of design and implementation models that realize the use cases. The developers review each successive model for conformance to the use-case model. The testers test the implementation to ensure that the components of the implementation model correctly implement the use cases. In this way, the use cases not only initiate the development process but bind it together. *Use-case driven* means that the development process follows a flow—it proceeds through a series of workflows that derive from the use cases. Use cases are specified, use cases are designed, and at

the end use cases are the source from which the testers construct the test cases.

While it is true that use cases drive the process, they are not selected in isolation. They are developed in tandem with the system architecture. That is, the use cases drive the system architecture and the system architecture influences the selection of the use cases. Therefore, both the system architecture and the use cases mature as the life cycle continues.

The architecture must be designed to allow the system to evolve, not only through its initial development but through future generations.

THE UNIFIED PROCESS IS ARCHITECTURE-CENTRIC

The role of software architecture is similar in nature to the role architecture plays in building construction. The building is looked at from various viewpoints: structure, services, heat conduction, plumbing, electricity, and so on. This allows a builder to see a complete picture before construction begins. Similarly, architecture in a software system is described as different views of the system being built.

The software architecture concept embodies the most significant static and dynamic aspects of the system. The architecture grows out of the needs of the enterprise, as sensed by users and other stakeholders, and as reflected in the use cases. However, it is also influenced by many other factors: the platform the software is to run on (such as computer architecture, operating system, database management system, and protocols for network communication), the reusable building blocks available (such as a framework for graphical user interfaces), deployment considerations, legacy systems, and nonfunctional requirements (such as performance and reliability). Architecture is a view of the whole design with the important characteristics made more visible by leaving details aside. Since what is significant depends in part on judgment, which, in turn, comes with experience, the value of the architecture depends on the people assigned to the task. However, process helps the architect to focus on the right goals, such as understandability, resilience to future changes, and reuse.

How are use cases and architecture related? Every product has both function and form. One or the other is not enough. These two forces must be balanced to get a successful product. In this case function corresponds to use cases and form to architecture. There needs to be interplay between use cases and architecture. It is a "chicken and egg" problem. On the one hand, the use cases must, when realized, fit in the architecture. On the other hand, the architecture must allow room for realizations of all the required use cases, now and in the future. In reality, both the architecture and the use cases must evolve in parallel.

Thus the architects cast the system in a form. It is that form, the architecture, that must be designed so as to allow the system to evolve, not only through its initial development but through future generations. To find such a form, the architects must work from a general understanding of the key functions, that is, the key use cases, of the system. These key use cases may amount to only 5 percent to 10 percent of all the use cases, but they are the significant ones, the ones that constitute the core system functions. Here is the process in simplified terms:

- The architect creates a rough outline of the architecture, starting with the part of the architecture that is not specific to the use cases (such as platform). Although this part of the architecture is use-case independent, the architect must have a general understanding of the use cases prior to the creation of the architectural outline.
- Next, the architect works with a subset of the identified use cases, the ones that represent the key functions of the system under development. Each selected use case is specified in detail and realized in terms of subsystems, classes, and components.
- As the use cases are specified and they mature, more of the architecture is discovered. This, in turn, leads to the maturation of more use cases.

This process continues until the architecture is deemed stable.

THE UNIFIED PROCESS IS ITERATIVE AND INCREMENTAL

Developing a commercial software product is a large undertaking that may continue over several months to possibly a year or more. It is practical to divide the work into smaller slices or mini-projects.

Each mini-project is an *iteration* that results in an *increment*. Iterations refer to steps in the workflow, and increments, to growth in the product. To be most effective, the iterations must be controlled; that is they must be selected and carried out in a planned way. This is why they are mini-projects.

Developers base the selection of what is to be implemented in an iteration upon two factors. First, the iteration deals with a group of use cases that together extend the usability of the product as developed so far. Second, the iteration deals with the most important risks. Successive iterations build on the development artifacts from the state at which they were left at the end of the previous iteration. It is a mini-project, so from the use cases it continues through the consequent development work—analysis, design, implementation, and test—that realizes in the form of executable code the use cases being developed in the iteration. Of course, an increment is not necessarily additive. Especially in the early phases of the life cycle, developers may be replacing a superficial design with a more detailed or sophisticated one. In later phases increments are typically additive.

