

# Features, Use Cases, Requirements, Oh My!

---

By Dean Leffingwell  
A Rational Software White Paper

**Rational**<sup>®</sup>  
the e-development company<sup>™</sup>

## **Table of Contents**

<b>INTRODUCTION .....</b>	<b>1</b>
<b>THE PROBLEM DOMAIN VS THE SOLUTION DOMAIN .....</b>	<b>2</b>
The Problem Domain.....	2
On User and Stakeholder Needs .....	2
The Solution Domain.....	3
<b>COMMON REQUIREMENTS TERMS IN THE SOLUTION DOMAIN .....</b>	<b>4</b>
Features of a Product or System .....	4
Use Cases.....	4
Vision Document .....	5
Software Requirements.....	5
Functional Requirements .....	6
Elaborating the Use Case.....	7
Nonfunctional Requirements .....	7
Design Constraints.....	8
Hierarchical Requirements.....	8
Traceability .....	8
Impact Analysis and Suspectness .....	8
Change Requests and the Change Management System.....	9
<b>SUMMARY .....</b>	<b>9</b>
<b>SUGGESTED READING .....</b>	<b>11</b>
<b>BIBLIOGRAPHY .....</b>	<b>11</b>

## Introduction

As a follower and proponent of object-oriented technology in the BU (before-UML) days, I must admit to a certain fascination with the various methods and notations spread by the industry thought leaders at the time. At about two to four years BU, you could walk into a room full of OO advocates and ask the following question:

I think this OO technology shows great promise but tell me, since the object shares behavior and data, what do you call this thing an object does to fulfill its behavioral obligations?

You might get the following answers:

“It’s a responsibility!” (Wirfs-Brock)  
“It’s an operation!” (Booch)  
“It’s a service!” (Coad/Yourdon)  
“It’s a (virtual) function!” (Stourstrup)  
”It’s a method!” (many others)

And if that wasn’t enough confusion, don’t even think about asking how you would graphically represent that thing we called an object and a class. (It’s a rectangle, it’s a cloud, it’s a... whatever.) While this seems silly, the reality is that some of the most significant agreements of our software engineering leaders—inheritance, relationships, encapsulation—were hidden, or at least were confused, by minor differences in terminology and notation. In other words, neither the science of OO engineering nor the benefits to be gained could advance further because the language to describe the science had not yet been invented. Of course, gaining agreement among these authors, methodologists<sup>1</sup>, and independent thinkers was a non-trivial problem but, eventually, along came the UML and the science of software engineering marched forward again.

While it’s perhaps not as bad as the tower of Babel wrought by the pre-UML competing OO methodologies, the methodology of *requirements management* suffers from some of the same issues—specifically the prevalence of ambiguous, inconsistent, and overloaded usage of common terms. These terms, including such seminal constructs as “Use Cases”, “Features”, and “Requirements”, are common everyday terms that “everyone understands” and yet to which each individual attaches their own meaning in a given context. The result is often ineffective communications. And this occurs in a domain wherein, perhaps like few others, success is defined simply by having *achieved* a common understanding. Booch [Booch 1994] points out that Stepp observed

an omnipresent problem in science is to construct meaningful classifications of observed objects and situations. Such classifications facilitate human comprehension of the observations and subsequent development of a scientific theory.

In order to advance the “scientific theory” of requirements, we are going to have to come to terms with terms!

The purpose of this article is to take a small step forward in the discipline of software engineering by defining and describing some of the most common terms and concepts used in describing requirements for systems that contain software. In doing so, we hope to provide a basis for common understanding among

---

<sup>1</sup> At Rational Software, it has been my privilege to work with some of the industry’s leading methodologists—Grady Booch, Ivar Jacobson, Jim Rumbaugh, Philippe Kruchten, Bran Selic, and others. While this has been a rewarding and fascinating part of my career, it’s not something I could recommend for everybody. In other words, please do NOT try this at home.

Features, Use Cases, Requirements, Oh My!

the many stakeholders involved: users, managers, developers, and others. Certainly if we communicate more effectively and thereby gain a common view, it's possible to more quickly develop and deliver higher quality systems.

This article is not an overview of the requirements management discipline—for that we can refer you to a number of books on the topic under the heading Suggested Reading. The goal of this article is simply to help practitioners in the field improve their ability to answer the following, fundamental question: “What, exactly, is this system supposed to do?”

