Defining the Software Process

Software development can be exceedingly complex and there are often many alternative ways to perform the various tasks. A defined process can help guide the software professionals through these choices in an orderly way. With an established process definition they can better understand what they should do, what they can expect from their co-workers, and what they are expected to provide in return. This allows them to focus on doing their jobs. A defined software process also provides organizations with a consistent working framework while permitting individual adjustments to unique needs.[1, 7] As Deming says, operational definitions are “something everyone can communicate about and work to.”[6]

Software engineering, however, is not a routine activity that can be structured and regimented like a repetitive manufacturing or clerical procedure. We are dealing with an intellectual process that must dynamically adjust to the creative needs of the professionals and their tasks. A trade-off is thus required between the individual need for flexibility and the organizational need for standards and consistency. Some of the factors to be considered are:

1. Since software projects have differences, their software engineering processes must have differences as well.

2. In the absence of a universal software engineering process, organizations and projects must define processes that meet their own unique needs.

3. The process used for a given project must consider the experience level of the members, current product status and the available tools and facilities.
13.1 Process Standards

While the need for project-unique process definitions is clear, there are also compelling reasons for standardization:

- Process standardization helps to reduce the problems of training, review, and tool support.
- With standard methods, each project's experiences can contribute to overall process improvement.
- Process standards provide the basis for process and quality measurements.
- Since process definitions take time and effort to produce, it is impractical to produce new ones for each project.

The conflicting needs for customization and standardization can often be resolved by establishing a process architecture, which consists of a standard set of unit or "kernel" process steps with rules for describing and relating them. Customization is then achieved through appropriate interconnections of these standard elements into tailored process models.

13.2 Definitions

Before proceeding further, several terms should be defined. "The term software refers to a program and all the associated information and materials needed to support its installation, operation, repair, and enhancement."[12] This is consistent with Brooks's definition of a programming product as a program together with all those items required to make it intelligible, usable, and extendable.[4] A program, of course, is merely the set of instructions that run on a computer. A program product thus includes all the user and support documentation, as well as any ancillary programs needed to use or support it. With this understanding the term software is used to refer to programming products.

Some definitions of software engineering, the software engineering process, software process architecture, and software process models are:[12]

- **Software engineering** "The disciplined application of engineering, scientific, and mathematical principles, methods, and tools to the economical production of quality software"

- **Software engineering process** "The total set of software engineering activities needed to transform a user's requirements into software"

- **Software process architecture** "A framework within which project-specific software processes are defined"
Software process model “One specific embodiment of a software process architecture”

Software process The set of activities, methods, and practices that are used in the production and evolution of software

Since most software is maintained and enhanced throughout its life, these definitions are intended to encompass new development, enhancement, and repair.

While software process models may be constructed at any appropriate level of abstraction, the process architecture must provide the elements, standards, and structural framework for refinement to any desired level of detail.

13.3 Levels of Software Process Models

Software process models can be defined at any of three levels. The U, or Universal, process model provides a high-level overview. The W, or Worldly, process model is the working level that is familiar to most programmers and managers. The A, or Atomic, process model provides more detailed refinements.

13.3.1 U Process Models

The Waterfall Model, described by Royce in 1970, is still the best known and most widely used overview framework for the software development process.[19] As shown in Fig. 13.1, it describes the basic process steps and provides general guidance on their role and order. While this model has been helpful in explaining the software development process, it has several shortcomings:[3]

- It does not adequately address changes.
- It assumes a relatively uniform and orderly sequence of development steps.
- It does not provide for such methods as rapid prototyping or advanced languages.

To address these concerns, Boehm has proposed the Spiral Model, as shown in Fig. 13.2.[3] This U process model sheds light on some of these issues and can help in identifying several of the key risks in software development.

Unfortunately, the real world of software development doesn’t conform neatly to either of these models. While they represent the general work flow and provide overview understanding, they are not easily decomposed into the progressively finer levels of detail that are needed to guide the work of the software professionals.

First, it is important to recognize that a complete model of a complex process must be complex. The operative work is complete. If one wants to use a model for a specific purpose, it can presumably be tailored to that purpose and, compromise
FIGURE 13.1
The Waterfall Process Model.[19]
completeness in other respects. These compromises, however, must be made with care, or the resulting simple model representation could easily be misleading.

The fundamental problem with simplistic U-level software process models is that they do not accurately represent what is really done.[13] The reason is that traditional process models are extremely sensitive to task sequence; consequently, simple adjustments can destroy the entire framework. This results from an over-emphasis on modeling tasks. While modeling tasks seems like a natural way to guide task-oriented people, it limits human flexibility and tends to arbitrarily

---

**FIGURE 13.2**
Spiral Model of the Software Process [3]
impose rigidity. With the requirements-design-code-test sequence, decisions to re-examine the requirements during test, for example, cannot be readily accommodated without a complete restructure of the process model. In short, with traditional models it is difficult to adequately address the behavioral aspects of processes.

The traditional task-oriented approach to process models results naturally from our task-oriented view of our work. This, for example, is what led to the Waterfall Model: the need for a general prescription of human activity. While this task structure is quite appropriate and relatively easy to understand when the tasks are simply connected, it becomes progressively less helpful as the number of possible task sequences increases. While it can, in principle, still produce an accurate model, it becomes more difficult to do so and progressively less understandable.

