

Chapter 2

The Four *Ps*: People, Project, Product, and Process in Software Development

The end result of a software **project** (Appendix C) is a product that is shaped by many different types of people as it is developed. Guiding the efforts of the people involved in the project is a software development process, a template that explains the steps needed to complete the project. Typically, the process is automated by a tool or set of tools. See Figure 2.1.

Throughout this book we will use the terms *people*, *project*, *product*, **process** (Appendix C), and *tools*, which we define as follows:

- *People*: The architects, developers, testers, and their supporting management, plus users, customers, and other stakeholders are the prime movers in a software project. People are actual human beings, as opposed to the abstract construct of *workers*, which we will introduce later on.
- *Project*: The organizational element through which software development is managed. The outcome of a project is a released product.
- *Product*: Artifacts that are created during the life of the project, such as **models** (Appendix A), source code, executables, and documentation.
- *Process*: A software engineering process is a definition of the complete set of activities needed to transform users' requirements into a product. A process is a template for creating projects.
- *Tools*: Software that is used to automate the activities defined in the process.

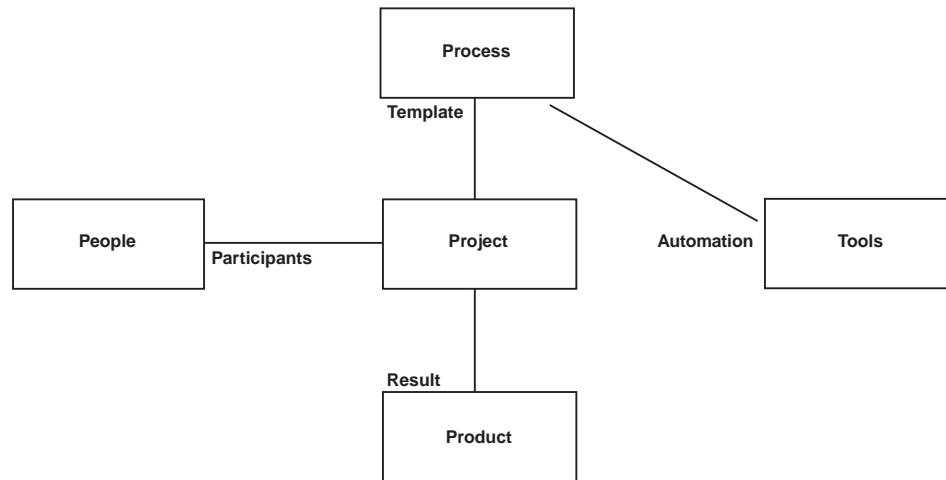


FIGURE 2.1 The 4 Ps in software development.

2.1 People Are Crucial

People are involved in the development of a software product throughout its entire life cycle. They finance the product, schedule it, develop it, manage it, test it, use it, and benefit from it. Therefore, the process that guides this development must be people oriented, that is, one that works well for the people using it.

2.1.1 Development Processes Affect People

The way in which a software project is organized and managed profoundly affects the people involved in the project. Concepts such as feasibility, risk management, team organization, project scheduling, and project understandability all play important roles:

- *Project feasibility:* Most people do not enjoy working on projects that are deemed infeasible—nobody wants to go down with the ship. As we saw in Chapter 1, an iterative approach to development allows the feasibility of a project to be assessed early on. Infeasible projects can then be terminated at an early stage, thus alleviating morale problems.
- *Risk management:* Similarly, when people sense that risks have not been analyzed and reduced, they become uneasy. The exploration of significant risks in the early phases mitigates this problem.
- *Team structure:* People work most effectively in small groups of six to eight members. A process that yields meaningful work for small groups, such as assessing a particular risk, developing a subsystem (Appendix A), or perform-

ing an iteration, provides this opportunity. A good architecture with well-defined interfaces (Appendix A; see also Chapter 9) between subsystems and components (Appendix A; see also Chapter 10) makes such a division of effort possible.

- *Project schedule:* When people believe that a schedule is unrealistic, morale will plummet—people don't like to go to work knowing that no matter how hard they try, they will never be able to produce the expected results. The techniques used in the inception and elaboration phases allow developers to get a good idea of what the end result of the project should be—that is what the released product should do. Since there is a good feeling of what the product should do, a realistic project plan for the effort involved as well as the time needed to accomplish the goals may be created, thus alleviating the “we'll never finish” people problem.
- *Project understandability:* People like to know what they are doing; they want to understand the whole picture. The architecture description provides an overview for everyone involved in a project.
- *Sense of accomplishment:* In an iterative life cycle, people receive frequent feedback, which in turn, provides closure. Frequent feedback and the resulting closure increases the work tempo. A fast work pace combined with frequent closure heightens people's sense of accomplishment.