## The Problem Domain vs the Solution Domain

Before we start describing specific terms, however, it's important to recognize that we will need to define terms from two quite different worlds—the world of the problem and the world of the solution. We'll call these the *problem domain* and *solution domain*, respectively.

### **The Problem Domain**

---

If we were to fly over the *problem domain* at a fairly low level, we would see things that look very much like the world around us. If we flew over the HR department, we might see employees, payroll clerks, and paychecks. If we flew over a heavy equipment fabricator, we might see welders, welding controllers, welding robots, and electrodes. If we flew over the World Wide Web, we'd see routers and server farms, and users with browsers, telephones, and modems. In other words, in any particular problem domain we can most readily identify the things (entities) that we can see and touch. Occasionally, we can even see relationships among those things; for example, there seems to be a one-to-one relationship between web users and browsers. We might even see messages being passed from thing to thing: “the welder appears to be programming a sequence into the welding robot's ‘brain’ ”.

If we are really observant, we might see things that look like *problems* just waiting to be resolved: “the welder seems really frustrated with his inability to get the sequence right” or “notice that nasty time delay between the time that the employee enters their payroll data and the day they receive their check!” Some of the problems seem to just *beg* for a solution. Perhaps we can build a system (better programmable controller, more efficient payroll processing) to help those poor users down there fix those problems.

### **On User and Stakeholder Needs**

Before we build that new system, however, we need to make sure that we understand the *real needs* of the users in that problem domain. If we don't, we may discover that the welder was grimacing only because he was suffering from a painful corn on his toe, and it turns out that neither he nor his management are interested in purchasing our brand new “SmartBot” automated welding control unit. We also notice that when we try to sell the SmartBot, the manager seemed to emerge as a key stakeholder in the purchasing decision. We don't remember seeing her in our fly-over. (Perhaps she was in the smoking lounge, our cameras don't seem to work as well in there.) In other words, not all stakeholders are users and we're going to have to understand the needs of both communities (stakeholders and users) if we hope to have a chance to sell the SmartBot. To keep things simple, we call all of these needs *stakeholder needs*, but we'll constantly remind ourselves that the potential users of the system appear to represent a very important class of stakeholders indeed.

We'll define a stakeholder need as:

a reflection of the business, personal or operational problem (or opportunity) that must be addressed to justify consideration, purchase or use of a new system.

Stakeholder needs, then, are an expression of the issues associated with the problem domain. They don't define a solution, but they provide our first perspective on what any viable solution would need to accomplish. For

## Features, Use Cases, Requirements, Oh My!

example, if we interview the plant manager for a heavy equipment fabricator, we may discover that welding large repetitive weldments consumes a significant amount of manufacturing time and cost. In addition, welders don't seem to like these particular jobs and they are constantly in danger of burnout. Worse still, the physical aspects of the job—repetition, awkward manual positions, danger to eyesight, and so on—present personal safety issues and long term healthcare concerns.

With these insights, we could start defining some of these *stakeholder needs*:

- We need an automated way to fabricate large repetitive weldments without the welder having to manually control the electrode.
- We are happy to have a welder present, but we need to remove him to a safety zone outside of the weldment and away from any moving machinery.
- We need an easy to use “training mode” so that average welders can “train” the machine to do the majority of the welding for them.
- We need to allow more flexibility in the training mode and recognize that this may contradict some aspects of the need for user-friendliness.



As we understand these various aspects of the system, we'll mentally “stack” these discoveries in a little pile called stakeholder needs.

### ***The Solution Domain***

---

Fortunately, our fly-over of the problem domain doesn't take very long and (usually) what we find there is not too complicated. We start to appreciate the problem when we leave the airplane, and set off to build a solution to the problems and needs we have observed. Yes, we've reached the beginning of the hard part: *forming a solution to the problem*. We consider the set of activities (system definition, design, and so on), the “things” we find and build to solve the problem (computers, robot arms, and the like), and the artifacts we create in the process (such as source code, use cases, and tests) part of the *solution domain*.

In the solution domain, there are many steps and activities we must successfully execute to define, build, and eventually deploy a successful solution to the problem. They include:

- 1) Understand the User's Needs
- 2) Define the System
- 3) Manage Scope and Manage Change
- 4) Refine the System Definition
- 5) Build the Right System

In a nutshell, the steps above define a simplified process for requirements management. This paper won't discuss these steps in much detail; for this we refer you to the references and selected reading, including the text “Managing Software Requirements”, [Leffingwell, 1999]. This paper is particularly consistent with that reference work and most of the definitions provided here are from that reference.