The real danger of attempting to use task-oriented models in such complex situations is that they must be simplified to permit human comprehension, and these simplifications tend to limit flexibility in task sequencing. When there are, for example, ten possible actions that could usefully be performed, a simplified task-oriented process model would presumably only show one or two. While this might be perfectly acceptable under normal circumstances, the other alternatives might then be viewed as abnormal or unauthorized. When such models are used to guide process automation, project management, or contract administration, the resulting process rigidity can cause serious problems. The question, therefore, is not “what is the right way to model the process?” but “what is the most appropriate way to model this process for this purpose?”

A final point concerns actual process execution. When the process model is used to guide or control the detailed performance of the process by people or machines, a comprehensive model is required, and it must include a task-oriented view. Indeed, a complete process model needs to contain functional, behavioral, structural, and conceptual views.[16, 17] For process management purposes, however, a simpler process model is appropriate as long as it does not artificially constrain process execution. There is good reason to believe that task-oriented models are not always the best for this purpose and that an entity view may be more appropriate in many cases. Entity process models are discussed briefly in a later section of this chapter. For this discussion we will deal with the more traditional and easily understood task-oriented process models. As will be noted later, however, many of the same considerations apply to entity process models as well.[13]

### 13.3.2 A Process Models

At the opposite extreme from U-level models, Atomic- (A-) level process models can be enormously detailed. They are needed by anyone who attempts to automate a specific process activity or use a standardized method or procedure to guide
execution of a task. Precise data definitions, algorithmic specifications, information flows, and user procedures are essential at this level. The amount of detail to be included in such models must be determined by their use. For example, an experienced developer who is repeating known manual tasks will not need as detailed a standard as a new trainee. When the task is to be automated, however, a great deal of detail is generally required.

Atomic process definitions are often embodied in process standards and conventions. These can be treated as process abstractions in the higher level W or U process models.

13.3.3 W Process Models

The Worldly (W) process level is of most direct interest to practicing software engineers. It guides the sequence of their working tasks and defines task prerequisites and results. When reduced to operational form, these models generally look like procedures. They specify who does what when. Where appropriate, they reference the A level that specifies standard task definitions or tool usage. For each task, W models define the anticipated results, the appropriate measures, and the key checkpoints.

13.3.4 Examples of Process Models

The three levels of process models can be viewed as embodied in policies at the U level, procedures at the W level, and standards or tools at the A level. Policies establish a high-level framework and set of principles that guide the overall behavior of organizations. They are particularly helpful in unanticipated circumstances where no precedents have been established. Some appropriate policy-level statements on the software process might be:

- All work will be subjected to an inspection before it is incorporated in a baseline.
- The quality of each product, at the time of new shipment, shall be better than its predecessor or leading competitor.
- All commitments for software cost or delivery will be supported by a documented and approved software engineering plan.
- Quality Assurance will review the software development process to assure senior management that the work is done according to established standards and procedures and in conformance with the intent of the stated policies.

At the W level, procedures are established to implement the policies. This W-level process model refers to any available Atomic-level standards that define precisely how tasks are to be performed.
At the W level, for example, a procedure might define the points at which Quality Assurance reviews are to be conducted and how the resulting issues are to be handled. This might specify what percent of the work is to be reviewed, how statistical samples are to be selected, and whether, when, and how SQA independently tests or monitors the software engineering work as it is being done.

Atomic-level standards then serve as the basis for directing the work and for the SQA review. For example, a code inspection standard would specify what code is to be reviewed, when, the methods to be used, the reports to be produced, and the acceptable performance limits. The developers would use this standard to guide their actions, and the SQA people would review their actions and work products against this standard.

13.4 Prescriptive and Descriptive Uses of Models

Process models can be used either to describe what is done or to characterize what is supposed to be done. In a descriptive case, models can provide useful information about the process and its behavior. After such models have been calibrated to reasonably represent actual behavior, they can be used to simulate process performance under varying conditions. This can help identify potential process problems before they occur.

This book uses process models in a prescriptive sense. The approach is to define how the process should be conducted and suggest where appropriate policies, procedures, and standards could help guide the work. As with any standard, process standards must not be overspecified. A coding standard, for example, that precisely specifies commenting rules would probably not be appropriate. While such rules are important, the needs are different for different languages and conditions. If, however, the scope of the standard were constrained to specific languages, it could very well be reasonable.

13.5 A Software Process Architecture

Since most organizations have at least some policies, procedures, and standards, they are also generally following some intuitive U-, W-, and A-level models both prescriptively and descriptively. To be fully effective, however, these process models should be explicit and should relate to each other. The problem of building process models, in fact, is much like that of building software systems. An architectural framework is needed to define the basic elements, how they relate, and how they are decomposed into greater detail.

The process architecture must, therefore, encompass all process levels. U process models permit global understanding and provide a framework for estab-
lishing effective policies. W process models guide the daily work, while A process
models provide the atomic detail for training and task mechanization. These levels
all connect and should be derivable from each other in a consistent and cohesive
way. This calls for a family of refineable and relatable process elements that can be
coupled to produce the specific models needed to address project needs.

13.6 Critical Software Process Issues

The reason for defining the software process is to improve the way the work is
done. By thinking about the process in an orderly way, it is possible to anticipate
problems and to devise ways to either prevent or to resolve them. Some of the
major software process issues concern quality, product technology, requirements
instability, and complexity. In summary, these issues are:

- **Quality**  Humans are error-prone; they make mistakes. Each error, when
  found, is a surprise whose correction is both expensive and disruptive. When
  the organization’s quality performance has not been adequate, some process
  changes are generally required. Examples might be the introduction of design
  and code inspections, establishing quality plans and measurements, im-
  proved testing, or more intensive Software Quality Assurance.