2.1.2 Roles Will Change

Since the key activities of software development are executed by people, there is a need for a uniform development process that is supported by tools and a Unified Modeling Language (now available in UML) (Appendix C) to enable people to be more effective. Such a process will enable developers to build better software in terms of time-to-market, quality, and cost. It enables them to specify requirements that better meet users' needs. It enables them to select an architecture that allows systems to be built in a cost-effective, timely manner. A good software process has another advantage: It helps us build more complex systems. We noted in Chapter 1 that as the real world becomes more complex, so customers will require more complex software systems. Business processes and the corresponding software will have a longer life. Because changes in the real world will continue to occur throughout these life cycles, software systems will have to be designed in such a way as to enable them to grow over longer periods of time.

To understand and support these more complex business processes and to implement them in software, developers will find themselves working with many other developers. To work effectively in larger and larger teams, a process is needed to provide guidance. This guidance will result in developers “working smarter,” that is, to restrict one's effort to that which adds value to the customer. One step in this direction is use-case modeling, which focuses effort on what the user needs to do. Another step is an architecture that will permit systems to continue to evolve for years to come. A third step is to buy or reuse as much software as possible. That, in turn, can

be accomplished only if there is a consistent way of integrating reusable components with newly developed elements.

In the coming years, most software people will begin to work closer to the mission they support, and they will be able to develop more complex software thanks to an automated process and reusable components. People will be crucial to software development for the foreseeable future. In the end, it is having the right people that makes us succeed. The issue comes down to making them effective and allowing them to do what only humans can—to be creative, to find new opportunities, to use judgment, to communicate with customers and users, and to understand a rapidly changing world.

2.1.3 Turning “Resources” into “Workers”

People fill many different positions in a software development organization. Preparing them for these positions takes education and pin-pointed training followed by careful assignment supported by mentoring and helpful supervision. An organization faces a substantial task when it moves a person from a latent “resource” to a particular position as a “worker.”

We have selected the word **worker** (Appendix C) to stand for the positions to which people may be assigned and which they accept [4]. A *worker type* is a role that an individual may play in software development, such as use-case specifier, architect, component engineer, and integration tester. We do not use the term *role* (instead of *worker*) for primarily two reasons: it has a precise and different meaning in UML, and the concept of a worker needs to be very concrete; we need to think in terms of individual workers as the positions taken by individuals. We also need to use the term *role* to talk about roles of a worker. A worker may play roles in relation to other workers in different workflows. For instance, the worker component engineer may participate in several workflows and in each workflow he plays a particular role.

Each worker (i.e., a worker instance) is responsible for a whole set of activities, such as the activities involved in the design of a subsystem. To work effectively, workers need the information that is required to carry out those activities. They need to understand what their roles are relative to those of other workers. At the same time, if they are to do their work, the tools they employ must be adequate. The tools must not only help workers carry out their own activities but shield them from information that is not relevant. To accomplish these objectives, the Unified Process formally describes the positions—that is, the workers—that people can take in the process.

Figure 2.2 illustrates how individual people may be different workers in a project.

A worker may also be realized as a set of individuals working together. For example, an architect worker may be realized as an architectural board.

Each worker has a set of responsibilities and performs a set of activities in developing software.

When allocating resources to workers in a project, the project manager needs to identify the competencies of individuals and match them with the required competen-

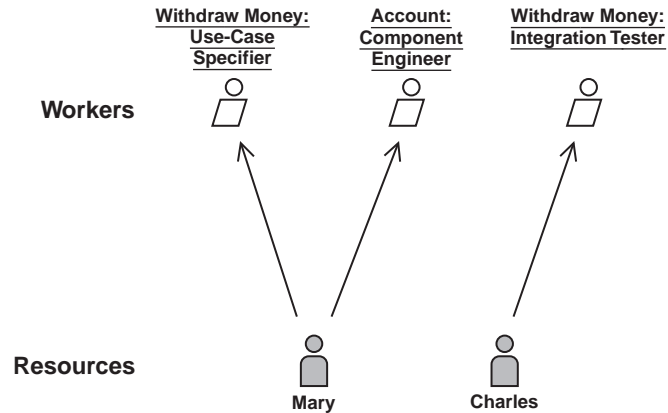


FIGURE 2.2 Workers and resources that realize them.