In every iteration, the developers identify and specify the relevant use cases, create a design using the chosen architecture as a guide, implement the design in components, and verify that the components satisfy the use cases. If an iteration meets its goals—and it usually does—development proceeds with the next iteration. When an iteration does not meet its goals, the developers must revisit their previous decisions and try a new approach.

To achieve the greatest economy in development, a project team will try to select only the iterations required to reach the project goal. It will try to sequence the iterations in a logical order. A successful project will proceed along a straight course with only small deviations from the course the developers initially planned. Of course, to the extent that unforeseen problems add iterations or alter the sequence of iterations, the development process will take more effort and time. Minimizing unforeseen problems is one of the goals of risk reduction.

There are many benefits to a controlled iterative process:

♦ Controlled iteration reduces the cost risk to the expenditures on a single increment. If the developers need to repeat the iteration, the organization loses only the misdirected effort of one iteration, not the value of the entire product.

♦ Controlled iteration reduces the risk of not getting the product to market on the planned schedule. By identifying risks early in development, the time spent resolving them occurs early in the schedule when people are less rushed than they are late in the schedule. In the "traditional" approach, where difficult problems are first revealed by system test, the time required to resolve them usually exceeds the time remaining in the schedule and nearly always forces a delay of delivery.

♦ Controlled iteration speeds up the tempo of the whole development effort because developers work more efficiently toward results in clear, short focus rather than in a long, ever-sliding schedule.

♦ Controlled iteration acknowledges a reality often ignored—that user needs and the corresponding requirements cannot be fully defined up front. They are typically refined in successive iterations. This mode of operation makes it easier to adapt to changing requirements.

These concepts—use-case driven, architecture-centric, and iterative and incremental development—are equally important. Architecture provides the structure in which to guide the work in the iterations, whereas use cases define the goals and drive the work of each iteration. Removing one of the three key ideas would severely reduce the value of the Unified Process. It is like a three-legged stool. Without one of its legs, the stool will fall over.

By identifying risks early in development, time spent resolving occurs early when people are less rushed.

Now that we have introduced the three key concepts, it is time to take a look at the whole process, its life cycle, artifacts, workflows, phases, and iterations.

The Life of the Unified Process

The Unified Process repeats over a series of cycles making up the life of a system. Each cycle concludes with a product release to customers.

Each cycle consists of four phases: inception, elaboration, construction, and transition. Each phase is further subdivided into iterations, as discussed earlier. See Figure 2 (on the next page).

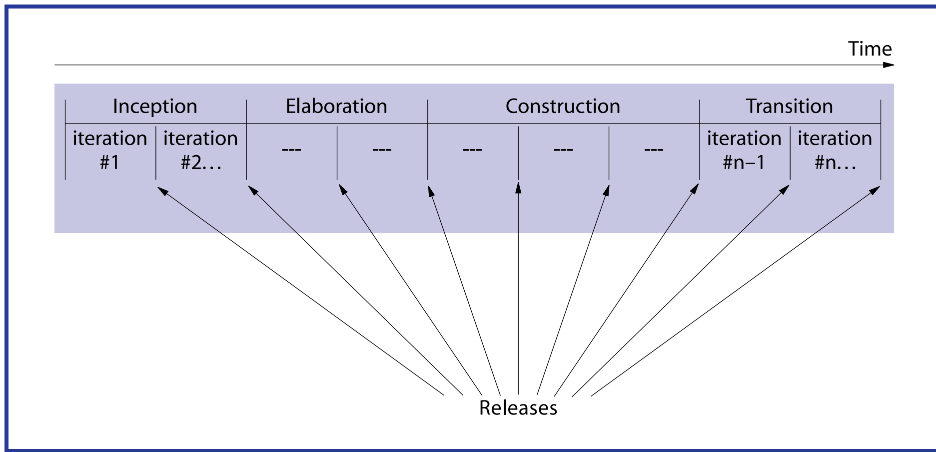

Figure 2. A cycle with its phases and its iterations.