- **Product technology**  Often with new developments it is not clear how to
  implement an algorithm, to meet a performance goal, or to pack function into
  a limited configuration. Since later patching of unsuccessful attempts can be
time-consuming and disruptive it is often wise to make process provisions for
  early experiments to provide this knowledge. Subsequent development can
  then proceed in a more orderly and effective way.

- **Requirements instability**  This is probably the most important single soft-
  ware process issue for many organizations. To design, to build, and to test a
  program the required functions, interfaces, and environments must be stable.
  While these may change during development, the changes must be tem-
  porarily frozen while development proceeds. At planned intervals, batches of
  changes can be considered and the design adjusted accordingly. If change is
  not controlled in this way, the development process will become unstable,
  and productivity and quality will be adversely affected. There are three basic
types of requirements changes that can be addressed in different ways by the
  process:

1. **Unknown requirements**  The users think they know what is wanted, but
during initial use they discover that their real needs are not what they had
thought. This is the normal consequence of automating a manual process
and can be countered by either installing early prototype systems or de-
composing the system into small incremental phases that are progressive-
ly developed, installed, used, evaluated, and then enhanced.
2. **Unstable requirements**  While the general requirements may be known, the specifics remain fluid. In embedded systems, for which the hardware and the software are often developed simultaneously, hardware changes may affect software requirements. Changing the hardware under a developed program is about as unsettling as changing the foundation while trying to build a house. While the frame may remain, patching all the cracks can be expensive. Here the process approach is to try to anticipate the instabilities and to isolate them. With an ill-defined aircraft flight instrument, for example, an abstraction could represent it to the rest of the program. If done properly, instrument changes could then be contained in the code abstraction that implements them.

3. **Misunderstood requirements**  Even when the requirements are known and stable, the implementers often do not understand them in the detail required to produce a satisfactory product. A typical example is end user interfaces. Here, user tests could be made with interface prototypes, or early versions of the user documentation could be tested in simulated use.

   □ **Complexity**  Application programs are often easier to develop than complete systems because their environment is generally more stable. This does not mean the work is less challenging but just that fewer things are changing simultaneously. During development of a new computer operating system, for example, everything is generally in flux: the basic control program, the compilers, the data management system, and often the computer architecture itself. To get anything done at all, special process provisions are required to provide at least intermittent stability.

Typically stability is achieved by establishing a modular design with functional and interface definitions. Disciplined change control then allows each module to have known and stable requirements during the intervals between changes. That is, the system is built in a succession of drivers. The requirements for the modules for driver 1 are defined, and, after it has been designed, driver 2 requirements are established, and so on. As long as the high-level design is well conceived and controlled, the modules can be successively improved, tested, and integrated into a progressively more refined final system.

### 13.7 A Preliminary Process Architecture

Organizations that face the issues of quality, product technology, requirements stability, and/or complexity need to define ways to address them. A process architecture permits them to represent and manipulate the process at the U level and then selectively to refine it to the W and A levels. This, of course, needs an overall architectural framework and a set of definitions.
The basic element of the process architecture is the unit cell, as shown in Fig. 13.3. Each cell is defined to accomplish a specified task and is uniquely identified. Each cell also has required entry conditions specified for task initiation that include the inputs (one or more with their sources). The task standards, procedures, methods, responsibilities, and required measurements are also defined. Finally, the exit conditions define the results produced, their level of validation, and any other post-task conditions. Cell feedback refers to any data provided to or received from other stages in the process.

As an example of how this cell can be used, Fig. 13.4 shows the full development cycle in one cell. While this abstract U-level model is not terribly informative, it does show the kind of information required for every process cell.

A somewhat more refined U-level model of the development process is shown in Fig. 13.5. Here the development cycle is broken into the basic cells of the Waterfall Model. The E, T, X, M (entry, task, exit, measure) specifications for this cell are shown in Table 13.1.

---

**Specifications:**
- **Entry:** The conditions to be met before task initiation.
- **Exit:** The results to be produced and how embodied.
- **Feedback:**
  - In: Any feedback from other stages.
  - Out: Any feedback to other stages.
- **Task:** What is to be done, by whom, how, and when, including appropriate standards, procedures, and responsibilities.
- **Measurements:** The required task measures (activities, resources, time), output (number, size, quality), and feedback (number, size, quality).

---

**FIGURE 13.3**
Basic Unit Cell
Specifications:
Entry: Approved requirements document and clarifications from requirements organization and requirements changes.
Exit: Delivered, tested software, including source and object code, design and user documentation, and support software.
Feedback:
In: User problems.
Out: Any areas where requirements need clarification from requirements organization, as determined in design, implementation, or test.
Task: The development organization develops the software using specified standards, procedures, and methods.
Measurements: Measurements of the task (changes, programmer time, schedule), the product (LOC, pages, defects), and the feedback (requirements issues, user problems).

FIGURE 13.4
The Single-Cell Development Process

13.7.1 Standard Process Elements

When looked at from the highest, or U, level, software processes tend to look much the same. This is because they are described in broad generic terms like "design," "implementation," or "test." When these activities are broken into more detail, however, significant differences show up. For example, the development of a new application program generally involves different activities than the enhancement of a system program. A control program for a new real-time computer or a system support program would also be handled differently. While these efforts could all be represented by the same U model, the actual software tasks are quite different.

With all this variation at the W level, however, many software activities can be relatively standardized across different projects. It is thus possible to establish
some basic process cells that can be interconnected in different ways to meet project-unique needs. The detailed structures of these standard cells are then further defined by A-level models as needed.