cies of the workers. This is not an easy task, especially the first time the Unified Process is used. The skills of the resources (i.e., real people) must be matched against the competencies specified by the various workers needed by the project. The competencies needed by some workers may be achieved through training, while the competencies needed by other workers may be gained only through experience. For example, the skills needed to be a use-case specifier may be learned through training, but those of an architect are typically acquired through experience.

One individual may be many workers during the life of a project. For example, Mary may start as a use-case specifier and later become a component engineer.

When allocating resources, the project manager should minimize the handoff of artifacts from one resource to another in a way that makes the flow of the process as seamless as possible. For example, the use-case engineer of the Withdraw Money use case (Appendix A; see also Chapter 7) will acquire a lot of knowledge about the responsibilities of the class Account (Appendix A), so he or she would be a logical choice to be the component engineer of the Account class. The alternative would be to train a new person to take on this work, which could be done, but it would be less efficient because of the loss of information, risk of misunderstandings, and so on.

2.2 Projects Make the Product

A development project results in a new release of a product. The first project in the life cycle (i.e., the first development cycle, sometimes called the “green-field project”) develops and releases the initial system, or product. Successive project cycles extend the life of the system over many releases. See [9] and [10] for more complete presentation of project management.

Throughout its life cycle, a project team has to be concerned with change, iterations, and the organizational pattern within which the project is conducted:

- *A sequence of change:* System development projects result in products, but the course along the way is a series of changes. This fact of project life has to be borne in mind as workers proceed through phases and iterations. Every cycle, every phase, and, yes, every iteration changes the system from one thing to something else. The first development cycle is a special case that changes the system from nothing into something. Each cycle leads to a *release*, and beyond a sequence of cycles, change continues for *generations*.
- *A series of iterations:* Within each phase of a cycle, workers carry out the activities of the phase through a series of iterations. Each iteration implements a set of related use cases or mitigates some risks. In an iteration developers proceed through a series of workflows: requirements, design, implementation, and test. Since each iteration goes through each of these workflows, we can think of an iteration as a *miniproject*.
- *An organizational pattern:* A *project* involves a team of people assigned to accomplish a result within business constraints, that is, time, cost, and quality. The people work as different workers. The idea of “process” is to provide a pattern within which people as workers execute a project. This pattern or template indicates the types of workers the project needs and the artifacts with which it is to work. The process also offers a lot of guidelines, heuristics, and documentation practices that help the assigned people do their job.

2.3 Product Is More Than Code

In the context of the Unified Process, the product developed is a software system. The term *product* here refers not just to the code that is delivered but to the whole system.

2.3.1 What Is a Software System?

Is a software system the machine code, the executables? It is that, of course, but what is machine code? It is a description! It is a description in binary form that can be “read” and “understood” by a computer.

Is a software system the source code? That is, is it a *description* that is written by programmers that can be read and understood by a compiler? Yes, that may be the answer.

We can continue in this manner to ask similar questions about the design of a software system in terms of subsystems, classes, **interaction diagrams** (Appendix A), **statechart diagrams** (Appendix A), and other artifacts. Are they the system? Yes, they are part of it. What about requirements, testing, sales, production, installation, and operation? Are they the system? Yes, they are also part of the system.

A system is all the artifacts that it takes to represent it in machine or human readable form to the machines, the workers, and the stakeholders. The machines are tools, compilers, or target computers. Workers include management, architects, developers, testers, marketers, administrators, and others. Stakeholders are the funding authorities, users, salespeople, project managers, line managers, production people, regula-

tory agencies, and so on. In this book, we will use the term *worker* for these three categories collectively, unless we explicitly note otherwise.

2.3.2 Artifacts

Artifact (Appendix C) is a general term for any kind of information created, produced, changed, or used by workers in developing the system. Some sample artifacts are UML diagrams and their associated text, user-interface sketches and **prototypes** (Appendix C; see also Chapters 7 and 13), components, test plans (see Chapter 11), and test procedures (see Chapter 11).

