Rational. software

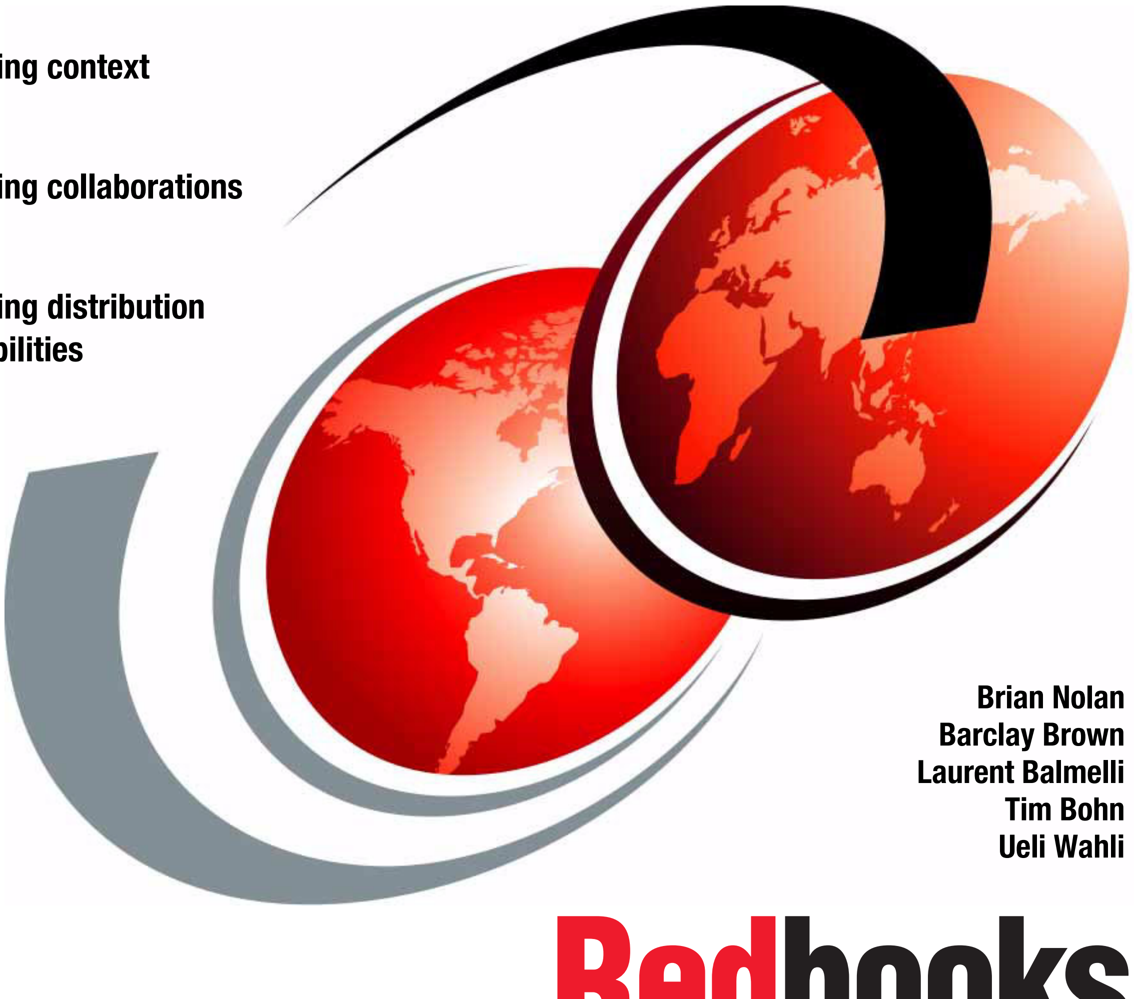
IBM

Model Driven Systems Development with Rational Products

Understanding context

Understanding collaborations

Understanding distribution of responsibilities



Brian Nolan
Barclay Brown
Laurent Balmelli
Tim Bohn
Ueli Wahli

Redbooks

ibm.com/redbooks

International Technical Support Organization

Model Driven Systems Development with Rational Products

February 2008

SG24-7368-00

Note: Before using this information and the product it supports, read the information in "Notices" on page ix.

First Edition (February 2008)

This edition applies to IBM Rational Systems Developer, Version 7.

Contents

Notices

Trademarks

Preface

This IBM® Redbooks® publication describes the basic principles of the Rational® Unified Process® for Systems Engineering, which is IBM Rational's instantiation of model-driven systems development (MDSD).

MDSD consists of a set of transformations that progressively refine knowledge, requirements, and design of complex systems. MDSD begins with activities and artifacts meant to promote an understanding of the system's context.

Requirements problems often arise from a lack of understanding of context, which, in MDSD, means understanding the interaction of the system with entities external to it (actors), understanding the services required of the system, and understanding what gets exchanged between the system and its actors. Managing context explicitly means being aware of the shifts in context as you go from one model or decomposition level to the next.

MDSD suggests that a breadth-first collaboration based approach across multiple viewpoints is more effective than a traditional depth-first functional decomposition in creating an architecture that will not only meet requirements, but will prove to be more resilient in the face of inevitable change. MDSD also seeks to provide an effective distribution of responsibilities across resources. Joint realization and abstractions such as localities provide an effective and elegant way of accomplishing this.

Finally, the ability to attach attributes and values to modeling entities and the parametric capabilities of SysML provide a basis for doing simulations or other models to meet cost, risk, and other concerns.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Brian Nolan is a course developer for IBM Software Group, Rational Learning Solutions® and Services, specializing in model-driven development. Prior to his current position, he was the regional practice lead for the Rational Unified Process for Systems Engineering. Dr. Nolan holds a Ph.D. degree in the classics from Ohio State University.

Barclay Brown is an executive consultant in the system engineering practice in IBM Global Business Services. Prior to this, he was the Worldwide Community of Practice leader for Rational Solution Architecture. He leads client engagements in aerospace and defense, system development, and IT enterprise architecture, helping clients transform their engineering organizations using IBM technologies, methods, and tools. Barclay has been a practitioner, consultant, and speaker on system engineering methods for over 8 years. His experience spans some 24 years in project management, system engineering, architectural modeling, and requirements analysis. His current specialization includes model-driven system development, enterprise architecture, estimation methods, and solution architecture. He is the designer of the model-driven system development course, offered by IBM. Barclay holds degrees in electrical engineering, psychology, and business.

Dr. Laurent Balmelli is a manager at IBM in charge of architecting the new generation of offerings and tools for systems engineering and product development. He has been a research staff member at T.J. Watson Research Center and IBM Tokyo Research Labs, and a member of several leadership councils in IBM since 2000. Since 2003, Dr. Balmelli has represented IBM within the SysML standard team and is one of the lead authors of the SysML language specification. He was recently awarded the position of invited professor at Keio University in Tokyo, Japan, where he currently resides.

Tim Bohn is currently the Worldwide Community of Practice Leader for Solution Architecture. Tim has been active in the Systems community for many years, helping customers adopt MDSD in their practice. Tim has been with IBM Rational Software for 12 years, in both technical and management roles. Prior to joining Rational, Tim worked as a software engineer and systems engineer for 16 years. Tim holds a BS and MS degree from the University of Southern California, where he also guest lectures.

Ueli Wahli is a Consultant IT Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO over 20 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide about WebSphere® Application Server and WebSphere and Rational application development products. In his ITSO career, Ueli has produced more than 40 IBM Redbooks. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology.

Thank you

We would like to thank the following individuals for their help with this book:

- Thanks to several authors who participated, but whose contributions we were not able to include in this edition: Christopher Alderton, Keith Bagley, James Densmore, Steven Hovater, and Russell Pannone
- Thanks to the reviewers, especially David Brown, who made extensive suggestions for improvement throughout
- Thanks to our managers for their support
- Thanks to Dr. Murray Cantor, for his thought, leadership, encouragement, and support
- Thanks to Yvonne Lyon, IBM Redbooks Editor, for editing this book
- Thanks to our families for their patience, support, and encouragement throughout this project

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbooks publication dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  `ibm.com/redbooks`

- Send your comments in an e-mail to:

  `redbooks@us.ibm.com`

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

1 Introduction

This book is based on work done at IBM Rational by Dr. Murray Cantor and others. In a series of articles for Rational, Dr. Cantor sets out the basic principles of the Rational Unified Process for Systems Engineering (RUP® SE), which is IBM Rational's instantiation of model driven systems development (MDSD)[1].

This chapter provides an introduction to MDSD, discusses the challenges it was designed to address, and some of the benefits of using it. It provides a core set of concepts to enhance understanding the methodology, and provides an overview of the rest of the book. It also indicates what knowledge is needed as a prerequisite to understanding the material we present.

[1] L. Balmelli, D. Brown, M. Cantor, and M. Mott, *Model-driven systems development*, IBM Systems Journal, vol 45, no. 3, July/September 2006, pp. 569-585 is the most recent.
`http://www.research.ibm.com/journal/sj/453/balmelli.html`
See also the series of articles in the Rational Edge, August-October, 2003

The challenges of systems development

As the world moves into the Information Age, the rate of change is increasing. Information is enabling new business models, such as eBay or Amazon.com, and as a result new demands are placed upon the information systems. System complexity is increasing in response to the capability of languages, technology, and global information flow. Coincident with increasing complexity, the pace of change is creating a need to reduce the time required to deliver solutions. Systems development has not kept pace with the demands to deliver more capability in less time. Development teams, using traditional methods, often still fail to deliver capability, which can be fatal to a business in the Information Age.

The changed context for systems development

Computing technology has advanced so that modern systems are thousands of times more powerful than their predecessors. This change removed resource constraints and is changing the approach to system delivery in fundamental ways. Historically teams struggled to deploy as much functionality using as little computer resource as possible. The development team's primary goal was to delivery a working system—cost, especially over a system's life cycle, was a secondary solution. Solutions were often highly customized and proprietary. Development life cycles were longer, and we could regularly schedule updates.

In modern systems, fewer components provide more functionality and therefore have greater code counts. Integration is critical. Our systems must integrate with today and tomorrow's systems **now**. Within the systems themselves, we must integrate components from a variety of sources. We have many technology choices, and software permeates everything. We have improved software development productivity, but our software has increased tenfold in size[2]. We must update our systems constantly, yet reduce costs across the life span of the system. We must innovate, but also manage risk; we must meet new technical challenges, but also manage cost.

Within the aerospace and defense markets, the changes are especially dramatic due to the changing nature of threats in conjunction with the changes to technology. During the Cold War, defense agencies and suppliers built large and expensive systems. Because these systems were focused on defending against other high technology threats, the high cost and time to develop was not seen as a major issue. With the threats posed by terrorism, this has changed. Terrorists cause disruption with relatively low cost devices and also change their tactics

[2] David Longstreet, *Software Productivity Since 1970*, 2002 (`http://www.softwaremetrics.com/Articles/history.htm`).
cited in Cantor, *Rational Unified Process for Systems Engineering*, Part 1: Introducing RUP SE Version 2.0, The Rational Edge, August 2003

rapidly. Hence the methods that worked for the Cold War do not work in the current environment. In today's world, defense systems require agility and net-centricity. Systems must become much more agile and capabilities must be deployed more quickly. Our development methods must help us integrate and deploy complex and scalable functionality more quickly.

Management of complexity

Our world is very complex—and becoming more complex daily.[3] We must manage complexity, before it overwhelms us. Methods for managing complexity can help us prosper in our complex world. Model driven systems development (MDSD) is such a method.

At its core, MDSD is quite simple, but very powerful in its simplicity; extraordinarily complex things are built from simple pieces.[4] It applies across a wide range of domains, and across a wide range of levels of abstraction from very abstract to very concrete, from business modeling to the modeling of embedded software. MDSD is not just a method for reasoning about and designing software systems, it is a method for reasoning about and designing large complex systems consisting of workers, hardware and software.[5]

The power of MDSD lies in the power of its abstractions.

Creative/dynamic and transactional complexity

In building systems, we are faced with two different kinds of complexity: Creative/dynamic complexity and transactional complexity:

- We face creative/dynamic complexity because we need teams of people to work together creatively to architect optimal, robust systems.
- We face transactional complexity when we try to manage all the components that make up a complex system.

Transactional complexity can be managed with MDSD.

[3] Cantor, *Rational Unified Process for Systems Engineering, Part 1: Introducing RUP SE Version 2.0*, The Rational Edge, August 2003, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/aug03/f_rupse_mc.pdf

[4] Booch covers this point in *Object-Oriented Design and Analysis with Applications*, 3rd Edition, Addison Wesley, Reading, MA, 2007. *When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently. In this manner, we satisfy the very real constraint that exists upon the channel capacity of human cognition …*, page 19.

[5] See Blanchard and Fabryky's definition: Blanchard and Fabryky, *Systems Engineering and Analysis*, third edition, Prentice Hall, 1998, quoted by Murray Cantor (see footnote 3).

Creative/dynamic complexity can be managed with a governance process. (The governance process must be enabling and not confining.)

Governance more and more becomes a matter of managing risk in an innovative world; of balancing innovation and risk.

Overview of model-driven systems development

Model-driven systems development is the progressive, iterative refinement of a set of models to drive development of your system.

The benefits of modeling

Why do we model? We model to manage complexity, to simplify and abstract essential aspects of a system. We model so that we can test inexpensively before we build, so that we can erase with a pencil before we have to demolish with a sledgehammer.[6]

The models *are* the architecture—they provide us with multiple views of the system and promote our understanding.

Model-driven systems development leverages the power of modeling to address a set of problems that have plagued systems development. We discuss some of these problems in the sections that follow. MDSD uses a set of transformations to iteratively refine our models and our understanding of the system to be built.

Central problems MDSD addresses

MDSD addresses a core set of system development problems:

- Overwhelming complexity: Managing complexity by managing levels of abstraction and levels of detail
- Not considering appropriate viewpoints: Multiple views to address multiple concerns
- System does not meet functional, performance and other system concerns: Integration of form and function
- Lack of scalability: Isomorphic composite recursive structures and method to address scalability

[6] This is an adaptation of a quote from Frank Lloyd Wright: *An architect's most useful tools are an eraser at the drafting board, and a wrecking bar at the site*

Managing complexity by managing levels of abstraction and levels of detail

Very often, when dealing with a system of systems, it is difficult to manage the details of system design at different levels of abstraction and detail. Issues at one level of the system get intertwined with issues at another; requirements and design at one level get confused with requirements and design at another.

Think of it this way—if your concern is to travel from Cambridge, England to Rome, Italy, you will be thinking about planes, trains, and automobiles—you probably do not want to be thinking about the wiring in the airplane, or the details of the air control system, or the brake system in the car.

Engineers have a tendency to want to jump down to the details. So when they talk about a system for getting you to your destination, they are as likely to talk about problems with the air control software or the wiring of a piece of hardware as they are to talk about larger-grained issues. This can lead to confusion and errors—diving too deep too early causes integration problems and constrains a solution too early. Requirements are usually best understood in context; jumping levels leads to a loss of context.

In our consulting practice at IBM, we have found it useful to manage the level of abstraction, and to use the appropriate level of detail for the level of abstraction under consideration. Also, we use a formal meta model to provide rigor to our reasoning.[7] Briefly, we consider two kinds of levels: model levels and levels of decomposition. Model level refers to what phase of our thinking we are in—analysis models should be less detailed than design models, for example. Decomposition level refers to how deep we are in the structural hierarchy of the system.

This is one of the foundational concepts for MDSD. For example, if we are creating a model for analysis, and we want to reason about distribution issues, we should use entities that do not commit us too early to design decisions.[8] If we are reasoning about the enterprise, we use entities that are appropriate for that level of decomposition, and keep our thinking at that level until it is appropriate to go to the next level of decomposition.

[7] L. Balmelli, J. Densmore, D. L. Brown, M. Cantor, B. Brown, and T. Bohn, *Specification for the Rational Unified Process for Systems Engineering—Semantics and Metamodel*, Technical Report RC23966, IBM Thomas J. Watson Research Center, Hawthorne, NY 10532, (May 2006)

[8] Localities in MDSD are a good example of this. See the discussion in chapters 2 and 5.

Multiple views to address multiple concerns

Our life is complicated, our systems are complex.[9] They are built from many parts; often there are many systems working together to accomplish a goal. Our minds do not handle certain kinds of complexity well. In mathematics, when we deal with multi-variable equations, we isolate variables, solve for them, and substitute them back into the equation.

We must provide a mechanism for doing the same thing with systems.[10] We do the same thing when we design and construct buildings. A building is a system. When we construct a building, we draw up many different plans: One for the electricity, another for the plumbing, different views of the exterior. To address the complexity of our systems, we have to create viewpoints that address multiple concerns. These can vary from system to system. Common viewpoints might include the logical viewpoint (what is the functionality), the distribution viewpoint (where does the functionality take place), the data viewpoint (what domain entities are manipulated), and the worker viewpoint (what human roles are involved). MDSD is explicitly designed to promote the creation of different viewpoints to address different concerns.

Integration of form and function

Function does not occur in a vacuum. It is hosted by physical form. Form exists to carry out function. We build systems to accomplish goals. The systems that we build do not exist in a vacuum—they are physical things. The goals that we have for a system, the functionality that we would like it to exhibit, are realized by forms or structures. The form that a system takes must support the goals that we have for it. Both the functionality of the systems and the systems themselves are constrained: we want something to occur within a specified amount of time; we do not want the system to harm its users or innocent bystanders.

Our systems generally must fit into certain spaces, weigh less than a certain amount. The goal of system design is to create a set of forms that will provide desired functionality within a set of constraints. MDSD ensures that system goals are met by distributing functionality across cooperating entities while reasoning about system performance, and other constraints.

[9] See the discussion on increased complexity in Cantor and Roose, *Hardware/software codevelopment using a model-driven systems development (MDSD) approach*, The Rational Edge, IBM developerWorks®, December 2005,
http://www.ibm.com/developerworks/rational/library/dec05/cantor/index.html?S_TACT=105AGX15&S_CMP=EDU

[10] See the discussion of abstraction, decomposition, and other topics in Booch et al., *Object-Oriented Analysis and Design with Applications*, 3rd Edition, Addison-Wesley, 2007, chapters 1 and 2

Two analogies

Consider two analogies here: Project management and restaurant ownership.

Project management

If you are a project manager, you want to complete your project on schedule and within budget. You have a set of people who will carry out a set of tasks. Your job is to schedule the tasks, assign them to workers, and ensure that the project remains on schedule and finishes within budget. Now consider a system to be a project—not the task of building the system, but the system itself. There is a set of tasks that you want the system to perform, you must distribute those tasks to a set of resources, and you want the tasks to be accomplished within a certain schedule, budget, and other constraints. Reasoning about this distribution problem is a core pillar of MDSD.

Restaurant ownership

Now imagine that you want to start a restaurant. Your goals might be varied and personal, but one of them better be to make a profit. There will be many aspects involved in making a profit, but one of them will be to maximize your throughput—that is, to serve as many quality meals as possible to as many customers as possible. You have many options at your disposal to accomplish this. Each option has a cost associated with it. You have to balance costs with the return inherent in each option.

You might start with a short-order cook in front of a stove, behind a counter with stools for the customers. Your rent is low, because you need very little space. Your salaries are low, because you only have to hire a cook or two. But the cook has to invite the customer to sit down, then take the order, cook it, deliver it, and wash dishes. You soon discover that your one employee can only handle a small number of customers at one time, because he or she has to do virtually everything. Your cook is very good, so word gets around. People come to the diner in droves, but soon get frustrated because of the long wait and lack of seating. Your cook gets burnt out, because he or she has to be constantly on the go. The throughput of your restaurant is limited, as are its profits.

You could add tables and some wait staff. Your rent has gone up because your space has increased, as have your salaries because your staff is increased, but you can increase the output of the cook because he or she can focus on the cooking, and the throughput of the restaurant through the division of responsibilities. Still, you will likely be constrained by the capabilities of the wait staff. Now they have to greet the customers, seat them, take their orders, bring them to the kitchen, retrieve the orders, carry them to the tables, give the customers their bills, collect the money, clear the table, and set it again for the next customers. Customers are frustrated because it takes so long to get seated, get their meals, and get their checks. You risk losing customers. So you add staff to clear and set the tables.

You can see how the situation progresses. Many restaurants now have someone to greet the customer, someone to seat them, someone to take their order, someone to pour beverages, someone to cook the order, someone to deliver it to the table, someone to deliver and collect the bill, someone to clear and set tables. The end goals remain the same, the tasks to be performed remain the same, but specialized roles are created to increase the restaurant's capacity and throughput. However, as noted before, the increased capacity comes at a cost, both in increased salaries and increased management complexity—you now have quite a staff to manage. The cost must be balanced against the increased capacity.

Finally, as opposed to suffering through these options by painful experience and trial and error, you could model the various options and run simulations to learn what could happen and to better understand the implications of your options. You might save yourself a lot of pain, suffering, and the loss of your time and money. You would certainly be better informed about your options, and increased knowledge reduces uncertainty and risk.

MDSD provides ways to reason about these issues—both for systems and for business processes.

Scalability: Isomorphic composite structures and recursion

Systems are composite structures; that is, they are made up of distinct pieces. Not only are they composite structures, they are isomorphic;[11] that is, each piece of the composite structure has a similar or identical structure itself. Composite isomorphic structures lend themselves to being processed recursively. MDSD is scalable because it is a recursive methodology. We can use it to reason about a system of any size. At each level of abstraction (or more precisely, at each model level, and at each level of decomposition)[12] we perform basically the same activities: understand the context of the system under consideration, understand the collaboration required to achieve the system's desired goals, and understand how function is distributed across form to achieve system goals within a set of constraints.

Benefits of model-driven systems development

MDSD provides many benefits. These are some of of the more significant ones:

- Reduction of risk
- Enhanced team communication

[11] Isomorphic comes from the Greek ισο (iso) meaning "same" and μορφοσ (morphos) "form"
[12] See Chapter 2 discussion of model levels.

- Explicit processes for reasoning about system issues and performing trade studies
- Early detection of errors
- Integration as you go, better architecture
- Traceability

Reduction of risk

MDSD, in conjunction with appropriate governance, can significantly reduce the risks of system development. The goal of many of the activities of MDSD is to reduce risk. The creation of models is the creation of an architecture. We build models to increase understanding, increased understanding reduces what is unknown both technically in the domain space, and operationally in the project management space—our technical knowledge increases as we complete iterations. At the same time, as we produce concrete deliverables we gain better estimates of time to completion. Increased levels of specificity reduce the variance in a solution space. However, MDSD does not create an artificial level of specificity at any point; the creation of false levels of specificity is often an unrecognized trap leading to false confidence and nasty surprises. Increase in knowledge and reduction of variance are prime risk reducers.

Enhanced team communication

Words can be slippery, elusive, and imprecise. Models can improve communication because they make specific a particular aspect of a system.[13] They also can make system issues *visible* through the use of diagrams. Often it is easier to point to a picture or diagram than it is to describe something in words. The very act of modeling or diagramming can force you to be concrete and specific. We have seen many times in our consulting practice (and many years of experience across many industries) the value of looking at a diagram, set of diagrams, or models. In one customer we worked with, MDSD diagrams were printed out on a plotter, posted in a central lobby, and became the focal point for discussions about the system across a broad set of stakeholders.