13.7.2 Implementation Cells

In defining a standard set of software process cells, we start with some relatively detailed software tasks. The basic implementation cell, \( C_0 \), is shown in Fig. 13.6. Since it is not very useful by itself, it is typically augmented with some form of testing, as shown in Fig. 13.7. This is the quick kernel, or \( K_q \), which forms one of the basic elements of the software process. Since \( K_q \) often produces poor-quality code, more comprehensive testing or inspection is generally required before useful programs result.

Before modifying \( C_0 \) to include inspections or testing, however, an inspection operator is established, as shown in Fig. 13.8. With this operator inspections can be applied to any arbitrary process activity (A). The final inspection action is to consider the inspection results and to decide whether to pass the product to the next development stage, to recycle back to A for fixes, or backtrack to a previous stage. Backtracking is often required when an inspection finds problems with earlier work.[5]

The significance of backtracking is that software process actions produce results that depend on them. If, for example, a design decision on some data type specification were later changed, every design and every implementation using this data type must also be examined. This requires knowledge of a growing number of dependencies as development progresses. When a change is made, this
TABLE 13.1
U-LEVEL DEVELOPMENT PROCESS CELL SPECIFICATIONS

<table>
<thead>
<tr>
<th>Cell:</th>
<th>Entry</th>
<th>Exit</th>
<th>Feedback In</th>
<th>Feedback Out</th>
<th>Task</th>
<th>Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Approved requirements, changes, and development plan</td>
<td>Inspected and approved design and changes</td>
<td>Inspected and approved design and changes</td>
<td>Inspected and approved design and changes</td>
<td>Inspected, tested, and approved software</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design issues</td>
<td>Implementation issues</td>
<td>Requirements issues</td>
<td>Requirements and design issues</td>
<td>Implementation, inspection, and unit test</td>
<td></td>
</tr>
</tbody>
</table>

dependency tree must be traced to identify every single resulting design or code change. If an error is not found in the process stage where it was introduced, its consequences can pervade all subsequent work, and correction costs can become quite large. The complexity of this process is one reason why it is so expensive to find and fix errors late in the development process and why abstractions are increasingly used in an attempt to reduce the impact of such changes.

13.7.3 The Unit Implementation Kernel

When the inspection operator is applied to each task in cell C₀, we get the implementation cell C₁, shown in Fig. 13.9. Here backtracking is shown both within C₁ and to prior process stages.

The next step is to add unit test, which results in the unit implementation kernel K₁, shown in Fig. 13.10. This kernel, which is used to implement small program modules, has the following characteristics:
1. Cell 001 uses the implementation cell $C_1$ to produce code, design, and documentation.

2. The design from 001 is used to define the unit test high-level design in 003. Since this is a white box test of program structure, the tests must be based on the structural design.

3. A $C_1$ development process is used in 004 to produce the necessary test cases and drivers.

4. At the conclusion of unit test, 002, the results are reviewed to see if the product should be passed to the next development phase or returned for further work.
5. Backtracking is possible from cell 001 or 002 and theoretically possible from any cell.

6. The output of unit implementation kernel $K_1$ is a final unit-tested program module with a documented design.

### 13.7.4 The Cell Specification

Once the general flow of a process is known, it is important to define each process cell. One approach is to use what is called the ETVX paradigm.[18] This refers to
FIGURE 13.10
Unit Implementation Kernel $K_1$
the explicit characterization of the Entry, Task, Verification, and Exit criteria for each process action. A slightly modified version of this, the Entry-Task-Exit (ETX) specification, is used here for cell definition. The ETX specifications for C₁ are shown in Table 13.2 and the ETX specifications for the unit implementation kernel K₁ are shown in Table 13.3. These process specifications for each project should include explicit responsibilities for task performance and should refer to the applicable standards and procedures.

Such detail may seem unnecessary since all the steps in K₁ are generally handled by a single programmer. Few implementers, however, have the background or training to know how to do every step of K₁ in the most effective way. A defined process can provide them guidance as well as setting the standard for management review and SQA audit.

13.8 Larger Process Models

Once some basic process cells have been defined, it is possible to construct larger process models. This is done by interconnecting these basic cells in various ways. The idea is consciously to design the development or maintenance process to address the anticipated issues and problems. The issues of quality, product technology, unknown requirements, unstable requirements, misunderstood requirements, and complexity generally can be dealt with by such process models. The following sections give examples of how this can be done.

13.8.1 Product Technology Unknowns

In advanced software systems there are often significant technical unknowns. Hardware engineers have long known that initial implementations of novel products rarely work as expected and are never directly shippable. They thus build "breadboards" to test their technical concepts and experiment with alternative approaches. Typically they implement the critical system elements in simplified form and rig the balance of the system in a suitably expedient way.

With software it is also appropriate to breadboard critical, complex, or unusually demanding functions. There are many potential implementation questions, and programmers often are unsure which ones to pursue until they have actually built something and run it. Unfortunately, most such choices are made in a standard product implementation process that does not allow time to try alternatives and throw away early versions.