Basically, there are two kinds of artifacts: engineering artifacts and management artifacts. This book focuses on the engineering artifacts created during the various phases of the process (i.e., requirements, analysis, design, implementation, and test).

However, software development also requires management artifacts. Several management artifacts have a short lifetime—they live only during the life of a project. To this set belong artifacts such as the business case, the development plan (including release and iteration plans), a plan for the allocation of individual people to workers (i.e., to the different positions, or responsibilities, in the project), and laying out the worker activities in the plan. These artifacts are described in text or diagrams, using any kind of visualization needed to specify the commitment made by the project team to the funding stakeholders. Management artifacts also include the specifications of the development environment—process automation software as well as the hardware platform required for developers and as a repository for the engineering artifacts.

2.3.3 A System Has a Collection of Models

The most interesting type of artifact employed in the Unified Process is the model. Every worker needs a unique perspective of the system (see Figure 2.3). When designing the Unified Process, we identified all the workers and every perspective that the workers could possibly need. The collected perspectives of all the workers are structured into larger quanta, that is, models, in such a way that a worker can retrieve any particular perspective from the set of models.

Building a system is thus a process of model building using different models to describe all the different perspectives of the system. Selecting the models for a system is one of the most important decisions the development team makes.

In Chapter 1 we introduced the primary models of the Unified Process (see Figure 2.4).

The Unified Process provides a carefully selected set of models with which to start. This set of models illuminates the system for all the workers, including customers, users, and project managers. It is selected so as to satisfy those workers' need for information.

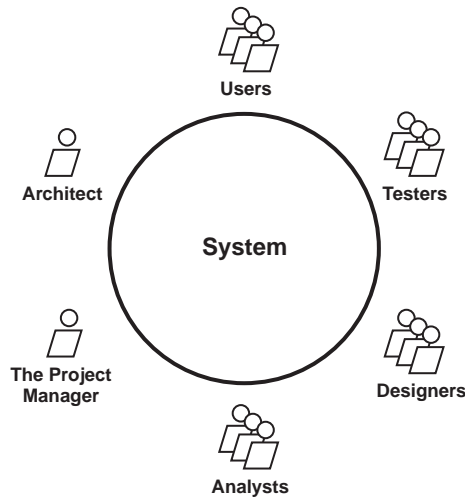


FIGURE 2.3 Workers participating in software development. (Some are singleton workers; others are multitypes and multiobjects.)

2.3.4 What Is a Model?

A model is an abstraction of a system, specifying the modeled system from a certain viewpoint and at a certain level of abstraction [1]. A viewpoint is, for instance, a specification view or a design view of the system.

Models are abstractions of the system that the architects and developers build. For instance, the workers modeling the functional requirements think of the system as having users outside the system and having use cases inside the system. They don't care about what the system looks like from the inside, only what it can do for its users. The workers making up the design think about structural elements such as subsystems and classes; they think in terms of how these elements work in a given context and how they collaborate to provide the use cases. They understand how these abstract things work, and they have a particular interpretation in their minds.

2.3.5 Each Model Is a Self-Contained View of the System

A model is a semantically closed abstraction of the system. It is a self-contained view in the sense that a model user doesn't need other information (e.g., from other models) to interpret it.

The idea of self-containment means that the developers intended there to be only one interpretation of what will happen in a system when an event described by the model is triggered. In addition to the system under consideration, a model also has to describe interactions between the system and its surroundings. Thus, apart from the system being modeled, the model also has to contain elements that describe relevant parts of its environment, that is, its **actors** (Appendix A; see also Chapter 7).



FIGURE 2.4 The primary model set of the Unified Process¹.

Most engineering models are defined by a carefully selected subset of the UML. For example, the use-case model consists of the use cases and the actors. That is basically all a viewer needs to understand it. The design model describes the subsystems and classes of the system and how they interact to realize the use cases. Both the use-case model and the design model describe two different but mutually consistent interpretations of what the system will do given a set of external stimuli from the actors. They are different because they are intended to be used by different workers with different tasks and missions. The use-case model is an outside view of the system, the design model is an inside view. The use-case model captures the uses of the system, whereas the design model represents the building of the system.

2.3.6 Inside a Model

A model always identifies the system being modeled. This system element is then the container of other elements. The **top-level subsystem** (Appendix B) represents the system being built. In the use-case model, the system contains use cases; in the design model, it contains subsystems, interfaces, and classes. It also contains collaborations (Appendix A) that identify all participating subsystems or classes, and it may contain more, such as statechart diagrams or interaction diagrams. In the design model every subsystem can itself be a container of similar constructs. This implies that there is a hierarchy of elements in this model.

