Rational Objectory Process

Introduction
## Contents

1 About this Manual 1

2 What is Objectory? 3

3 Process Architecture 5
   What is a Software Engineering Process 5
   Two Dimensions 6
   Process Structure 7
   Lifecycle Structure 9

4 Objectory Process Components 15
   Process Representation 15
   Process Components and Models 18
   Requirements Capture 18
   Analysis & Design 21
   Implementation 23
   Test 25

5 Features of Objectory 27
   Object Technology 27
   Use-Case-Driven Development 27
   Use Cases in the Development Process 28
   Controlled Iterative Development 29
   Requirements Management 31
   A Strong Emphasis on Architecture 32
   Component-Based Development 33
   Process Configurability 34

6 Getting Started with Objectory 37
   Objectory Manuals 37
   Objectory Online 40
   Learning the Process 40
   Guide for Projects 47

Appendix A: Key Modeling Concepts 53
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors</td>
<td>53</td>
</tr>
<tr>
<td>Use Cases</td>
<td>53</td>
</tr>
<tr>
<td>Objects</td>
<td>54</td>
</tr>
<tr>
<td>Classes</td>
<td>54</td>
</tr>
<tr>
<td>Generalization</td>
<td>55</td>
</tr>
<tr>
<td>Models</td>
<td>55</td>
</tr>
<tr>
<td>Appendix B: What Is a Use Case?</td>
<td>57</td>
</tr>
<tr>
<td>Appendix C: What Is Software Architecture?</td>
<td>64</td>
</tr>
<tr>
<td>Architecture Description</td>
<td>65</td>
</tr>
<tr>
<td>Architectural Focus</td>
<td>66</td>
</tr>
<tr>
<td>Appendix D: Differences Between Objectory Versions</td>
<td>71</td>
</tr>
<tr>
<td>From Objectory 4.0 to Objectory 4.1</td>
<td>71</td>
</tr>
<tr>
<td>From Objectory 3.8 to Objectory 4.0</td>
<td>71</td>
</tr>
<tr>
<td>Appendix E: Glossary</td>
<td>73</td>
</tr>
<tr>
<td>Appendix F: References and Further Reading</td>
<td>81</td>
</tr>
<tr>
<td>Index</td>
<td>84</td>
</tr>
</tbody>
</table>
1  About this Manual

This manual is an introduction to the Rational Objectory Process. After reading it you should be able to locate, read, and use all other parts of Objectory. This manual is intended for every member of a project that uses Objectory. To understand this manual you need only a general knowledge of software development.

- Chapter 2, *What is Objectory?* gives a brief overview of the process, including its goals and principles.
- Chapter 3, *Process Architecture*, explains the basic notions we use to describe and explain the process: in terms of its phases, iterations, and process components. You must read this chapter before using any other parts of Objectory.
- Chapter 4, *Objectory Process Components*, describes the organization of the process in terms of its representation: process component, workflows, artifacts, and workers.
- Chapter 5, *Features of Objectory*, describes salient Objectory features as being component object technology-based, use-case driven, while placing an emphasis on requirements management, and architecture.
- Chapter 6, *Getting Started with Objectory*, describes the manuals and online facilities that constitute the product, and gives guidance for managers, developers, and process engineers on how to use them.
- Appendix A, *Key Modeling Concepts*, explains the basic concepts of objects, classes, use cases, actors, and the concept of a model, for readers not familiar with object technology.
- Appendix C, *What Is Software Architecture?* is for those who want to better understand what software architecture is, and how to represent it.
1 About this Manual

- Appendix D, *Differences Between Objectory Versions*, is for readers familiar with earlier versions of the Rational Objectory Process, and describes the differences between this version and earlier versions.

- Appendix E, *Glossary*, contains definitions of the significant terms used in this manual.

- Appendix F, *References and Further Reading*, lists a number of books and articles on process, object-oriented-technology and other related topics.
2 What is Objectory?

The Rational Objectory Process is a Software Engineering Process. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end users, within a predictable schedule and budget. The Objectory process captures many of the best practices in modern software development in a form that can be tailorable for a wide range of projects and organizations.

Objectory is an iterative process. Given today’s sophisticated software systems, it is not possible to sequentially first define the entire problem, design the entire solution, build the software, and finally test the product. An iterative approach is required that allows an increasing understanding of the problem through successive refinements, and to incrementally grow an effective solution over multiple iterations. This approach gives better flexibility in accommodating new requirements or tactical changes in business objectives, and allows the project to identify and resolve risks earlier.

Objectory is a controlled process. This iterative approach is only possible however through very careful requirements management, and change control to ensure at every point in time a common understanding of the expected functionality, the expected level of quality, and to allow a better control of the associated costs and schedules.

Objectory activities create and maintain models. Rather than focusing on producing large amounts of paper documents, Objectory emphasizes the development and maintenance of models—semantically rich representations of the software system under development.

Objectory focuses on early development and baselining of a robust software architecture, which facilitates parallel development, minimizes rework, increases reusability and
maintainability. This architecture is used to plan and manage the development around the use of software components.

Objectory development activities are driven by use cases. The notion of use cases, and scenarios drive the process flow from requirements capture through testing, and provides coherent and traceable threads through both the development and the delivered system.

Objectory supports object-oriented techniques. Several of the models are object-oriented models, based on the concepts of objects, classes, and associations between them. These models, like many other technical artifacts, use the Unified Modeling Language (UML) as the common notation.

Objectory supports component-based software development. Components are nontrivial modules, subsystems that fulfill a clear function, and that can be assembled in a well-defined architecture, either ad hoc, or some component infrastructure such as the Internet, CORBA, COM/DCOM, for which an industry of reusable components is emerging.

Objectory is a configurable process. No single process is suitable for all software development. Objectory fits small development teams, as well as large development organizations. Objectory is founded on a simple and clear process architecture that provides commonality across a family of processes, and yet can be varied to accommodate different situations. It contains guidance on how to configure the process to suit the needs of a given organization.

Objectory encourages objective on-going quality control. Quality assessment is built into the process, in all activities, involving all participants, using objective measurements and criteria, and not treated as an afterthought, or a separate activity performed by a separate group.

Objectory is supported by tools that automate large parts of the process. They are used to create and maintain the various artifacts—models in particular—of the software engineering process: visual modeling, programming, testing, and so on. They are invaluable in supporting all the bookkeeping associated with the change management, as well as the configuration management that accompanies each iteration.
3 Process Architecture

What is a Software Engineering Process

A process is a set of partially ordered steps intended to reach a goal; in software engineering the goal is to build a software product, or to enhance an existing one; in process engineering, the goal is to develop or enhance a process.

Expressed in terms of business engineering, the software development process is a business process; the Rational Objectory Process is a generic business process for object-oriented software engineering. It describes a family of related software engineering processes sharing a common structure, a common process architecture.

This common process architecture is the subject of this chapter.

When a software system is developed from scratch, development is the process of creating a system from requirements. But once the systems has taken form (or in Objectory terms, once the system has passed through the initial development cycle), any further development is the process of conforming the system to the new or modified requirements. This applies throughout the system’s lifecycle.

![Diagram: Software-Engineering Process]

The software-engineering process is the process of developing a system from requirements, either new (initial development cycle) or changed (evolution cycle).
Two Dimensions

To understand the form of the Rational Objectory Process, consider the following two dimensions:

- Along process components, which groups activities logically by nature.
- Along time, the lifecycle aspects of the process as it unfolds

The process is organized both in time (phases), and content (process components).

The first dimension represents the static aspect of the process: how it is described in terms of process components, activities, workflows, and so on.

The second dimension represents the dynamic aspect of the process, as it is enacted, and is expressed in terms of cycles, phases, iterations, and milestones.

The description of the process often refers to two different viewpoints:

- A technical viewpoint, where the focus is on the artifacts and deliverables attached to the product being developed.
Process Structure

A managerial viewpoint, where the focus is on time, budget, people, process, and other economic considerations.

Process Structure

This section describes the static content of the process.

Process Components

In the Rational Objectory Process - Process Manual, the process is organized in process components that are described in terms of activities, workers and artifacts.

The Rational Objectory Process is composed of seven process components, four engineering process components:

- Requirement capture
- Analysis & Design
- Implementation
- Test

and three supporting components:

- Management
- Deployment
- Environment

Activities

Each process component comprises a set of correlated activities. An activity describes the tasks—thinking steps, performing steps, reviewing steps—done by workers to create or modify artifacts, together with the techniques and guidelines to perform these tasks, and possibly including the use of tools to automate some of these tasks.

Workflows

Workflows describe the sequencing of activities inside or across process components. For each process component, Objectory describes a typical workflow. There are also a few
typical “mid-level” workflows cutting across process components for different kinds of iterations.

Workers

A worker defines the behavior and responsibilities of an individual, or a set of individuals working together as a team. This is an important distinction because it is natural to think of a worker as the individual or team itself, but in Objectory the worker is more the role defining how the individuals should carry out the work.