The quick kernel, Kₜ, is an appropriate way to produce experimental code. Since such code has a short useful life and is only for use by the implementer, the
### TABLE 13.2
ETX SPECIFICATIONS FOR THE IMPLEMENTATION CELL C₁

<table>
<thead>
<tr>
<th>Entry</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspected and approved high-level design</td>
<td>Detailed design and problem fixes</td>
<td>Inspected and approved detailed design and problem fixes</td>
<td>Source and object code and problem fixes</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>Detailed design and problem fixes</td>
<td>Inspected and approved detailed design and problem fixes</td>
<td>Source and object code and problem fixes</td>
<td>Inspected software and inspection report</td>
</tr>
<tr>
<td>Feedback In</td>
<td>Inspection problems</td>
<td>Inspection problems and inspection approval/disapproval</td>
<td>Problems found</td>
<td>Problems found and approval/disapproval</td>
</tr>
<tr>
<td>In</td>
<td>Inspection approval/disapproval</td>
<td>Problems found</td>
<td>Detailed design and high-level design problems</td>
<td>Problems found and approval/disapproval</td>
</tr>
<tr>
<td>Feedback Out</td>
<td>High-level design problems</td>
<td>Produce the detailed design, documentation, and problem fixes</td>
<td>Inspect and approve detailed design and problem fixes</td>
<td>Implement and document detailed design and problem fixes</td>
</tr>
<tr>
<td>Task</td>
<td>Produce the detailed design, documentation, and problem fixes</td>
<td>Inspect and approve detailed design and problem fixes</td>
<td>Implement and document detailed design and problem fixes</td>
<td>Implement and document detailed design and problem fixes</td>
</tr>
<tr>
<td>Measures</td>
<td>Resources expended, work product produced, problems found</td>
<td>Resources expended, work product inspected, problems found</td>
<td>Resources expended, work product produced, problems found</td>
<td>Resources expended, work product produced, problems found</td>
</tr>
</tbody>
</table>

The need for documentation, inspection, or testing is minimal. If the code works sufficiently well to prove the concept, the simple $K₄$ process should suffice. Where the implementer concludes that documentation, an informal inspection, or some other provisions are needed they can, of course, be added.

The full experimental kernel, $K_6$, is shown in Fig. 13.11. The key to this subprocess is the feedback from 002 to 001 that determines whether the experiment should be revised and repeated or if enough has been learned to proceed with a full high-level design (003) and $K_1$ implementation (004). It is, of course, possible to learn from these steps something that invalidates prior work and causes backtracking to previous stages. The ETX specifications for $K_6$ are shown in Table 13.4.
TABLE 13.3
ETX SPECIFICATIONS FOR THE UNIT IMPLEMENTATION KERNEL K_1

<table>
<thead>
<tr>
<th>Cell:</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>Inspected and approved high-level design</td>
<td>Inspected and approved source and object code and test cases</td>
<td>Inspected and approved detailed design</td>
<td>Test case requirements and plan</td>
</tr>
<tr>
<td>Exit</td>
<td>Inspected and approved software and inspection report</td>
<td>Unit tested and inspected software</td>
<td>Test case high-level design</td>
<td>Inspected and approved test cases</td>
</tr>
<tr>
<td>Feedback In</td>
<td>Unit test problems approval/disapproval</td>
<td>Test case high-level design problems</td>
<td>Test case problems</td>
<td></td>
</tr>
<tr>
<td>Feedback Out</td>
<td>High-level design problems</td>
<td>Unit test problems Unit test approval/disapproval</td>
<td>Detailed design problems</td>
<td>Test case high-level design problems</td>
</tr>
<tr>
<td>Task</td>
<td>Implement and inspect software</td>
<td>Perform unit testing</td>
<td>Produce test case high-level design and plan</td>
<td>Implement and inspect test cases</td>
</tr>
<tr>
<td>Measures</td>
<td>Resources expended, work product produced, problems found</td>
<td>Resources expended, work product produced, problems found</td>
<td>Resources expended, work product produced, problems found</td>
<td>Resources expended, work product produced, problems found</td>
</tr>
</tbody>
</table>

13.8.2 The Problem of Complexity

The process kernels described thus far have focused on small tasks that could be performed by one or two programmers. While these kernels provide useful guidance for small tasks, their real value is the insight they can provide for larger projects. Here multiple modules are typically involved, and the work of many professionals must be coordinated.

One effective way to handle the problems of large-scale software process complexity is shown in Fig. 13.12. Here the high-level design (001) establishes the basic requirements for each module, which is then implemented with the appropriate kernel process for that module (002...003). The design is also the foundation for integration planning (005), which produces the integration and test requirements and plan. This in turn defines the required test cases, the drivers, and
FIGURE 13.11
Experimental Kernel $K_a$
TABLE 13.4
ETX SPECIFICATIONS FOR EXPERIMENTAL KERNEL K_e

<table>
<thead>
<tr>
<th>Cell:</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entry</strong></td>
<td>Approved requirements</td>
<td>Experimental program design and code Test objectives</td>
<td>Experiment results, inspected and approved requirements</td>
<td>Inspected and approved high-level design</td>
</tr>
<tr>
<td><strong>Exit</strong></td>
<td>Experimental program design and code Test objectives</td>
<td>Experiment results</td>
<td>Inspected and approved high-level design</td>
<td>Inspected and tested software</td>
</tr>
<tr>
<td><strong>Feedback In</strong></td>
<td>Test problems and questions</td>
<td></td>
<td>Implementation problems</td>
<td></td>
</tr>
<tr>
<td><strong>Feedback Out</strong></td>
<td>Requirements problems</td>
<td>Test questions and interim results</td>
<td>Requirements problems</td>
<td>High-level design problems</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>Develop experimental model</td>
<td>Run experiment, measure, document, and decide if objectives met</td>
<td>Develop and inspect high-level design</td>
<td>Implement, inspect, unit test, and document software</td>
</tr>
<tr>
<td><strong>Measures</strong></td>
<td>Resources expended, work product produced, problems found</td>
<td>As specified by test objectives</td>
<td>Resources expended, work product produced, problems found</td>
<td>Resources expended, work product produced, problems found</td>
</tr>
</tbody>
</table>

the stubs to be produced (006). All the modules and facilities are delivered to the integration cell (004), which integrates them together into the final system.