Improved communication across a development organization also occurs as a result of MDSD. Engineers in different disciplines have a unifying language they can use to deal with systems issues. Systems engineers can create models that can be handed to the engineers in multiple disciplines (hardware, software, and others) as specification for their design; common use case models can drive system development, testing, and documentation.

[13] Again, see Booch et al., *Object-Oriented Analysis and Design with Applications*, 3rd Edition, Addison-Wesley, 2007, chapter 1: Models provide a means to reason about a part of the system—necessary due to cognitive limits of the human—while maintaining on overall coherence of the parts

Common languages promote common understanding. Unified Modeling Language (UML) and Systems Modeling Language (SysML) derive from the same meta object framework; products in one or the other are likely to be understandable across diverse disciplines. By focusing on usage, collaboration, and distribution, better cross-organizational discussions can take place. Use cases, or common system threads, can unify stakeholders, developers, and users. Beyond systems and software engineering MDSD also provides the framework for reasoning about the integration of concerns across all of the engineering disciplines (for example, thermal, structure, electrical, and navigation).

Explicit processes for reasoning about system issues

Often, many of our design decisions are implicit, the result of many years of experience. While this can be valuable (we do value experience), it can also lead to premature design decisions, or decisions that have not been adequately reasoned through, communicated, tested, or verified.

Complexity also demands explicit processes. A commercial pilot would not think of taking off with a plane full of passengers without a checklist of tasks and safety checks. We follow a repeatable process to improve quality and consistency. By designing the process to address specific issues and risks, we increase our chances for success.

MDSD has been designed to address a specific set of issues in the development of complex systems. Explicit processes also improve communications. Design decisions are taken out of the heads of engineers, documented through models, and progressively refined. In MDSD, process is not just the checking off of steps, but performing repeatable tasks to produce quality artifacts—the quality of the process is judged by the quality of the results—where possible by executable results, that is, a running system or piece of a system.[14]

Early detection of errors

One of the benefits of a well designed process for designing systems is the early detection and resolution of errors. Figure 1-1 shows the cost of errors rising exponentially as they are discovered later in the system development life cycle.

[14] See Walker Royce, *Software Project Management: A Unified Framework*, Addison-Wesley, 1998. Also Kurt Bittner and Ian Spence, *Managing Iterative Software Development Projects*, Addison-Wesley, 2006.

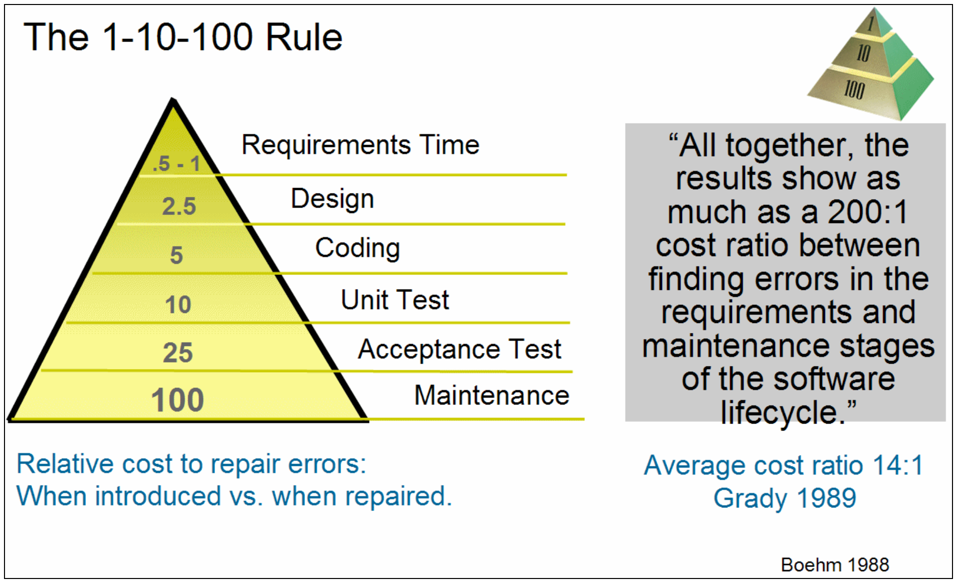

The 1-10-100 Rule
1
10
100
.5 - 1
Requirements Time
2.5
Design
5
Coding
10
Unit Test
25
Acceptance Test
100
Maintenance
"All together, the results show as much as a 200:1 cost ratio between finding errors in the requirements and maintenance stages of the software lifecycle."
Relative cost to repair errors:
When introduced vs. when repaired.
Average cost ratio 14:1
Grady 1989
Boehm 1988

Figure 1-1 High cost of requirements errors

Our experience has shown us that iterating through the production of a set of artifacts improves both the artifacts themselves and the system that is the end product. Each progressive step in the process of defining context, defining collaborations, and specifying the distribution of responsibilities across a set of cooperating entities highlights ambiguities in previous steps, uncovers problems or issues in design, and provides the opportunity to correct mistakes early in the development process at a much lower cost than when they go undetected until later.

MDSD is based on many years of experience across a wide range of customers and projects. We have seen the benefits of well designed activities applied iteratively to a set of concrete artifacts that can be tested.

Integration as you go—better architecture

One of our greatest challenges in developing systems is to integrate functionality successfully, avoid duplication of functionality, and avoid brittle architectures.

Cantor provides the following example:

> *One image satellite ground support system that is currently being fielded was built with a functional decomposition architecture. The system requirements included the ability to plan missions, control the satellites, and process the collected data for analysis. Accordingly, the developer built three subsystems: mission planning, command and control, and data processing. Each of these*

subsystems was given to an independent team for development. During the project, each team independently discovered the need for a database with the satellite's orbital history (the satellites can, to some extent, be steered to different orbits as needed). So each team built its own separate database, using separate formats. But the information needs to be consistent for the overall system to operate correctly, and now, the effort required to maintain these three databases is excessive and could easily have been avoided had the team done some kind of object analysis, including a study of the enterprise data architecture.[15]

MDSD seeks to avoid this kind of duplication of functionality by promoting a breadth-first analysis of functionality across a set of collaborating entities. Collaboration, both in the development process, and in system functionality is at the heart of MDSD.

Traceability

Traceability is usually a requirement for the systems that we build. Often, it is an explicit contract item: You *shall* provide traceability matrices to demonstrate how the requirements of the system have been implemented and tested. Apart from contract requirements, traceability is needed to do effective fault or impact analysis: If something goes wrong, we must determine what caused the fault; if some requirement must be changed, or added, we must determine what parts of the system will be affected.

Providing traceability can be an onerous requirement. Many times it is done manually at significant cost both in the original development and later through testing and maintenance. Manual methods of providing traceability are difficult to maintain and error-prone.

MDSD can help lighten the burden of providing and then maintaining traceability information. Three of the core processes of MDSD, operations analysis, logical decomposition and joint realization tables, allow for a great deal of the traceability problem to be automated. SysML provides semantic modeling support for traceability. The Rational Software Delivery Platform also provides tools and support for traceability.

Well defined semantics

Talking about the various parts of a system, at their different levels, and talking about their relationships, can be difficult and confusing without well defined semantics. MDSD has a well defined meta model which promotes clarity of discussion (see the aforementioned citation[15]).

[15] Cantor, *Thoughts on Functional Decomposition*, The Rational Edge, April 2003, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/apr03/FunctionalDecomposition_TheRationalEdge_Apr2003.pdf

Core processes of model-driven systems development

Model-driven systems development is essentially a simple process, but no less powerful because of its simplicity; in fact, we believe its elegance and simplicity contributes to its power. Furthermore, it is correct in that it is constructed from first principles. It starts with the definition of a system and then provides constructs for defining each of the parts of the system. It also provides an underlying meta model to maintain coherence of the model design as a team reasons about the various parts of the system.[16]

Model-driven systems development is an extension to the Rational Unified Process (RUP). As such, it has a well defined set of roles, activities, and artifacts that it produces. Furthermore it exists as a plug-in for the Rational Method Composer (RMC). Within the context of the Rational Unified Process, however, its essential simplicity is not necessarily immediately apparent within the phases, work flows, and activities. One of the goals of this document is to demonstrate its essential simplicity and power.

The various activities of MDSD are centered around three goals:

- Defining context
- Defining collaborations
- Distributing responsibilities

These activities are carried out at each model level, and at each level of system decomposition. As noted previously, MDSD is a recursive or fractal process—this is part of what makes it simple and powerful.

Defining context

Confusion about context is one of the prime causes of difficulty in system development and requirements analysis. If you are not sure what the boundaries of your system are, you are likely to make mistakes about what its requirements are. Lack of clarity at this point in the development process, if carried through to deployment of the system, can be extraordinarily expensive—systems get delivered that do not meet the expectations of their stakeholders, or faults occur in expensive hardware systems after they have been deployed, and have to be recalled, redesigned, and redeployed. Or the system never gets deployed at all, after millions of dollars have been spent in its development.

Defining context means understanding where the system fits in its enterprise, domain, or ecosystem. Understanding context in a system of systems also means understanding where the various pieces of the system fit and how they relate to each other.

[16] Correspondence with Michael Mott, IBM Distinguished Engineer

One of the most difficult areas of defining or understanding context is being aware of context shifts, especially in systems of systems. A context shift occurs when you go from talking about a system within an enterprise to talking about one of its subsystems. At that point, you are considering the subsystem to be a system in its own right. It will have its own set of actors, likely to be other subsystems of the original system under consideration. It is important to manage these context shifts carefully, and to keep straight where in the system you are at a particular point. Technically, we call this set of levels within the system its decomposition levels.[17] An explicit transformation between black box and white box views are one of the ways MDSD manages this context shift.[18]

Understanding the intended usage of a system is one of the most powerful means of analyzing it and its requirements effectively. Usage drives the functional requirements for the system. What we want the system to do determines its functionality. In MDSD, use cases represent the most important usages of the system. Use cases help define the context of the system; use cases also help put other related requirements into a context.

An essential set of artifacts is produced as we reason about context at any level:

- Context diagram
- Use case model
- Requirements diagram (optional using SysML)
- Analysis model

Defining collaborations

Brittle, stove-piped architectures are expensive and difficult to maintain or extend. MDSD promotes horizontal integration by emphasizing collaborations at the core of the methodology. Even when we are examining the context of a system, we investigate how it collaborates with other entities in its domain or enterprise. As we analyze candidate architectures and perform trade studies, we investigate how the internal pieces of the system collaborate together to realize its functionality.

Scalability is achieved through system decomposition and operational analysis.[19] The interaction of a set of systems at any given level of abstraction or decomposition determines the interactions of subsequent levels.

Essential list of artifacts:

- Sequence diagrams
- Analysis model
- Package diagram/overview of logical architecture

[17] See the aforementioned citation (footnote 15).
[18] See Chapter 2, “Transformation methods” on page 28, and discussion in Chapters 3 and 4.
[19] Ibid.

Distributing responsibilities

Perhaps the greatest challenge in developing any system, but most especially in developing large, complex, systems of systems, is to ensure that all constraints on the system are met while still delivering the desired functionality. How we distribute functional responsibilities across both the logical and distribution entities is the third major theme of MDSD.

Two concepts are used in MDSD to facilitate this. The first is the use of what is called a *joint realization table*. The second is the use of *localities*.

- Joint realization tables help us reason about functionality across a set of system viewpoints—logical, distribution, data, process, and worker, for example.
- Localities help us reason about quality of service measures at a level of abstraction that promotes flexibility in eventual implementation. One of the temptations of Systems Engineering is to jump ahead to an implementation based on experience rather than explicit reasoning and design. Localities are intended to encourage explicit documentation of design decisions and trade-offs. They can form the basis for trade studies in the trade space.

Essential list of artifacts:

- Locality diagrams
- Joint Realization tables
- Deployment diagrams (design level and lower)

Prerequisites/required foundational concepts/languages

Basic familiarity with the Rational Unified Process is assumed, but is not strictly necessary to understand this book.

Iterative development is at the core of the Rational Unified Process. We assume that in any innovative, high-risk project (and what new systems development project is not, in one way or another?) some form of iterative development will be used because it is a major risk reducer.[20]

The Rational Unified Process, and MDSD as an extension of it, are both use case driven. We discuss use cases in Chapter 3, "Black-box thinking: Defining the system context" on page 35, as a core part of MDSD, but we do not cover in detail how they can serve as the basis for effective iterative development; nor how to manage an iterative development project based on use cases.

[20] We do not discuss program or project management as such in this document. For the important role of iterative development, see Walker Royce, *Software Project Management: A Unified Framework,* and Kurt Bittner and Ian Spence, *Managing Iterative Software Development Projects,* (both cited in footnote 14).

For this, readers should refer to RUP's project management discipline and Bittner's book just cited.

UML

Knowledge of the basics of UML is assumed. Readers should be familiar with the basic structure and behavioral diagrams in UML, and should know the pieces that make up the diagrams. They should have knowledge of the basic entities of UML such as classes, operations, use cases.[21]

SysML

This book assumes basic knowledge of SysML.[22]

The most important parts of SysML to be considered in this book are:

- Requirements modeling
- Structure modeling with blocks
- Parametrics

The use of SysML is not required to get benefits from MDSD; however, MDSD is optimized by using SysML semantics and capabilities. SysML was created with the intent to provide richer semantics for systems than UML provides. Some of the central issues that MDSD addresses were drivers behind important semantics in SysML. We will provide discussion of these as they occur in this book.

How the book is organized

This chapter provides an introduction to MDSD. Chapter 2 covers definitions, design points and key concepts, while Chapters 3, 4, and 5 cover the core of MDSD. Chapter 6 discusses model structure and use of Rational Systems Developer to create MDSD artifacts. Chapter 7 gives an overview of those SysML concepts required for MDSD, and suggestions for using SysML with MDSD. These can be read independently, while Chapters 2, 3, 4, and 5 stand as a virtual unit.

[21] There is no lack of material available on UML. A good starting point might be Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd edition, 2003. The standard references are James Rumbaugh, Ivar Jacobsen, and Grady Booch, *Unified Modeling Language Reference Manual*, 2004, and Grady Booch, James Rumbaugh, and Ivar Jacobsen, *Unified Modeling Language User Guide*, 2005

[22] The Object Management Group developed and manages the SysML specification: http://www.omgsysml.org

2

Definitions, design points, and key concepts

To understand MDSD, we must set forth some key definitions and discuss key concepts and design points. This chapter defines important terms as used in MDSD, discusses some of the key concepts of MDSD, and sets out some of the motivations for its design.[1]

[1] This chapter uses material from, and adapts, two articles: L. Balmelli, D. Brown, M. Cantor, and M. Mott, *Model-driven systems development*, IBM Systems Journal, vol 45, no. 3, July/September 2006, pp. 569-585, and Cantor, *Rational Unified Process for Systems Engineering*, The Rational Edge, August 2003. Used with permission.

Definitions

The following definitions are important to an understanding of MDSD. We provide them here for clarity in the discussions in the rest of the chapters of this book.

System

A *system* is a set of resources that is organized to provide services. The services enable the system to fulfill its role in collaboration with other systems to meet some useful purpose. Systems can consist of combinations of hardware, software (including firmware), workers, and data. This definition of systems is extremely general: a product, such as an automobile or a computer, is a system; a business or its components are also systems. Businesses can be organized into larger enterprises that are also systems, for example, the health-care system.

Service

At a high level, a *service* is a mechanism by which the needs or wants of the requestor are satisfied. In a given context, the term service represents either a service specification or a service implementation, or both. A *service specification* is the definition of a set of capabilities that fulfill a defined purpose. A *service implementation* realizes the behavior described in the service specification and fulfills the service contract.

In MDSD, the service specification can be a UML or SysML interface. The service implementation is represented by the logical and distribution projections or viewpoints of the model.[2]

Requirement

A *requirement* is a condition or capability to which the system must conform.

Model

A *model* is defined as a collection of all the artifacts that describe the system.

[2] Wikipedia's article on Service (System Architecture) defines service as follows: *In the context of enterprise architecture, service-orientation and service-oriented architecture, the term service refers to a discretely defined set of contiguous and autonomous business or technical functionality.* Organization for the Advancement of Structured Information Standards (OASIS) defines service as *a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.* In this document we use the term somewhat loosely, as defined in the text.

Generally, model-driven development (MDD) is a technique for addressing complex development challenges by dealing with complexity through abstraction. Using this technique, complex systems are modeled at different levels of specificity. As the development program proceeds, the model undergoes a series of transformations, with each transformation adding levels of specificity and detail.

This last quote is very important in regard to the process to be described in the following chapters, and also sets the stage for the possibility of automation through transformations as in Rational Software Architect and Rational Software Modeler (RSx).

Artifact

An *artifact* is defined as any item that describes the system, including a diagram, matrix, text document, or the like.

Use case

A *use case* is a sequence of events that describes the collaboration between the system and external actors to accomplish the goals of the system. In other words, the use case is a way to specify the behavior required of the system and external entities in response to a given sequence of stimuli.

This definition is different from the standard definition of use case as found in virtually all the literature on use cases. The authors of the Systems Journal article explain:

> *In working with the systems community, who typically interact with large teams requiring precise communications, we found that the common informal definition of a use case (namely, a description of a service that the software provides, which provides value to the actor) is inadequate for a variety of reasons. A service … is a behavior of the system. The actual semantics of use cases more closely resemble collaboration than behavior. Value is far too subjective a term to be included in the definition of a framework element. In any case, the entity receiving benefit from the system behavior might not include the actors in the collaboration. In addition, the software definition of a use case does not provide for scalability.*

This definition provides scalability because it is isomorphic with the definition of an operation, that is, they both consist of a sequence of events. In fact, the difference is one of context, as will be seen below. Operations at any given level are instances of one or more use cases for entities at the next lower level. Also note that this does not emphasize a *sequence of steps*, but rather emphasizes the collaboration.

Operation

An *operation* is defined as follows in the UML 2.0 specification:

> An operation is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior.

The MDSD meta model defines operations as follows:

> An operation represents a service delivered by a system.

Actor

An *actor* is anything that interacts with the system. Examples of actors include users, other systems, and the environment, including time and weather. There is often confusion between *users* and *workers*. In systems engineering, users are external to the system, and thus are actors. The specification of workers in a system is captured in the worker viewpoint[3]—that is, how one would elaborate on what the workers must do, and how to produce a set of instructions for them.

Locality

Finally, we explain a concept introduced by Cantor to facilitate reasoning about the distribution of functionality across physical resources, localities.

A *locality* is defined as a member of a system partition representing a generalized or abstract view of the distribution of functionality. Localities can perform operations and have attributes appropriate for capturing non-functional characteristics.[4]

Localities can be represented either as stereotyped SysML blocks or as stereotyped UML classes.

Associated with localities are connections. Figure 2-1 shows two localities and one connection.

[3] This document discusses the difference between actors and workers, but does not deal in detail with the worker viewpoint.

[4] Original discussion of localities occurs in M. Cantor, *RUP SE: The Rational Unified Process for Systems Engineering*, The Rational Edge, November 2001, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/archives/nov01.html

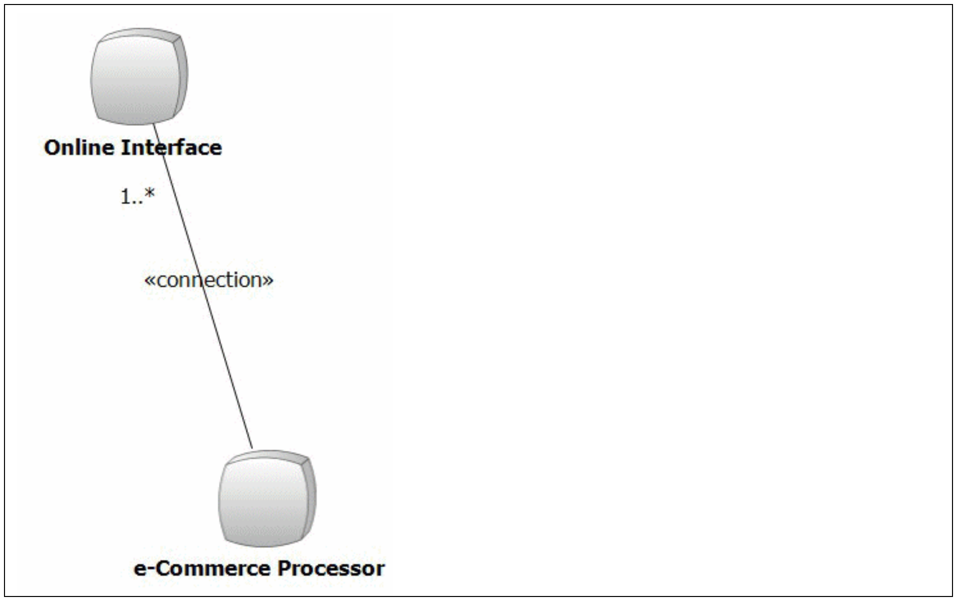


Figure 2-1 Two localities and a connection

Connection

Connections are defined as generalized physical linkages. Connections are characterized by what they carry or transmit and the necessary performance and quality attributes in order to specify their physical realization at the design level. They are linked to the concept of a *flow port* in SysML, which allows the designer to specify what can flow through an association and its ports (data, power, fuel).

In UML, connections are represented by stereotyped associations.

Design points

MDSD is intended to provide a framework for reasoning about the whole spectrum of systems concerns.

Four basic principles

MDSD provides support for constructing a sound architecture on the basis of four principles: separation of concerns, integration, system decomposition, and scalability.

Separation of concerns

Separation of concerns allows developers to address each set of stakeholder concerns independently.

Integration

Integration is achieved by requiring the use of a common set of design elements across multiple sets of concerns.

System decomposition

System decomposition subdivides the system by structure, rather than by function, enabling the framework to provide levels of structure that enable parallel development.

Scalability

Scalability is achieved by using the same framework, whether the system under construction is an enterprise or a product component or anything in between.

This last point gives MDSD great power and elegance—we can use it to reason effectively about any system, from organization to product component. It dispenses with artificial complexity introduced by having a different methodology at each level, and identifies powerful abstractions common to each. It creates a methodology that is easily internalized by practitioners and applicable to many domains.

Additional design points

The design of MDSD is also intended to:

- Apply the RUP framework to systems development
- Employ the appropriate semantics and modeling languages
- Provide tool assets
- Maintain all model levels as program assets

Let us now take a look at each one.[5]

Apply the RUP framework to systems development

The RUP life cycle and disciplines are shown in Figure 2-2. MDSD follows the RUP in these ways:

- **Life cycle**: Focusing on removing risks, MDSD follows RUP's four phases by leveraging the team's evolving understanding of the project details.

[5] The following material is adapted from Cantor, *Rational Unified Process for Systems Engineering*, The Rational Edge, August 2003.