The product

Each cycle results in a new release of the system, and each release is a product ready for delivery. It consists of a body of source code embodied in components that can be compiled and executed, plus manuals and associated deliverables. However, the finished product also has to accommodate the needs, not just of the users, but of all the stakeholders, that is, all the people who will work with the product. The software product ought to be more than the machine code that executes.

The finished product includes the requirements, use cases, nonfunctional requirements, and test cases. It includes the architecture and the visual models—artifacts modeled by the Unified Modeling Language. In fact, it includes all the elements we have been talking about in this chapter, because it is these things that enable the stakeholders—customers, users, analysts, designers, implementers, testers, and management—to specify, design, implement, test, and use a system. Moreover, it is these things that enable the stakeholders to use and modify the system from generation to generation.

Even if executable components are the most important artifacts from the users' perspective, they alone are not enough. This is because the environment mutates. Operating systems, database systems, and the underlying machines advance. As the mission becomes better understood, the requirements themselves may change. In fact, it is one of the constants of software development that the requirements change. Eventually developers must undertake a new cycle, and managers must finance it. To carry out the next cycle efficiently, the developers need all the representations of the software product (Figure 3):

- ♦ A use-case model with all the use cases and their relationships to users.
- ♦ An analysis model, which has two purposes: to refine the use cases in more detail and to make an initial allocation of the behavior of the system to a set of objects that provides the behavior.
- ♦ A design model that defines (a) the static structure of the system as subsystems, classes, and interfaces and (b) the use cases realized as collaborations among the subsystems, classes, and interfaces.
- ♦ An implementation model, which includes components (representing source code) and the mapping of the classes to components.
- ♦ A deployment model, which defines the physical nodes of computers and the mapping of the components to those nodes.
- ♦ A test model, which describes the test cases that verify the use cases.
- ♦ And, of course, a representation of the architecture.

The system may also have a domain model or a business model that describes the business context of the system.

All these models are related. Together, they represent the system as a whole. Elements in one model have trace dependencies backwards and forwards with the help of links to other models. For instance, a use case (in the use-case model) can be traced to a use-case realization (in the design model) to a test case (in the test model). Traceability facilitates understanding and change.

Phases within a cycle

Each cycle takes place over time. This time, in turn, is divided into four phases, as shown in Figure 4. Through a sequence of models, stakeholders visualize what goes on in these phases. Within each phase managers or developers may break the work down still further—into iterations and the ensuing increments. Each phase terminates in a *milestone*. We define each milestone by the availability of a set of artifacts; that is, certain models or documents have been brought to a prescribed state.

The milestones serve many purposes. The most critical is that managers have to make certain crucial decisions before work can proceed to the next phase. Milestones also enable management, as well as the developers themselves, to monitor the progress of the work as it passes these four key points. Finally, by keeping track of the time and effort spent on each