The integration process can be better appreciated by examining a further decomposition (Fig. 13.13). Several modules M_1 through M_j are first integrated (001) to form the basic system nucleus, or initial driver build B_j. While this is a big bang integration, it is kept as limited as possible while still producing a rudimentary driver capability. Once the initial driver is available, additional modules can be incorporated a few at a time. Module test cases then are run against each build to ensure that it was properly incorporated and that it functions according to specifications. These system builds are then made available for use in test or component build.

The build cell is shown in more detail in Fig. 13.14. Here one or more new modules M_j are added to the previous build in spin j (001). The result is tested against the j test cases IT_j in 002, and a decision is made on whether the module passes the integration test criteria and should remain in the spin or if it should be removed. A de-spin (004) is used to remove any troublesome modules.
Figure 13.12 Integration Cycle
If the integration decision is passed, regression tests are run (003) to help ensure that the changes in this spin do not invalidate previously integrated functions. Again, the test results are used to decide whether the spin should be accepted. The regression test bucket is updated (005) to reflect the new module and the integration test experience. The regression test data is used in building the next regression bucket, RB(j+1).

13.8.3 Requirements Instability

To this point, it has been assumed that the software requirements were known, stable, and understood. Since this is rarely the case, compensating process provisions are needed. The appropriate provisions depend on which of the three basic types of requirements instability is involved:

1. $KSU'$ The requirements are known and stable, but the implementers do not understand them sufficiently well to produce an adequate implementation (not U, or U').

2. $KS'U$ The requirements are known and understood, but they are not stable ($S'$).

3. $K'SU$ The requirements are stable and understood, but they are not fully known ($K'$).

Case 1 seems reasonable, but the other two appear illogical. How, for example, could a requirement be known and understood but unstable? This, unfortunately, is a very common problem. In many systems the requirements are firm and well understood right up until the need for a change is recognized. This often happens when many parts of a large system are being developed simultaneously and unanticipated design or implementation problems in one unit affect several others.

Case 3 was aptly described by Mr. Dooley, Finley Peter Dunne’s fictional philosopher, when he said, “It ain’t what you don’t know that hurts you—it’s what you know that ain’t so.” When software is part of a new application that has yet to be tried, the planners often believe they know the requirements, but they do not. The new application will change its environment, which will change system behavior, modify the user’s perceptions, and change the requirements. This is the basic software uncertainty principle: The more an application is changed, the less accurate are its requirements. It is important not to confuse precision with accuracy. Requirements may be very detailed and precise but not accurately represent the real needs of the users.

13.8.4 Prototype Process Models

Prototype programs can be built to learn the potential customers’ reactions. These prototypes demonstrate one or more facets of system behavior for test with the
intended users. They can then be used to try to reduce the requirements uncertainties. The requirements instability case 1, KS'U', can be addressed by using the $K_e$ experimental development cycle in Fig. 13.11. Here the measurement and evaluation step should involve end users or others who understand the application and can spot operational problems. This figure, for example, is the process model that represents a user interface simulation.

The process in Fig. 13.15 deals with requirements that are known and understood but unstable (KS'U). Here the development process is broken into releases, and the requirements changes are batched for periodic introduction. An initial high-level design and $K_1$ development process (001) are used to develop the initial modules needed for the initial product build $B_1$. These are integrated in 002, and this initial build is used as the base for integrating the next batch of required function in 004. This process continues until the final build $B_6$ is produced at 006. This incremental development process provides temporary requirements stability while accommodating changing user needs.

The process in Fig. 13.16 is used when the requirements are not known in advance (K'SU). Here the only answer is to install a suitably performing prototype product in the customer's environment for trial use. Once the requirements are known, they are returned to development, where the prior version is updated or a new product is built. The process shown in Fig. 13.16 uses whichever development process, $D_1$, is appropriate to the situation. For relatively simple systems not requiring integration, high-level design and $K_1$ kernels could be used. With larger systems, the full integration process of Fig. 13.12 might be used. Here again, the approach is to ensure that the full development process is only used with reasonably stable requirements or with relatively modest incremental requirements changes. With real unknowns, of course, the experimental kernel $K_e$ should be used. With truly revolutionary systems, several rebuild cycles may be needed before the system performs as desired.

**The Use of Prototyping.** Each of these prototyping methods could be used at any of the process levels. Instead of using $K_1$ for implementation, the full integration cycle in Fig. 13.12 could be used, for example. The specific process selected should depend on the problems and risks with the system and each of its modules.