- **Iterations**: MDSD advocates a series of system builds based on risk identification and mitigation; an iteration will generally include at least one system build. In particular, all of the artifacts, including the detailed project plans, evolve through iterations. A key feature that RUP SE inherits from RUP is a rejection of waterfall development and the use of iterative development.
- **Disciplines**: MDSD follows the focus areas, or *disciplines* shown in Figure 2-2, which provide a number of views into the underlying process definition and the effort that will be carried out by the team in developing the system. Although the RUP project team contains systems engineers, there is no separate systems engineering discipline. Rather, systems engineers play one or more RUP roles and participate in one or more RUP disciplines. Note that the disciplines' work flows and activities are modified to address broader system problems. These modifications are described in the following sections.

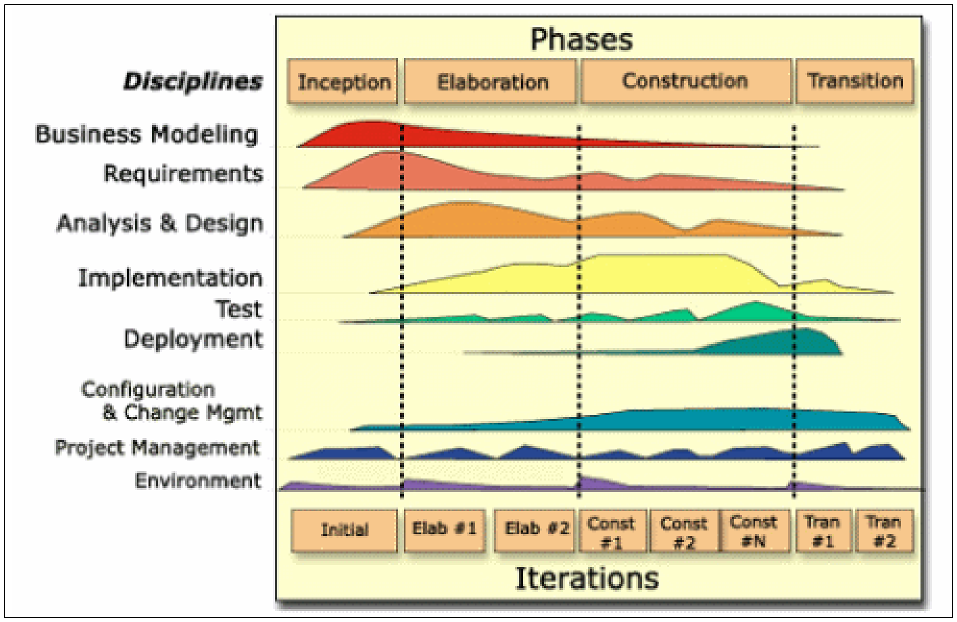


Figure 2-2 RUP Process Framework (adopted by MDSD)

As explained next, MDSD supplements RUP with additional artifacts, along with activities and roles to support the creation of those artifacts. These are described in more detail in “Creating MDSD artifacts” on page 109.

In addition, as a RUP framework plug-in, MDSD provides the opportunity to employ these underlying RUP management principles to systems development:

- Results-based management
- Architecture-centric development

Employ the appropriate semantics and modeling languages

SysML was developed in response to the same kind of issues that MDSD addresses. In fact, concepts from MDSD influenced the design of SysML: Several of the constructs in SysML were developed with MDSD (or RUP SE at the time) in mind. In particular, the use of parametrics enables effective reasoning about many systems engineering concerns.

However, you can also use UML 2.0 to express MDSD concepts. This document is written to accommodate the use of both modeling languages.

Provide tool assets

To support MDSD, IBM Rational Software provides an RMC plug-in that describes the MDSD extension to RUP in detail, along with Rational Software Delivery Platform (SDP) and Rational RequisitePro® tool add-ins.

Maintain all model levels as program assets

A systems life span often outlasts the initial requirements and enabling technologies. Over time this leads to either outdated or otherwise insufficient functionality, or unacceptably cost of ownership. It follows, therefore, that an effective architecture framework should maintain model views at increasing levels of specificity: The top levels establish context and specification; the lower levels establish components and bills of materials. Traceability should be maintained throughout.

Maintaining these levels provides the setting for reasoning about the impact of the changes. Changes in mission usually results in changes at the top level in the model that flow to the lower levels. Changes in technology permit either different design trades or different realizations of the current design. MDSD provides the needed model levels and traceability.

Key concepts

The MDSD framework consists of two kinds of artifact: *static artifacts*, namely, representations of the system in its context and the things that comprise the system; and *dynamic artifacts*, namely, how the static elements collaborate to fulfill their role in the system. The static artifacts enable separation of concerns and scalability and provide the semantics for system decomposition. The dynamic artifacts enable integration of concern. The framework consists of three types of element, namely *model levels*, *viewpoints*, and *views*.

Model levels

A model level is defined as a subset of the system model that represents a certain level of specificity (abstract to concrete); lower levels capture more specific technology choices. Model levels are not levels of decomposition; in fact, a model level can contain multiple levels of decomposition.

Model levels are elements designed to group artifacts with a similar level of detail and are customizable to meet your needs and terminology. However, the levels discussed in the following have proved to be useful in practice (Table 2-1).

Table 2-1 Model levels in the RUP SE architecture framework

Model level	Expresses
Context	System black box—the system and its actors (though this is a black-box view of the system, it is a white-box view of the enterprise containing the system)
Analysis	System white box—initial system partitioning in each viewpoint that establishes the conceptual approach
Design	Realization of the analysis level in hardware, software, and people
Implementation	Realization of the design model into specific configurations

Context level

The context level treats the entire system as a single entity, a black box. This level addresses the system's interaction with external entities.

Note that in Table 2-1 the system black box is a white-box view of the enterprise. Understanding this shift in context is essential to success with MDSD. That is, when we expand the enterprise black box to a white-box view, the system and other entities in the enterprise will be represented. When we shift our focus to a system black box, the other entities will be its actors.

Analysis level

At the analysis level, the system's internal elements are identified and described at a relatively high level. Which elements are described at this level depends upon the viewpoint. For example, in the logical viewpoint [see Table 2-2], subsystems are created to represent abstract, high-level elements of functionality. Less abstract elements are represented as sub-subsystems, or classes. In the distribution viewpoint, localities are created to represent the places where functionality is distributed.

Design level

At the design level, the decisions that drive the implementation are captured. In the transition from the analysis to the design level, subsystems, classes, and localities are transformed into hardware, software, and worker designs. This is *not* a direct mapping from system elements to designs, rather, design decisions are made by deriving the design from the functionality represented in the subsystems and classes. These design decisions are constrained by the supplementary requirements and distribution choices represented by the localities and their attributes. The resulting design transformation realizes all of the specifications from the analysis level. In other words, the system architecture is specified at the analysis level, creating requirements that the design level must satisfy.

Implementation level

At the implementation level, decisions about technology choices for the implementation are captured. Commercial products can be specified, or items might be specified for internal implementation. As before, moving from the design level to the implementation level is a transformation, but this time the mapping is more direct. For example, at the design level, the functional activities of a worker are mapped to a position specification with a defined set of skills. Then, at this level, the specification can be fulfilled either by hiring someone with the correct skill set (similar to choosing a commercial product with certain capabilities) or by training an individual to acquire the required skills (similar to doing an internal implementation).

Viewpoints

A viewpoint is defined as a subset of the architecture model that addresses a certain set of engineering concerns. The same artifact can appear in more than one viewpoint. Viewpoints allow framework users to separately address different engineering concerns while maintaining an integrated, consistent representation of the underlying design. Table 2-2 describes the core RUP SE viewpoints.

Table 2-2 Core SE RUP viewpoints

Viewpoint	Expresses	Concern
Worker	Roles and responsibilities of system workers	Worker activities, human.system interaction, human performance specification
Logical	Logical decomposition of the system as a coherent set of SysML blocks that collaborate to provide the desired behavior	► Adequate system functionality to realize use cases ► System extensibility and maintainability ► Internal reuse ► Good cohesion and connectivity
Distribution	Distribution of the physical elements that can host the logical services	Adequate system physical characteristics to host functionality and meet supplementary requirements
Information	Information stored and processed by the system	Sufficient system capacity to store data; sufficient system throughput to provide timely data access
Geometric	Spatial relationships between physical systems	Manufacturability, accessibility
Process	Threads of control that carry out computational elements	Sufficient partitioning of processing to support concurrency and reliability needs

The set of viewpoints is fluid and has grown over time. Most development efforts do not require all of the viewpoints shown in Table 2-2. Further, viewpoints are extensible to address program domain specific needs, such as security or safety. Generally these extended viewpoints can reuse the semantics of the core set of viewpoints.

A particular viewpoint might not be useful at all model levels. For example, hardware developers are a category of (internal) program stakeholders concerned with the allocation of functionality and distribution of hardware within the system. However, at the analysis model level, decisions about where functionality will be implemented (in hardware, software, or workers) have not yet been made. As a result, there is typically no need for a hardware viewpoint at the analysis model level. However, if the system involves actual hardware development, then one certainly does need a hardware viewpoint at the more specific (lower) model levels.

Although different architectures require different sets of viewpoints, almost all require the logical and distribution viewpoints.

Views

Views constitute the intersection of viewpoints and model levels. Views contain artifacts (that is, objects used to document engineering data) that describe how the viewpoint's engineering concern is addressed at a particular model level. Table 2-3 includes a sample set of view artifacts. In practice, each program chooses the view artifacts that meet its individual needs. The project's set of view artifacts is what the RUP calls the development case, which includes the choice of artifacts and prescriptive guidance on how to document them, along with guidelines, templates, and checklists.

The framework might leave the impression that the views contain unrelated artifacts. In reality, there are many relationships between the artifacts. These relationships are captured in the MDSD meta model.[6]

Table 2-3 RUP SE architecture framework (cells shows sample model views)

Model levels	**Model viewpoints**					
	Worker	**Logical**	**Information**	**Distribution**	**Process**	**Geometric**
Context	Role definition, activity modeling	Use case diagram specification	Enterprise data view	Domain-dependent views		Domain-dependent views
Analysis	Partitioning of system	Product logical decomposition	Product data conceptual schema	Product locality view	Product process view	Layouts
Design	Operator instructions	Software component design	Product data schema	ECM (electronic control media) design	Timing diagrams	MCAD (mechanical computer-assisted design)
Implemen-tation	Hardware and software configuration					

Transformation methods

MDSD includes novel, related artifacts for transformation methods between model levels. The generation of these artifacts and their relationships requires new techniques. These techniques are described next.

[6] See footnote 7 on page 5

MDSD starts with system decomposition, that is, the division of a system into elements in order to improve comprehension of the system and the way in which it meets the needs of the user. Because of the limited capability of humans to understand complexity, a *divide and conquer* system decomposition approach is appropriate.[7] In this approach, the system is decomposed into a comprehensible set of elements, each of which has a comprehensible set of requirements. Sometimes, to manage complexity in very large systems, system decomposition must be applied recursively.

Effective application of system decomposition requires the means of modeling the system from a variety of viewpoints and at increasing levels of specificity. In addition, a set of transformations between model levels is required as a basis of the development process. These transformations provide a means of deriving the next level of specificity while maintaining traceability and coherence for the entire model. MDSD consists of creating the model artifacts as a means of specifying the system elements and their integration. An artifact is defined as any item that describes the architecture, including a diagram, matrix, text document, or the like. This model provides a common means for facilitating collaboration across the engineering disciplines, coordinating iterative development methods, and assigning technical and managerial responsibilities.

System of systems decomposition

In this subsection, we describe a method of object oriented logical decomposition to describe a hierarchical system of systems. Additionally, we discuss a number of principles, found in traditional systems development, that underpin the MDSD framework discussed.

A system encapsulates the resources it requires to deliver its services. Systems can be decomposed into systems, each of which also encapsulates all of their resources. Because systems control their resources and can encapsulate other systems, a *system of systems* is a recursive pattern. A process can therefore be applied to recursively decompose a system into other systems, which are themselves decomposed further. During such recursive decomposition it is important to understand at which *level* in the hierarchy we stand during a discussion. Although terms such as *superordinate system* and *subordinate system* are relevant when discussing the pattern, it is sometimes more useful to discuss *system levels* because more than two levels can be considered.

The term *system level* indicates the relative position in the overall hierarchy: System level 1 represents the root system (by definition, there is always exactly one system level 1 system). An overview of the key artifacts in two system levels is shown in Figure 2-3.

[7] B. Blanchard and W. Fabrycky, *Systems Engineering and Analysis*, 3rd Edition, Prentice Hall, 1998

Figure 2-3 shows the pattern that allows the framework to support recursive system decomposition. The dotted lines between the systems indicate UML dependencies.

These system levels are called decomposition levels in the MDSD meta model.

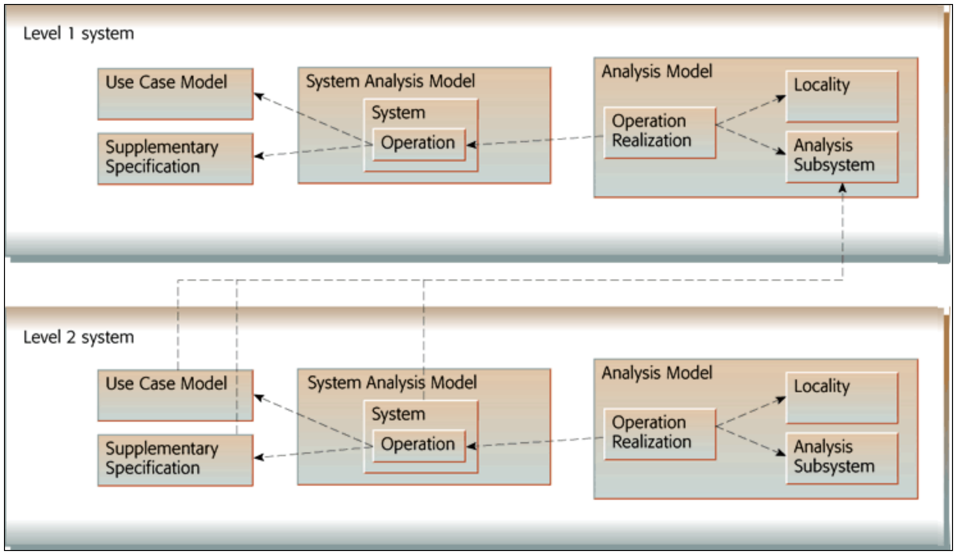


Figure 2-3 Levels of system decomposition

Operations analysis

Classical use case analysis is a form of requirements decomposition; therefore, it is inadequate to meet the needs of systems development.[8]

[8] L. Balmelli, D. Brown, M. Cantor, and M. Mott, *Model-driven systems development*, IBM Systems Journal, vol 45, no. 3, July/September 2006, p. 571: Requirements-driven systems development methods define requirements early in the life cycle, after which the techniques of functional decomposition are applied to determine the mapping of requirements to system components. At every level of the hierarchy, functional analysis derives requirements, and engineering methods derive measures of effectiveness. Once the requirements are described in sufficient detail, detailed design activities begin. As systems become more complex and integrated, with fewer components delivering more capability, this traditional approach becomes unwieldy due to the large number of possible mappings. It is common for a modern system, such as an automobile, to have thousands of detailed requirements and thousands of components, resulting in millions of possible mappings. Faced with this dilemma, developers limit the level of integration, resulting in systems that may be highly capable but are brittle and difficult to maintain. MDSD methods mitigate this explosion of mappings by providing levels of abstraction.

In MDSD, the techniques of use case analysis are extended to operations analysis. Operations analysis consists of the following recursive pattern:

1. Decompose the system to create a context for the system elements.
2. Treat the system operations as use case scenarios for the elements.
3. Describe the scenarios in which the elements, as black boxes, interact to realize the system operations.
4. Derive the operations of the elements from the scenarios.

This pattern can be applied starting at the enterprise, which contains the system of interest (hence the context level for the MDSD framework). In this application of the pattern, the enterprise is treated as a system and the system to be developed as a component.[9]

The system decomposition creates the context for the elements; thus, context is maintained at every level of the system hierarchy. The operations analysis provides a method for creating traceability between the use cases, which define the business or mission needs, and the system components that satisfy those needs. The maintenance of this context at each level of the hierarchy was a key insight during our development of MDSD. The use cases at the top level of the system hierarchy define the interactions of the system with external entities in order to fulfill its mission. These interactions are analyzed to identify the operations that the system provides in order to fulfill its role in the use cases. Operations analysis forms the basis of the use case realization. The operations are combined into interfaces or services.

Operations analysis uses sequence diagrams to recursively derive system component black-box requirements at every level of the hierarchy. An operation realization is created for each operation, and the realization is performed concurrently across the system components identified in the architectural analysis activity. This will be treated at greater length below and in “Operation analysis” on page 72.

Joint realization

In developing the system model, use cases are written, system components are defined, and the interactions between the components are described. This is standard practice for modeling a system. For large-scale developments, we must design across multiple viewpoints concurrently, distributing functionality to the various pieces of the system. We also decompose the system, divide and suballocate the requirements, and develop links for traceability purposes.

[9] This is elaborated in chapters 3 and 4.

The new mechanism for connecting all of these items is a joint realization table (JRT). The joint realization method is how the JRT is completed, and is therefore the process by which decomposition is accomplished within MDSD. Joint realization is covered in “Joint realization” on page 86.

Requirement derivation

With current requirements-driven development methods, the system’s nonfunctional requirements (NFRs) are often found in a software requirements specification or similar document. The engineers decompose the functional requirements and then document them in a specification tree. The objective is to continue to suballocate functionality into ever-finer levels of granularity until the details are sufficiently documented for development to proceed. MDSD differs from this approach by decomposing the system into components, in contrast to traditional methods that decompose the requirements into a specification tree. MDSD is able to recursively define the component architecture at each level of the hierarchy; after this, the NFRs are suballocated to the components. The JRT is used in this approach to link the system behavior, logical components, distribution components, and NFRs into a coherent model that maintains context and traceability throughout the system analysis. With this method, MDSD provides a robust means for system decomposition and modeling.

Summary: The core MDSD process

We have discussed a set of transformations that form the basis of MDSD.

The first transformation is black box to white box, from specification to realization. This is both structural and behavioral; we decompose the system structurally (system → subsystems) through system decomposition. We decompose the system behaviorally in the context of collaborations through operation analysis. We unify these transformations with joint realization.

First of all, we would like to point out the alternation between specification and realization—in the black-box view, we specify or derive the functional requirements (use cases and operations), the constraints on those functional requirements, and we specify the constraints on the system as a whole. These requirements are analyzed in the context of collaborations with system's actors.

In the white-box view(s), we analyze how the system will *realize* those requirements, and how it will meet the constraints imposed on it (both constraints on the behavior and constraints on the system itself). This involves understanding collaborations across multiple viewpoints. We look at both the collaborations from the perspective of a single viewpoint with sequence diagrams, and across multiple viewpoints with joint realization tables.

Operation analysis also involves a black box to white box transformation—first we specify the system operations (derived from the white-box *enterprise* analysis) in a system black-box view, then we realize those operations in system white-box diagrams consisting of collaborating subsystems.

The aforementioned alternation noted occurs in both the model levels and the system decomposition levels:

- In model levels, specification at one level is realized in the next. Note, however, that the realization becomes the specification for the next lower level. So, specification at the context level is realized in the analysis level. This is turn is the specification for the design level.
- In system decomposition level, specifications at the enterprise level [or level N] are realized in the system level [or level N + 1]. This set of realizations becomes the specifications for the subsystem level.

We discuss these transformations in detail in the following chapters.

3

Black-box thinking: Defining the system context

Model-driven systems development helps to manage the complexity of designing a system. This chapter discusses the importance of understanding context, how context drives usage, and how usage helps us discover requirements that ensure that the system meets the stakeholder needs.

The importance of understanding context

Understanding context is critical in creating systems that accomplish the goals for which they are built.

> *In systems engineering, context includes the set of things (people, other systems, and so forth) with which the system interacts and how those interactions proceed so that the system can fulfill its role in the enterprise.*[1]

Understanding context, then, means understanding the interaction of the system with entities external to it (actors), understanding the services required of the system, and understanding what gets exchanged between the system and its actors. Understanding context is also important for ensuring that the appropriate requirements exist or will be developed.

Managing context explicitly means being aware of the shifts in context as you go from one model or decomposition level to the next. In this chapter we discuss how to delineate the boundaries of the system, how it relates to its enclosing enterprise, and how we proceed from a black-box perspective to a white-box perspective while maintaining context.

Context and description

Describing something seems at first glace to be a simple task. In practice, however, a number of issues arise. Consider an ordinary pencil. How would you describe it? While it is tempting to leap into writing an actual description, consider the question literally.

> *How* would you describe the pencil, that is, how would you proceed to arrive at a description?

The answer depends on the viewpoint from which the describer is operating. Now imagine that you are an engineer working for a pencil manufacturing company. Does this viewpoint affect how you would describe the pencil? Certainly—you would probably focus on the construction aspects, dimensions, specifications, and materials of the pencil. An accountant from the same firm might focus on the labor and material costs of the same pencil. A buyer from an office supply company would likely be more interested in the price, packaging, and market appeal of the pencil.

How you describe something depends on your particular viewpoint. Which description is the real one, or the right one? Of course, none is more real or right than any of the others—all have their purposes.

[1] Balmelli et al., *Model-driven systems development*, IBM Systems Journal, vol. 45 no. 3, p. 576

The system in context

In MDSD, we consider multiple viewpoints in describing a system. We must make choices about what to describe, where to start, and how to know we are done. To begin, we place the system in its context. This might seem like an obvious step, but many systems are described without reference to their context; or, if context is considered, it does not play a central role in the development methodology. It is natural to describe the pencil in isolation, considering only, or mainly, the attributes and qualities of the pencil in a vacuum, so to speak.

If we wish to describe the pencil in its context, then we must first choose the context in which the pencil exists. We might consider the pencil as a stock-keeping unit (SKU) in an inventory system. This would give us one kind of contextual description. Yet another context would be the pencil as an item being manufactured, a participant in the many shaping, assembly, and finishing processes it undergoes. The context we choose is determined by our needs.

Consider also a car. The context in which we intend to use it will determine many of its features and requirements. If it is to be used in an urban setting for daily transportation, it will be a very different car than a stock car to be raced on a track, or a Formula One racer. The context will impose a different set of features and services required from the car.

An important context: Usage

In MDSD, one of the most important contexts to consider is usage, that is, how a system is used, and how it interacts with entities outside itself as it is used. Why? Because our purpose is to develop a system, or enhance an existing one, one of our most important considerations should be that the system is *useful*. If we can base our designs on the actual usages to which the system is to be put, we will be assured that we build what is needed. After all, systems are built to be used!

Relating this to a set of services is fairly straightforward. The system will be used through the services it provides. In fact, the usage provides context for the services. How the system will be used, either by people or other systems, helps determine what services the system needs to provide.

This dynamic—of describing a system in the context of its usage—might seem completely obvious, but in our experience it is rarely done, or if done, is minimized in importance. Most large systems are built based on requirements written by teams of people with varying ideas and requirements, each with some idea of how the system is to be used. Seldom is a unified and comprehensive picture of the system's usage created. Required features of the system are listed and even elaborated, without being connected to actual usages.