2.3.7 Relationships between Models

A system contains all the **relationships** (Appendix A) and constraints between model elements contained in different models [1]. Thus a system is not just the collection of its models but the relationships between them as well.

For instance, every use case in the use-case model has a relationship with a collaboration in the analysis model (and vice versa). Such a relationship is in UML called a trace dependency, or simply a trace (Appendix A). See Figure 2.5, in which traces in only one direction are indicated.

There are also traces between, for instance, collaborations in the design model and collaborations in the analysis model, and between components in the

1. In UML terms, these “packages” represent «business entities» (or «work units») in the Unified Process and not model elements to model a particular system. See also the explanation in the sidebar in Section 7.1.



FIGURE 2.5 Models are tightly linked to one another through traces.

implementation model and subsystems in the design model. Thus we can connect elements in one model to elements in another model using traces.

The fact that the elements in two models are connected does not change what they do inside the models to which they belong. Trace relationships between elements in different models add no semantic information to help understand the related models themselves; they just connect the models. The ability to trace is very important in software development for reasons such as understandability and change propagation.

2.4 Process Directs Projects

The word *process* is an overused term. It is used in many different contexts, such as business process, development process, and software process, with many different meanings. In the context of the Unified Process, we mean the key “business” process in a software development business, that is, an organization that develops and supports software (on designing a software development business, see [2]). In this business there are other processes as well, such as the support process, which interacts with users of the products, and a sales process, which starts with an order and delivers a product. However, our focus in this book is the development process [3].

2.4.1 Process: A Template

In the Unified Process, *process* refers to a concept that works as a template that can be reused by creating instances of it. It is comparable to a class form, which you can use to create objects in the object-orientated paradigm. *Process instance* is a synonym for *project*.

In this book, a *software development process* is a definition of the complete set of activities needed to transform users’ requirements into a consistent set of artifacts that represents a software product and, later, to transform changes in those requirements into a new, consistent set of artifacts.

The word *requirement* is used in a general sense, meaning “needs.” At the outset, these needs are not necessarily understood in their entirety. To capture these requirements, or needs, more completely, we may have to understand the business of the customers and the environment in which their users work more fully.

The value-added result of the process is a consistent set of artifacts, a baseline that represents one application system or a family of such systems that comprise a software product.

A process is a definition of a set of activities, not their execution.

Finally, a process covers not just the first development cycle (the first release) but the most common later cycles. In later releases, an instance of the process takes incremental changes in the requirements and produces incremental changes to the artifact set.

2.4.2 Related Activities Make Up Workflows

The way we describe a process is in terms of workflows, where a workflow is a set of activities. What is the source of these workflows? We don't get them by splitting the process into a number of smaller interacting subprocesses. We don't use traditional flowcharts to describe how we decompose the process into smaller chunks. Those are not efficient ways to devise the workflow structure.

Instead, first we identify the different kinds of workers that participate in the process. Then we identify the artifacts that we need to create during the process for each type of worker. This identification, of course, is not something you can do in a blink. The Unified Process relies on a lot of experience in finding the feasible set of artifacts and workers. Once we have identified this set, we can describe how the process flows through the different workers and how they create, produce, and use each other's artifacts. In Figure 2.6 we show an **activity diagram** (Appendix A) that describes the workflow in use-case modeling. Note the “**swim lanes**” (Appendix A)—there is one for each worker—how the work flows from one worker to another, and how activities (represented by cogwheels) are performed in this flow by the workers.

Now we can easily find activities that these workers need to execute when they are activated. These worker-activities are meaningful work for one person acting as a worker. Moreover, from these descriptions we can immediately see if any individual worker needs to be involved more than once in the workflow.