For example, from the reference [Leffingwell, 1999], we find that requirements management is a systematic approach to eliciting, organizing, and documenting the requirements of the system, and a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system.

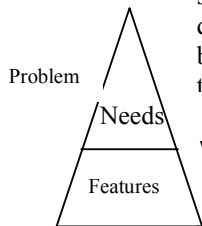
But let's move on to discovering and defining some more of the requirements management terms we'll need to describe the system we are about to build.

## Common Requirements Terms in the Solution Domain

### **Features of a Product or System**

---

As we start thinking about solutions to the problems we've identified, it's very natural to start jotting down the *features* of a system. Features occupy an interesting place in the development of a system. They seem to fit somewhere between an expression of the user's real needs and a detailed description of exactly how the system fulfills those needs. As such, they provide a handy construct, a "shorthand" if you will, for describing the system in an abstract way. Since many possible solutions exist for the problem that needs to be solved, in a sense features provide the initial bounds of a particular system solution; they describe what the system is going to do and, by omission, what it will not.



We'll define a feature as  
a service that the system provides to fulfill one or more stakeholder needs.

Features are easily represented in natural language, using terms familiar to the user. Example features might include:

- The system runs off standard North American power.
- The tree browser provides a means to organize the defect information.
- The home lighting control system has interfaces to standard home automation systems.

Since features are derived from stakeholder needs, we position them at the next layer of the pyramid, below needs. In so doing, we've also moved from the problem domain (needs) to the first level of the solution domain (features).

It's important to notice that features are NOT just a refinement (with increasing detail) of the stakeholder needs. Instead, they are a direct response to the *problem* offered by the user and they provide us with a top-level *solution* to the problem.

Typically, we should be able to describe a system by defining 25–50 features that characterize the behavior of the system. If you find yourself with more than 50 features on your hands, it's likely that you've insufficiently abstracted the true features of the system or it may also be the case that the system is too large to understand and you may need to consider dividing the system into smaller pieces.

Features are described in natural language so any stakeholder who reads the list can immediately gain a basic understanding of what the system is going to do. The features list will lack fine-grained detail. That's all right. We're simply trying to communicate the intent and, since many stakeholders are likely to be non-technical, too much detail can be confusing and may even interfere with understanding. By example, a partial list of features for our SmartBot automated welding robot might include:

- A "lead through path" training mode that allows the welder to teach the robot what paths will be welded.
- A "step-and-repeat" feature that supports repetitive welding sequences.

### **Use Cases**

---

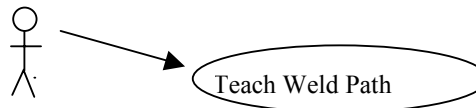
As we think further about the way in which the system needs to do its job for the user, we might find it beneficial to use the *use case technique* for further describing system behavior. This technique has been well developed in a number of books [Jacobson 1992] and is also an integral technique in the industry-standard Unified Modeling Language [Booch 1999].

Technically, a use case  
describes a sequence of actions, performed by a system, that yields a result of value to the user.

## Features, Use Cases, Requirements, Oh My!

In other words, the use case describes a series of user and system interactions that helps users accomplish something they wanted to accomplish. Stated differently, the use case describes HOW users and the system work together to realize the identified feature.

Use cases also introduce the construct of an actor, which is simply a label for the user who is using the system at that time. In UML, a use case is represented by a simple oval, whereas an actor is represented by a stick figure with a name. So you can illustrate both with a simple diagram as we see below.



The use case technique prescribes a simple step-by-step procedure for how the actor accomplishes the use case. For example, a use case for Step and Repeat might start out as follows:

Step 1: The welder presses the step and repeat button to initiate the sequence.

Step 2: The welding system releases power to the drive motors so that the robot's arms can be moved manually.

Step 3: The welder grabs the trigger, moves the arm to the weldment, and holds down the "weld here" button for each path to be welded.

The use case technique provides a number of other useful constructs, such as pre and post descriptions, alternate flows, and so on. We'll talk about these later as we examine the use case in more detail. For now, we simply need to know that use cases provide an excellent way to describe *how* the features of the system are achieved.

For planning purposes, it's likely that more than use cases will be necessary to describe how a particular feature is implemented. A small number of use cases (perhaps 3–10) may well be necessary for each feature. In describing the use cases, we are elaborating on the behavior of the system. Detail increases as additional specificity is attained.