The process is ironic in fact. Those writing the requirements for the system clearly imagine using the system as they write, but what they write are requirements, features, and attributes. They usually do not fully describe the usages they are imagining that give rise to those features. Then, system engineers and designers read these requirements and attempt to re-imagine how the system will be used! Misunderstandings and unfortunate assumptions result in a system that is only a partial fit for the intended uses.

Even when a document (or documents) such as a CONOPS (concept of operations) is provided, the context is not maintained, nor is traceability provided throughout the whole development process.

So, while there are many possible contexts from which to describe a system, the most important one is its usage. By placing a system in the context of the people and other systems with which it interacts, identifying the usages that deliver value, and describing the precise nature of those usages, we describe a system in the most *useful* way possible!

Usage-driven versus feature-driven system design

To make this important idea clear, let us consider an example. Automobile navigation systems based on the Global Positioning System (GPS) satellite network have become fairly common in recent years. From examining and comparing these systems and how they operate, it seems clear that for the most part, they were designed by considering the features they should have instead of the usages they should perform. If a designer (or more likely, a committee of designers) were to sit down and try to write the requirements for a new GPS navigation product, they would likely write a list of features similar to this:

- **GPS navigation system features**:
  - Plot route from current location to an address.
  - Enter addresses by choosing the city, then street, then street number.
  - Select fastest, shortest, or highway-avoiding routes.
  - Locate nearest point-of-interest by category (restaurant, fuel station).
  - Display remaining distance and time to destination.
  - Resume navigation to destination after power on.
  - Warn when off route and re-route based on new current location.
  - Retrace my route back to my starting point.

Nothing here is bad or incorrect. Such a list, however, ignores a number of important aspects of how such a system might be used in actual practice. If, instead of trying to list features, the designers try to list how the system will actually be used, quite a different picture emerges. Asking *What will the system be used for?* instead of *What should the system do?* produces a list more like this:

- **GPS navigation system usages**:
  - Help me identify my destination using the information I know.
  - Guide me to a destination.
  - Find a Mexican restaurant that is on my way to my destination.
  - Show me the hotels that I can reach in about 5 more hours of driving.
  - Where are the truck stops in the cities I will pass through today?

This is quite a different kind of list. By describing the actual usages to which the system will be put, and basing our designs on those, we are assured that the system we design will meet the real needs. It is also interesting to note that many of these usages can be accomplished with little additional development effort, and no additional hardware. They are a matter of imagination. By combining existing elements, we can perform interesting new usages, provided we imagine these in our design process.

The important question to ask at this point is, *What is the relationship between the features and the usages?* The answer to this is one of the keys to understanding the MDSD modeling process. Usages are, in a way, combinations of various features or services arranged in a sequence so as to provide value.

Instead of using a set of features as the sole statement of requirements of a system, what if we were to describe a comprehensive set of system usages, and then from these, derive the necessary features and functions? This would result in an architecture optimized for usage. We would be sure that we have all of the capabilities needed to perform (or *realize*) the usages, and we would be sure we have not required any unnecessary functions.

Then, if we took it a step further, and used the same usage-based models to design subsystems and components within the overall system, we could provide comprehensive traceability. We could show precisely how even the most minute operation of a component contributes to particular system usages. Changes to any part of the system could be analyzed for impact to all other system elements, and we would be assured of complete requirements coverage.

This is the kind of model MDSD can produce through system decomposition and operation analysis, as introduced in “System decomposition” on page 22 and “Operations analysis” on page 30, and explained in “Operation analysis” on page 72. Of course, we still have to consider how constraints on functionality and on the system itself will influence the architecture, and we will do that when we consider localities and joint realization.

MDSD Step 1: Define the system context

Defining the system context is the first step in the MDSD process.[2] First of all, we define the context of any system to be an enterprise. If we consider the system to be level 1 in system decomposition levels, then the enterprise is level 0. As noted before, this also applies more generically—the entity under consideration, our *system* is level n in some hierarchy of system decomposition, and our *enterprise* is level n-1.

By examining the enterprise, its goals, and its components, we will understand the system in its context. The goals of an enterprise will be realized by its collaborations with external entities and supported by the collaboration of internal components. These internal components (or entities, to use a slightly less overloaded term) will collaborate through a set of enterprise operations to support the enterprise's collaboration with its enclosing context. Any enterprise operation that our system under consideration participates in will in fact be a candidate, if not an actual, system use case. To determine what the enterprise operations are, we must analyze the enterprise's use cases and actors. In other words, we must understand the collaboration of the enterprise with its actors to discover its operations. These operations lead to system use cases. Additionally, the other internal entities of the enterprise are usually our system's actors.

Actors and boundaries

In the following sections we discuss discovering actors and use cases as part of understanding the context of the system under consideration.

MDSD Step 2: Finding actors

After choosing an entity in your MDSD model, the next step is to find actors for this entity.[3] Actors represent the roles played by entities (either a person or another system) in relation to the entity under consideration. By definition, they are outside the entity and interact with it.

For example, if we are building a guidance system within a commercial aircraft, and the aircraft is our entity, then it is likely the passengers would be its actors, while the captain and crew can be represented as part of the aircraft, and thus are not actors. To be a little more exact, we are not representing the passenger as an actor, we are actually representing the passenger role. Actors represent the roles played by people and outside systems in relation to our entity. Other actors for the commercial aircraft might include the control tower, regional air traffic control center, and the ground crew.

[2] See also Task: Define the system context in the Rational Unified Process (RUP) v7

[3] Ibid, Task: Find Actors and Use Cases

Finding actors in the MDSD modeling process is only slightly different from finding actors in ordinary software-focused use case modeling. The difference is usually one of scale or context. With a software application as the system, we are really only looking for people and systems that interact with, or use the application to be our actors. With actors in MDSD we take a broader view, and must look for any entity that interacts with ours. This term *interact* is important. Not all things that touch a system interact with it. For example, should rain be an actor to the aircraft? Well, it depends on whether the aircraft has a requirement to interact with the rain. If, for instance, as with some cars, the presence of rain triggers the windshield wipers and defogger, then the rain is indeed causing an interaction and should be shown as an actor.

In finding actors we are looking for entities that take part in interactions that involve system functionality. Remember that the purpose of the model is to describe system functionality through usage scenarios, so it is the participants in those scenarios that we seek for actors. Can inanimate, passive objects be actors? Probably not, unless they are systems themselves. A voting machine does not interact with a ballot, nor does a gun interact with the bullet. These items will be captured later in the model as I/O entities.

Primary and secondary actors

Primary, or *initiating* actors are those who initiate system usage while secondary, or *participating* actors, are those who interact with the system in the course of it performing some function initiated by a primary actors. As I order a book from an online store, that store's system interacts with my bank's system to validate my credit card. To the store's system, I am a primary actor (customer) and the bank system is a secondary actor. The bank system only interacts with the online store system in the processing of doing something for me. Without me, there is no need for an interaction with the bank. This is not to say that primary actors are more important than secondary actors, or that somehow the system is more *for them*. The notion of primary and secondary actors is important because not all actors will initiate usages of the system—some will simply participate in usages initiated by others.

Note that we cannot designate primary and secondary actors as such in the model, because a particular actor might function as the initiator of one system usage, while being only a participant in another. We simply use this distinction to aid in discovering all of the actors. Often, primary actors are mentioned first, and in thinking about what the system does for them, other secondary or participating actors are discovered as well.

A common trap that befalls new MDSD modelers, is to try to come up with usages for all of the actors discovered. Because some of the actors will be secondary (participating) actors, they will not have their *own* use cases.

For example, take an online bookseller. Actors identified are the customer and the bank credit card system. Both are valid actors, though it is likely only the customer will be a primary actor who initiates a system usage (purchase book). The bank credit card system will likely turn out to be a participating actor in this usage.

At this stage in the modeling process we seek to identify all actors—those who will turn out to be primary, secondary, or both.

Questions to discover actors

The following questions, based on those used in software application use case modeling, can be helpful in identifying actors:

- Who/what uses the system?
- Who/what gets or receives something from this system?
- Who/what provides something to the system?
- Where in the company (or in the world) is the system used?
- Who/what supports and maintains the system?
- What other systems use this system?
- What outside conditions or events must the system detect and respond to?
- Who/what can request or command the system to do something?
- Who/what must the system communicate with to do anything identified in the aforementioned questions?

Actors and value

Value is a difficult term to define clearly.[4] Most definitions of actors state that a use case always provides a meaningful result of value to the actor. In reality, it is easy to see that while value is always created by a use case, it is not always the actor who receives that value. Take the case of a payroll clerk printing paychecks using a payroll system. Does the payroll clerk receive value from this? Perhaps, if one of the paychecks is the clerk's own, but the lion's share of the value accrues to the enterprise itself. An even more vexing case is the common situation in aerospace and defense systems of a system firing a weapon at an enemy target. Clearly the enemy target is an actor, but does it receive value? One could perhaps say whimsically that it receives negative value, but the clearer answer is that the firing of the weapon produces value for the enterprise by defending the fleet, or maintaining a position.

In MDSD, we find it best to simply require that use cases provide a meaningful result of value, without requiring that the value be assigned to an actor.

[4] See "Use case" on page 19

In this actor discovery process, two opposite concerns often emerge. To some it seems that the identification of the actors is a limited, even trivial concern and they resist doing this work. The obvious response to this is that if the activity is trivial, then go ahead and do it in a few minutes and be done with it. In reality of course, it is usually much more interesting work, takes more than a few minutes, and fosters interesting conversations about the system almost immediately.

The other concern often raised is that the number of actors is unlimited, and thus the task of identifying all of them is enormous. This usually results from a misunderstanding of the nature of actors and how they represent roles, not individual people or systems. For instance, a system might interact with hundreds of different employees across several divisions to collect time sheet information. There might be a tendency to think that an actor is needed for each employee, for each division's employees, or perhaps for each type of employee (manager, technician, engineer). In actuality, probably only one actor is needed. An actor like *staff member* might capture the role that all of these employees play with respect to the system. So in identifying actors, the key question is not so much *Who uses the system?* but *What roles are there interacting with the system?*

Actors and the system boundary

In systems engineering, we pay a great deal of attention to system boundaries, interfaces and interface specifications. MDSD includes this kind of analysis explicitly. By identifying all of the entities with which a system interacts (actors) and all of the information and physical items (I/O entities) exchanged with the system, an MDSD model captures what is needed to specify system interfaces. As the model proceeds to develop deeper levels of decomposition, more detailed subsystem interface specifications can be captured in the same way. In a sense, you can produce such system interface specifications *for free* from an MDSD model. This is useful to note, since much work is often devoted to producing interface specifications as a separate activity, and this might be redundant effort when using MDSD.

In fact, system quality can be positively affected by the integration of such efforts into the overall MDSD modeling activity, instead of assigning them as separate efforts by separate teams, as is often done. Part of the effectiveness of MDSD comes from its comprehensiveness—that it integrates a number of often disparate system engineering or enterprise architecture activities, including:

- Requirements modeling
- Specification trees
- Traceability analysis
- Interface specifications
- Concept-of-operation analysis
- Functional block diagrams
- Logical or conceptual architecture

MDSD Step 3: Create a context diagram

A context diagram in MDSD is a diagram that shows a system element in the context of the entities with which it interacts. In the case of an enterprise context diagram, we represent the enterprise, and all of the enterprise actors discovered, each with a relationship to the enterprise. The enterprise, of course, is treated as a black box in this diagram, since no internal workings are shown—only the interfaces it has with the outside world.

It is surprising how illuminating such a diagram is in the early stages of developing a system! By showing an entity and everything with which it interacts in a single view, it becomes straightforward (though not necessarily easy) to reason about the precise positioning of the entity in relation to its world.

An example context diagram is shown in Figure 3-1.

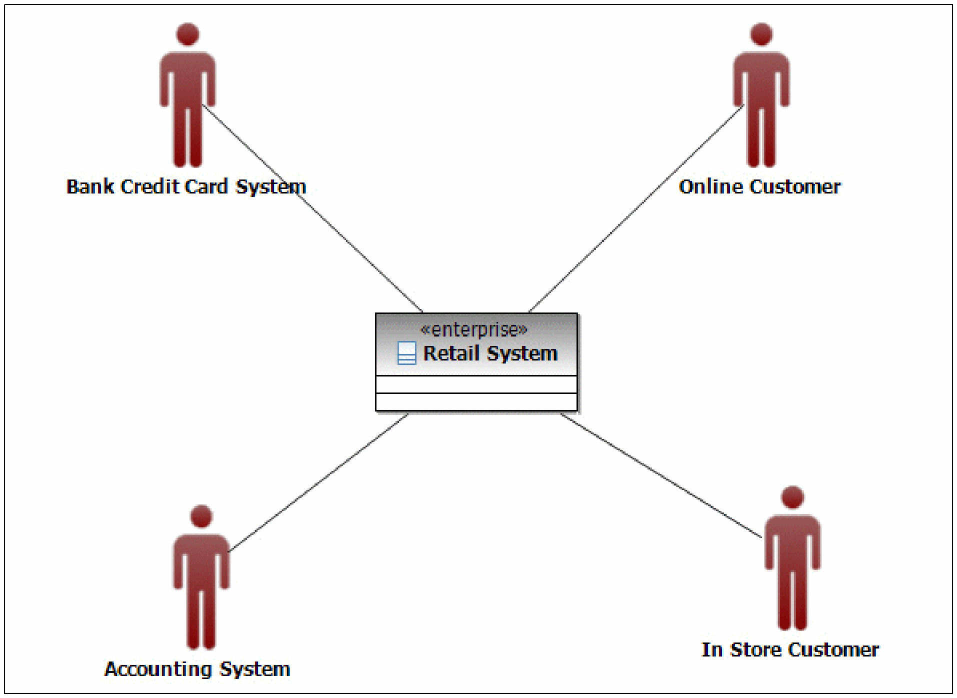


Figure 3-1 Sample context diagram

I/O entities

Here, we consider I/O entities and how they can be identified.

MDSD Step 4: Finding I/O entities

As actors are identified and placed on the context diagram, I/O entities can begin to be identified. An I/O entity is something that is exchanged between an actor and the system under consideration. It can be information or a physical item, and can be either sent or received by the system to or from the actor. Each I/O entity is associated with an actor and is designated as either sent or received (or both) by that actor.

If the system under consideration were an online bookstore, I/O entities would include books (received by actor) and money (sent by actor). I/O entities are drawn on the context diagram with associations to actors.

I/O entities are useful in several ways. In the early stages of the model, they are used to more fully understand actors and the nature and purpose of their interaction with the system. As the model develops, I/O entities are also used as parameters to fully specify operations, and also form the basis for a domain model that can be created later. I/O entities are often simply identified in the early stages of the model and are later elaborated with attributes as the model develops.

With the addition of I/O entities, the static portion of the context model is complete, and we move on to the behavioral aspects of it—finding use cases and operations.

An example of a context diagram with I/O entities is shown in Figure 3-2.

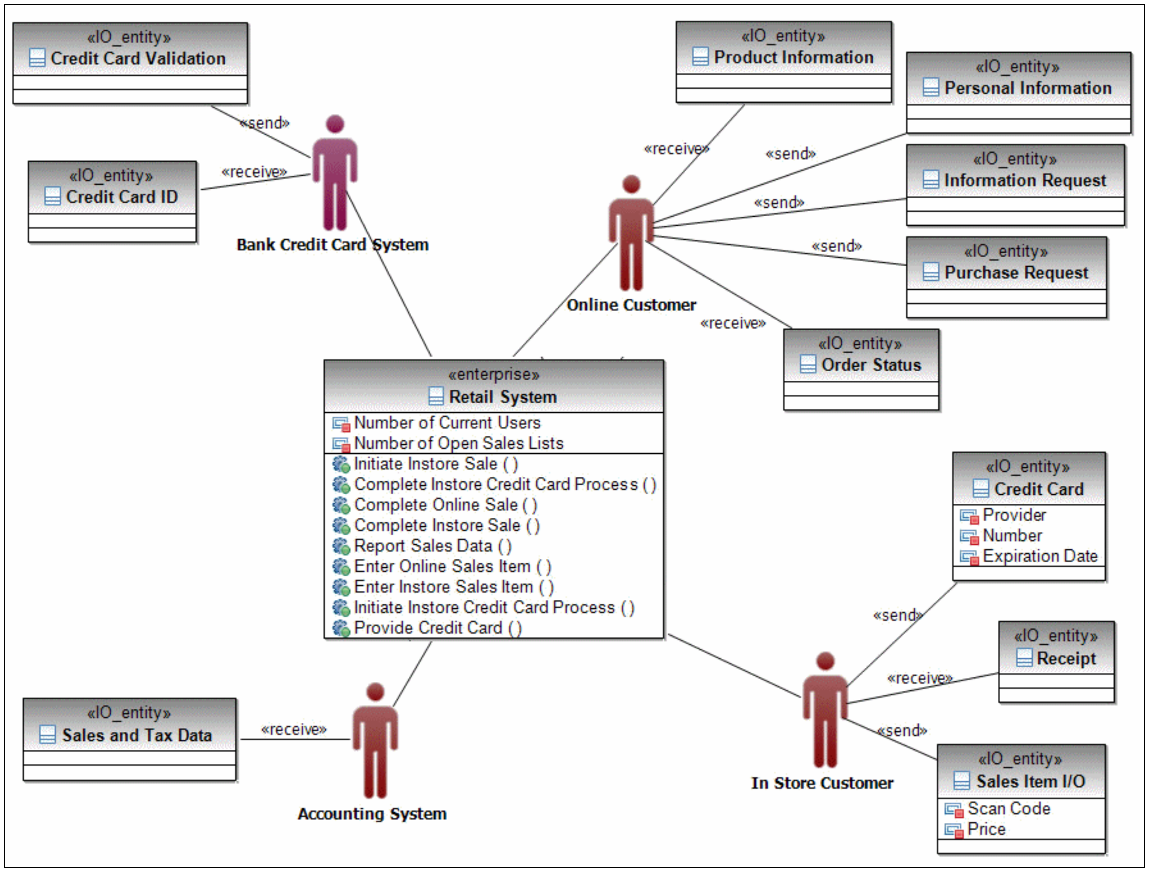

«IO_entity» Credit Card Validation
«send»
«IO_entity» Credit Card ID
«receive»
Bank Credit Card System
«IO_entity» Product Information
«IO_entity» Personal Information
«receive»
«send»
«IO_entity» Information Request
«send»
«send»
«IO_entity» Purchase Request
Online Customer
«receive»
«IO_entity» Order Status
«enterprise» Retail System
Number of Current Users
Number of Open Sales Lists
Initiate Instore Sale ( )
Complete Instore Credit Card Process ( )
Complete Online Sale ( )
Complete Instore Sale ( )
Report Sales Data ( )
Enter Online Sales Item ( )
Enter Instore Sales Item ( )
Initiate Instore Credit Card Process ( )
Provide Credit Card ( )
«IO_entity» Credit Card
Provider
Number
Expiration Date
«send»
«IO_entity» Receipt
«receive»
«send»
«IO_entity» Sales Item I/O
Scan Code
Price
In Store Customer
«IO_entity» Sales and Tax Data
«receive»
Accounting System

Figure 3-2 Retail system context diagram

Use cases

Use case modeling in MDSD is done very much like traditional use case modeling for software applications, so all of the guidance in the many books and courses on use case modeling, such as *Mastering Requirements Management with Use Cases* from Rational University (course REQ480) applies in general. In the following sections, therefore, we highlight the important aspects of use case modeling as it related to MDSD.

In the previous chapter, we explained how MDSD involved the following conceptual steps:

1. Decompose the system to create a context for the system elements.
2. Treat the system operations as use case scenarios for the elements.
3. Describe the scenarios in which the elements, as black boxes, interact to realize the system operations.
4. Derive the operations of the elements from the scenarios.

Let us bring this up a level to the enterprise, and change terms appropriately:

1. Decompose the enterprise to create a context for the enterprise elements.
2. Treat the enterprise operations as use case scenarios for the elements (one of which will be our system.
3. Describe the scenarios in which the elements, as black boxes, interact to realize the enterprise operations.
4. Derive the operations of the elements from the scenarios. These elements will be the element use cases.

Because this is a recursive process, we also apply it to lower level elements such as subsystems. In each case, the same process applies—all that changes is the context.

Note that in step 2 we treat the entity operations as use case scenarios. We can do this because use cases and operations are essentially isomorphic, that is, they have the same structure; only their context is different.

A use case is defined variously. The standard definition is that a use case represents a dialog or sequence of steps between a system and its actors that returns a result of value. MDSD defines a use case as described in “Use case” on page 19:

> *A use case is a sequence of events that describes the collaboration between the system and external actors to accomplish the goals of the system. In other words, the use case is a way to specify the behavior required of the system and external entities in response to a given sequence of stimuli.*

An operation also consists of a sequence of steps, performed by the entity under consideration and its actors. It also has a return value. It also represents a collaboration of entities to achieve the return value.

If we are using UML as our modeling language, it is no accident that we use a UML collaboration to represent both use case and operation realizations, or that a sequence diagram is considered to be a representation of that collaboration.

Therefore, in the discussion that follows, much of what is said about use cases also applies to operations.

MDSD Step 5: Finding use cases

Identifying use cases is an important step in this process, and is, at the same time simple, profound, and vitally important. Use cases form the basis from which the dynamic part of the MDSD model is derived. What we are seeking to do here is to identify the complete set of planned usages of an entity when the entity is treated as a black box. This is probably the hardest part—staying to a black-box perspective of an entity. Often those doing this kind of modeling have deep knowledge of the internals of the particular entity and it is a challenge to keep the focus at a high level. We often find ourselves reassuring such teams that we will get to work on the lower-level interactions—which interest them far more—soon enough and emphasize that the purpose is to derive the lower level interactions from an analysis of the fundamental usages of the higher level entity—its reason for being. By keeping to this high level focus, the higher level use cases can be developed more quickly and the lower levels developed in due time.

Finding use cases involves *stepping back* and looking at the entity as a black box, and asking, how do these actors we have identified interact with the entity? What are the complete entity usages? What are the major results of value produced by the entity? When we next expand the entity to its white-box view, we will be asking the same questions of the subentities. In the case of an enterprise, we will look at the system and its actors in the white-box expansion.

What do you use your car for?

We often use this illustration in our MDSD courses to help people understand use cases. If I ask a group what they use their cars for, the first response will likely be, *to get from point A to point B*. I ask them where those places are because I have never seen them on a map. I also ask them if they wake up in the morning and say to themselves, *today I want to get from point A to point B*. They laugh and realize that getting from point A to point B is not a real usage. It is too vague. So I ask them to forget that they know anything about use cases or computers for that matter, and just answer the question: *What do you use your car for?*

With some thought, we come up with a number of complete usages of the car, such as:

- Commute to work (and back home)
- Go shopping
- Go on vacation
- Take the kids to school
- Travel to a remote bike ride

The first point of discussion is whether these are actually separate usages or all just aspects of some *master* use case, such as *travel from point to point*. This is an important consideration. The question boils down to, how similar are these usages, and how different? This can be a difficult question to answer until the details of each usage are specified, so our general guidance is, if it seems they might be different enough to warrant separate use cases, keep them separate until it is clear they can be combined. Note also that similar usages might give rise to new and important requirements. If we omit the *go on vacation* use case, we might build a car with a two gallon fuel tank—great for commuting and shopping, but no good for long trips. On balance, it pays to try to discover the required usages and then combine them as possible.