Artifacts

The activities of the process have input and output, which are artifacts; some of these artifacts are deliverables.

The artifacts produced and managed by a project can be grouped into five main “buckets” or sets:

- Requirement set
- Design set
- Implementation set
- Deployment set
- Management set

Models play a central role in the Rational Objectory Process. The models are semantically consistent descriptions of the software system, developed for a specific purpose. It is possible to trace information from one model to another.

- Requirement set
  This regroups all information describing what the system must do. This may comprise a domain model, a use-case model, and other forms of expression of the user’s need (storyboards, mock-ups, interface prototypes) and other requirements, such as regulatory constraints.
- Design set
  This regroups all information describing how the system if to be constructed, to capture decisions about the system to be built, taking into account all of the constraints of time,
budget, legacy, reuse, quality objectives, etc. This comprises a design model, a test model, and other forms of expression of the system nature (prototypes, performance).

- Implementation set
  
  This regroups all information about the elements of software comprising the system: source code in programming language, configuration files, data files, etc., together with the information describing how to assemble the system. This comprises an implementation model.

- Deployment set
  
  This regroups all information about how the software is actually packaged, shipped, installed, and run on the target environment. This may comprise a deployment model.

- Management set
  
  This regroups all information pertaining to plans, schedule, cost, business case, risks, and so on, used to manage the project.

Lifecycle Structure

This is the dynamic organization of the process along time.

The software lifecycle is broken into cycles, each cycle working on a new generation of the product. The Ojectory process divides one development cycle in four consecutive phases:

- Inception
- Elaboration
- Construction
- Transition

Each phase is concluded with a well-defined milestone—a point in time at which certain critical decisions must be made, and therefore key goals must have been achieved.
Inception Phase

During the inception phase, you establish the business case for the system and define the project scope. To accomplish this you must identify all external entities with which the system will interact (actors) and define the nature of this interaction at a high-level. This involves identifying all use cases, and describing a few significant ones. The business case includes success criteria, risk assessment, an estimate of the resources needed, and a phase plan showing dates of major milestones.
At the end of the inception phase, you examine the lifecycle objectives of the project and decide whether or not to proceed with the development.

**Elaboration Phase**

The goals of the elaboration phase are to analyze the problem domain, establish a sound architectural foundation, develop the project plan, and eliminate the highest risk elements of the project. Architectural decisions must be made with an understanding of the whole system. This implies that you describe most of the use cases, and take into account some of the constraints: supplementary requirements. To verify the architecture, you implement a system that demonstrates the architectural choices, and executes significant use cases.

At the end of the elaboration phase, you examine the detailed system objectives and scope, the choice of an architecture, and the resolution of major risks.

**Construction Phase**

During the construction phase, you iteratively and incrementally develop a complete product that is ready to transition to its user community. This implies describing the remaining use case, fleshing out the design, completing the implementation, and testing the software.

At the end of the construction phase, you decide if the software, the sites, and the users are all ready to “go operational.”

**Transition Phase**

During the transition phase you transition the software to the user community. Once the product has been put in the hands of the end users, issues often arise that require additional development to adjust the system, correct undetected problems, or finish some of the features that may have been postponed. This phase typically starts with a “beta release” of the system.
At the end of the transition phase you decide whether the lifecycle objectives have been met, and possibly if you should start another development cycle. This is also a point where you wrap up some of the lessons learned on this project to improve the process.

**Iterations**

Each phase in the Objectory process can be further broken down into iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows incrementally from iteration to iteration to become the final system.

Each iteration within an Objectory process phase results in an executable release of the system.

Each iteration goes through all aspects of software development, that is all process components, although with a different emphasis on each process component depending on the phase.

This is shown in the following diagram, by the intensity of work in each process component as you progress from iteration to iteration across all four phases.
Phases in Objectory consist of iterations, in which all process components are involved. Each process component is responsible for a set of artifacts. The height of the curves indicate the amount of resources.

The main consequence of this iterative approach is that the artifacts described earlier grow and mature over time, as shown in the following diagram.

Information set evolution over the development phases.
4 Objectory Process Components

This chapter describes the organization of the process in terms of its representation: components, workflows, artifacts and workers. It also describes salient Objectory features as being object technology-based, use-case driven, while placing an emphasis on requirements management and architecture.

Process Representation

The process components are described in Rational Objectory Process - Process Manual, in terms of workers, activities and workflows. The accompanying artifacts are described in the Artifact manual.

- A **worker** defines the behavior and responsibilities of an individual, or a set of individuals working together as a team. This is an important distinction because it is natural to think of a worker as the individual or the team itself. In Objectory, the worker is more of a role that defines how the individuals should carry out the work.

- An **activity** is the smallest piece of work that is relevant. For example, it is not reasonable to do only part of an activity. Dividing the work in this manner makes it easier to monitor development. It is better (easier) to know that the project has completed three out of five activities rather than 60% of one activity.

- **Artifacts** are the modeling constructs and documents that activities evolve, maintain, or use as input.

Each worker has a set of associated **cohesive** activities, “cohesive” meaning that the activities are best performed by one individual. The responsibilities of each worker are usually defined relative to certain artifacts, for example documents. Examples of workers are Use-Case Designer, Designer, Architect, and Process Engineer.
Each worker has its own set of activities and artifacts.

Through the associated set of activities, the worker also implicitly defines the abilities expected to perform the work.

Individuals available to a project manager are people with specific sets of abilities.

The project manager will associate these individuals to workers in such a way that a given individual performs a worker’s associated activities and behavior, and is responsible for all the artifacts associated with that worker.
Each individual in the project is assigned to one or several workers.

The association of individual to worker is dynamic over time. It is driven by the activities that need be performed at a given time, and constrained by the availability of individuals with adequate competence. This association has several constants, however:

- An individual may act as several different workers during the same day: Sylvia may be a Design Reviewer in the morning and a Use-Case Designer in the afternoon.
- An individual may act as several workers simultaneously: Jane may be both the Architect and the Designer for a certain class.
- Several individuals may act as the same worker to perform a certain activity together, acting as a team: Both Paul and Mary may be the Use-Case Designer of the same use case.

When planning the various process components and activities, the project manager should try to allocate the individuals in a seamless fashion, that is, minimize as much as possible the hand-off of artifacts from one individual to another.

For example, the Designer of a class can also be its Implementer. For a given use case, the same individual can
act as the Use-Case Specifier and as the Use-Case Designer. This "seamlessness" is again constrained by the range of each individual’s abilities.

Process Components and Models

As explained earlier, a process component has an associated set of activities and artifacts. The most important artifacts are the models that each process component yields: use-case model, design model, implementation model, and test model. The next figure shows the relationship of the process components and models.

![Process Components and Models Diagram]

Each process component is associated with a particular model.

Requirements Capture

The goal of the Requirements Capture process component is to describe what the system should do and allows the developers and the customer to agree on that description. To achieve this, we delimit the system—define its surroundings
and the behavior it is supposed to perform. Customers and potential users are important sources of information as well as any system requirements that may exist.

Requirements capture results in a use-case model and some supplementary requirements. The use-case model is essential for both the customer, who needs the model to validate that the system will become what he expected, and for the developers, who need the model to get a better understanding of the requirements on the system.

The use-case model is relevant to all people involved in the project.

The use-case model consists of actors and use cases. Actors represent the users, and any other system that may interact with the system being developed. Actors help delimit the system and give you a clearer picture of what it is supposed to do.

Use cases represent the behavior of the system. Because use cases are developed according to the actor’s needs, the system is more likely to be relevant to the users. The following figure shows an example of a use-case model for a recycling-machine system.

An example of a use-case model with actors and use cases.

Each use case is described in detail. The use-case description shows how the system interacts step by step with the actors and what the system does.
The use cases function as a unifying thread throughout the system’s development cycle. The same use-case model is used during requirements capture, analysis & design, and test.

Workers and Workflow

Each worker is responsible for a set of activities and behavior. Each process component has its own set of workers and workflow, a logical way in which worker activities progress. The following workers are defined for requirements capture:

- Use-Case Model Architect
- Use-Case Specifier
- Requirements Reviewer
- Architect

The following figure is an overview of the workflow in requirements capture. See *Rational Objectory Process - Process Manual* for more details.

The workflow in requirements capture, shown in terms of workers and their activities. The arrows indicate a logical order between the activities.
The goal of the Analysis & Design process component is to show how the system will be realized in the implementation phase. You want to build a system that:

- Performs—in a specific implementation environment—the tasks and functions specified in the use-case descriptions.
- Fulfills all its requirements.
- Is structured to be robust (easy to change if and when its functional requirements change).

The use-case model is the basis for design, along with the supplementary specifications.

Analysis & Design results in a design model that serves as an abstraction of the source code; that is, the design model acts as a “blueprint” of how the source code is structured and written. Design also results in “inside-view” descriptions of the use cases, or use-case realizations, which describe how the use cases are realized in terms of the participating objects/classes.