## ***Vision Document***

---



In many development efforts, a statement of the problem, key stakeholder, and user needs, a list of the features of a system, and perhaps example use cases may be found in a document called the Vision document. It may be called by a variety of other names, such as Project Charter, Product Requirements Document, Marketing Requirements Document, and so forth. No matter what it's called, the Vision document highlights the overall intent and purpose of the system being built and, as such, it's a natural container for the features and illustrative use cases that have been identified to date. In other words, the Vision document captures the gestalt of the system and uses *stakeholder needs, features, and use cases* to communicate the intent.

However, we cannot simply dump these features and initial use cases into the hands of the development team and expect them to rush off to develop a system that really satisfies the stakeholder needs. We will probably need to be a lot more definitive about what we want the system to do, and we'll probably have a variety of new stakeholders involved, including developers, testers, and the like. That's the need addressed by the next layer of the system definition—the software requirements.

## ***Software Requirements***

---



Software requirements provide the next level of specificity in the requirements definition process. At this level, we must specify requirements and use cases sufficiently so that developers can write code and testers

## Features, Use Cases, Requirements, Oh My!

can test to see that the code meets the requirements. Graphically, software requirements provide the base of our pyramid.

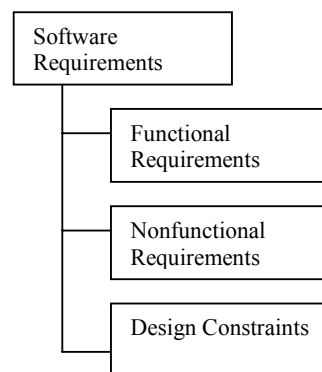
What is a software requirement? Although there have been many definitions used throughout the years, we have found the definition provided by requirements engineering authors Dorfman and Thayer [Dorfmann 1990] to be quite workable:

- *a software capability needed by the user to solve a problem that will achieve an objective, or*
- *a software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documentation.*

Applying this definition, the team can develop a more specific set of requirements to refine, or elaborate, the features list discussed earlier. Each requirement serves some feature and vice versa. Notice the simplicity of this approach. We have a list of features and we then elaborate those features by writing a set of requirements that serve those features. *We don't write any other requirements.* This avoids the temptation to simply sit down, stare at the ceiling, and “think up some requirements for this system.”

The process is straightforward, but not necessarily easy. Each feature is reviewed and then requirements are written to support the feature. Inevitably, writing the requirements for one feature will spur ideas for new or revised requirements for a feature that has already been examined.

Of course, you know it's not easy to write down requirements and there can be a large number of such requirements that must be specified. We've found it helpful to think about three types or categories of software requirements—functional requirements, nonfunctional requirements, and design constraints.



We find these three categories helpful in the way we think about the requirement and what role we expect the requirement to fill. Let's look at these different types of requirements and see how you can use them to define the differing aspects of the proposed system.

### **Functional Requirements**

Functional requirements express what the system does. More specifically, functional requirements describe what the inputs are, what the outputs are, and how it is that specific inputs are converted to specific outputs at various times. Most software applications conceived to do useful work are rich with functional requirements. When specifying these requirements, it's important to strike a balance between being too vague (“When you push the On button, the system turns on”) and being too specific about the functionality. It's important to give the designers and implementers as wide a range of design and implementation choices as possible. If we're too specific, we may over-constrain the team and if we're too wishy-washy, the team won't know what the system is supposed to achieve.

## Features, Use Cases, Requirements, Oh My!

There isn't just one right way to specify requirements. One technique is simply to take a declarative approach and write down each detailed thing the system needs to do.

For example:

During the time in which the "weld here" input is active, the system digitizes the position of the electrode tip by reading the optical encoders every 100 msec.

### **Elaborating the Use Case**

In many systems, it's easier to organize the specification activity by refining the use cases defined earlier and developing additional use cases to fully elaborate the system. Using this technique, you refine the steps of the use case into more and more detailed system interactions. You'll also need to define pre-conditions and post-conditions (states the system assumes before and after the use case), alternative actions due to exception conditions, and so on.

Since use cases are semantically well defined, they provide a structure into which to organize and capture the system behavior. Here is a representative use case for the Smartbot.