To continue our example, a little more thinking should produce additional usages for the car such as these:

- Listen to music.
- Watch a movie.
- Cool off (this was mentioned by a group in Florida in the summer).
- Put the baby to sleep (all mothers know that car motion can be sleep inducing).
- Take a nap (just check the parking lots during lunch time for evidence of this; one vehicle I know allows the heater to run with the engine off to keep a napper warm for a while).

Each usage must be complete, that is, it must reflect a complete goal that someone has. By *listen to music*, we do not mean listen to music as one drives to a destination, we mean using the car to listen to music. This is also an important point. With use cases, we are after the main, complete usages. It is always useful to ask the question: *Could this use case be a part of some larger usage?* This is not an attempt to consolidate or combine use cases just so that there are fewer, but an attempt to find the real, complete usages of the system.

An example might help here. If we ask what the stakeholders for a large supply chain system use the system for, we might get answers like, look up inventory levels, determine re-order points, and so forth. Are these complete usages? They could be, and they will work as use cases, but it should be considered that maybe there is one larger usage that encompasses both of these smaller interactions. One could ask if determining re-order points is one of the purposes of the system, or is it really in the service of some larger goal, such as *maintain inventory levels*? If the latter, then we could try using that as the use case and see if it can be expressed as a flow of events. If so, we have found something closer to the heart of the system's purpose, and a better use case.

Writing a brief description

As these use cases are identified, a brief description should be written. This serves several purposes. It clarifies the author (or group) thinking on what the use case really encompasses. Often good use case names are brief, and not too specific. For instance, does the use case *maintain inventory levels* include the receiving ordered goods, or only the ordering and purchasing side of the process? This can be stated in the brief description. Often such decisions are clarified when the use cases are identified and initially discussed, but such discussions are easily forgotten unless recorded in the brief description of the use case.

The best brief descriptions read like a Reader's Digest condensation of the actual use case. They state who accomplishes what with the system in the specific usage. They are written much like a use case flow of events, but in very broad terms. A possible brief description for *maintain inventory levels* could be:

> *Marketing determines needed inventory levels based on sales projections. Warehousing and distribution report on current levels. Systems determines needed order quantities weekly and generates purchase orders for approval by procurement staff.*

If these use cases are being identified in a workshop setting, have someone in the workshop create a brief description based on the group discussion at the time the use case is identified. This is a good check—if there is not enough known to write a brief description, then perhaps the use case is too vague, or we do not have the right stakeholders and subject matter experts in attendance.

As we have noted previously, it can be very useful to analyze at least a portion of the enterprise to understand its use cases and operations, especially those which involve our system under consideration, If the enterprise is large and complex, we might not want to analyze all of its use cases and operations, but only those that we can identify as involving our system. It might be useful to draw a use case diagram for the enterprise level. Later we will draw them for other levels as well, but we will keep them separate. In an enterprise level use case diagram, the enterprise is considered as the system, and thus is not shown, so the diagram must be labeled so that it is clear to what system the use cases refers (Figure 3-3).

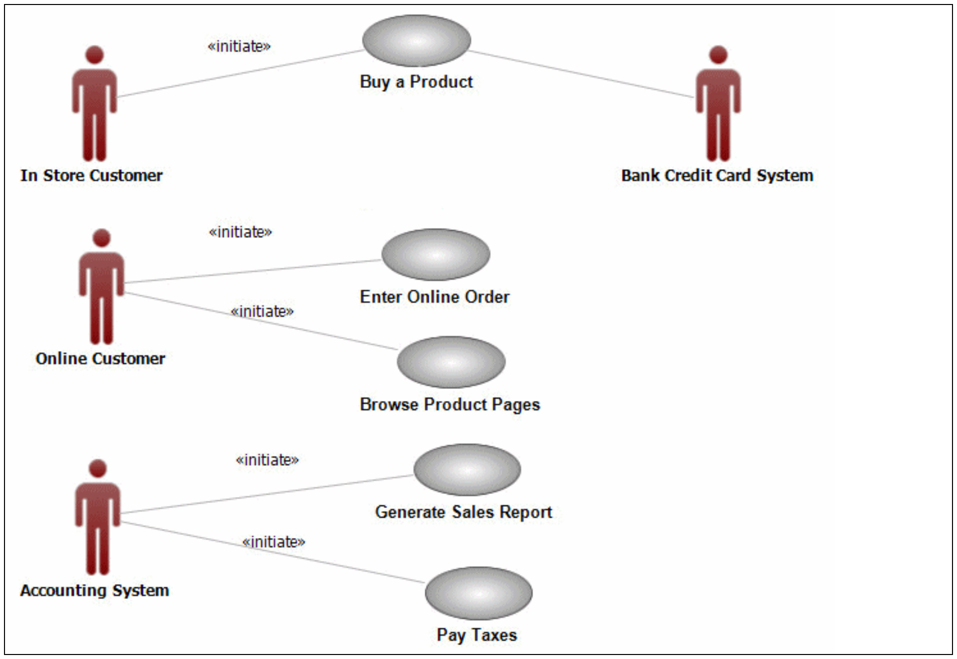


Figure 3-3 Retail use case diagram

An enterprise use case diagram can show all use cases for the enterprise or a subset of them as just noted for clarity. What is important is that all the use cases shown are at the same level of decomposition, that is, the enterprise level, or level 0. Actors shown are enterprise actors—the same ones shown on the enterprise context diagram if one has been developed. Because the enterprise is treated as a black box, no workers are shown. Workers (people inside the system) will likely become actors at lower levels of abstraction.

Actor involvement in use cases

One of the most common omissions made in use case modeling for MDSD is to overlook some actor interaction. It is easy enough to identify the primary, or initiating actor associated with a use case, but it is easy to overlook other actors who have a supporting role in the carrying out of the use case. In MDSD, this is a particularly serious omission, because the actor interactions allow the identification of the operations the system must perform to realize the use case. This will be seen in later steps as the operations analysis proceeds, but for now, understand that all actor interactions must be captured. Such omissions can, of course, be discovered and provided later, but the recommendation here is to try to identify all of the system interactions—do not skip any for the sake of brevity or speed.

Use case flows of events

Here we discuss how to write use case flows of events.

MDSD Step 6: Write use case flows of events

With the use cases identified, the next step is to write flows of events. As noted before, use case modeling is, for the most part, done in MDSD exactly as in traditional use case modeling, Here we offer just a few highlights of the most important things to remember in writing a flow of events for MDSD.

Level of detail in use case flows

One of the common questions asked about use cases is *How much detail should be included in a use case?* The question implies that there is a sort of sliding scale of detail that one can increase or decrease. Actually, it is simpler than that. Use cases should contain enough detail to fully explain the actor interactions necessary to accomplish the use case. Thus the use case will keep to the black-box perspective, and not contain any details about what happens inside the system to accomplish the use case. Some exceptions can be made to this rule, but let us consider the dangers before we explain those.

If, while writing a use case, we begin to include details about what is happening inside the system, we risk spiraling down into system details that will prevent us from seeing the important aspects of the level of abstraction we are examining. Remember that the focus of the use case is the interactions between the elements outside the system and the system itself.

Use cases are statements of requirements, and thus should not include white-box design decisions, even if they are known at this point. For one thing, they can change multiple times as the design is validated, and for another, such details will be specified at a lower level of abstraction, and thus would be redundant here.

That use cases should keep to a black-box perspective is not to say that they should not be specific and detailed within that perspective. Sometimes we see use cases that contain steps akin to this:

> *The user enters the important information into the system.*

Use cases should indeed specify what information is required, either by stating the data items directly or by specific reference to a data dictionary or other outside source. As we will see, this information can be included in the model in the form of operation signatures as the use cases are analyzed, and it can also be further modeled in the domain diagrams.

While use case flows of events can be written in many formats, we find that a simple numbered list of steps is the most useful. Remember that one of the main purposes of use cases is to be readable by many stakeholders. To make this possible, use cases should be written in plain language and using terms familiar to the organization. It does no good to write in IT-oriented technical language, even if this is more precise, since it will hinder understanding and genuine agreement from stakeholders.

The MDSD template for a use case is shown in Appendix A, “MDSD use case specification template” on page 181. Note that this template has two alternate formats for the flow of events. The plain numbered list of steps should be used for enterprise and element use cases, and the table format, with columns for both black- and white-box steps, should be used for operation realizations, derived in the flowdown process as described in succeeding sections.

Initiation of the use case

In MDSD, we require that actors initiate all use cases. Why is this? Since we are building a model in which we will ultimately express all system functionality as operations of system elements, what we are after is all of the functionality that can be requested of these elements. We will derive the needed operations from this set of requests. It will be seen later why system functionality that is assumed to be initiated by the system itself must be represented as part of a larger behavior that is initiated by an actor, but for now, simply write use cases as if they are initiated by an actor. Here is how.

It might take some looking to determine the correct actor to represent the initiator of the use case. A common case is behavior that is initiated based on a schedule. If such behavior is actually initiated based on an outside scheduler system, then this can be the actor. If the behavior is initiated by a clock, and the clock is external to the system, then the clock can be represented as an actor. In the rare case when the behavior is initiated by system, based on time, and the only time reference or clock is also inside the system, the best choice is to have an actor called *time*. This allows behavior to be modeled as if time is requesting it to happen. This might seem awkward, but by doing this all behavior will be captured as part of operations.

We have also found it best to adopt the convention of beginning each use case with the phrase *This use case begins when…* followed by the event that starts the use case. Some examples are shown here.

Examples of use case initiation:

- This use case begins when the console operator selects to review the program log.
- This use case begins at 4:00 am daily.
- This use case begins when the scheduling system requests the nightly reconciliation process to begin.
- This use case begins when it is time to check for the presence of rain.

Using activity diagrams

If the flow of events in a use case is complex, and especially if there are numerous or complex alternative flows of events, it might be helpful to draw an activity diagram to illustrate the entire flow of events. Activity diagrams have the advantage of being able to show all alternate flows in one view, but have the disadvantage of obscuring the main flow. Swimlanes can also be added to these diagrams to show the responsibilities of the actors and the system.

We do not use activity diagrams in place of sequence diagrams in the MDSD flowdown process. We have found that sequence diagrams have clearer semantics for operations analysis, and that it is easier to extract traceability information from the models using sequence diagrams.[5] For now, it should just be clear that activity diagrams are used in MDSD as an optional view, to help illustrate complex use case flows of events. We have seen many situations where they were not used at all, with no ill affects, and others where they were used only for complex use cases.

Understanding collaboration from a black-box perspective

If we have completed our work through the previous MDSD step, what we have now is a complete set of use cases. The next step is to answer the question, *What operations must the entity be capable of, in order to make possible all of the usages described in these use cases?* To answer this question, we perform operation identification.

[5] Swimlanes and call operation actions in activity diagrams provide an alternative for those who are more comfortable using activity diagrams. We do not treat this option in this document.

Identifying operations

Here we discuss operation identification by using sequence diagrams.

MDSD Step 7: Operation identification

Operation identification involves the use of a sequence diagram. Sequence diagrams show the same flow of events described in a use case, but use a very specific format and method to show them. The flow of events of each use case is shown as a series of interactions, more specifically requests from one entity to another. The use case is carried out as entities makes requests of one another.

We create two kinds of sequence diagrams in MDSD—black-box and white-box. In a black-box sequence diagram, only the entity and its actors are placed on the diagram while in a white-box sequence diagram, multiple elements within the entity are used in addition to the actors. For operation identification, we need only a black-box sequence diagram. We will use white-box sequence diagrams later.

For each use case, draw a black-box sequence diagram with lifelines for the entity and each of the actors involved in the main flow of that use case, or any of its alternate flows. Then, following the flow of events in the use case, write a sequence of requests that fulfill the use case. For example, consider the use case *commute to work* mentioned before. The entity is the car. The flow of events might initially be written as follows:

1. This use case begins when the driver approaches and unlocks the car.
2. The driver starts the car and allows it to warm up.
3. The driver drives the car to the work location.
4. And so forth...

We must *transform* this plain language flow of events into a series of requests. We do this by asking, for each step or set of steps in the use case, what request is being made of the system do to something. Sometimes this takes a combination of imagination and reading ahead in the use case to determine the actual purpose of things.

In the example here, we might ask what request is being made in the first step. By approaching the car, is the driver making some request of the car? It might be tempting to draw this on a sequence diagram as an arrow from the driver to the system, and label it as *approach car* but this is not correct (Figure 3-4).

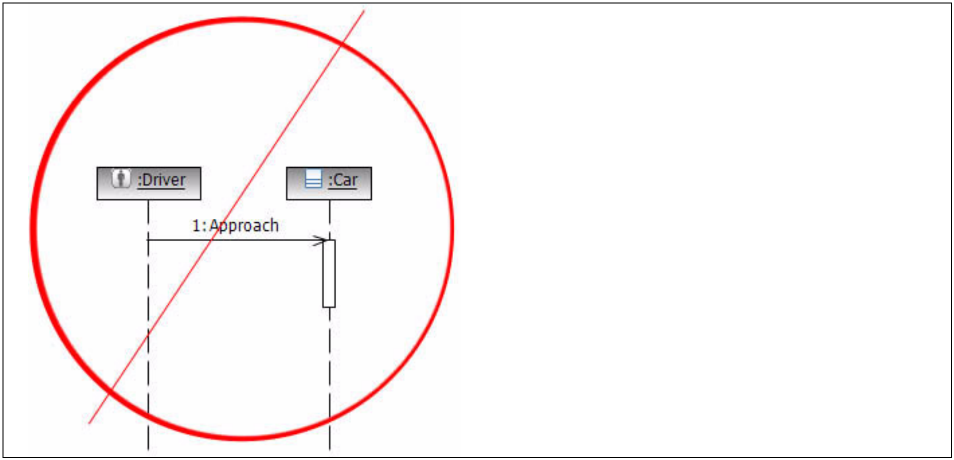


Figure 3-4 Incorrect sequence diagram

This would mean that the driver is requesting the car to approach. What is the right way to represent this? We get the answer from the second part of that step in the use case. When the driver approaches the car, he or she is actually requesting the car to unlock. We thus draw a message arrow from the driver to the car and label it *unlock*. Note that this allows great flexibility in implementation—the unlocking can be accomplished by an automatic proximity key, a biometric sensor, a conventional key, or any other means. This is one of the important features of MDSD. Because we treat the car as a black box in describing this use case, we *abstract away* all of the details of how the car performs the required behavior.

One might note here that after the analysis of this use case fragment, the driver approaching the car turns out not to be significant in the design of the system. Unless we are planning on designing a car that somehow detects the driver approaching, the use case should really begin with the driver unlocking the car (Figure 3-5).

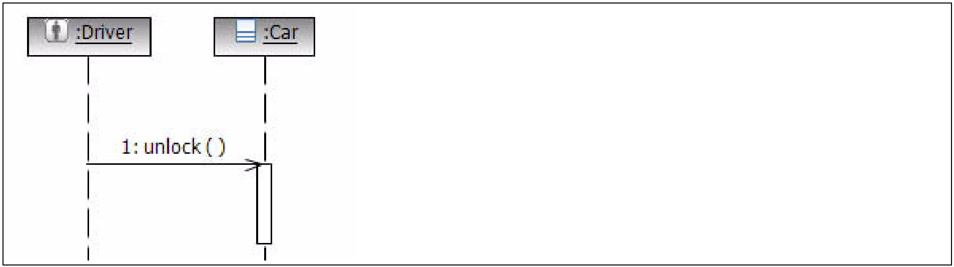


Figure 3-5 Correct sequence diagram

Requests: The key to operations

The concept of characterizing all behavior of the system as a series of requests is one of the most difficult for the new MDSD practitioner to grasp, so its purpose and conceptual basis bears a bit more explanation here. When we think about the idea of a system performing some action, it is tempting to think of this *in a vacuum*, that is, with no reference to any other element. So the car unlocking is something that the car does, and that is enough said. This leads us to think of systems as composed of elements each performing some set of functions.

When we model a system in this way, we are tempted to produce something akin to process flow diagrams (or block diagrams) that simply show the order in which functions are performed. What it leaves out, is precisely how these functions are made to perform in sequence, and how the parts of the system collaborate to produce desired behavior. Tacit in these diagrams is some kind of master control flow that causes things to happen. If the master controller is made explicit, and shown as controlling or collaborating with other pieces of the system, fine; but often the controller is implicit in the control flow, and we have found implicit designs or assumptions to be problematical. Systems in reality are not so mysterious. Behavior happens as a result of parts of the system interacting with each other and the world, not through some hidden, unspecified master controller, as some process diagrams imply.

In MDSD, we characterize systems as collections of elements that communicate with each other by, in essence, if not literally, making requests of each other. So instead of describing the unlocking of the car as the action of the driver (unlock the car) and the action of the car (unlock), we describe this behavior as the driver requesting the car to unlock. Sometimes the request is not so easy to determine. If the behavior I am trying to describe is a home owner sending in a mortgage payment, it is tempting to think of this as the homeowner's action (send mortgage payment). Instead, we ask, what is the homeowner requesting the mortgage company to do here? If we were to read ahead a bit in such a use case, we would find that the next thing that happens is that the mortgage company receives the payment and applies it to the homeowner's mortgage account.

Instead of describing this as series of actions taken by actors and elements (send, receive, apply) we can describe this behavior as the homeowner requesting the mortgage company to apply their mortgage payment. Apply mortgage payment, when shown as a request the homeowner makes of the mortgage company, is a much more concise and specific description of the behavior. It has the added benefit of speaking directly to a purpose of the system. Systems do not exist to send and receive data. They exist to do things such as applying mortgage payments.

Specifying request signatures

We can make such a request more complete by including the notation of the entities carried along with it. A request from the homeowner to the mortgage company to apply the payment must be accompanied by the actual payment. Thus we would write the request fully as:

```
apply mortgage payment (mortgage payment)
```

A full signature also specifies the entities that travel back to the requester as a result of the request. If the mortgage company is expected to send back a statement as a result of the payment, the full signature would then read:

```
apply mortgage payment (mortgage payment, statement)
```

In practice, we sometimes omit these full signatures (request along with entities items passed back and forth) in the early stages of building the model. If including the signature adds clarity and does not slow down the modeling process, then by all means it can be included as the models are developed. If additional research or thinking is required to fully specify the signatures, then a decision can be made to either spend that time on the first pass, return later, or perhaps delegate this work to a sub-team.

Entities included in signatures should match the level of decomposition at which the modeler is working. When working with an enterprise use case for instance, we might use customer information to refer to a set of information that at a lower level would be further described as a set of specific fields. These entities exchanged between system elements and actors also appear in the model as the I/O entities discussed earlier in the section on context diagrams. They also become the foundation for the more complete domain model described in a later section.

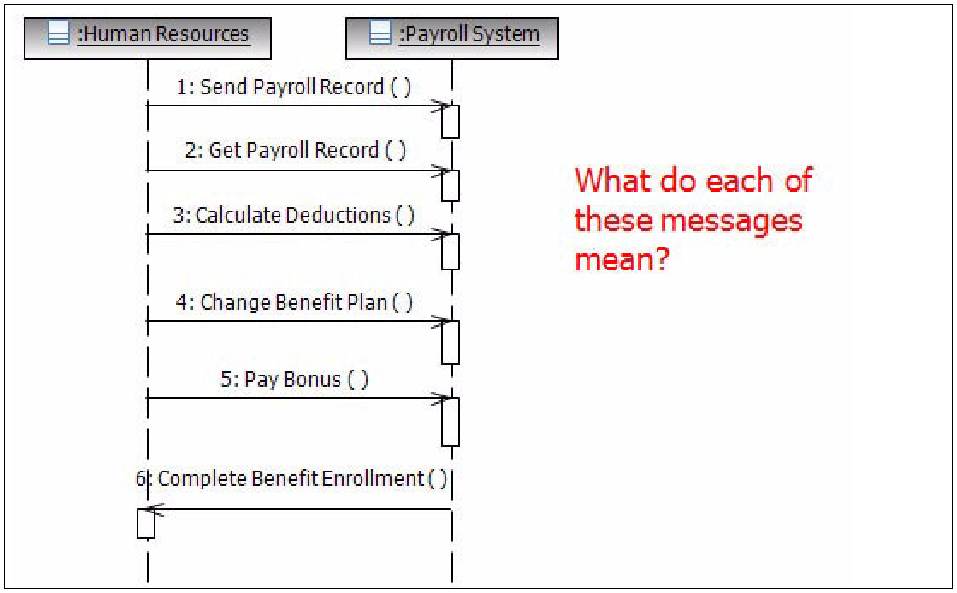
Information in the MDSD model

An MDSD model is an abstraction of the system being developed, in fact, multiple abstractions at different levels. Thus we seek to represent information in the model also in an abstract way. The information entities that appear in the signatures of messages are one way to do this. In these messages, we show information at a high level, for example, we might show something like *customer information*, instead of listing out name, address, phone, account number, purchasing history, and so forth. This allows us to show the information used at a high level, recognizable by all stakeholders. Most stakeholders are not able to makes sense of a detailed information design, such as a database schema or data dictionary, and these would be far too much information for the purposes of the higher levels in the model.

I/O entities are another way to abstract information, and can also represent physical items as well. Information entities can also be I/O entities if they are sent or received outside the system. Both I/O entities and information entities can be used to create a domain model, or even multiple domain models at various levels of abstraction in the model. A domain model is a UML class or SysML block diagram showing the entities and their relationships, such as multiplicity (one-to-one, one-to-many) and generalization/inheritance.

Message naming: A quiz

Because this topic is so important, let us review the principles covered so far with a little exercise. Which of the messages in the following diagram seem to be correct, and which seem to contain an error? It should be noted that this is merely a grouping of independent messages for presentation purposes, no sequence is implied (Figure 3-6).



:Human Resources
:Payroll System
1: Send Payroll Record ( )
2: Get Payroll Record ( )
3: Calculate Deductions ( )
4: Change Benefit Plan ( )
5: Pay Bonus ( )
6: Complete Benefit Enrollment ( )
What do each of these messages mean?

Figure 3-6 Which messages are correct?

The best way to do this quiz is to read each message in its full plan language form using the term *requests*:

- The first message would be read: *The human resources system requests the payroll system to send the payroll record*. If this sounds like a correct statement of the behavior of the system, then this message is well-named—it does and it is. It means that the payroll system must be capable of sending a payroll record, which seems sensible.

- The second would be read: *The human resources system requests the payroll system to get the payroll record*. This now seems odd. It implies that the payroll system must *get* the payroll record. From where? From some other system? Would not the payroll system be expected to *have* the payroll record in its database somewhere? This message likely indicates a very common error. The use case step probably reads something like this: *The human resources system gets the payroll record from the payroll system*.