In a large system some modules may require some form of prototyping, while others could be developed directly. With external interfaces, for example, the involved modules should often be prototyped and tested with the end users. Few technical professionals are capable of appreciating the operational details that make systems attractive to their users, though many may think they can. The normal result is a hard-to-use system that is expensive and time-consuming to fix. An appropriate process would ensure that these user interfaces were prototyped and checked before implementation.
**Prototyping Issues.** Evidence shows that product development through prototyping is substantially faster and less expensive than traditional methods.\[2, 8, 9\] Some key considerations for using prototypes are:

- Even if it is only intended as a quick experiment, the prototype’s objectives should be clearly established before starting to build it.
- Define the prototype process in advance. Prototyping does not mean hacking code in any way that comes to mind. The methods used should be appropriate to the specific prototype objectives.
- The results of the prototyping effort must be documented and analyzed before a decision is made on how to proceed.
- Complete the initial prototype and analyze the results before starting another prototype iteration. Prototyping is habit-forming, and unplanned iterations can be very expensive.
- There is no “right” way to prototype. The result may be thrown away, used as a foundation for enhancement, or re-packaged as a product. It all depends on the original objectives, the process used, and the desired quality of the result.
- When the prototype is to be included as part of the final product, the need for design records, service facilities, and user documentation must be recognized at the outset and suitable process provisions made. If these needs are not addressed in the initial implementation, later retrofit is generally more expensive than throwing the prototype away and building a product from scratch. With the knowledge gained, this is often far faster than trying to clear up the undocumented patchwork that often results from prototype enhancements. A
carefully developed prototyping plan and process specification are thus of paramount importance.

13.9 Detailed Process Models

While one can say a great deal about knitting without knowing how to make stitches, one can't actually do knitting. The same is true of programming. All the process models in the world will not help to produce programs unless they can be reduced to the level of programming. This is the Atomic process level.

With the A-level process, the detail used in task definition should be appropriate to the knowledge and skill of the professionals. One example is the partial decomposition of the process for building regression test buckets in Fig. 13.17 and Fig. 13.18. In Fig. 13.17, the regression test process starts with the regression test plan (001). The regression bucket is then built (002) using the planning data, the previous bucket, and the test case library, which includes both the prior and new tests. This regression bucket is then run (003) and the results used to decide whether to fail the product or proceed to the next stage.

Regression test planning is further decomposed in Fig. 13.18 as follows:

1. The build schedule and prior build records are examined to determine the new modules that will be included (001).
2. The expect list for these new modules is examined to determine the functions that will be included and the modules they are expected to affect (002).
3. The test records for the impacted modules are examined to identify the test cases available for them (003).
4. The test profiles are examined to determine those test cases that apply to the new functions to be included. A subset of the highest yield functional test cases is also included (004).
5. Previous regression test data and the test histories for each test case and module are examined to determine which of the older tests in the regression bucket should be retained, which should be dropped, and which tests should be added (005).

With this information the final regression bucket can be built for the next build spin. This process could be decomposed into further detail, depending on the specific needs of the project and the experience level of the people.

13.10 Entity Process Models

The process models we have discussed so far are similar to state models of a software system. In simplest terms, the process is either in an idle state before the
FIGURE 13.17
Regression Test Refinement
FIGURE 13.18
Regression Test Planning
entry criteria for the first cell are met, or the process is in a state represented by one of the succeeding cells. After the final output from the last cell is produced, the model returns to the idle state, waiting for further requirements. As the implementation tasks become more complex, however, this simplistic picture may provide a less realistic representation of actual task behavior. For example, when a design question is fed back from implementation, implementation work generally proceeds while the design question is being resolved. These small-scale process iterations are thus not visible in the higher-level models.

One alternative is to consider basing process models on entities, similar to those used by Jackson in Jackson System Development.[14] Here one deals with entities and the actions performed on them. Each entity is a real object that exists and has an extended lifetime. That is, entities are living things rather than ephemeral objects that are transiently introduced within the process.

Examples of such entities are the requirements, the finished program, the program documentation, or the design. While these sound like some of the items discussed in the Waterfall Model, they are really quite different. The traditional Waterfall Model deals with tasks such as producing the requirements. This task is then presumed completed before design starts. In reality, the requirements entity must survive throughout the process. While it undergoes various transformations, there is a real requirements entity that should be available at all later times in the process. Exactly the same is true of the design, the implementation, and the test suite.

Entity process models (EPMs) provide a useful additional representation of the software process because they are often more accurate than task-based models for complex and dynamic processes. The reason is that EPMs deal with real objects that persist and evolve through a relatively small and defined sequence of states. The transitions between these states result from well-defined causes, and the relationships of these states and transitions can be relatively invariant to the particular task sequence employed.

For this purpose, entities are real things, not just artifacts introduced to assist in the work. When, for example, the full software product life cycle is considered, a number of “artifacts” produced during development become extremely important. Examples are the requirements, the design, the test documents, the test cases, and the test plans. What is not so clearly recognized, however, is that the traditional view of these items as end products of a development phase can lead to the belief that they are completed and will undergo no further changes. As a result, there are often no official process provisions for keeping the design and requirements documentation up to date throughout implementation and test.

The question then is what are the most appropriate entities to use in modeling the software process. Some obvious entities are:

- The deliverable code
- The users’ installation and operation manuals
Some other items that should generally be considered as entities that persist after conclusion of the initial software development work are:

- The requirements documents
- The design
- The test cases and procedures

It is clear that each of these entities cycles through a set of states during the software process. By focusing on these states and the actions required to cause state transitions, the process of producing an entity process model becomes relatively straightforward:

1. Identify the process entities and their states.
2. Define the triggers that cause the transitions between these states.
3. Complete the process model without resource constraints (unconstrained process model, UPM).[13]
4. Impose the appropriate limitations to produce a final constrained process model (CPM).[13]

While this procedure is conceptually simple, it can become quite complex when applied to large-scale processes. EPMs provide a useful way to characterize complex and highly feedback activities at the U or W level, but they must be transformed to a task structure to actually guide the work or its automation. Since they can provide a more accurate high-level representation of the work, however, they provide a useful perspective for planning and tracking purposes.