The design model consists of design classes structured into design packages; it also contains descriptions of how objects of these design classes collaborate to perform use cases. The next figure shows part of a sample design model for the recycling-machine system in the use-case model shown in the previous figure.
Part of a design model with communicating design classes, and package group design classes.

The design activities are centered around the notion of architecture. The production and validation of this architecture is the main focus of early design iterations. Architecture is represented by a number of architectural views. These views capture the major structural design decisions. In essence architectural views are abstractions or simplifications of the entire design, in which important characteristics are made more visible by leaving details aside. The architecture is an important vehicle not only for developing a good design model, but also for increasing the quality of any model built during system development.

**Workers and Workflow**

The following workers are defined in analysis & design:

- Architect
- Use-Case Designer
- Designer
- Design Reviewer

The following figure gives an overview of the workflow in analysis & design. The workflow is divided into architecture-
level design, and class-level design. See *Rational Objectory Process - Process Manual* for more details.

The workflow in analysis & design, described in terms of workers and their activities. The arrows indicate a logical flow between the activities.

**Implementation**

The system is realized through implementation producing the sources (source-code files, header files, makefiles, and so on) that will result in an executable system. The sources are described in an *implementation model* that consists of modules structured into implementation packages. The design model is the basis for implementation.

Implementation includes testing the separate classes and/or packages, but not testing that the packages/classes work together. That is described in the next process component, “Test”.

---

Rational Objectory Process - Introduction 23
Workers and Workflow

The following workers are defined in implementation:

- Architect
- System Integrator
- Implementer
- Code Reviewer

The following figure provides an overview of the workflow in implementation. The workflow spans activities from defining the implementation view, to implementing classes, to planning and performing integration. See *Rational Objectory Process - Process Manual* for more details.
Test verifies the entire system. You first test each use case separately to verify that its participating classes work together correctly. Then you test (certain aspects of) the system as a whole with use-case descriptions as input to this test. At the end of test, the system can be delivered.

Workers and Workflow

The following workers are defined in test:

- Test Designer
- Integration Tester
- System Tester
- Designer
- Implementer

The following figure provides an overview of the workflow in test. The workflow spans activities from planning, designing, implementing and executing test procedures. See *Rational Objectory Process - Process Manual* for more details.
The workflow in test, shown in terms of workers and their activities. The arrows indicate a logical order between the activities.
5 Features of Objectory

This section explains the core ideas behind the Objectory process. Reading it should help you understand the reason for the Objectory process concepts and constructs. This, in turn, should help you use the Objectory process more effectively.

Object Technology

Many projects today employ object-oriented programming languages to obtain reusable, change-tolerant, and stable systems. To obtain these benefits, it is even more important to use object technology in design. Objectory produces an object-oriented design model that is the basis for implementation.

An object-oriented model aims at reflecting the world we experience in reality. Thus, the objects themselves often correspond to phenomena in the real world that the system is to handle. For example, an object can be an invoice in a business system, or an employee in a payroll system.

A model, correctly designed using object technology is:

- *Easy to understand*. It clearly corresponds to reality.
- *Easy to modify*. Changes in a particular phenomenon concern only the object that represents that phenomenon.

The use of object technology is described in more detail in *Rational Objectory Process - Modeling Guidelines*.

Use-Case-Driven Development

It is often difficult to tell from a traditional object-oriented system model how a system does what it is supposed to do. This difficulty stems from the lack of a “red thread” through the system when it performs certain tasks. In Objectory, *use cases* are that thread because they define the behavior performed by a system (see appendix A, *What is a Use Case?*, for a more precise definition of use case). Use cases are not...
part of “traditional” object orientation, but their importance has become more and more apparent. Other object-oriented methods provide use cases, but use different names for them.

Use Cases in the Development Process

Chapter 2 explained that one advantage of Objectory is its “use-case driven approach.” What is meant by that is that use cases defined for a system are the basis for the entire development process.

Use cases play a role in each of the four engineering process components: requirements capture, analysis & design, implementation, and test.

- The use-case model is a result of requirements capture. In this early process you need the use cases to model what the system should do from the user’s point of view. Thus, use cases constitute an important fundamental concept that must be acceptable to both the customer and the developers of the system.

- In analysis & design use-case descriptions are used to develop a design model. This model describes, in terms of design objects, the different parts of the implemented system, and how the parts should interact to perform the use cases.

- During implementation the design model is the implementation specification. Because use cases are the basis for the design model, they are implemented in terms of design classes.

- During test the use cases constitute test cases. That is, the system is verified by performing each use case.

Notice that a use case has several descriptions. For each use case there is a Use-Case Description, which describes what the system should do from the user’s point of view; there is a Use-Case Realization, which describes how the use case is performed in terms of interacting objects.

Use cases have other roles as well:

- They are used as a basis for planning an iterative development.
• They form a foundation for what is described in user manuals.
• They can be used as ordering units. For example, a customer can get a system configured with a particular mix of use cases.

Controlled Iterative Development

The Objectory iterative approach is generally superior to a linear or waterfall approach for several reasons:

• *It lets you take into account changing requirements.* The reality is that normally requirements will change. Requirements change and requirements “creep” have always been a primary source of project trouble, leading to late delivery, missed schedules, unsatisfied customers, and frustrated developers. As Fred Brooks wrote 25 years ago, “Plan to throw one away, you will anyhow.”

• *Elements are integrated progressively—integration is not one “big bang” at the end.* The Objectory iterative approach is almost a continuous integration. What used to be a long, uncertain, and troublesome time, taking up to 40% of the total effort at the end of a project, is now broken into six to nine smaller integrations that begin with fewer elements.

• *It lets you mitigate risks earlier, because integration is generally the only time risks are discovered or addressed.* As you unroll the early iterations you go through all process components, exercising many aspects of the project: tools, off-the-shelf software, people skills, and so on. Some perceived risks may prove not to be risks, and new, unsuspected risks may emerge.

• *It provides management with a way to do tactical changes to the product; for example, to compete with existing products.* You can decide to release a product with reduced functionality earlier to counter a move by a competitor, or you can adopt another vendor for a given technology.

• *It facilitates reuse, because it is easier to identify common parts as they are partially designed or implemented, instead of identifying all commonality up front.* Identifying and developing reusable parts is hard. Design reviews in early
iterations allow architects to identify unsuspected potential reuse, and develop and mature common code in subsequent iterations.

• *It results in a more robust architecture because you are correcting errors over several iterations.* Flaws are detected even in the early iterations as the product moves beyond inception. Performance bottlenecks are discovered at a time when they can still be addressed, instead of on the eve of delivery.

• *Developers learn along the way, and the various competencies and specialties are more fully employed during the whole life cycle.* Testers start testing early, technical writers write early, and so on. In non-iterative development, the same people would be waiting around to begin their work, making plans and honing their skills. Training needs, or the need for additional (perhaps external) help is spotted early on, during assessment reviews.

• *The process itself can be improved, and refined along the way.* The assessment at the end of an iteration not only looks at the status of the project from a product and schedule perspective, but also analyzes what should be changed in the organization and in the process itself to perform better in the next iteration.

A customer once said: “With the waterfall approach, everything looks fine until near the end of the project, sometimes up until the middle of integration. Then everything falls apart. With the iterative approach, it is very difficult to hide the truth for very long.”

Project managers often resist the iterative approach, seeing it as a kind of endless “hacking.” In Objectory, the interactive approach is very controlled; iterations are planned, in number, duration, and objective. The tasks and responsibilities of the participants are defined. Objective measures of progress are captured. Some rework does take place from one iteration to the next, but this, too, is carefully controlled.
The two key elements behind a controlled iterative process are requirements management and change control. Requirements management is a systematic approach to eliciting, organizing, communicating, and managing the changing requirements of a software-intensive system or application.

The benefits of effective requirements management include:

- **Better control of complex projects**
  Lack of understanding of the intended behavior as well as “requirements creep” are common factors in out-of-control projects.

- **Improved software quality and customer satisfaction**
  The fundamental measure of quality is “does this system do what it is supposed to do?” This can be assessed only when all stakeholders have a common understanding of what must be built and tested.

- **Reduced project costs and delays**
  Errors in requirements are very expensive to fix; decreasing these errors early in the development cycle cut projects costs and schedules.

- **Improved team communication**
  Requirements management facilitates early involvement of users to ensure that the application meets their need; well-managed requirements build a common understanding of the project needs and commitments among all stakeholders: users, customers, management, designers, testers.

Focused more closely towards the needs of the development organization, change control is not only a systematic approach to managing changes in requirements, design, implementation, but also covers the important activities of keeping track of defects, misunderstandings, project commitments, and being able to associate these with specific artifacts and releases.
5  Features of Objectory

A Strong Emphasis on Architecture

Use cases drive the Objectory process end-to-end over the whole lifecycle, but the design activities are centered around the notion of architecture—system architecture, or for software-intensive systems, software architecture. The main focus of the early iterations of the process—mostly in the elaboration phase—is to produce and validate a software architecture, which in the initial development cycle takes the form of an executable architectural prototype that gradually evolves to become the final system in later iterations.