  This correct line in the use case flow was mistranslated into the message just illustrated. The message should have been translated as a request from the human resources system to the payroll system to send (or provide, deliver) the payroll record.

- Taking the third, fourth, and fifth messages in the illustration, we should find that if we read them in their full English version as shown, they do indeed make sense, and that the indicated operations make sense as operations of the payroll system:
  - Calculate deductions
  - Change benefit plan
  - Pay bonus
- The final message reads: *The payroll system requests the human resources system to complete benefit enrollment*. Assuming that completing benefit enrollment is something the human resources system has to be capable of doing, this message is shown correctly.

Toward better requests

When first creating MDSD models, practitioners tend toward using transactional-sounding names such as send, receive, accept, provide, and the like. Using the earlier example of a car, when the driver goes to unlock the car, we might be tempted to write a request from the driver to the car to accept the key, followed by an internal function of the car to unlock the door, as shown in Figure 3-7.

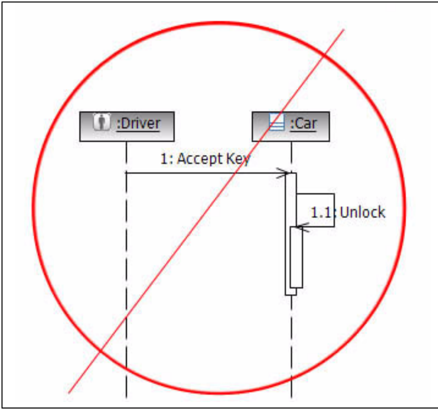


Figure 3-7 Unlocking car: Cumbersome sequence

While this might be technically correct, it is less than optimal in the model for two main reasons. First, the car does not exist for the purpose of accepting a key. Even if unlocking the car were required to be by key versus some other means, saying that the required function for the car is to accept a key is not true, and misleading. Second, it requires an internal function, shown as a reflexive arrow on the sequence diagram, to be clear about what is going on.

This pattern, or we should say, anti-pattern, of a transaction-oriented message immediately followed by an internal function is quite commonly used by new practitioners. The solution is to combine the two by asking, what is the *real* function that is required of the car? To get at this, we can simply ask, why is the driver inserting the key (or sending the data) into the car. The answer is of course that the driver is really not just requesting the car to accept the key, but requesting the car to unlock. Thus we can better model it as a single message, unlock. Optionally, we can add key as a parameter on the unlock request, since the key is passed between the driver and the car as part of the request (Figure 3-8).

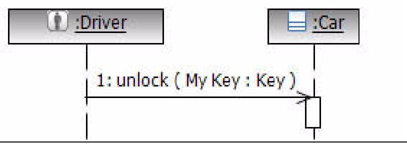


Figure 3-8 Unlocking car: Better sequence

In addition to making the model more compact and succinct, this fits our intuitive understanding of what is happening. As a driver, what I want the car to do is unlock. If it accepted my key without unlocking, I am not happy. So the real requirement on the car is for it to be able to unlock, and this single request shows that.

The questions to ask in creating the requests that populate a sequence diagram, are first, who is requesting what or whom to do what? In the aforementioned example, while the use case states *the driver unlocks the car* the request is actually from the driver to the car to unlock. It is the car that unlocks itself in response to a request from the driver. In many cases, asking *who is requesting what?* leads to a good, solid message name that clearly indicates the real action of the system at that point. Such messages are somehow satisfying in that they clearly communicate the meaning and intent of the request, and not just its form. If the messages in your model tend to be of the form, *send* this, or *receive* that, or *get* this, or *provide* that, then the real purpose of the system interaction is hidden behind these generic, transaction-oriented terms.

The way around this, when confronted with, say, a *send customer profile,* message is to ask, *why* is the system sending the customer profile to this other entity? Perhaps the answer is that the other entity needs the customer profile so it can validate the customer's credit limit, in which case *validate credit limit* would be a much better name for that message. Keep asking why, until you get good, solid answers about what is going on.

It is also important to try to keep messages named in the commonly used language and jargon of the enterprise in which you are working. While modelers who have trained analytical minds might come up with superior terms, it is more important to keep models in a language that can be readily understood by business stakeholders. In a recent engagement, models were printed on large rolls of paper and hung in a high traffic area so that everyone in the company could see them. With only a brief explanation of what the models represent, stakeholders with no UML or modeling training could understand the models, primarily because they were couched in familiar business language.

Identifying operations from the sequence diagram

Once we have developed the black-box sequence diagrams for each of the use cases, we are ready to identify the operations—our reason for doing all the sequence diagrams. Looking at a black-box sequence diagram, focus on an element and you see that some of the arrows are pointing in toward the element's lifeline (the vertical line dropping from the element at the top of the diagram) and some arrows point away from this lifeline.

To determine an entity's operations on a sequence diagram, note the arrows pointing into the lifeline and originating from another lifeline only. When working with paper models—such as those on flip charts—we often circle these arrowheads in a bright color to emphasize their importance. Each arrow pointing in toward the enterprise element's lifeline represents a candidate operation for the enterprise. Why? An arrow represents a message that carries a request—a request being made of the entity. If the system is to work, the entity must be capable of responding to that request when it is initiated. Thus the entity must have an operation that corresponds to the request.

To put it simply, if the entity is at some point requested, by any actor, to unlock, then it is required that the entity have an operation called *unlock*. It is as simple as that. So, we can read the operations for the entity right off of the sequence diagram we have just drawn, by simply noting the arrows that point in towards its lifeline (Figure 3-9).

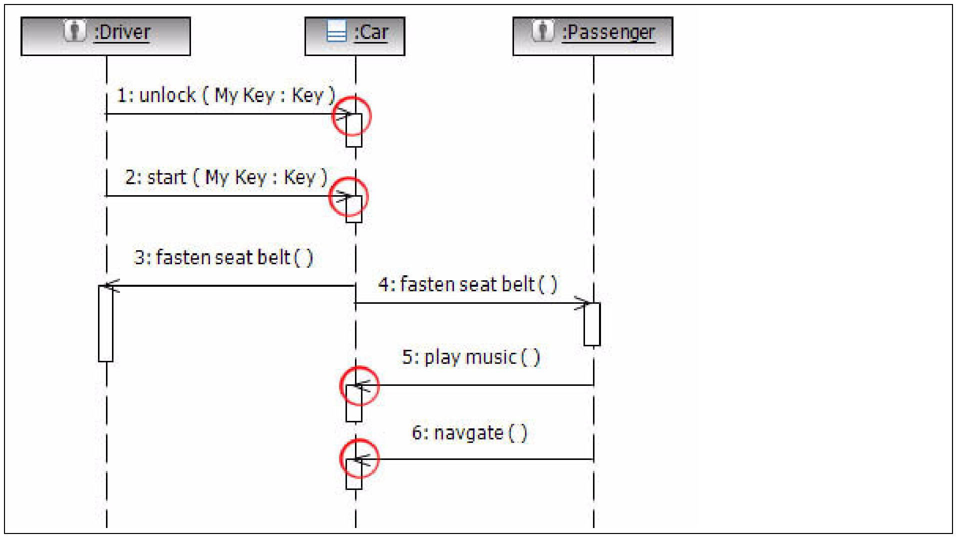


Figure 3-9 Sequence diagram with arrowheads circled in red

At this point it is often asked why arrows that point out (away) from the element's lifeline do not represent operations of the element; after all, they seem to be something the element must do. Indeed, the system must issue the requests represented by those arrows, but the system does not just make these requests at any time. Because we have modeled the actual sequence of operations, we know when the system must take such an action, and it is as a part of fulfilling the previously requested operation.

For example, consider a message arrow going from the driver to the navigation system, requesting the navigation system to route to destination, followed by an arrow from the navigation system to the GPS satellite, requesting it to confirm current location. In this case, *route to destination* becomes an operation of the navigation system, while *confirm current location* does not. Why? Because *confirm current location* is performed by the GPS satellite, and the navigation system requests this as part of *route to destination.* Requesting the navigation system to route to destination implies that the navigation system must determine the current position, and it does this by requesting the GPS satellite to do it. There is no need to think through all this though—just take only the arrows pointing in towards the enterprise as the operations for the entity (Figure 3-10).

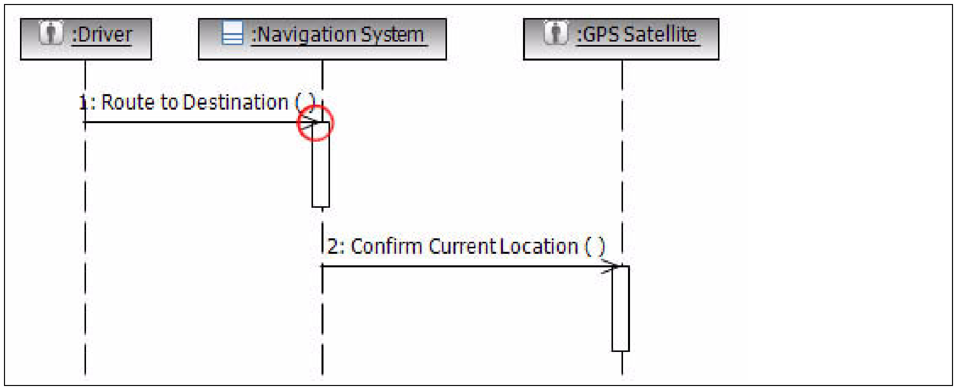


Figure 3-10 Sequence with red circle only on the arrowhead of route to destination}

In the initial modeling stages, which are often done using flip charts rather than a modeling tool, one must be careful to identify the operations using this principle. When the models are transferred into a UML or SysML modeling tool, we can assign an operation to the receiver of the message, if one already exists that corresponds to our message, or we can create an operation and it will be assigned to the receiving class.

Incidentally, what do messages that represent requests of actors mean on this diagram? Because we are not designing and building the actors, we do not take them to indicate design requirements on the actors, however, they do indeed represent *interface* requirements on these actors. What the model says is that these are effectively requests by the system for the actors to do something. As awkward as it might seem for the system to be making requests of actors, this formulation is actually quite useful, because it expresses specifically how actors will interact with the system. In the case of non-human actors, that is, other systems, these interactions must match the interface capabilities of those systems, an important point of coordination. In fact, this is true of human actors as well—just try asking a service representative for a service they do not offer!

This is a benefit of an MDSD model—it maps these interaction requirements in the same model with system functional requirements and usage scenarios, ensuring consistency.

Having now determined our set of candidate operations, by producing sequence diagrams for all use cases (including alternate flows), we now move to the next major step, during which we will produce a consistent, optimal set of operations.

Refactoring operations

Here we consider refactoring and consolidating operations.

MDSD Step 8: Refactoring and consolidating enterprise operations

It might seem that we have determined all of the operations necessary for an entity to fulfill all of its use cases, but there is one final step. In most situations, we find that due to the elapsed time it takes to create a complete use case model, and the fact that usually multiple modelers are involved, we must ensure that the operations determined from the analysis of the entire collection of use cases do not include redundant or overlapping operations.

To do this, review the list of operations that you have identified from analysis of all the entity's use cases. Look for any operations that might be similar but named slightly differently. For example, if in one use case an operation was identified called *start-up* and in another *initialize* we might look more closely into these to see if they could be treated as the same operation. If so, then rename one or both of them so they are the same, and make any necessary adjustments to the use case flows of events and black-box sequence diagrams to make it all consistent.

In the early stages of an MDSD model, you can expect lots of this kind of refactoring and rethinking of the model.

More about operations

Now that we have identified the set of operations necessary to fulfill (or accomplish) the use cases, let us look more closely at what an operation is and what it represents in an MDSD model. Operations are like use cases, in that they are flows of events that accomplish something. In addition, they do show primarily interactions between system elements and actors, while hiding functionality internal to those elements.

They are unlike use cases in that they are not complete system usages, but are more atomic. Operations also *run to completion*, meaning that once invoked, they continue until they are finished (or fail) without requiring the actors to invoke any further operations. If there were a need for an additional operation invoked by the actor, that would be the end of this operation and the start of another. Operations can have interactions with other elements and actors as they run, but have only a single invocation by the element or actor who invokes them.

For example, when I request my car to start by turning the key in the ignition, the car starts, or does not, with no further interaction with me. The car can have additional interactions with other actors, say a GPS satellite, in the course of starting, but it runs to completion without needing me for anything. Based on how we derive operations using sequence diagrams, this *run to completion* feature takes care of itself—no special attention to it is necessary.

In an optimal system architecture, we would expect operations to be used in more than one use case. We would also expect most use cases to need more than one operation for their fulfillment. There are exceptions. A use case in which the system interacts only with its initiating actor, and only once at initiation of the use case, would be accomplished by a single operation.

If no operations participate in collaborations for multiple use cases, then the architecture might be taking a *stove-piped* pattern, which is usually non-optimal. For example, if I ended up with a separately implemented customer information subsystem in each of my enterprise applications, I have probably failed to achieve good optimization. At the same time, if accomplishing a use case involves many rapid interactions between system elements, performance might suffer. MDSD does not solve this automatically. If it did, human architects would be unnecessary! MDSD does provide a way to reason about these kinds of trade-offs. The objective is to create an optimal set of operations for an entity, and, as we will soon see, other elements within it.

With operations in hand, we can proceed to the next decomposition level of the system.

Figure 3-11 shows a completed context diagram with the entity under consideration, its actors and I/O entities, and entity operations. Note that there is a significant amount of information in this diagram: we have a better sense of the boundaries of the entity, we have a better understanding of what functionality it must provide, and we have a high level view of what information gets passed between the entity and its actors. In other words, we have a better understanding of its context.

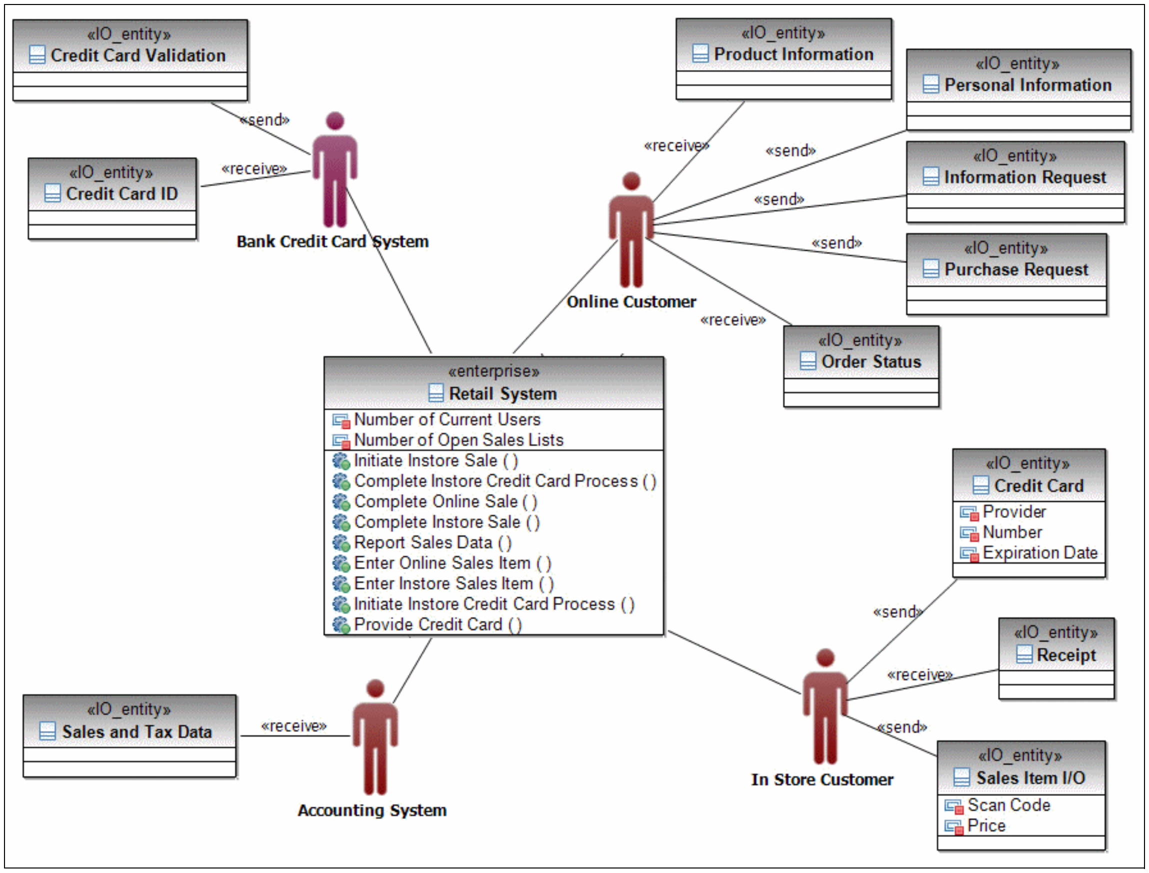

«IO_entity» Credit Card Validation
«IO_entity» Credit Card ID
«send»
«receive»
Bank Credit Card System
«IO_entity» Product Information
«IO_entity» Personal Information
«IO_entity» Information Request
«IO_entity» Purchase Request
«IO_entity» Order Status
«receive»
«send»
«send»
«send»
«receive»
Online Customer
«enterprise» Retail System
Number of Current Users
Number of Open Sales Lists
Initiate Instore Sale ( )
Complete Instore Credit Card Process ( )
Complete Online Sale ( )
Complete Instore Sale ( )
Report Sales Data ( )
Enter Online Sales Item ( )
Enter Instore Sales Item ( )
Initiate Instore Credit Card Process ( )
Provide Credit Card ( )
«IO_entity» Credit Card
Provider
Number
Expiration Date
«IO_entity» Receipt
«IO_entity» Sales Item I/O
Scan Code
Price
«send»
«receive»
«send»
In Store Customer
«IO_entity» Sales and Tax Data
«receive»
Accounting System

Figure 3-11 An enterprise context diagram showing actors, I/O entities, and enterprise operations

Summary

We have spent this chapter looking at a black-box point of view. We have considered the system, enterprise, or entity as a black box and explored its context so that we can understand what is expected of it, and what collaborations it participates in within that context.

Having gained this explicit understanding, we proceed to the next larger step in MDSD's transformations, that of examining the entity as a white box, exposing the internal elements, collaborations, and distribution of responsibilities within it. As noted previously, we also will be transitioning from specification to realization; in looking at the black box, we discover what is required of the entity. In looking at the white box, we begin to design how the entity will realize what is required of it.

4

White-box thinking: Understanding collaboration

In the previous chapter, we examined the system from a black-box perspective to understand what services are required of the system we are considering, and how it collaborates with other entities outside of it to fulfill the goals of the larger enterprise. In this chapter, we break open the black box, and look at the system from a white-box perspective.

We begin with the logical viewpoint. This tends to lead to more flexible architectures, as opposed to beginning with the distribution viewpoint. We address the distribution viewpoint in the next chapter.[1]

[1] See article by Murray Cantor, *The role of logical decomposition in system architecture*, August, 2007, http://www.ibm.com/developerworks/rational/library/aug07/cantor/

Operation realization

Here we discuss logical system elements and the use of context diagrams.

MDSD Step 9: Operation realization

The question we must answer now is, how are the operations we have uncovered accomplished using a collaboration of elements at the next level of decomposition? So far, we have treated the system under consideration as a single black box, and avoided any mention of elements inside. Now, we will determine the logical system elements within the entity and map out how they collaborate to accomplish each of the system operations. Indirectly of course, this also shows us how they collaborate to fulfill the system use cases.

The logical viewpoint

When thinking about what would make good elements at the next level, it is tempting to move toward a physical decomposition of the system under consideration and use these as our logical elements. If we have been considering a car, we might be tempted to put physical subsystems such as the drivetrain, suspension, electrical and fuel systems as our next level elements. In some cases, where the physical constraints on the system might in fact determine how much functionality we can provide, we will need to proceed in this way. However, in cases where the physical constraints are not as important, starting with the physical, while perhaps a familiar method, has the potential disadvantage of stifling innovation by pre-supposing a specific implementation.

Creating a logical, rather than physical architecture first, allows more creative reasoning about the overall architecture of the system. In thinking this way, similar elements can be grouped together, while disparate concerns can be separated, increasing modularity. Trade-offs between coupling (interconnections between elements) and cohesion (tightly connected elements combined into one) can be evaluated and decided. In our next steps, specifically joint realization, we will consider how the different viewpoints must be overlaid one upon the other to create an overall architecture.

The creation of any particular logical architecture requires real domain expertise and experience and involves many factors beyond the scope of this book. While there are architectural principles that can be applied, MDSD does not automatically create these elements. It does, however, provide frameworks for reasoning explicitly about the kinds of issues that directly influence the architecture. The process of designing the architecture is an interactive one, involving initial formulations and revisions. The practical approach is to make an initial draft of a set of elements, perform the next steps in the flowdown, and use this to either validate or refine the element choices.

Note that logical elements can be either system elements that contain some combination of hardware, software, people and information, or can be workers. A worker is a human that is part of the system at the level above, and thus is not represented as an actor. For example, if my enterprise (level 0) system is an aircraft, we would likely consider the pilot to be inside the system of the aircraft, thus the pilot does not appear as an actor at level 0—in fact, the pilot does not appear at all at level 0. At level 1 we have the pilot, along with logical elements such as navigation, weapons, environment, and so forth.

So the pilot could come out as a worker—a *human system* element. The pilot is still inside the enterprise, so we do not call him or her an actor, but within the scope of level 1, all the system elements—system and worker—interact with each other and are in a sense actors to each other. Note also that this is a choice—the pilot could remain as a worker, hidden inside another system 1 element, say something like *aircraft command and control*. In this case, the pilot would not appear at level 1, and could come out as a worker at level 2.

MDSD Step 10: Creating element context diagrams

As logical elements are determined, it helps to create context diagrams to show these elements and their relationships to actors, and to each other. To create a context diagram for a level 1 system element, we draw the element, along with all of other elements with which it interacts. The elements can be one of three possible types:

- Actors, which also appear on the level 0 context diagram
- Other level 1 system elements
- Level 1 workers

Context diagrams can be created for each logical element. Like an enterprise context diagram, these show a certain element, its actors, and their I/O entities. When drawn in a UML or SysML modeling tool, these context diagrams also serve as collecting points for the operations that will be derived for these elements (Figure 4-1). Note the shifting focus or context here—if we choose to look at each element in a particular level as our system under consideration, the other elements at that same level will be its actors.[2]

With an initial cut at the logical elements for this model level or level of decomposition, we are ready to proceed to the realization of the operations.

[2] Currently no modeling tool handles this issue well. Several workarounds are possible—differing coloration of the elements in different diagrams is a possibility.