### 13.11 Process Model Views

The three basic views of process models are the state view, the organizational view, and the control view. The state view is what we have discussed so far, with the states either representing various stages of the process (tasks) or stages of the product (entities). The state view thus has two complementary representations: task-oriented and entity-oriented. The task-oriented representation explicitly portrays the tasks, while the entity state changes are implied. In the entity-oriented representation, the entity states are explicitly portrayed, while the tasks are implied as causing the transitions between these states.

A view of the development process that defines the responsibilities for each activity is shown in Fig. 13.19. While such a picture is not very informative from a software engineer’s point of view, it does identify the essential responsibilities. Again, such a view could be refined to any desired level of detail.

The third view relates to measurement and control. A simplified view of the data gathering and approval aspects of the development process are shown in Fig.
13.20. The relationships between the management, control, and support activities are shown for data gathering, approval, and reporting. Since these relationships have a critical impact on the behavior of a software organization, they must be clearly defined. This is generally done at the highest level in the organization through organization charts and responsibility statements.

These are different views rather than alternatives. They each present an essential perspective of the process that must be understood, defined, and managed. If any view is not addressed, an important facet of software management will likely be overlooked.

It is possible to produce higher-level models of the software process for use in guiding software engineering activities. While the specific methods are not described here, they are similar to those used in defining traditional administrative procedures, only somewhat more complex.[18] When more detailed process refinements are desired, it is desirable to use automated tools. While there has been only limited experience with such approaches, some techniques and methods have been published.[10, 11, 13, 15, 16, 17]

13.12 Establishing and Using a Process Definition

Each software organization should establish a process architecture and process models tailored to its particular needs. This tailoring is done as follows:
1. Define a standard process as a foundation for tailoring. It is helpful to identify likely tailoring options in this standard model.

2. Establish the ETX specifications for the standard process model.

3. Make provisions to gather and track the resulting process measurements.

4. Establish checkpoints and standards for SQA review. While such checkpoints have not been illustrated here, they typically involve the review of test and inspection data and the observation of selected tasks. SQA may also conduct independent tests.

5. Incorporate specific measurement and reporting provisions.

6. Instruct the development personnel on the use and value of the process architecture, the standard process models, and when, why, and how they should be tailored.

Since many projects will likely need their own unique process definitions, they should start with the standard process and then take the following steps:
1. Identify the unique project issues, problems, and success criteria:
   - Knowns and unknowns
   - Key technical and management risks
   - Constraints
   - Measurement, tracking, and review

2. Document the adjustments required to the standard process to produce a basic overall project process. This should address:
   - The requirements unknowns
   - The integration and regression issues
   - The product design and application unknowns
   - The management and assurance requirements
   - The staffing and training needs (K may be needed just to build an adequate experience level)

3. For each software system component, repeat these definitions.
4. Once each program module has been identified, consider the process definition for it as well and make any necessary adjustments.

---

13.13 Basic Process Guidelines

Finally, some guidelines are needed for developing and using a process architecture and its process models:

- Establish objectives for each project’s process.
- Define the basic process architecture, make sure it meets the needs of the projects, and then enforce it as an overall process framework.
- Remember that each project, component, and module is unique and its process should be uniquely determined. While some combination of the standard process kernels will generally be appropriate, be open to truly new and unique situations and establish process provisions for them when they occur.
- Establish process definition standards.
- Change the process model dynamically as the problems change.
- Require that all deviations for the standard process be documented, reviewed, and approved.

In developing a process architecture and a set of process models, it is important to remember that these can become quite complex. Rather than attempt to establish a fully defined process, it is wisest to create a high-level framework architecture and
then refine a few areas at a time. This refinement process should be driven by the needs of the projects. It should also give priority to those areas in which the professionals need the most guidance and should consider the available process skills. Treat process architecture and design much the same as a complex new system development: Start with a high-level prototype and refine it as you gain knowledge and experience.

13.15 Summary

A defined software process provides organizations with a consistent process framework while permitting adjustment to unique needs. The conflicting needs for customization and standardization can be met by establishing a process architecture with standard unit or “kernel” process steps and rules for describing and relating them. Customization is then achieved through their interconnection into process models.

Software process models can be defined at the Universal (U) process level, the Worldly (W) process level, and the Atomic (A) process level. These are typically embodied in policies at the U level, procedures at the W level, and standards at the A level.

When looked at from the U level, software processes tend to look much the same. When one attempts to break these down in more detail, however, significant project differences begin to appear. At the W level, however, many of the tasks are relatively standardized across projects. They may be used in different combinations and sequences, but it is possible to identify some basic process cells that can then be interconnected to meet unique project needs. The detailed structures within these standard cells would then be A-level models.

Each software organization should establish a process architecture and process models that are tailored to its particular needs. The appropriate steps are: identify the unique project issues, problems, and success criteria; establish a basic overall project process; and repeat these definitions for each component and program module.

Some guidelines on developing and using a process architecture are: establish objectives, define the basic process architecture, make sure it meets the needs of the projects, and then enforce it as an overall process framework. Also remember that each project, component, and module is unique and its process should be uniquely determined. Process definition standards are also needed. The process models should be changed as the problems change, and all deviations from the standard process must be documented and approved.

Above all, remember that process definitions can be very complex. They should thus be approached with the same care as the design of a large system; start
with a prototype and add enhancements as requirements knowledge and development experience are gained.

References