For an introduction to the notion of architecture—and more specifically software architecture—and an explanation of why this notion is crucial, see Appendix C, What Is Software Architecture?

The Objectory process provides a methodical, systematic way to design, develop and validate an architecture. It offers templates for architectural description around the concepts of multiple architectural views, and the capture of architectural style, design rules, and constraints. The design process component contains specific activities aimed at identifying architectural constraints and, architecturally significant elements, as well as guidelines on how to make architectural choices. The management process shows how the planning of the early iterations takes into account the design of an architecture and the resolution of the major technical risks. See Rational Objectory Process - Project Management and all activities associated with the worker ‘Architect’ in Rational Objectory Process - Process Manual.

Architecture is important for several reasons:

- It lets you gain and retain intellectual control over the project, to manage its complexity, and to maintain system integrity.

A complex system is more than the sum of its parts, more than a succession of small independent tactical decisions. It must have some unifying coherent structure to organize those parts systematically, and provide precise rules on how to grow the system without having its complexity “explode” beyond human understanding.
The architecture establishes the means for improved communication and understanding throughout the project by establishing a common set of references, a common vocabulary with which to discuss design issues.

- **It is an effective basis for large-scale reuse.**
  By clearly articulating the major components and the critical interfaces between them, an architecture lets you reason about reuse, both internal reuse—the identification of common parts—and external reuse—the incorporation of ready made, off-the-shelf components. But it also allows reuse on a larger scale: the reuse of the architecture itself in the context of a line of products that addresses different functionality in a common domain.

- **It provides a basis for project management.**
  Planning and staffing are organized along the lines of major components. Fundamental structural decisions are taken by a small, cohesive architecture team; they are not distributed. Development is partitioned across a set of small teams each responsible for one or several parts of the system.

### Component-Based Development

A software component can be defined as a nontrivial piece of software, a module, a package, or a subsystem, that fulfills a clear function, has a clear boundary and can be integrated in a well-defined architecture. It is the physical realization of an abstraction in your design.

Components come from different avenues:

- In defining a very modular architecture, you identify, isolate, design, develop, and test well-formed components. These components can be individually tested and gradually integrated to form the whole system.

- Furthermore, some of these components can be developed to be reusable, especially the components that provide common solutions to a wide range of common problems. These reusable components, which may be larger than just collections of utilities or class libraries, form the basis of
reuse within an organization, increasing overall software productivity and quality.

- More recently the advent of commercially successful component infrastructures such as CORBA, the Internet, ActiveX or JavaBeans, triggers a whole industry of off-the-shelf components for various domains, allowing you to buy and integrate components rather than developing them all in-house.

The first point exploits the old concepts of modularity, and encapsulation, bringing the concepts underlying object-oriented technology a step further. The last two points shift software development from programming software (a line at time) to composing software (by assembling components).

Objectory supports component-based development in several ways.

- The iterative approach allows you to progressively identify components, decide which one to develop, which one to reuse, and which one to buy.
- The focus on software architecture allows you to articulate the structure: the components and the ways in which they integrate: the fundamental mechanisms and patterns by which they interact.
- Concepts such as packages, subsystems, and layers are used during analysis & design to organize components, and specify interfaces.
- Testing is organized around components first, then gradually larger sets of integrated components.

**Process Configurability**

The Rational Objectory Process is general and complete enough to be used by a wide-range of software development organizations. In many circumstances, this software engineering process will need to be modified, adjusted, extended and tailored to accommodate the specific characteristics, constraints and history of the adopting organization.
The process elements that are likely to be modified, customized, added or suppressed include: artifacts, activities, workflows, workers and process components. See *Rational Objectory Process - Process Configuration* manual for more details.
6 Getting Started with Objectory

The Objectory product consists of:

- A set of manuals that describe the process.
- An online version of the Objectory process description in HTML.

The content of the manuals, and online description are the same, except for a few diagrams.

Additional supporting material consists of templates for reports and documents, tool wizards, programming guidelines.

Training on the Objectory process can be ordered separately.

Objectory Manuals

Each manual has its own purpose, but relates strongly to the others. At times, basic information is duplicated in several manuals both for clarity and for the convenience of the reader. If you do not understand a concept in one manual, another manual may treat it in more depth. Cross-references are provided.
The Rational Objectory process manuals.

- *Rational Objectory Process - Introduction* explains what you need to know before you start work with the Objectory process.

- *Rational Objectory Process - Process Manual* contains the guidelines you need when working with the Objectory process. It explains what to do and how and when to do it. It contains descriptions of the activities, workflows and workers.

- *Rational Objectory Process - Artifacts* contains descriptions of the elements of information that are input or output of the process, and how this information can be packaged in documents and reports.

detailed advice on modeling elements: form, identification, quality.

- *Rational Objectory Process - Process Configuration* explains how to adapt the process to a specific project. To adapt the process, you decide which parts of the process (activities, modeling elements, documents) to use, and what individuals will do what activities.


Objectory Online

Objectory online makes the contents of the manuals possible to view with any of the popular web browsers and support frameworks, such as Netscape Navigator™ and Microsoft Internet Explorer™. Objectory online has a lot of hypertext links and interactive images for the user to browse.

You can view Objectory online.

Learning the Process

This section gives guidelines on how to learn the process by reading the manuals, either with hard copy or online. Remember that one of the most effective ways of learning something is to actually do it. So try to combine your reading with some practical work.
To assist you in the learning process, there are reading “assignments” for the following key project roles:

- **Project manager**, the manager of the project.
- **Process engineer**, the person who configures or tailors the process to be suitable for the project organization. A person acting as a process engineer is usually responsible for the methods in the project.
- **Developer**, which comprises the various workers (use-case designer, architect, and so on).

Guide for Project Managers

**Learning goals:** To learn the fundamentals of Objectory and get a deeper insight into how to manage a project, using a controlled iterative process.

1. Read *Rational Objectory Process - Introduction*, Chapters 1 through 6 to get acquainted with fundamental concepts, and gain an understanding of Objectory’s features. Chapters 3 and 4 are important in understanding how the process is organized.

2. Read all of *Rational Objectory Process - Project Management*.


Guide for Process Engineers

Learning goals: To learn the fundamental concepts of the Objectory process, and gain a deeper understanding of how to configure or tailor the process.

1. Read *Rational Objectory Process - Introduction*, Chapters 1 through 6 to get acquainted with the fundamental concepts, and gain an understanding of Objectory's features.


3. Read *Rational Objectory Process - Project Management*, Chapters 2 through 5 to get an understanding of how configuring the process fits into a project's lifecycle.


To be able to configure the process, you need a deeper understanding of the process components you are going to configure.

1. To understand requirements capture,
understand how the work is performed. You can decide later which steps to perform and in what order.

- Read *Rational Ojectory Process Artifacts*, Chapter 4, *Requirements Capture*, to understand what artifacts and documents to produce during requirements capture.

- Read *Rational Ojectory Process - Process Manual*, Chapter 5, *Workers in Requirements Capture*, to understand what each worker should do. An important part of configuring is to decide if you should use the set of worker suggested here.

2. To understand analysis & design,


- Read *Rational Ojectory Process - Modeling Guidelines*, Chapter 2, *Guidelines for Analysis & Design*, to understand the notation used in analysis & design (that is, in the design model).

- Read *Rational Ojectory Process - Process Manual*, Chapter 7, *Workflow in Design*, to understand how the work is performed. You can decide later which steps to perform and in what order.


- Read *Rational Ojectory Process - Process Manual*, Chapter 9, *Workers in Design*, to understand what each worker should do. An important part of configuring is to decide if you should use the set of worker suggested here.
3. To understand implementation,
   - Read *Rational Objectory Process - Modeling Guidelines*, Chapter 3, *Guidelines for Implementation*, to understand the notation used in implementation (that is, in the implementation model).
   - Read *Rational Objectory Process - Process Manual*, Chapter 11, *Workflow in Implementation*, to understand how the work is performed. You can decide later which steps to perform and in what order.
   - Read *Rational Objectory Process - Process Manual*, Chapter 13, *Workers in Implementation*, to understand what each worker should do. An important part of configuring is to decide if you should use the set of worker suggested here.

4. To understand test,
   - Read *Rational Objectory Process - Modeling Guidelines*, Chapter 4, *Guidelines for Testing*, to understand the notation used in test (that is, in the test model).
work is performed. You can decide later which steps to perform and in what order.


- Read *Rational Objectory Process - Process Manual*, Chapter 17, *Workers in Test*, to understand what each worker should do. An important part of configuring is to decide if you should use the set of worker suggested here.

**Guide for Developers**

**Learning goals:** To learn the fundamental concepts of the Objectory process and gain a deeper understanding of the part(s) of the process you work with:

1. Read *Rational Objectory Process - Introduction*, Chapters 1 through 6 to get acquainted with the fundamental concepts, and gain an understanding of Objectory’s features.

To be able to work within one or several process components, you need a deeper understanding of them. Each of the references below is to *Rational Objectory Process - Process Manual*, unless otherwise specified.