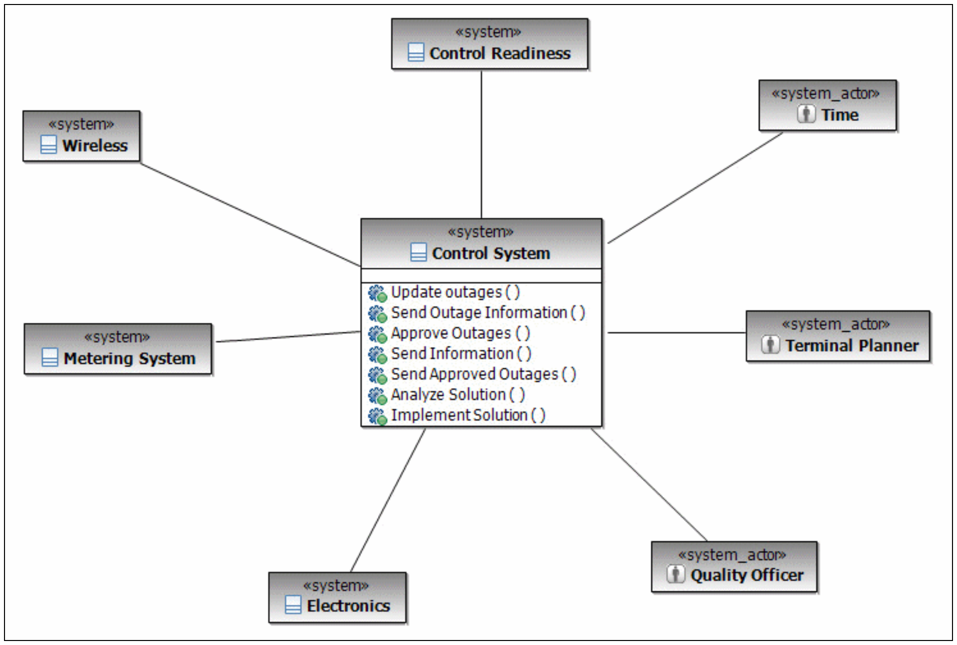


Figure 4-1 Level 1 context diagram showing both human and non-human actors

Operation analysis

For each operation, the question we next need to answer is, how is this operation accomplished (or realized) as a collaboration of elements at the next level of decomposition? To answer this, we first write out the flow of events for the operation. Normally, when we write a flow of events, say for a use case, we keep to a black-box perspective exclusively. To realize an operation as a collaboration of lower level elements, clearly we need both black- and white-box perspectives.

In a way, we already have the black-box perspective of each operation. Look at the black-box sequence diagram of any use case that uses this operation. You will see a series of messages (requests) beginning with the one that invokes the operation. Follow this series of messages until you hit the next operation on the same element, or the end of the use case, whichever comes first and stop. What you have traced is the set of black-box interactions that accomplish this operation.

In Figure 4-2 the operation reject trade, from a black-box perspective, would encompass the Enterprise Application's request of the Quality Officer to update trade status, as well as Enterprise Application's request of the Quality Engineer to update trade status.

Note that some operations consist of only one black-box interaction, the one that invokes that operation. This is the case with Provide Energy or AS Trade Details in the same diagram.

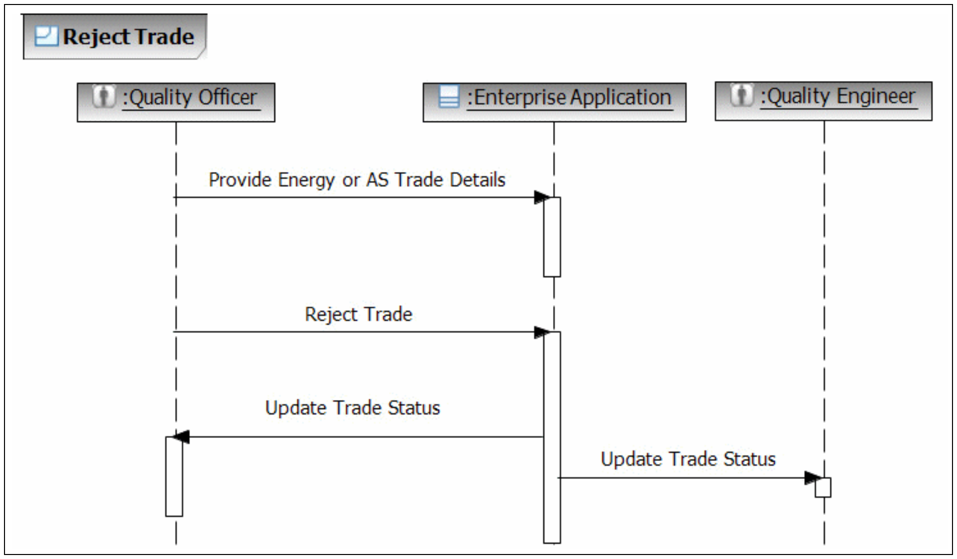


Figure 4-2   Black-box sequence diagram

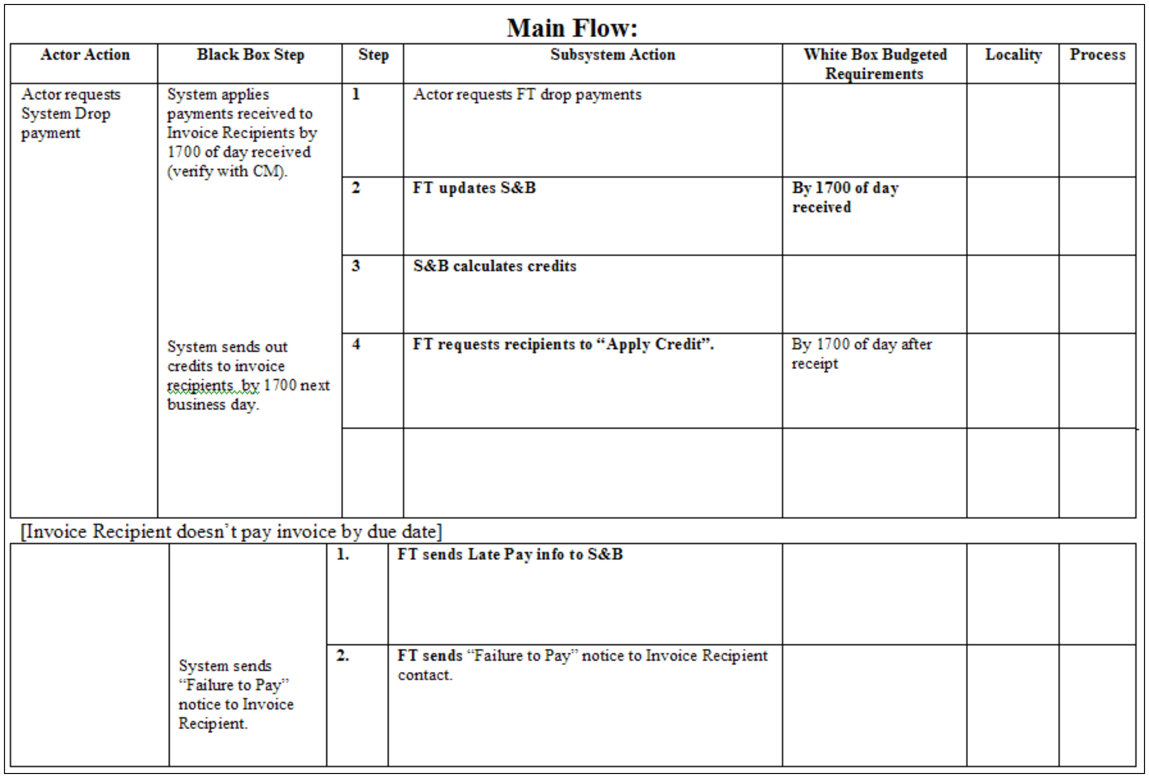
It helps to keep this black-box sequence in mind as we proceed to the work of creating the white-box expansion of the operation. To create this expansion, first we will write an operation specification for each operation. This operation specification, like a use case specification, describes a sequence of events to accomplish a goal. To write one for an operation, we work our way through the black-box description of the operation, and elaborate the black-box actions into white box, by explaining how the elements at the next decomposition level collaborate to accomplish the operation.

In the accompanying example, we show how the operation initiate new sale is realized by a collaboration of the point of sale and order processing elements (here called subsystems).

Enterprise operation: Initiate New Sale

- Black-box perspective:
  - Clerk starts new sale
  - The system enables the scanner
- White-box perspective:
  - The Point-of-Sale subsystem clears the transaction, brings up a new sales screen, and requests the Order Processing subsystem to create a new sales list
  - The Order Processing subsystem starts a new sales list
  - The Point-of-Sale subsystem enables the scanner

So that we can add additional items to the white-box expansion, we use a tabular format for an operation specification such as Figure 4-3. This template can be customized to meet the needs of specific modeling situations.



Main Flow:

Actor Action	Black Box Step	Step	Subsystem Action	White Box Budgeted Requirements	Locality	Process
Actor requests System Drop payment	System applies payments received to Invoice Recipients by 1700 of day received (verify with CM).	1	Actor requests FT drop payments			
		2	**FT updates S&B**	**By 1700 of day received**		
		3	**S&B calculates credits**			
	System sends out credits to invoice recipients by 1700 next business day.	4	**FT requests recipients to "Apply Credit".**	By 1700 of day after receipt		

[Invoice Recipient doesn't pay invoice by due date]

		1.	**FT sends Late Pay info to S&B**			
	System sends "Failure to Pay" notice to Invoice Recipient.	2.	**FT sends** "Failure to Pay" notice to Invoice Recipient contact.			

Figure 4-3 Operation specification example

The heart of an operation specification is the flow of events. The columns to the left, system actor action and black-box step, allow the modeler to show the black-box flow of events for the operation. This can be useful as the white-box flow is being developed. Because this black-box flow also appears identically in the use case specification in which this operation is used, in practice we sometimes delete this black-box information after the operation's white-box sequence has solidified. The white-box steps incorporate all of the behavior specified in the black-box steps, described at this lower level of abstraction.

In the white-box sequence, notice that we do not use the term *system* nor do we use the name of the enterprise. Anytime the black-box flow named something that the system or the enterprise does, we must translate that into what the elements of the system or enterprise do. Main flows are thus expanded, followed by any alternate flows as shown in the example.

The table also contains columns for process and locality, which are not completed initially, but will be used later to express joint realization of the operations.

With the flow of events created, we now draw a white-box sequence diagram to allow us to determine the operations that the elements at this level must perform to realize the operation from the level above. White-box sequence diagrams are quite similar to the black-box sequence diagrams. The difference is that instead of a single UML classifier (or SysML block) to represent the system, we instead use multiple UML classifiers (or SysML blocks) representing the logical elements at this decomposition level.

We then translate the white-box expansion flow of events developed before, into requests made between these logical system elements and the actors. In the example of Figure 4-4, the actors In Store Customer and Bank Credit Card System interact with the Sales Clerk (modeled here as an element, but could have been shown as a worker (if we do not plan to further decompose) and six logical system elements.

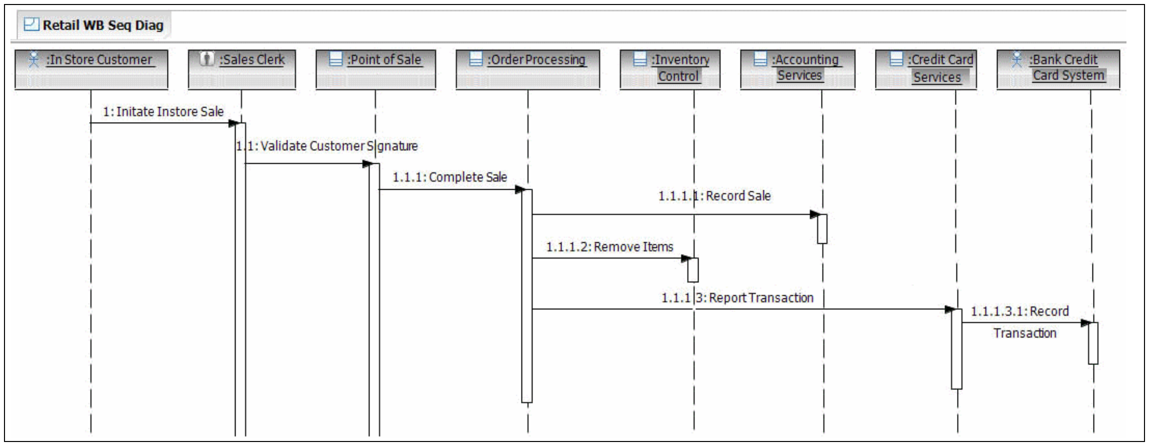


Figure 4-4 White-box sequence diagram for operation Compute Online Sale

In the same way as described for black-box sequence diagrams, operations are identified for system elements by looking for arrows pointing in towards each logical element. Note that with the black-box sequence diagram, we identify only operations on a single system element (the enterprise) while with the white-box sequence diagram we identify operations on all the elements at the next level. Thus we only use the black-box sequence diagram to get started (for example, when analyzing the enterprise use cases), and can use white-box sequence diagrams at every level below that.

As the operations for each element at this level are identified by realizing each operation from the level above, they are refactored and consolidated in the same way we described previously.

Flowdown to further levels

To continue the flowdown to levels below level n, the same process is used. Each operation of each level n logical element is realized using an operation specification and white-box sequence diagram, thus identifying operations on elements at the next level.

MDSD Step 11: Create use case models at levels below the enterprise

A common observation at this point in the flowdown process is that it seems we do not need use cases at levels below the enterprise. We need enterprise use cases to get the flowdown going, but then flowdown proceeds from operation to operation without requiring use cases at other levels, right? Well, yes and no.

It is true that flowdown to determine logical system elements and the collaborations and operations does not require the development of use cases, but for the model to achieve completeness, attention should be given to use cases at levels 1, 2, 3, and so forth.

Use cases at level 1 (and below) are useful for several purposes similar to the widely known uses of use cases, namely for testing and project management. They are also useful for documentation, since they show how the element is used, that is, how its operations are used in sequence to accomplish a specific result. The team responsible for building and testing a level 1 element, can use the use cases for this element to schedule iterative builds and releases, and also to derive test cases. Yes, they would also test using the element's operations, but these operations are atomic and do not always reflect complete usages.

Use cases for elements at any level can be determined from the operation realizations at the level above. For example, looking at the white-box sequence diagram of an enterprise operation, imagine shining a flashlight beam down the page from the level 1 element. The light would illuminate only the interactions with that element. The set of these interactions comprise a use case for that element. The sequence of events, including both the requests made of the element and the requests made by the element, are precisely one case of usage. This sequence shows how this element is used to accomplish a higher level purpose, namely the realization of the enterprise operation, and in turn the higher level purpose of fulfilling the enterprise use case.

It is important to see the interdependence between the use cases of elements at level 1. The complete usages of each level 1 element are intertwined with those of the other level 1 elements with which it collaborates to fulfill an enterprise operation. You can think of an enterprise operation realization as a use case for each level 1 element that participates in its realization. In practice, depending on the purposes of the model being developed, it might or might not be necessary to do the work to pull these use cases out of the realizations, using the flashlight technique mentioned before, and to draw them out as use cases, complete with use case diagrams and flows of events.

In general, if there is a team chartered to build an element, then this kind of work is useful at that level; if the element in question is simply for analysis purposes and will not be designed and built as such, then this work might not be justified.

5

Understanding distribution of responsibility

In the previous two chapters, we have examined the general transformation from looking at the system as a black box to looking at it as a white box, and understanding its context, its collaboration with other entities, and the distribution of responsibilities across logical entities in both the black-box and white-box perspectives. In doing so, we have concentrated primarily on the logical viewpoint. In this chapter, we turn our focus to the distribution viewpoint.

Joint realization is the MDSD technique for integrating various viewpoints in one table, allowing us to reason about systems concerns across as many viewpoints as necessary. Localities are the means for reasoning visually about distribution of logical responsibilities to locations where processing will take place. We discuss them first.

Localities

Here we consider the importance of localities in relation to systems engineering.

MDSD Step 12: Developing a locality model

The logical viewpoint is useful for reasoning about system functionality, segmentation, element interaction, collaboration and interfaces at various model and decomposition levels. The distribution viewpoint is needed to reason about a different set of concerns. In virtually every system, we need to reason about where functionality should be deployed, not just what functionality should be implemented. Distributing the system elements and their functions involves concerns such as space, time, and communication pathways. Decisions made here affect performance, maintainability, reliability, and cost.

Localities and systems engineering

In systems engineering, the physical resources are a part or aspect of the system. It follows that semantics need to be provided to reason about the properties of the elements of the physical realization of the system. More specifically, the outcome of a systems engineering effort includes a detailed specification of the hardware to be built or acquired. Note that systems engineering does not include the hardware engineering disciplines (mechanical, electrical) but does include sufficient specification to be used as input to the hardware design team.

As we have discussed, MDSD uses an analysis level, distribution viewpoint diagram called *system locality view*. In the distribution viewpoint, the system is decomposed into elements that host the logical subsystem services. Locality diagrams are the most abstract expression of this decomposition. They express where processing occurs without tying the processing locality to a specific geographic location, or even the realization of the processing capability to specific hardware. Locality refers to proximity of resources, not necessarily location, which is captured in the design model. For example, a locality view might show that the system enables processing on a space satellite and a ground station. The processing hosted at each locality is an important design consideration.

The locality diagrams show the initial partitioning, how the system's physical elements are distributed, and how they are connected. The term locality is used because locality of processing is often an issue when considering primarily nonfunctional requirements.

Locality semantics

Localities are used to capture the distribution characteristics of the system class, In particular, localities have class and instance attributes, and measures of effectiveness captured as tagged values. Because localities are parts of the system that host or implement functionality, they are used to reason about nonfunctional or quality aspects of the system.

Localities have two default sets of tags:

- **Quality**: Reliability, availability, performance, capacity, and so forth
- **Management**: Cost and technical risk

These locality characteristics form a nominal set. Each development team should determine the best set of characteristics for their project. This determination could be a development case specification activity.[1]

Locality characteristics are set to meet their derived requirements. There is a subtle difference between characteristics and requirements. For example, for good engineering reasons, you might specify a locality that exceeds requirements. In “Localities, services, and interfaces” on page 82 we show that localities host subsystem services.

Connection semantics

Localities are joined by connections, which represent the physical linkages between localities. Connections are stereotyped associations with tagged values, again capturing characteristics. Nominal connection tags are:

- **Throughput**: Transfer rate, supported protocols
- **Management**: Cost, technical risk

Because localities host services, connections must pass service invocations. In fact, there are at least three types of flow we have to consider in systems:

- Control flow
- Data flow
- Material flow

Consider, for example, the throttle in an automobile. The throttle linkage is the *control* connection that transmits the service requests (open or close) to the throttle. The gas line is also a connection to the throttle. The gasoline itself is not a service request, but rather a *raw material* used by the throttle to perform its services. Finally, there can be a network *data* connection to the throttle containing an ongoing stream of environment and automobile status data that is used to adjust the response to the throttle.

[1] A development case is a RUP artifact to customize a development process.

Localities and nodes

The UML documentation states that UML nodes are classifiers that have processing ability and memory. Used in deployment diagrams, the UML node semantics support reasoning about the hosting processors for the software components. The implicit assumption is that the physical resources are outside the software under consideration. For example, in software engineering, the hardware is often seen as an enabling layer below the operating system. UML does provide design and implementation-level artifacts for deployment diagrams:

- **Descriptor diagrams:** For the design level
- **Instance diagrams:** For the implementation level

In particular, instance deployment diagrams are meant to capture configurations and actual choices of hardware and software, and to provide a basis for system analysis and design, serving as an implementation level in the distribution viewpoint.

The UML reference manual describes an instance version of a deployment diagram as *a diagram that shows the configuration of run-time processing nodes and component instances and objects that live in them*.

In MDSD, this intent is to model the places where services are performed, that is, where the functionality described in the logical models happens. Modeling localities allows for reasoning about the distribution of functionality. Localities express a set of constraints on the realization of the functionality performed by hardware, software and people. Using localities, engineers can model what functionality can (and cannot) be grouped together.

Localities, services, and interfaces

A locality specifies places where logical services are provided. In practice, each locality will provide a subset of the services of one or more of the logical subsystems. The determination of those services is an outcome of the joint realization.

The set of hosted subsystem services for a given locality should be captured with UML or SysML interfaces. Subsystems are classifiers, and their services are classifier operations. Both UML and SysML allow operations, and therefore subsystem services, to be organized into interfaces. That is, an interface is a subset of subsystem services. In this approach, we define the needed interfaces for each of the subsystems and then assign them to the appropriate localities. Generally, there will be more than one interface associated to a locality[2].

[2] See further discussion and illustration (Figure 5-1)

Design trades

Design trades is the name of a common systems engineering technique: Building a set of alternate design approaches; analyzing the cost, quality, and feasibility of the alternatives; and then choosing the best solution. The locality view supports design trades by containing more than one locality diagram, each representing a different conceptual approach to the physical decomposition and distribution viewpoint of the system. It also supports reasoning about the various parameters associated with the localities through their tagged values in UML and the parametrics in SysML. These associated parameters can be used to drive simulations in external programs such as Matlab.

Figure 5-1 and Figure 5-2 are locality diagrams that document different engineering approaches to a click-and-mortar enterprise with a number of retail stores, central warehouses, and a Web presence.

The first solution (Figure 5-1) shows processing capability in the stores. The second solution (Figure 5-2) shows all terminals connected directly to a central office processor. In each case, characteristics can be set for the localities that are required to realize the design:

- The first solution uses in-store caching to improve performance, because system performance might be constrained by network bandwidth. This architecture, however, can come at a maintenance and hardware procurement cost due to distributed nature of hardware and software. Upgrades to software will have to be performed across the whole network.
- The second example becomes more attractive as bandwidth across the network increases, due, let us say, to the introduction of fiber optics. In this case, there is not so much a performance penalty, and maintenance and upgrades become easier and less expensive due to the centralized nature of the processing.

It is precisely for reasoning about these kinds of issues that we use localities and connections. Today, most people would agree that Figure 5-1 represents a better design; however, the solution in Figure 5-2 might be considered superior in a few years, as cost of increased bandwidth decreases and network reliability increases.