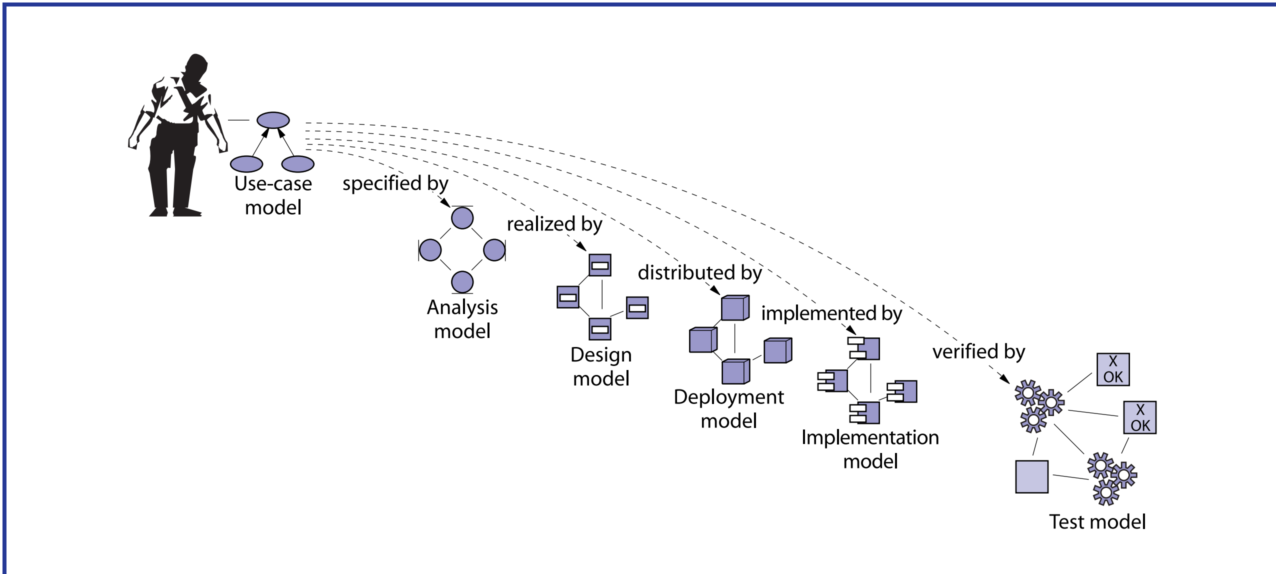

Figure 3. There are dependencies between many of the models of the Unified Process. As an example, the dependencies between the use-case model and the other models are indicated.

phase, we develop a body of data. This data is useful in estimating time and staff requirements for other projects, projecting staff needs over project time, and controlling progress against these projections.

Figure 4 lists the workflows—requirements, analysis, design, implementation, and test—in the left-hand column. The curves approximate (they should not be taken too literally) the extent to which the workflows are carried out in each phase. Recall that each phase usually is subdivided into iterations, or mini-projects. A typical iteration goes through all the five workflows as shown for an iteration in the elaboration phase in Figure 4.

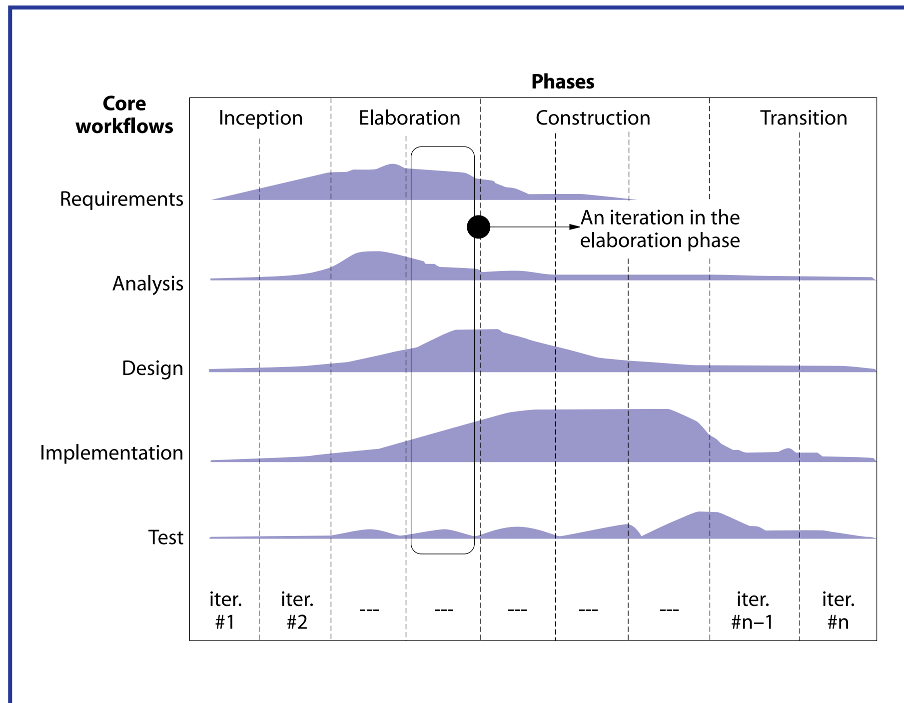

Figure 4. The five workflows—requirements, analysis, design, implementation, and test—take place over the four phases: inception, elaboration, construction, and transition.