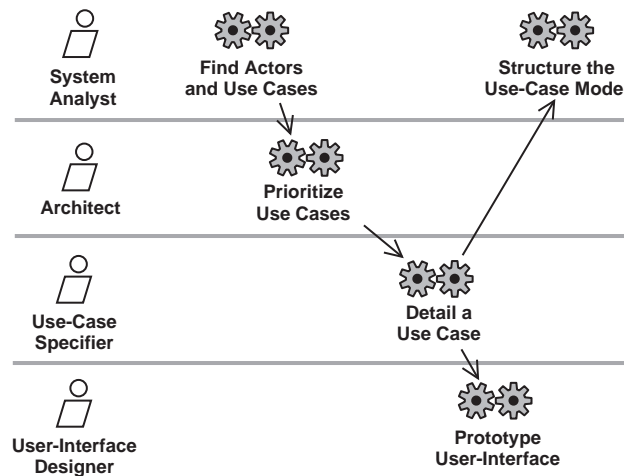


FIGURE 2.6 A workflow with workers and activities in “swim lanes.”

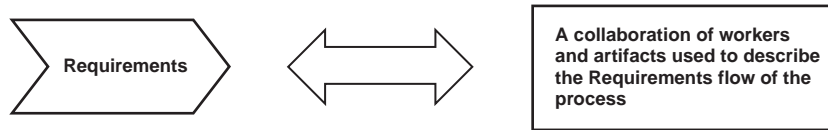


FIGURE 2.7 The “fish” notation is a shorthand for a workflow.

In other words, we describe the whole process in pieces called **workflows** (Appendix C). In UML terms, a workflow is a **stereotype** (Appendix A) of collaboration, in which workers and artifacts are participants. Thus the workers and artifacts that participate in one workflow may (and usually do) participate in other workflows as well. We will use the notation given in Figure 2.7 for workflows.

An example of a workflow is the requirements workflow. It includes the following workers: system analyst, architect, use-case specifier, and user-interface designer. It includes these artifacts: use-case model, use cases, and others. Other examples of workers are component engineers and integration testers. Other examples of artifacts are **use-case realizations** (Appendix B; see also Chapters 8 and 9), classes, subsystems, and interfaces.

2.4.3 Specializing Process

No single software development process can be applied everywhere! Processes vary because they exist in different contexts, develop different types of systems, and meet different kinds of business constraints (e.g., schedule, cost, quality, and reliability). Consequently, a real-world software development process must be adaptable and configurable to meet the actual needs of a specific project and/or organization. The Unified Process is designed to be specialized (on designing a process, see [6]). It is a generic process, that is, a process framework. Every organization that uses the Unified Process will eventually specialize it so that it fits its situation (i.e., its kind of application, its platform, etc.) (on specializing a process, see [8]).

The Unified Process may be specialized to fit different application and organizational needs. At the same time, it is desirable that within an organization, at least, the process be fairly consistent. This consistency will allow components to be used interchangeably, people and managers to transfer between projects readily, and accomplishment metrics to be comparable.

The main factors that influence how the process will differ are

- *Organizational factors*: Organizational structure, organizational culture, project organization and management, competence and skills available, previous experience, and existing software systems.
- *Domain factors*: Application domain, business process to support, user community, and offerings available from competitors.
- *Life cycle factors*: Time to market, expected life span of the software, the technology and the people expertise in developing the software, and planned future releases.

- *Technical factors:* Programming language, development tools, database, frameworks and underlying “standard” architectures, communication, and distribution.

These are the causes. What effect will they have? Well, you may decide to remove workers and artifacts from the Unified Process to better fit less mature development organizations. It may also happen that you will extend the process with new—not yet specified—workers or artifacts because these extensions would make the process more efficient for your project. You may also change the way you think a particular artifact should be described; you might impose a different structure on its description. Our experience is that people in the first projects pretty much use what the Unified Process suggests. As time goes by and they gain more experience, they develop their own minor extensions.

What is it in the design of the Unified Process that permits it to be specialized [6]? The answer is simple but not easy to understand at first. Objectory is itself designed using what are, in effect, objects: use cases, collaborations, and classes. The classes here, of course, are not software, but business objects, that is, workers and artifacts. They can be specialized or exchanged with others without changing the design of the process. In later chapters when we describe the workflows, you will see that we use objects to describe them. Those objects are workers and artifacts.

2.4.4 Merits of Process

A common process within and across development teams provides many benefits:

- Everyone on the development team can understand what he or she has to do to develop the product.
- Developers can better understand what other developers are doing—at earlier or later stages of the same project, in similar projects in the same enterprise, at different geographic locations, and even in projects in other companies.
- Supervisors and managers, even those who cannot read code, can, thanks to architectural drawings, understand what developers are doing.
- Developers, supervisors, and managers can transfer between projects or divisions without having to learn a new process.
- Training can be standardized within a company. Training can be obtained from colleges and shortcourses.
- The course of software development is repeatable, meaning that it can be scheduled and cost estimated with sufficient accuracy to meet expectations.