Use Case Name	Teach Weld Path
Actor	Welder
Brief Description	This use case prescribes the way in which the welder teaches the robot a single weldment path operation.
Flow of Events	Basic flow for the use case begins when the welder presses the "teach" button on the control console. The system turns off the power to the robot arms. The welder grabs the teaching electrode and positions the teaching tip at the start of the first weld. The welder presses the "weld here" trigger and simultaneously moves the teaching tip across the exact path to be welded. At the end of the path, the welder releases the "weld here" trigger and then returns the robot's arm to the rest position.
Alternative Flow of Events	At any time during the motion, the welder can press the pause button, the robot will turn on power to the motors, and hold the arms and teaching tip in the last known position.
Pre-conditions	The robot must have performed a successful auto-calibrate procedure.
Post-conditions	The traverse path and weld paths are remembered by the system.
Special Requirements	The welder cannot move the tip at a rate faster than 10cm/second. If faster motion is detected, the system will add resistance to the arms until the welder returns to the acceptable lead through speed.

### **Nonfunctional Requirements**

In addition to functional requirements such as inputs translating to outputs, most systems also require the definition of a set of nonfunctional requirements that focus on specifying additional system "attributes", such as performance requirements, throughput, usability, reliability, and supportability. These requirements are just as important as the input-output oriented functional requirements. Typically, nonfunctional requirements are stated declaratively using expressions such as "The system should have a mean time

Features, Use Cases, Requirements, Oh My!

between failure of 2,000 hours”, “The system shall have a mean time to repair of 0.5 hours”, and “The Smartbot shall be able to store and retrieve a maximum of 100 weld paths”.

### **Design Constraints**

As opposed to defining the behaviors of the system, this third class of requirements typically imposes limitations on the design of the system or process we use to build the system. We’ll define a design constraint as

restrictions upon the design of a system, or the process by which a system is developed, that do not affect the external behavior of the system, but must be fulfilled to meet technical, business or contractual obligations.

A typical design constraint might be expressed as “Program the welder control unit in Java”. In general, we treat any reasonable design constraints just like any other requirements although testing compliance to such constraints may require different techniques. Just like functional and nonfunctional requirements, these constraints can play an integral role in designing and testing the system.

### **Hierarchical Requirements**

Many projects benefit from expressing these requirements in a hierarchical or *parent-child* structure. A parent-child requirement is an amplification of the specificity expressed in a parent requirement. Parent-child requirements allow you a flexible way to enhance and augment your specification, while at the same time organizing the depth of detail presented. By looking only at the parents, it’s straightforward to present the top-level specification in a way that is easily understandable by the users. At the same time, the detailed “child” specification can be quickly inspected by the implementers to make sure that they understand all of the implementation details.

Note that hierarchical requirements consist up of the standard three types of requirements—functional, non-functional, and design constraints. It’s only the *elaboration relationship* between these requirements that is defined here.

### **Traceability**

---

In addition to the terms we have defined so far to describe the *things* we use to describe system requirements, we now turn our attention to a key *relationship*, traceability, which may exist between these things.

A significant factor in quality software is the ability to understand, or trace, requirements through the stages of specification, architecture, design, implementation, and test. Historical data shows that the impact of change is often missed and small changes to a system can create significant reliability problems. Therefore, the ability to track relationships, and relate these relationships when change occurs, forms a key thread throughout many modern software quality assurance processes, particularly in mission critical activities such as safety-critical systems (medical and transportation products), systems with high economic costs of failure (on-line trading), and so on.

Here’s how we define requirements traceability:

A traceability relationship is a dependency in which the entity (feature, use case, requirement) “traced to” is in some way dependant on the entity it is “traced from”.

For example, we’ve described how one or more Software Requirements are created to support a given feature or use case specified in the Vision document. Therefore, we can say that these Software Requirements have a *traceability relationship* with one or more Features.

### **Impact Analysis and Suspectness**

In addition, a traceability relationship goes beyond a simple dependency relationship because it provides the ability to do impact analysis using a concept that we call “suspectness”. A traceability relationship

Features, Use Cases, Requirements, Oh My!

goes “suspect” whenever a change occurs in the “traced from” (independent) requirement and, therefore, the “traced to” (dependent requirement) must be checked to ensure that it remains consistent to the requirement from which it is traced.

For example, if we use traceability to relate requirements to specific tests and if a requirement such as “The Smartbot shall be able to store and retrieve a maximum of 100 weld paths” becomes “The Smartbot shall be able to store and retrieve a maximum of 200 weld paths”, then the test traced from this requirement is *suspect*, in that it is unlikely any test devised to test the first requirement will be adequate to test the second one.