1. To learn about requirements capture,

   - Read Chapter 2, *Introduction to Requirements Capture*, for an overview of the requirements capture.

   - Read Chapter 3, *Workflow in Requirements Capture*, to get an overview of the workflow in requirements capture.

   - Read about the activities in detail in the sections of Chapter 4, *Activities in Requirements Capture*. Read Chapter 5, *Workers in Requirements Capture*, to help you get acquainted with a particular worker.

   - Read *Rational Objectory Process - Modeling Guidelines*, Chapter 1, *Guidelines for Requirements Capture*, as a reference to understand the notation used in requirements capture.
• Read *Rational Objectory Process - Artifacts*, Chapter 4, *Requirements Capture*, as references to understand what artifacts to produce and how to document results in requirements capture.

2. To learn about analysis & design,
   • Read Chapter 6, *Introduction to Analysis & Design*, to get an overview of design.
   • Read Chapter 7, *Workflow in Analysis & Design*, to get an overview of the workflow in analysis & design.
   • Read about the activities in detail in the subsections of Chapter 8, *Activities in Analysis & Design*. Read Chapter 9, *Workers in Analysis & Design*, to help you get acquainted with the activities and responsibilities associated with a particular worker.
   • Read *Rational Objectory Process - Artifacts*, Chapter 5, *Analysis & Design*, to understand what artifacts to produce, and how to document results in analysis and design.

3. To learn about implementation,
   • Read Chapter 11, *Introduction to Implementation*, to get an overview of implementation.
   • Read Chapter 12, *Workflow in Implementation*, to get an overview of the workflow in implementation.
   • Read about the activities in detail in the subsections of Chapter 12, *Activities in Implementation*. Read Chapter 13, *Workers in Implementation*, to help you get acquainted with the activities and responsibilities associated with a particular worker.
   • Read *Rational Objectory Process - Artifacts*, Chapter 6, *Implementation*, to understand what artifacts to
produce, and how to document results in implementation.

4. To learn about test,

- Read Chapter 14, *Introduction to Test*, to get an overview of test.
- Read Chapter 15, *Workflow in Test*, to get an overview of the workflow in test.
- Read about the activities in detail in the subsections of Chapter 16, *Activities in Test*. Read Chapter 17, *Workers in Test*, to help you get acquainted with the activities and responsibilities associated with a particular worker.

Guide for Projects

To successfully install a software-engineering process in a project (or entire organization), you must at the minimum consider the following issues:

- *Management support*. Without appropriate management support, it is hard to introduce a new process.
- *Training project members*. Every project member must understand the new process.
- *Configuring the process to suit the project’s needs*. Configuration must consider such factors as the problem domain, what kind of application, the company culture, and supporting tools. See *Rational Objectory Process - Process Configuration* for more details.
- *Supporting tools*. A software-engineering process requires tools to support all the activities in a system’s lifecycle. You need to select a set of supporting tools and tailor them to work with each other, as well as within the process.

It is important that management stand behind the introduction of a new process, and possibly a new tool set.
Management must be fully aware of what it takes to introduce a new process:

- **There are always initial costs associated with introducing a new process.** You need to train your staff, and possibly pay for mentoring. You may experience a lower productivity in your first project, at least if you include necessary training as a part of the project. It may be wise to exclude training from the initial project to get a fairer comparison with other projects. Over time you will see a productivity gain; experience shows that you can expect to see increased quality from the first project.

- **Most people like doing things in ways they are familiar with, especially when under stress.** This means that you can expect the staff to fall back on old procedures, especially when the pressure is extreme. As a manager, you must be prepared for this. Encourage team members to stick to the process. You may even find yourself falling back to old procedures when you see a problem. Try to go through at least one full project if you possibly can and then evaluate what went well, what did not go well, and what can be improved.

- **There are always risks when you introduce a new process, but there are also great rewards.** One way to reduce risks is to provide the appropriate training and to run a pilot project, which is discussed in more detail in the following paragraphs. Another way to reduce risk is to have skilled mentors assist you. They should be able to identify and eliminate some of the issues before they become costly problems. They also typically increase productivity by helping you to avoid pitfalls.

It is strongly recommended that you introduce the process in a small project. A project goal should be that people understand the process and can work with it. Another should be to develop a process configuration (define how the process should be used). It is also recommended that you train people in an environment that is as realistic as possible.

There are several ways to accomplish these goals, all of which are a mixture of training, mentoring workshops, and project work:
• Apply the process to a pilot project, in which you work on a fictitious prototype example. The goal is to learn the process without affecting the actual product.

• Apply the process only to the first iteration of a project. In which case, you would be working on parts of the product, but there would still be ample time to fix anything that goes wrong.

• Conduct a process “boot camp.” The goal is to let developers work through all (or most) phases of the development process in the shortest possible time. The system could either be a fictitious prototype or part of the real system. The advantage of a “boot camp” is that people are focused on learning. The goal is primarily to get through all phases and learn as much as possible, thereby gaining enough experience to be able to configure the process to meet a particular set of needs.

When you plan training, remember to deliver it when it is needed. This will maximize the motivation and capacity to learn. Deliver design training when people are about to start working with design, for example.
Supporting Tools

A software-engineering process requires tools to support all activities in a system’s lifecycle. An iterative development process puts special requirements on the tool set you use, such as better integration among tools and round-trip engineering between models and code. You also need tools to automate documentation, and possibly automate tests to make regression testing easier. The Objectory process can be used with a variety of tools, either from Rational or other vendors. However, Rational provides many well-integrated tools that efficiently support the Objectory process.

Below are listed the tools you will require, and some examples of tools Rational has to meet these needs.

- **A modeling tool** to develop the different models, such as use-case model and design model. The tool should have true round-trip engineering so that you can forward-engineer and reverse-engineer code without overriding the changes you have made in the models or code since the last generation.
  
  Example: Rational Rose

- **A requirements management tool**, to capture, organize, prioritize, trace all requirements.
  
  Example: Rational Requisite Pro

- **A documentation tool**, to support the project documentation. You need to be able to extract information from the modeling tool, as well as other sources such as the code, to create documents that present the models. If you do not have automated document generation, you will most likely have documentation that diverts from your models (or no documentation at all). A documentation tool must allow you to make changes manually in a document, and not override these changes when you regenerate documentation.
  
  Example: Rational SoDA
• *Programming tools*, to assist the developers: editors, compilers, debuggers, and so on. These should be integrated with the modeling environment and the test environment.

Example: Rational Apex/Ada, Rational Apex/C++ (Java ready)

• Tools that support the *project manager* in planning and steering the project.

• In a large project, you need to automate the allocation of a task to the individuals on the project. A *task management tool* can help project managers know what tasks to work on, enforce conformance to a certain process, and at the same time automate tedious tasks related to configuration management. It may also help project managers to continuously monitor the project’s progress.

Example: Rational Summit or ClearGuide

• A *configuration management tool*, can help you keep track of all artifacts produced, and their different versions. Both models, and especially code needs to be configuration-managed. Integration of coding environments, modeling tools, and configuration management tools is essential.

Example: Rational Apex/CMVC or ClearCase.

• In an iterative development process you test throughout the lifecycle. It is important that you use *test tools* to automate testing so that you can easily retest code (regression testing) to minimize resources and maximize quality. More specialized tools allow you to perform load testing.

Examples: Rational SQA Suite, Rational TestMate, Rational Visual Test
Appendix A

Key Modeling Concepts

This appendix introduces the concepts of actors, use cases, object-oriented concepts, and the general idea of models. More detailed definitions are included in *Rational Objectory Process - Modeling Guidelines*. Use cases have a special interest and are further explained in *Appendix B, What Is a Use Case?*. These concepts are vital to understanding the Objectory process. If you are already familiar with them, you need not read this section.

Actors

To fully understand the purpose of a system you have to know *who* the system is for, and *who* will use it. In Objectory, the different types of users are represented by *actors*. Also, any other system that interacts with the system is represented by an actor; thus actors delimit the system.

Use Cases

The functionality of a system is defined by *use cases*, each of which represents a specific way to use the system. The description of a use case defines what happens in the system when the use case is performed. Thus, a use case corresponds to a sequence of actions performed by the system, which yields an observable result of value to a particular actor. An *action* is an atomic set of activities, which is performed either entirely, or not at all.
In an automated teller machine, the client can, for instance, withdraw money from an account, transfer money to an account, or check the balance of an account. These operations correspond to flows of events in the system, which can be represented by use cases.

Objects

An object is an abstraction of something, in the domain of a problem domain, or in a system’s implementation. For instance, an object can be an invoice in a business system, or an employee in a payroll system.

An object encapsulates data and behavior, which is different from the traditional separation of functions and data. The object’s data is represented by attributes; its behavior by operations.

An object is defined by a class, which is described in the next section.