During the inception phase, a good idea is developed into a vision of the end product and the business case for the product is presented. Essentially, this phase answers the following questions:

♦ What is the system primarily going to do for each of its major users?

♦ What could an architecture for that system look like?

♦ What is the plan and what will it cost to develop the product?

A simplified use-case model that contains the most critical use cases answers the first question. At this stage the architecture is tentative. It is typically just an outline containing the most crucial subsystems. In this phase, the most important risks are identified and prioritized, the elaboration phase is planned in detail, and the whole project is roughly estimated.

During the elaboration phase, most of the product's use cases are specified in detail and the system

architecture is designed. The relationship between the architecture of a system and the system itself is paramount. A simple way to put it is that the architecture is analogous to a skeleton covered with skin but with very little muscle, the software, between the bone and the skin—just enough muscle to allow the skeleton to make basic movements. The system is the whole body with skeleton, skin, and muscle.

Therefore, the architecture is expressed as views of all the models of the system, which together represent the whole system. This implies that there are architectural views of the use-case model, the analysis model, the design model, the implementation model, and the deployment model. The view of the implementation model includes components to prove that the architecture is executable. During this phase of development the most critical use cases identified during the elaboration phase are realized. The result of this phase is an architecture baseline.

At the end of the elaboration phase, the project manager is in a position to plan the activities and estimate the resources required to complete the project. Here the key question is, Are the use cases, architecture, and plans stable enough, and are the risks under sufficient control to be able to commit to the whole development work in a contract?

During the construction phase the product is built—muscle, the completed software, is added to the skeleton, the architecture. In this phase, the architecture baseline grows to become the full-fledged system. The vision evolves into a product ready for transfer to the user community. During this phase of development, the bulk of the required resources is expended. The architecture of the system is stable, however, because the developers may discover better ways of structuring the system, they may suggest minor architectural changes to the architects. At the end of this phase, the product contains all the use cases that management and the customer agreed to develop for this release. It may not be entirely free of defects, however. More defects will be discovered and fixed during the transition phase. The milestone question is, Does the product meet users' needs sufficiently for some customers to take early delivery?

The transition phase covers the period during which the product moves into beta release. In the beta release a small number of experienced users try the product and report defects and deficiencies. Developers then correct the reported problems and incorporate some of the suggested improvements into a general release for the larger user community. The transition phase involves activities such as manufacturing, training customer personnel, providing help-line assistance, and correcting defects found after delivery. The maintenance team often divides these defects into two categories: those with sufficient effect on operations to justify an immediate delta release and those that can be corrected in the next regular release.

The Unified Process is component based. It uses the new visual modeling standard, UML, and relies on three key ideas—use cases, architecture, and iterative and incremental development. To make these ideas work, a multifaceted process is required, one that takes into consideration cycles, phases, workflows, risk mitigation, quality control, project management, and configuration control. The Unified Process has established a framework that integrates all those different facets. This framework also works as an umbrella under which tool vendors and developers can build tools to support the automation of the process, to support the individual workflows, to build all the different models, and to integrate the work across the life cycle and across all models. ❖

Adapted from Chapter 1 of The Unified Software Development Process, *by Ivar Jacobson, Grady Booch, and James Rumbaugh, ISBN: 0-201-57169-2. Reprinted by permission of Addison Wesley Longman, One Jacob Way, Reading, MA 01867.* *All rights reserved.* *Contact Addison Wesley Longman at (781) 944-3700, http://www.awl.com/cseng/.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.