Despite these advantages of a common process, some still insist that a common process does not solve the “really hard problems.” To that we answer simply: “of course not.” People still solve problems. But a good process helps people to excel as a team. Compare this to the organization of a military operation. Waging battle always boils down to individuals that do things, but the outcome is also decided by the effectiveness of their organization.

2.5 Tools Are Integral to Process

Tools support modern software development processes. Today, it is unthinkable to develop software without using a tool-supported process. The process and the tools come as a suite: the tools are integral to the process [5], [7].

2.5.1 Tools Impact Process

Process is strongly influenced by tool support. Tools are good at automating repetitive tasks, keeping things structured, managing large amounts of information, and guiding you along a particular development path.

With little tool support, a process would have to rely on a lot of manual work and would therefore be less formal. In practice, most of the formal work has to be postponed to the implementation activities. Without tool support that automates consistency across the life cycle, it would be hard to keep models and implementation up-to-date. Iterative and incremental development would be more difficult. Either they would end up inconsistent, or they would require a lot of manual work to update documents in order to maintain consistency. The latter would decrease the productivity of the team significantly. The team would have to do all the consistency checks manually. That is very hard, if not impossible, so there would be numerous flaws in the developed artifacts. And doing it that way would require more lead time.

Tools are developed to automate activities, fully or partially, to increase productivity and quality, and to shorten lead time. As we introduce tool support, we get a different, more formal process. We can introduce new activities that would be impractical to carry out without tools. We can work more precisely during the whole life cycle: We can use a formal modeling language like UML to ensure that each model is consistent within itself and with other models. We can use one model and from it generate parts of another model (e.g., design to implementation and vice versa).

2.5.2 Process Drives Tools

Process, whether explicitly or implicitly defined, specifies the tool functionality, that is, the use cases of the tools. The fact of “process” is, of course, the only reason we need any tools. The tools are there to automate as much of the process as possible.

The ability to automate a process depends on having a clear picture of which use cases each worker needs and which artifacts he or she needs to manage. An automated process provides an efficient means of allowing the whole set of workers to work concurrently, and it provides a means of checking consistency over all the artifacts.

The tools that implement an automated process should be *easy to use*. To make them highly usable means that tool developers need to give thoughtful consideration to the way in which software development is carried out. For instance, how will a worker approach a certain task? How will he become aware of what the tool can help him with? What tasks will be recurring and, hence, worth automating? What tasks will be rare and perhaps not worth embodying in a tool? How can a tool guide a worker into spending time on important tasks that only he can do, leaving repetitive

tasks that the tool can do better to the tool? To answer questions like these, the tool must be easy for workers to understand and use. Moreover, to be worth the time it takes to learn, it must provide a substantial productivity boost.

There are special reasons for the ease-of-use goal. Workers should be able to try out different alternatives, and they should easily be able to massage the candidate designs for each alternative. They should be able to select one approach and try it out. If it turns out to be infeasible, they should be able to move easily from that approach to another one. Tools should enable workers to reuse as much as possible; they should not have to start all over again for each approach tried. In sum, it is essential that tools should both support the automation of repetitive activities and the management of the information represented by the series of models and artifacts and encourage and support the creative activities that are the crucial core of significant development.

2.5.3 Balance Process and Tools

Thus to develop a process without thinking about how it will be automated is academic. To develop tools without knowing what process (framework) they are to support may be fruitless experimentation. There has to be a balance between process and tools.

On the one hand, process drives tool development. On the other hand, tools guide process development. The development of process and its tool support must take place concurrently. At every release of process there must also be a release of tools. At every release there must be this balance. To get balance closer to the ideal will take several iterations, and the successive iterations must be guided by user feedback in between the releases.

This process-tool relationship is another chicken-and-egg problem. Which one comes first? In the case of tools, many of them in recent decades have come first. Process was not yet well developed. As a result, the tools did not work as well as expected in the rather hit-or-miss processes to which users tried to apply them. Many of us had our faith in tools shaken. Software development continued to be a none-too-efficient handicraft. In other words, process must learn from tools and tools must support a well-thought-out process.