Classes

A class defines a template for the structure and behavior of all its objects. The objects created from a class are also called the instances of the class.
Generalization

A generalization shows that one class inherits from another. The inheriting class is called a descendant. The class inherited from is called the ancestor. Inheritance means that the definition of the ancestor—including any properties such as attributes, relationships, or operations on its objects—is also valid for objects of the descendant. The generalization is drawn from the descendant class to its ancestor class.

```
Deposit Item

<table>
<thead>
<tr>
<th>can</th>
<th>bottle</th>
<th>crate</th>
</tr>
</thead>
<tbody>
<tr>
<td>height : mm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight : kg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The classes Can, Bottle, and Crate have common properties height and weight. Each is a specialization of the general concept Deposit Item.

Models

Model building is widely accepted by all engineering disciplines, largely because it affords a look at the system before it is actually constructed. Models let you see early on problems that would be extremely expensive—if not impossible—to fix later.

In Objectory, you use several models of the system, because the system has several stakeholders, each with a different interest. When you discuss the system and its requirements, for example, the focus is on the end user, or customer, so the model must describe what the system should do, and
abstrats the implementation details. Later, during design, the model must focus on the needs of the designers who must be able to reason about decomposition, abstraction, and hierarchy at a level above the programming language.

Implementation and test also require different model characteristics relevant to the implementers and testers. Thus, as you move through the software lifecycle, you will evolve several models that describe different aspects of the system you are developing.
Appendix B

What Is a Use Case?

This appendix gives a more thorough discussion of the use-case concept. The text is an extract from Rational Objectory Process - Process manual.

Symbol

Definition

A use-case instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.

A use-case type defines a set of use-case instances.

There are several key words in this definition:

- **Use-case instance.** The sequence referred to in the definition is really a specific flow of events through the system, or an instance. Many flow of events are possible, and many may be very similar. To make a use-case model understandable, you should group similar flows of events into a use-case type. When we say that we identify and describe a use case, we really mean that we identify and describe the type.

- **System performs.** When we say a system “performs actions,” we mean that the system provides the use case. An actor communicates with a use case instance of the system.
- *An observable result of value.* You can put a value on a successfully performed use case. A use case should make sure that an actor can perform a task that has an identifiable value. This is very important in determining the correct level/granularity for a use case. Correct level refers to achieving use cases that are not too small. In certain circumstances, you can then use a use case as a planning unit in the organization that includes the actor.

- *Actions.* An action is a computational or algorithmic procedure; it is invoked either when the actor provides a signal to the system or when the system gets a time event. An action may imply signal transmissions to either the invoking actor or other actors. An action is atomic, i.e. it is performed either entirely or not at all.

- *A particular actor.* The actor is key to finding the correct use case, especially because the actor helps you avoid use cases that are too large. It is important to begin with individual (human) actors, or instances of actors. It is a good idea when determining suitable actors to name at least two or, if possible, three people who would be able to perform as the individual actor. Suppose you are developing an object-oriented modeling tool. There are really two actors in this example: a developer, someone who develops systems using the tool as support, and a system administrator, someone who manages the tool. Each of these actors has his own demands on the system, and will therefore require his own set of use cases.

**Explanation**

The functionality of a system is defined by different use cases, each of which represents a specific flow of events. The description of a use case defines what happens in the system when the use case is performed.
In an automated teller machine the client can, for instance, withdraw money from an account, transfer money to an account, or check the balance of an account. These operations correspond to flows, which you can represent by use cases.

Each use case has a task of its own to perform. The collected use cases constitute all the possible ways of using the system. You can get an idea of the task of a use case simply by observing its name.

Example:

A telephone call is an example of a use case in a switching system, as the figure below illustrates:

The use case *Telephone Call* begins when the party who initiates the call lifts her receiver and goes on until both parties have put down their receivers.
The Scope of a Use Case

It is often hard to decide if a set of user-system interactions, or *dialog*, is one or several use cases. Consider the use of a recycling machine. The customer inserts deposit items, such as cans, bottles, and crates, into the recycling machine. When she has inserted all her deposit items, she presses a button, and a receipt is printed. She can then exchange this receipt for money.

Is it one use case to insert a deposit item and another use case to require the receipt? Or is it all one use case? There are two actions, but one without the other is of little value to the customer. Rather it is the complete dialog with all the insertions and getting the receipt back, that is of value for the customer (and makes sense to her). Thus, the complete dialog, from inserting the first deposit item, to pressing the button and getting the receipt, is a complete case of use, a use case.

Additionally, we want to keep the two actions together, to be able to review them at the same moment, modify them together, test them together, write manuals for them and in general manage them as a unit. Especially in larger systems this becomes very obvious.

A Use Case Defines What the System Does

A use-case description describes *what happens* in the system, and how the system interacts with the actors when the use case is executed. The use-case description does not define *how* tasks are performed in the system. This is left for the object models to deal with.

Example:

In the telephone example (see above) the use-case description would indicate—among other things—that the system issues a signal when the receiver is lifted and that the system then receives digits, finds the receiving party, rings his telephone, connects the call, transmits speech, and so on.
In an executing system, an instance of a use case does not correspond to any particular object in the implementation model (for example, an instance of a class in the code). Instead, it corresponds to a specific flow that is invoked by an actor and executed as a sequence of events in the implemented objects. In other words, instances of use cases correspond to communicating instances of implemented objects. Often, the same objects participate to implement several use cases. For example, both the use cases Deposit and Withdrawal in a banking system may use a certain account object in their implementation. This does not mean that the two use cases communicate, only that they use the same object in their implementation.

You can view a flow as several subflows, which taken together yield the total flow. You can reuse the description of a subflow in use-case descriptions. Subflows in the description of one use case’s flow of events may be common to those of other use cases. In the design you should have the same objects perform this common behavior for all the relevant use cases; that is, only one set of objects should perform this behavior no matter which use case is executing. If the behavior is common to more than two use cases or forms an independent part, the model might be clearer if you model the behavior as a use case of its own used by the original use cases. See also Rational Objectory Process - Modeling Guidelines.

Example:

In an automated teller machine system the initial subflow is the same in the flows of the use cases Withdraw Money and Check Balance. The use cases’ flows start by checking the identity of the card and the client’s personal access code.

Use Case Types Have Many Possible Instances

A use-case instance can follow an almost unlimited, but enumerable, number of paths. These paths represent the choices open to the use-case instance in the description of its flow of events. The path chosen depends on events. Types of events include
What Is a Use Case?

- *Input from an actor.* For example, an actor can decide, from several options, what to do next.

  Example:

  In the use case *Recycle Items* in the *Recycling-Machine System* the *Customer* always has two options: hand in still another deposit item or get the receipt of returned items.

- *A check of values or types of some internal object or attribute.* For example, the flow of events may differ if a value is greater or less than a certain value.

  Example:

  In the use case *Withdraw Money* in an automated teller machine system, the flow of events will differ if the *Client* asks for more money than he has in his account. Thus, the use-case instance will follow different paths.

Instances of several use cases and several instances of the same use case work concurrently if the system permits it. In use-case modeling, you can assume that instances of use cases can be active concurrently without conflict. The design model is expected to solve this problem, since use-case modeling does not describe how things work. One way to view this is to assume that only one use-case instance is active at a time and that executing this instance is an atomic action. In use-case modeling, we consider the “interpreting machine” as infinitely fast, so that serialization of use case instances is not a problem.
Appendix C

What Is Software Architecture?

This appendix discusses software architecture, and how it is applied to Objectory. It is intended to deepen your understanding of how architectural concerns can affect a project.

Software architecture is a concept that is easy to understand, and that most engineers intuitively feel, especially with a little experience, but it is hard to define precisely. In particular, it is difficult to draw a sharp line between design and architecture—architecture is one aspect of design that concentrates on some specific features.

In *An Introduction to Software Architecture*, David Garlan and Mary Shaw suggest that software architecture is a level of design concerned with issues: “Beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.” [GS93]

But there is more to architecture than just structure; the IEEE Working Group on Architecture defines it as “the highest-level concept of a system in its environment” [IEEE]. It also encompasses the “fit” with system integrity, with economical constraints, with aesthetic concerns, and with style. It is not limited to an inward focus, but takes into consideration the system as a whole in its user environment and its development environment—an outward focus.

In Objectory, the architecture of a software system (at a given point) is the organization or structure of the system’s
significant components interacting through interfaces, with components composed of successively smaller components and interfaces.

Architecture Description

To speak and reason about software architecture, you must first define an architectural representation, a way of describing important aspects of an architecture. In Objectory, this description is captured in the Software Architecture Document.