## ***Change Requests and the Change Management System***

Finally, change is inevitable. For your project to have any hope of succeeding, a process for managing change is essential. Regardless of their source, and the sources are legion, all changes, including requests that affect features and requirements, need to be introduced and managed in an orderly manner. The key element of any change management system is the *Change Request* itself.

We’ll define change requests as

an official request to make a revision or addition to the features and/or requirements of a system.

Change Requests need to enter the system as a structured and formalized statement of a proposed change, and any particulars surrounding the change. In order to manage these changes, it’s important that each change has its own identity in the system. A simplified form of a change request might appear as:

Change Request	
Change Request Item	Value
Change Request ID	CR001
Change Request Name	Safety Feature on Power On Button
Brief Description of Change	Add hold time to Power On button that requires user to hold button for xx seconds before system turns on
Requested by...	Safety Supervisor

In most projects, you’ll find no shortage of changes! In fact, your problem becomes one of managing, integrating, and, where required, rejecting unnecessary changes in an orderly manner. In other words, you need a process to manage change. Your change management system should be used to capture *all* inputs and transmit them to the authority of a Change Control Board (CCB) for resolution. The CCB, consisting of no more than three to five people who represent the key stakeholders for the project (customers, marketing, and project management), administer and control changes to your system, and thereby play a key role in helping the project succeed.

## **Summary**

At the beginning, we noted that a goal of this article was to help practitioners in the field improve their ability to answer the fundamental question:

“What, exactly is this system supposed to do?”

As a step toward this goal, we defined and described some of the common terms—such as stakeholder needs, features, use cases, software requirements, and more—used by analysts and others who have responsibility for describing issues in the problem domain, and for expressing the requirements to be

## Features, Use Cases, Requirements, Oh My!

imposed upon any prospective solution. In so doing, we also illustrated some of the key concepts of effective requirements management. By using the terms and approaches outlined in this article, you will more effectively understand your user's needs and communicate the requirements for proposed solutions to developers, testers, and other technical team members who are responsible for building a system to meet the needs of your customers and users.

An important technique for further defining and communicating many additional key aspects of software solutions is the use of a standard modeling language. The Unified Modeling Language (UML) is a language for visualizing, specifying, and documenting the artifacts of a software intensive system, and provides a means for expressing these technical constructs in a more semantically precise manner. A companion paper, *Modeling the Requirements Artifacts with UML*, establishes the necessary UML constructs and extensions to more effectively perform the requirements management tasks.

## Suggested Reading

- [Leffingwell 1999] Leffingwell, Dean, and Don Widrig. *Managing Software Requirements: A Unified Approach*. Reading, MA: Addison Wesley Longman, 1999.
- [Weigers 1999] Weigers, Karl. *Software Requirements*, Redmond Washington: Microsoft Press, 1999.

## Bibliography

- [Booch 1994] Booch, Grady. *Object-Oriented Analysis and Design with Applications*, 2<sup>nd</sup> ed. Redwood City, CA. Benjamin Cummings, 1994.
- [Booch 1999] Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading, MA: Addison Wesley Longman, 1999.
- [Dorfmann 1990] Dorfmann, Merlin, and Richard H. Thayer. *Standards, Guidelines, and Examples of System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1990.
- [Jacobson 1992] Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Harlow, Essex, England: Addison Wesley Longman, 1992.
- [Rumbaugh 1999] Rumbaugh, James, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison Wesley Longman, 1999.

Features, Use Cases, Requirements, Oh My!



Corporate Headquarters  
18880 Homestead Rd.  
Cupertino, CA 95014

Toll-free: 800.728.1212  
Tel: 408.863.9900  
Fax: 408.863.4120  
Email: [info@rational.com](mailto:info@rational.com)  
Web site: [www.rational.com](http://www.rational.com)  
For International Offices: [www.rational.com/corpinfo/worldwide/location.jtmpl](http://www.rational.com/corpinfo/worldwide/location.jtmpl)

Rational, Rational logo, Rational the e-development company, Rational RequisitePro, Rational Rose, Rational ClearQuest, SoDA, Rational Suite, AnalystStudio, TestStudio, PerformanceStudio and the Rational Unified Process are trademarks or registered trademarks of Rational Software Corporation. References to other companies and their products use trademarks owned by the respective companies and are for reference purposes only. ALL RIGHTS RESERVED. Made in the U.S.A.  
© Copyright 2000 by Rational Software Corporation.

Subject to change without notice.