We want to spell this point out with utmost clarity: Successful development of process automation (tools) cannot be achieved without the parallel development of the process framework in which the tools are to function. This point must be obvious to everyone. If a shadow of doubt still lurks in your mind, ask yourself whether it would be possible to develop IT support for the business processes at a bank without knowing what those processes were.

2.5.4 Visual Modeling Supports UML

We have just established that tools are important in carrying out the purpose of process.

Let us look at a significant example of such a tool in the context of the support environment for the Unified Process. Our example is the modeling tool for UML.

The UML is a visual language. As such, it is expected to have features common in many drawing packages, such as in-line editing, formatting, zooming, printing, color, and automatic layout. In addition to these features, UML defines syntactic rules that specify how elements of the language might be used together. So the tool has to be capable of assuring that these syntactic rules are followed. This capability is beyond that possessed by common drawing packages, where no rules are enforced.

Unfortunately, enforcing syntactic rules of this sort without exception would make the tool unusable. For instance, during model editing, the model will frequently be syntactically incorrect, and the tool needs to be able to allow for syntactical incorrectness in this mode. For example, a message in a **sequence diagram** (Appendix A; see also Chapter 9) might be allowed before any operations are defined for the class.

UML includes a number of semantic rules that also need to be supported. These rules can be incorporated in the modeling tool, as either instant enforcement or on-demand routines that traverse a model and check for common mistakes or look for semantic or syntactic incompleteness. In summary, the modeling tool needs to incorporate more than knowledge of UML; it needs to allow developers to work creatively with UML.

Using UML as the standard language, the market will experience much better tool support than any modeling language has ever had. This opportunity for better support is due in part to the precise definition of UML. It is also attributable to the fact that UML is now a widely adopted formal industry standard. Instead of tool vendors competing to support many different modeling languages, the game has now become one of finding who best supports UML. This new game is better for users and customers of software.

UML is only the modeling language. It does not define a process of how to use UML to develop software systems. The modeling tool does not have to enforce a process, but if the user uses a process, the tool can support it.

2.5.5 Tools Support the Whole Life Cycle

There are tools to support every aspect of the **software life cycle** (Appendix C):

- *Requirements management*: Used to store, browse, review, track, and navigate the various requirements of a software project. A requirement might have a status attached to it, and the tool might permit a requirement to be traced to other artifacts in the life cycle, such as a use case or a test case (see Chapter 11).
- *Visual modeling*: Used to automate the use of UML, that is, to model and assemble an application visually. With this tool we integrate with programming environments and ensure that the model and implementation are always consistent with each other.
- *Programming tools*: Used to provide a range of tools, including editors, compilers, debuggers, error detectors, and performance analyzers.
- *Quality assurance*: Used to test applications and components, that is, to record and execute test cases that drive testing of a GUI and an interface of a compo-

ment. In an iterative life cycle, regression testing is even more essential than it is in conventional development. Automating test cases is essential to allow for high productivity. In addition, many applications also need to be exposed to stress and load testing. How will this application's architecture stand up to the use of 10,000 concurrent users? You want to know the answer to this question before you deploy to the 10,000th user.

In addition to these functionally oriented tools, there are other tools that are cross-life cycle. These tools include version control, configuration management, defect tracking, documentation, project management, and process automation.

2.6 References

- [1] OMG Unified Modeling Language Specification. Object Management Group, Framingham, MA, 1998. Internet: www.omg.org.
- [2] Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Reading, MA: Addison-Wesley, 1997.
- [3] Watts S. Humphrey, *Managing the Software Process*, Reading, MA: Addison-Wesley, 1989.
- [4] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology*, Reading, MA: Addison-Wesley, 1995.
- [5] Ivar Jacobson and Sten Jacobson, "Beyond methods and CASE: The software engineering process with its integral support environment," *Object Magazine*, January 1995.
- [6] Ivar Jacobson and Sten Jacobson, "Designing a Software Engineering Process," *Object Magazine*, June 1995.
- [7] Ivar Jacobson and Sten Jacobson, "Designing an integrated SEPSE," *Object Magazine*, September 1995.
- [8] Ivar Jacobson and Sten Jacobson, "Building your own methodology by specializing a methodology framework," *Object Magazine*, November–December 1995.
- [9] Grady Booch, *Object Solutions: Managing the Object-Oriented Project*, Reading, MA: Addison-Wesley, 1996.
- [10] Walker Royce, *Software Project Management: A Unified Framework*, Reading, MA: Addison-Wesley, 1998.