Architectural Views

We have chosen to represent software architecture in multiple architectural views. Each architectural view addresses some specific set of concerns, specific to stakeholders in the development process: end users, designers, managers, system engineers, maintainers, and so on.

The views capture the major structural design decisions by showing how the software architecture is decomposed into components, and how components are connected by connectors to produce useful forms [PW92]. These design choices must be tied to the requirements, functional, and supplementary, and other constraints. But these choices in turn put further constraints on the requirements and on future design decisions at a lower level.

A Typical Set of Architectural Views

Architecture is represented by a number of different architectural views, which in their essence are extracts illustrating the “architecturally significant” elements of the models. In Objectory, you start from a typical set of views, called the “4+1 view model” [KR95]. It is composed of:

- The Use-Case View, which contains use-cases and scenarios that encompasses architecturally significant behavior, classes, or technical risks. It is a subset of the use-case model.
• The *Logical View*, which contains the most important design classes and their organization into service packages and subsystems, and the organization of these subsystems into layers. It contains also some use case realizations. It is a subset of the design model.

• The *Implementation View*, which contains an overview of the implementation model and its organization in terms of modules into packages and layers. The allocation of packages and classes (from the *Logical View*) to the packages and modules of the *Implementation View* is also described. It is a subset of the implementation model.

• The *Process View*, which contains the description of the tasks (process and threads) involved, their interactions and configurations, and the allocation of design objects and classes to tasks. This view need only be used if the system has a significant degree of concurrency. In Objectory 4.1, it is a subset of the design model.

• The *Deployment View*, which contains the description of the various physical nodes for the most typical platform configurations, and the allocation of tasks (from the *Process View*) to the physical nodes. This view need only be used if the system is distributed.

The architectural views are documented in a *Software Architecture Document*. You can envision additional views to express different special concerns: user-interface view, security view, data view, and so on. For simple systems, you may omit some of the views contained in the 4+1 view model.

**Architectural Focus**

Although the views above could represent the whole design of a system, the architecture concerns itself only with some specific aspects:

• The *structure* of the model—the organizational patterns, for example, layering.

• The *essential elements*—critical use cases, main classes, common mechanisms, and so on, as opposed to all the elements present in the model.
• A few key scenarios showing the main control flows throughout the system.
• The services, to capture modularity, optional features, product-line aspects.

In essence, architectural views are abstractions, or simplifications, of the entire design, in which important characteristics are made more visible by leaving details aside. These characteristics are important when reasoning about
• System evolution—going to the next development cycle.
• Reuse of the architecture, or parts of it, in the context of a product line.
• Assessment of supplementary qualities, such as performance, availability, portability, and safety.
• Assignment of development work to teams or subcontractors.
• Decisions about including off-the-shelf components.
• Insertion in a wider system.

Architectural Patterns

Architectural patterns are ready-made forms that solve recurring architectural problems [BMR96]. An architectural framework or an architectural infrastructure (middleware) is a set of components on which you can build a certain kind of architecture. Many of the major architectural difficulties have been resolved in the framework or in the infrastructure, usually targeted to a specific domain: command and control, MIS, control system, and so on.

Architectural Style

A software architecture, or only an architectural view, may have an attribute called architectural style, which reduces the set of possible forms to choose from, and imposes a certain degree of uniformity to the architecture. The style may be defined by a set of patterns, or by the choice of specific components or connectors as the basic building blocks. For a given system, some of the style can be captured as part of the architectural description in an architecture style guide—part
of a design guidelines document in Objectory. Style plays a major role in the understandability and integrity of the architecture.

Architectural Blueprints

The graphical depiction of an architectural view is called an architectural blueprint. For the various views described above, the blueprints are composed of the following diagrams from the Unified Modeling Language [UML]:

- **Logical view.** Class diagrams, state diagrams, and object diagrams.
- **Process view.** Class diagrams and object diagrams (encompassing task—processes and threads).
- **Implementation view.** Component diagrams.
- **Deployment view.** Deployment diagrams.
- **Use-case view.** Use-case diagrams depicting use cases, actors, and ordinary design classes; sequence diagrams depicting design objects and their collaboration.

The Architecting Process

In Objectory 4.1, the architecture is primarily an outcome of the analysis & design process component. As the project reenacts this process component, iteration after iteration, the architecture evolves, refined, and polished. As each iteration includes integration and test, the architecture is quite robust by the time the product is delivered. This architecture is a main focus of the iterations of the elaboration phase, at the end of which the architecture is normally baselined.
Appendix D

Differences Between Objectory Versions

From Objectory 4.0 to Objectory 4.1

Objectory 4.1 represents a significant improvement over Objectory 4.0 in the following areas:

- Test and Implementation process components are now described in more details and incorporate particular elements from the SQA Process and Apex and Summit concept guides.
- The Analysis & Design process component makes a more thorough usage of the UML notation, and contains more modeling guidelines.
- A separate manual describes all the artifacts generated during process execution.
- Workflows are introduced that describe typical sequencing of activities across process components, at various stages of the development cycle.

From Objectory 3.8 to Objectory 4.0

The Rational Objectory Process version 4.0 released in October 1996 represented a major evolution over Objectory 3.8.

Objectory 3.8 was released in January 1996. It was the ultimate successor of a series of software-engineering processes developed and distributed by Objectory AB between 1987 and 1996. This process finds its origins in works by Ivar Jacobson et al., Object-Oriented Software Engineering, and The Object Advantage.
In October 1995, Rational Software Corp. and Objectory AB merged, and the two companies put together their assets, expertise, and experience. In particular, they merged their ideas on the software-development process which were already very similar, as well as being complementary in some areas.

Objectory 4.0 has inherited from 3.8 its underlying process architecture, the process description method, and the emphasis on use cases throughout the development cycle.

Compared with Objectory 3.8, Objectory 4.0 has some significant differences:

- A stronger emphasis on iterative controlled development.
- A greater emphasis on software architecture and architectural representation.
- The use of a subset of UML as the notation for all modeling elements.
- The use of Rational Rose 4.0 as the principal tool to support modeling activities.
- More detailed project management support.
- The use of HTML for the online process.

But Objectory 4.0 has also been simplified relative to Objectory 3.8 by omitting two process components: Robustness Analysis, and Domain Modeling, and their corresponding artifacts.
Appendix E

Glossary

Here we provide definitions of the most common terms used in the Objectory process. See also the master glossary in the Rational Objectory Process - Process Manual, Appendix B.

abstract Qualifies a class or type that cannot be directly instantiated (opposite of “concrete”).
abstraction The essential characteristics of an entity that distinguish it from all other kind of entities and thus provide crisply-defined boundaries relative to the perspective of the viewer.
activity Describes a piece of work a worker may be asked to perform.
actor Someone or something, outside the system or business that interacts with the system or business.
analysis The part of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses on what to do, design focuses on how to do it. See also “Design.”
analysis & design A process component in the Objectory process, whose purpose is to show how the system’s use cases will be realized in implementation: (general) activities during which strategic and tactical decisions are made to meet the required functional and quality requirements of a system. For the result of analysis and design activities, see “Design Model.”
architectural baseline The baseline at the end of the elaboration phase.
architectural view A view of the system architecture from a given perspective; focuses primarily on structure, modularity, essential components, and the main control flows.
architecture
The highest level concept of a system in its environment [IEEE]. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces.

artifact
A piece of information that is produced, modified, or used by a process, ii) defines an area of responsibility, and iii) is subject to version control. An artifact can be a model, a model element, or a document. A document can enclose other documents.

association
A relationship that models a bi-directional semantic connection among instances.

attribute
An attribute defined by a class represents a named property of the class or its objects. An attribute has a type that defines the type of its instances.

baseline
A reviewed and approved release of artifacts that constitutes an agreed basis for further evolution or development, and that can be changed only through a formal procedure, such as change management and configuration control.

change management
The activity of controlling and tracking changes to artifacts.

class
A class is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.

component
1. (General) An architectural “element,” defined by a set of attributes: name, responsibility, content, constraints, dependencies, interface.

2. A component represents a piece of software code (source, binary or executable), or a file containing information (a startup file or a ReadMe file). A component can also be an aggregate of other components, for example, an application consisting of several executables.
configuration management  | A supporting process whose purpose is to identify, define, and baseline items; control modifications and releases of these items; report and record status of the items and modification requests; ensure completeness, consistency and correctness of the items; and control storage, handling and delivery of the items. (ISO)

construction  | The third phase of the Objectory process, in which the software is brought from an executable architectural baseline to the point at which it is ready to be transitioned to the user community.

customer  | A person or organization, internal or external to the producing organization, who takes financial responsibility for the system. In a large system this may not be the end user. The customer is the ultimate recipient of the developed product and its artifacts.

cycle  | One complete pass through the four phases: inception, elaboration, construction, and transition. The span of time between the beginning of the inception phase and the end of the transition phase.

deliverable  | An output from a process that has a value—material or otherwise—to a customer.

deployment  | A process component in the software-engineering process, whose purpose is to ensure a successful transition of the developed system to its users. Included are artifacts such as training materials and installation procedures.

deployment view  | An architectural view that describes one or several system configurations; the mapping of software components (tasks, modules) to the computing nodes in these configurations.

design  | The part of the software development process whose primary purpose is to decide how the system will be implemented. During design, strategic and tactical decisions are made to meet the required functional and quality requirements of a system. See also Analysis.

design model  | An object model describing the realization of use cases; serves as an abstraction of the implementation model and its source code.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>development case</td>
<td>The software-engineering process used by the performing organization. It is developed as a configuration, or customization of the Objectory process product, and adapted to the project’s needs.</td>
</tr>
<tr>
<td>domain</td>
<td>An area of knowledge or activity characterized by a family of related systems.</td>
</tr>
<tr>
<td>elaboration</td>
<td>The second phase of the Objectory process, in which the product vision and its architecture are defined.</td>
</tr>
<tr>
<td>entity class</td>
<td>A class used to model information that has been stored by the system, and the associated behavior. A generic class, reused in many use cases, often with persistent characteristics. An entity class defines a set of entity objects, which participate in several use cases and typically survive those use cases.</td>
</tr>
<tr>
<td>environment</td>
<td>A supporting process component in the software-engineering process, whose purpose is to define and manage the environment in which the system is being developed. Includes process descriptions, configuration management, and development tools.</td>
</tr>
<tr>
<td>evolution</td>
<td>The life of the software after its initial development cycle; any subsequent cycle, during which the product evolves.</td>
</tr>
<tr>
<td>evolutionary</td>
<td>Qualifies an iterative development strategy that acknowledges that user needs are not fully understood and therefore requirements are refined in each succeeding iteration (elaboration phase).</td>
</tr>
<tr>
<td>generalization (between classes)</td>
<td>A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element can be used where the more general element is allowed.</td>
</tr>
<tr>
<td>generation</td>
<td>Release at the end of a software development cycle.</td>
</tr>
<tr>
<td>implementation</td>
<td>A process component in the software-engineering process, whose purpose is to implement and unit test the classes.</td>
</tr>
<tr>
<td>implementation model</td>
<td>The implementation model is a collection of components, and the implementation subsystems that contain them.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>implementation view</td>
<td>An architectural view that describes the organization of the static software elements (code, data, and other accompanying artifacts) on the development environment, in terms of both packaging, layering, and configuration management (ownership, release strategy, and so on). In Objectory it is a view on the implementation model.</td>
</tr>
<tr>
<td>inception</td>
<td>The first phase of the Objectory process, in which the seed idea, request for proposal, for the previous generation is brought to the point of being (at least internally) funded to enter the elaboration phase.</td>
</tr>
<tr>
<td>increment</td>
<td>The difference (or “delta”) between two releases at the end of two subsequent iterations.</td>
</tr>
<tr>
<td>incremental</td>
<td>Qualifies an iterative development strategy in which the system is built by adding more and more functionality at each iteration.</td>
</tr>
<tr>
<td>inheritance</td>
<td>The mechanism that makes generalization possible; a mechanism for creating full class descriptions out of individual class segments.</td>
</tr>
<tr>
<td>integration</td>
<td>Integration refers to the software development activity in which separate software components are combined into a whole.</td>
</tr>
<tr>
<td>iteration</td>
<td>A distinct sequence of activities with a baselined plan and evaluation criteria resulting in a release (internal or external).</td>
</tr>
<tr>
<td>layer</td>
<td>A specific way of grouping packages in a model at the same level of abstraction.</td>
</tr>
<tr>
<td>logical View</td>
<td>An architectural view that describes the main classes in the design of the system: major business-related classes, and the classes that define key behavioral and structural mechanisms (persistency, communications, fault-tolerance, user-interface). In Objectory, the logical view is a view of the design model.</td>
</tr>
<tr>
<td>management</td>
<td>A supporting process component in the software-engineering process, whose purpose is to plan and manage the development project.</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| **method** | 1. A regular and systematic way of accomplishing something; the detailed, logically ordered plans or procedures followed to accomplish a task or attain a goal.  
2. (UML) The implementation of an operation. The algorithm or procedure that effects the results of an operation. |
| **milestone** | An event held to formally initiate or conclude an iteration. |
| **model** | A semantically closed abstraction of a system. In Objectory, a complete description of a system from a perspective ('complete' meaning you don’t need any additional information to understand the system from that perspective); a set of modeling elements. Two models cannot overlap. |
| **object** | An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations and methods. An object is an instance of a class. |
| **object model** | An abstraction of a system’s implementation. |
| **operation** | A service that can be requested from an object to effect behavior. |
| **package** | A grouping of modeling elements. |
| **phase** | The span of time between two major milestones of the process, at which point a well-defined set of objectives is met, artifacts are completed, and decisions are made to move, or not move into the next phase. |
| **process** | 1. (Software) Any thread of control that can logically execute concurrently with other processes.  
2. (Business) A set of partially ordered steps intended to reach a goal; in software engineering the goal is to build a software product or to enhance an existing one; in process engineering, the goal is to develop or enhance a process model; corresponds to a business use case in business engineering. |
<table>
<thead>
<tr>
<th>Glossary Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>process view</td>
<td>An architectural view that describes the concurrent aspect of the system: tasks (processes) and their interactions.</td>
</tr>
<tr>
<td>product</td>
<td>Software that is the result of development, and some of the associated artifacts (documentation, release medium, training).</td>
</tr>
<tr>
<td>prototype</td>
<td>A release that is not necessarily subject to change management and configuration control.</td>
</tr>
<tr>
<td>release</td>
<td>A subset of the end product, which is the object of evaluation at a major milestone (see “Prototype,” “Baseline”).</td>
</tr>
<tr>
<td>requirement</td>
<td>A requirement describes a condition or capability of a system; either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document.</td>
</tr>
<tr>
<td>requirements capture</td>
<td>A process component in the software-engineering process, whose purpose is to define what the system should do. The most significant activity is to develop a use-case model.</td>
</tr>
<tr>
<td>reuse</td>
<td>Further use or repeated use of an artifact.</td>
</tr>
<tr>
<td>risk</td>
<td>An ongoing or upcoming concern that has a significant probability of adversely affecting the success of major milestones.</td>
</tr>
<tr>
<td>scenario</td>
<td>A described use-case instance. A specific sequence of actions that illustrates behaviors.</td>
</tr>
<tr>
<td>system</td>
<td>As an instance, an executable configuration of a software application or software application family; the execution is done on a hardware platform. As a class, a particular software application or software application family that can be configured and installed on a hardware platform. In a general sense, an arbitrary system instance.</td>
</tr>
<tr>
<td>test</td>
<td>A process component in the software-engineering process whose purpose is to integrate and test the system.</td>
</tr>
<tr>
<td>transition</td>
<td>The fourth phase of the process in which the software is turned over to the user community.</td>
</tr>
<tr>
<td>use-case</td>
<td>A sequence of actions a system performs that yields an observable result of value to a particular actor.</td>
</tr>
</tbody>
</table>
**Glossary**

**use-case model**  A model of what the system is supposed to do and the system environment.

**use-case view**  An *architectural view* that describes how critical use cases are performed in the system, focusing mostly on architecturally significant components (objects, tasks, nodes). In Objectory, it is a view of the use-case model.

**view**  A simplified description (an abstraction) of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective. See also “Architectural view.”

**vision**  The user’s view or customer’s view of the product to be developed.

**worker**  Role figure to be played by individual members in the business.

**workflow**  The sequence of *activities* performed in a business that produces a *result* of observable value to an individual *actor* of the business.
Appendix F

References and Further Reading

<table>
<thead>
<tr>
<th>Process</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOO95</td>
<td>Grady Booch, Object Solutions, Addison-Wesley, 1995.</td>
</tr>
<tr>
<td>JAC94</td>
<td>Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, The Object Advantage—Business process reengineering with object</td>
</tr>
</tbody>
</table>
References and Further Reading


**Object Oriented Technology**


References and Further Reading


Software Architecture


Index


Miscellaneous


Check also Rational Web site [http://www.rational.com](http://www.rational.com) for white papers on process, object-oriented technology, and tools.

Index

4  description 21

4+1 view model 66

A

actors 53

   in requirements capture 19

analysis & design process component
B
blueprints
architectural 69

C
classes 54
component based development 33
components
in architecture 66
connectors
in architecture 66
controlled iterative development
benefits of 29

D
descendant 55
design See analysis & design
design model
description 21

F
flow of events 58; 59
forms
in architecture 66
framework
architectural 68
Index

G
generalization 55
glossary 73

I
implementation process component
description 23
infrastructure
architectural 68
inheritance 55
instance 54
Introduction manual 38
iterative development
benefits of 29

M
manuals 37
models
in general 55
in Objectory 18

O
object 54
object instance 54
object technology
benefits of 27
Objectory manuals 37
Objectory online 40
operation 54

P
patterns
architectural 68
process components 7
description 18
process configurability 34
Process Configuration manual 39
Process Manual 38
Project Management manual 39

R
requirements capture process component
description 18
requirements management
benefits of 31

S
software architecture 65
Software Architecture Document 66
style
architectural 68

T
test process component
description 25
U

use case
  flow of events 59
use case 53; 58
  benefits of, in the development process 27
  definition 57
  flow of events 58
  in requirements capture 19

use-case model
  description 19
  use-case-driven development 27

W

waterfall approach
  compared to iterative approach 29
view
  architectural 66