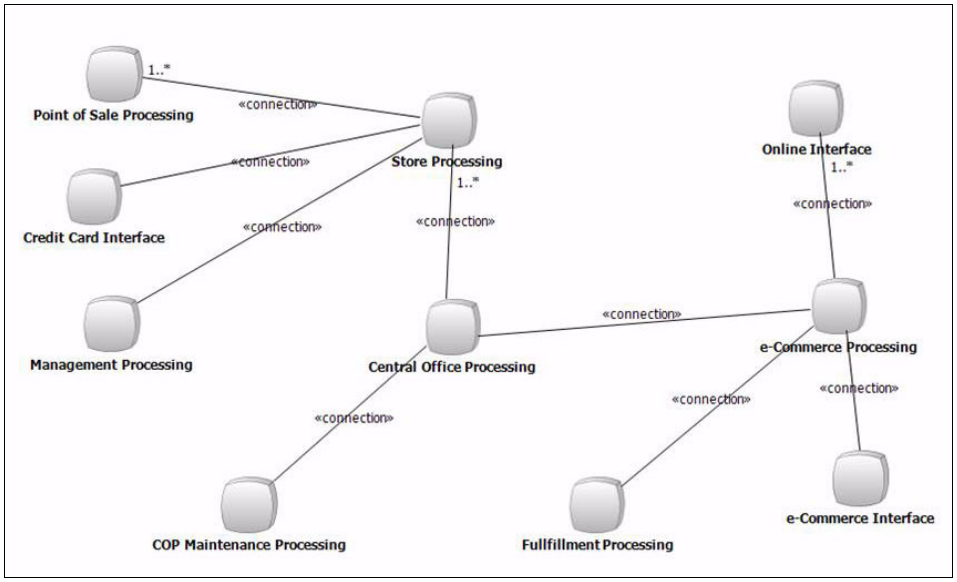


Figure 5-1   System locality view: Example 1

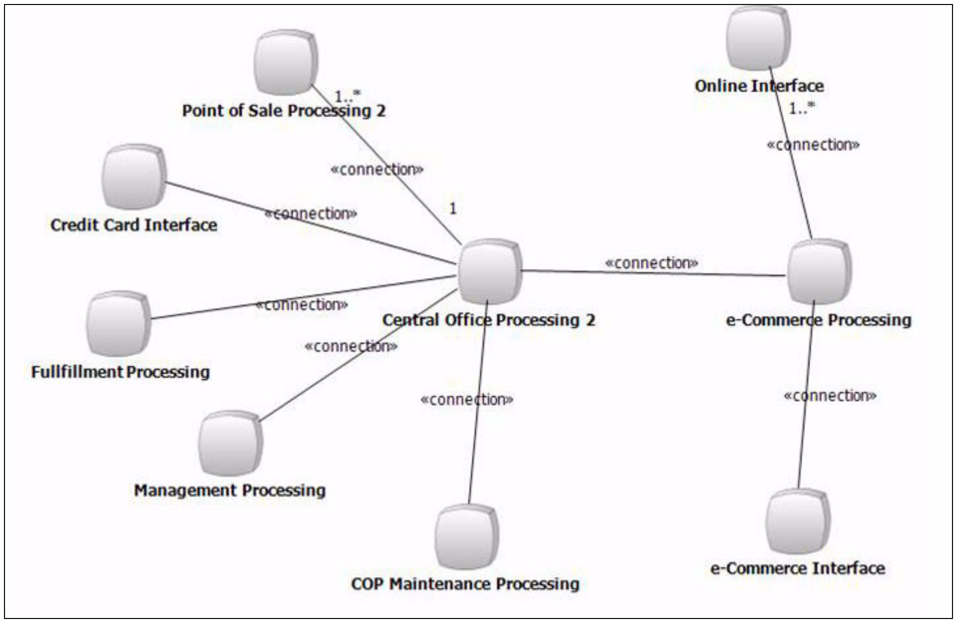


Figure 5-2   System locality view: Example 2

Sequence diagrams with localities

After drawing a locality view, the next step is to analyze how the operations on the various logical elements will be deployed at these places. To do this we construct a new sequence diagram, similar to the ones we have already done, but instead of the logical elements and actors, we use the localities and the actors as the lifelines. We create such a locality interaction (sequence) diagram for each operation at the level above which we are doing our locality analysis. Thus there will be a locality interaction diagram for each white-box sequence diagram at this level.

To determine the messages between the elements on our locality interaction diagram, we simply copy the messages from the white-box sequence diagram one-for-one onto the new diagram. The messages are the same; the difference is the elements to which the messages go:

- In the white-box sequence diagrams, messages are requests of some logical system element to perform some operation.
- In the locality interaction diagram, the same messages indicate where the operation is to be implemented.

We can think of it as a request being made of a distribution location, where part of the system is implemented. Notice that it is quite common to have numerous reflexive messages (messages that go from an element back to itself), because this means that a number of operations happen consecutively at one place. Figure 5-3 shows how the initiate new sale operation from an earlier illustration is distributed across the locations in the retail system.

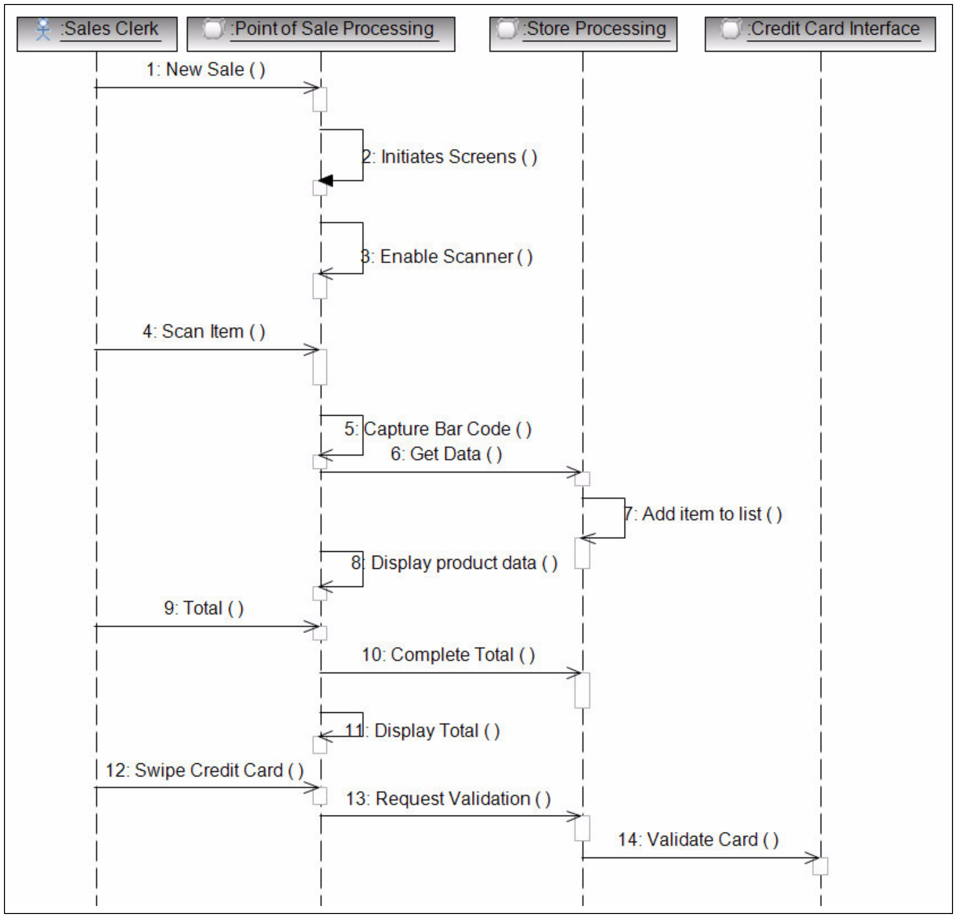


Figure 5-3 Sequence diagram with localities

Joint realization

At this point, our next step is joint realization. If we think of an individual operation in the system, at any level of decomposition, it has a tie to both its logical element, and to the distribution element where it is implemented. An analogy would be a person who is a citizen of one country, but a resident of another. The operation is a citizen of its logical element, where it was *born* and had its origin in the model. The same operation is a resident of the locality where it has been implemented; where it now *lives* and performs its work.

Because operations are nearly always implemented in groups, we can use a construct such as a UML interface to group them and show this joint realization relationship as shown in Figure 5-4. For example, the operation Analyze Data is an operation of the logical element Logical System3, and is implemented at the Data Analysis locality.

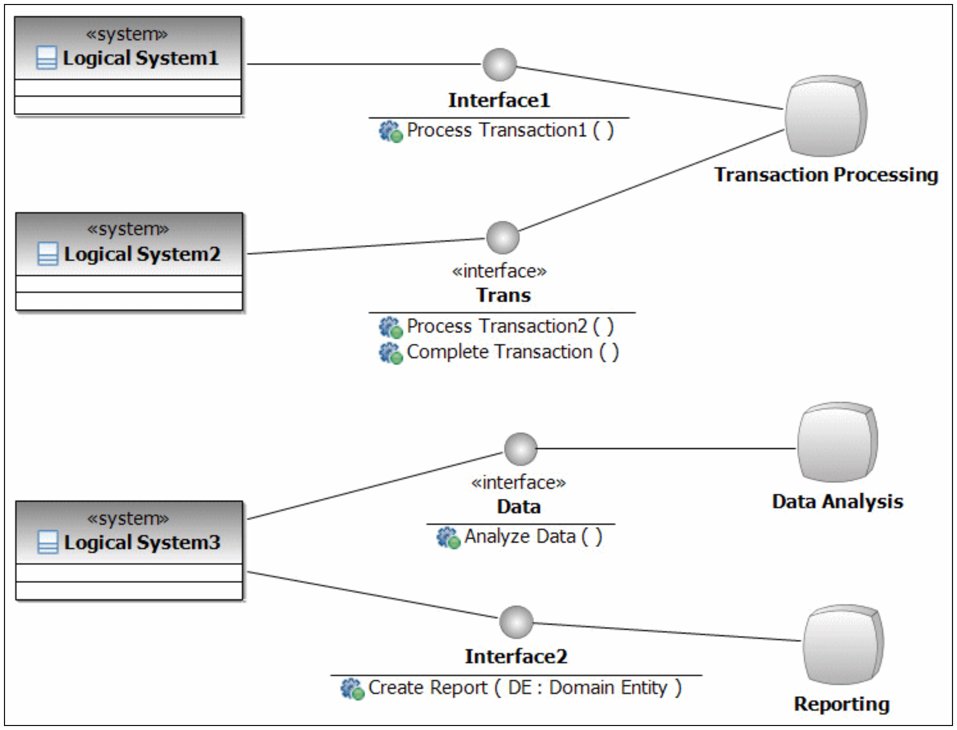


Figure 5-4 Joint realization diagram

Not all levels of decomposition have to include distribution models. Distribution models are included where they make sense and address concerns important to the system. For example, if our level 0 is a corporation, and our level 1 logical elements are major functions of the corporation (such as marketing, finance, human resources, manufacturing, and so forth), it probably does not make sense to do distribution modeling at that level. It is likely the various functions (operations) of say, marketing, are not easily located to a particular place. Distribution modeling would likely start at the next level.

In practice, the exact sequence of the modeling work at a given level varies depending on the needs. In some cases, the logical model is created fully before proceeding to the distribution model. In others, some of the logical model is created, and then validated using distribution modeling before more of the logical model is developed. Multiple iterations are often used to refine the models as more is learned over the course of the system development effort. Eventually the entire model comes together.

This is also true when deciding whether to reason about distribution issues first with sequence diagrams, or with joint realization tables. In all likelihood, we should perform both activities in parallel—we can use joint realization tables to get a view of multiple dimensions, and we can use sequence diagrams to focus on and reason about functionality in the viewpoint. We now turn to joint realization tables, which we have actually used before and partially filled in, as we did operation analysis in the logical viewpoint (refer to “Operation analysis” on page 72).

Joint realization tables

In MDSD, we distinguish between functional requirements and nonfunctional requirements (NFRs). Functional requirements describe the system behavior as well as the collaboration among system components to accomplish the system behavior. NFRs pertain to how a system performs its functions and include concerns such as quality, quantity, and timeliness.

Joint realization tables (JRTs) decompose the system behavior in the context of the logical and distribution architectures and, at the same time, assign nonfunctional requirements to these system behavior steps (services/operations). In a real sense, this is the missing link—the item that was needed to connect object-oriented development models to the needs of the engineering community developing large-scale systems.

A JRT example that decomposes the task of printing a page is shown in Table 5-1.

Table 5-1 Partial joint realization table for printing a page

White -box Step	Action Performed	White-box Budgeted Requirements	Distribution Reference (Locality)	Process Reference
1	**LRF1: I/O Services** WSB1: receives the block and stores in an available data buffer in memory.	SUP1: 10 ms	DRF1: Printer Control Unit	PRF1: Data_rec
2	**LRF2: I/O Services** WSB2: updates the input data buffer queue with the address of the received block and sends the awaiting process input data buffer queue address list to the Raster Image Processing subsystem.	SUP2: 2 ms	DRF2: Printer Control Unit	PRF2: Input_data _buff_mgt

White -box Step	Action Performed	White-box Budgeted Requirements	Distribution Reference (Locality)	Process Reference
3	**LRF3: Raster Image Processing** WSB3: reads the buffer queue address list and begins reading the data blocks. As the block are processed, one or more page bitmaps are rendered to memory and stored in available page bitmap buffers.		DRF3: Printer Control Unit	PRF3: Page_RIP
4	**LRF4: Raster Image Processing** WSB4: indicates the input data block is available for reuse after the block is read and processed.		DRF4: Printer Control Unit	PRF4: Input_data _buff_mgt

The header material for the Build Page operation provides context for elaborating the JRT. This JRT decomposition allocates the functionality of the single black-box operation to white-box printer entities:

- The Action Performed column captures both the logical entity performing the action and the logical step performed. In this example, two logical entities, I/O Services and Raster Image Processing, collaborate to print a page.
- NFRs are allocated to the logical white-box steps in the White-box Budgeted Requirements column—for example, 10 milliseconds are allocated to the I/O Services' operation that receives and stores an available data block in memory.
- The last two columns provide the distribution and process references. In this example the Printer Control Unit locality and Data_rec process must perform the operation of receiving a block and putting it into memory within the same 10 millisecond budget.

The JRT maintains context, captures the logical and distribution decomposition, and provides for the allocation of nonfunctional requirements. With the JRT in place, (or, as noted before, developing it in parallel), it is useful to represent the content in SysML as a coupled set of sequence diagrams showing the same flow in the different viewpoints. Figure 5-5 shows the sequence diagrams for the print page service.

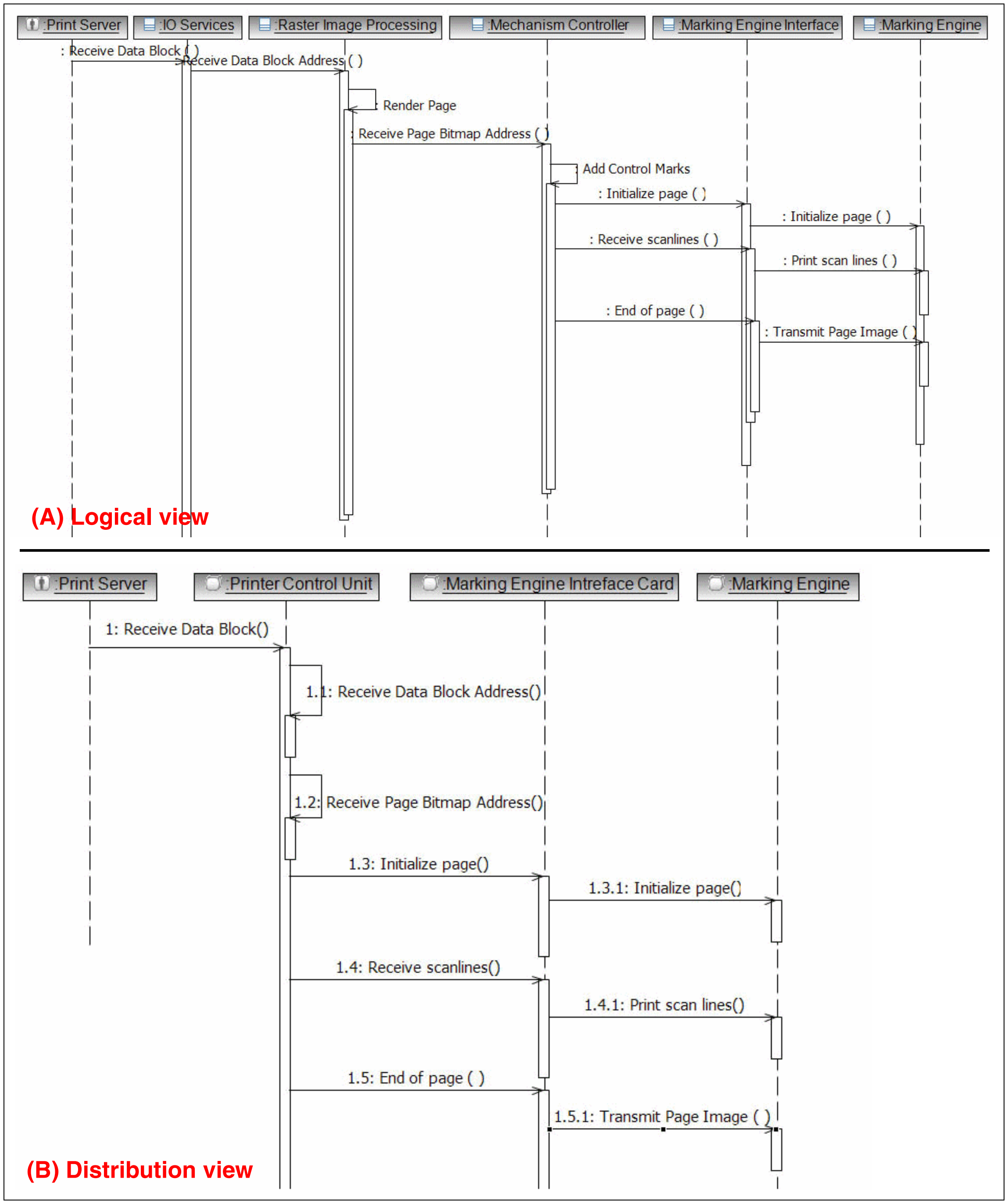


Figure 5-5 Logical and distribution sequence for print page flow: (A) Logical view (B) Distribution view

The insights gained by modeling the various elements (for example, analysis subsystems, localities) can lead to their refactoring and refinement until the needed set of interactions are identified and assigned to them. The candidate operations can also be refactored and refined as a result of the insights gained from the model.

Next, we must link the information in the JRT to a model of the system. To do so, it is necessary to identify the subsets of operations that are performed by a particular locality. Examples from the JRT are the Receive Data Block operations, which are performed by both the I/O Services subsystem and the Printer Control Unit locality. An initial set of interfaces can be derived by considering the mapping of operations to localities. In addition, cohesion principles should be applied to specify interfaces and then the mapping of operations to localities should be used as a check to ensure that the minimum requirement (the *split* of operations across localities for a given analysis subsystem) is satisfied. The resulting analysis-level logical and distribution views are shown in Figure 5-6.

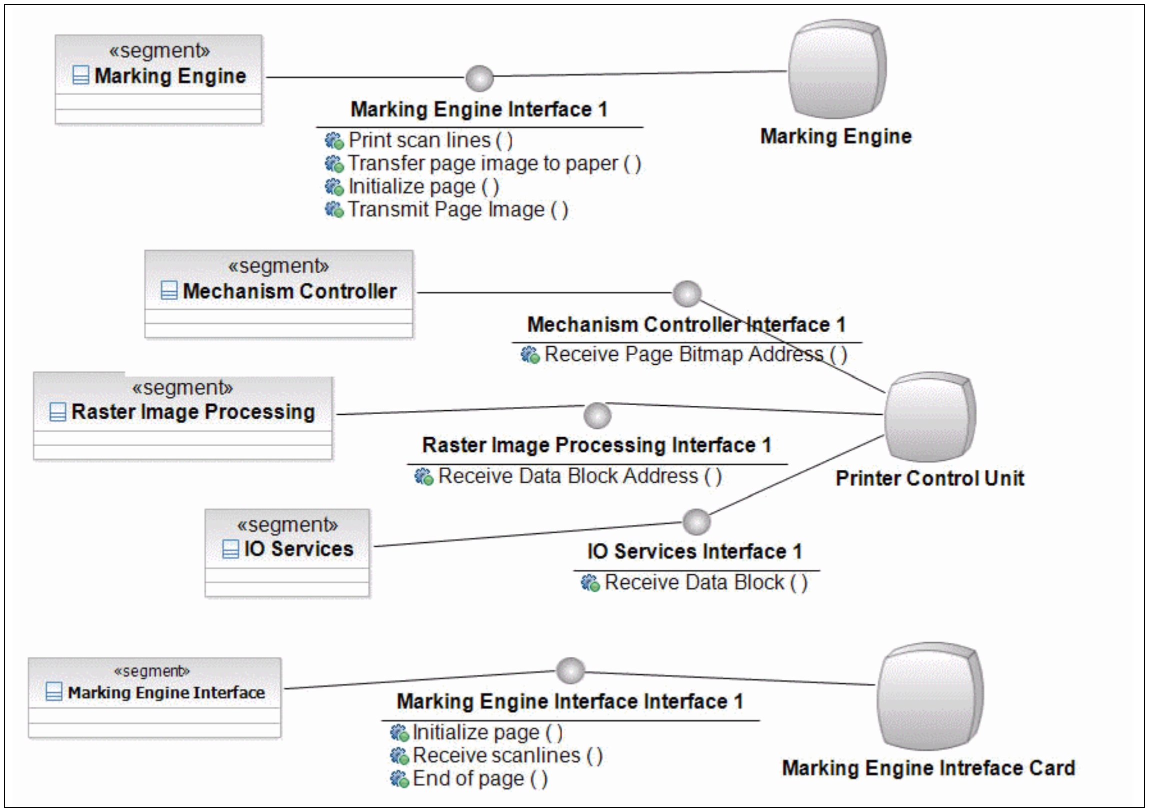


Figure 5-6 Association of logical entities, localities, and interfaces

This process of joint realization, using both sequence diagrams in the logical and distribution viewpoints, and through the use of joint realization tables, provides us with the means to reason about functional and non-functional requirements across a set of multiple viewpoints. We have given examples of the logical and distribution viewpoints, but we can also extend the concept to deal with other viewpoints as well.

In the sample JRT shown (Table 5-1 on page 88), we have a column for the process viewpoint. We could easily add other columns for other viewpoints as necessary (security and data, for example), as our problem domain dictates. We could also easily create stereotyped entities that would be able to be placed onto sequence diagrams as well.

Joint realization, then, is a robust technique to bridge the gap between software and systems engineering, while localities provide a good example of how UML and SysML can be extended to meet our analytical needs.

6

Tool support for MDSD

While many of the techniques that we have discussed in the previous chapters can first be captured manually on white boards and with other low-tech methods, MDSD really depends on tool support to be scalable and powerful. This chapter explains how to capture many of the artifacts already described in IBM Rational Systems Developer.

We begin by discussing a model structure to support MDSD, and then provide step-by step instructions for producing some of the most important artifacts.

Model structure

After beginning to create MDSD models, new modelers often ask, *What is the best way to represent and keep all this organized in a modeling tool such as Rational Software Modeler or Rational System Developer?* The answer of course is that there are many possible ways to represent the work products. What we describe here is one way to do this, which has been used with success during the course of work with several clients.

Organizing an MDSD model

MDSD models are best developed as creative acts, that is, they are developed as the system is explored and understood, not as an afterthought or documentation effort of something that is already understood. Because of this the development of the model is highly interactive, and best done by a team working together. Usually a core team of system engineers, architects or modelers, about three to five people, do the lion's share of the work, bringing in various stakeholders and subject matter experts throughout the process to supply important information.

In this kind of working environment, it can be useful to create the model, initially using flip charts and colored markers. This deceptively simple approach has a number of important benefits, including:

- Charts become a permanent record of the work from the beginning.
- Charts can be hung on the wall of a "project room," making the entire model visible at all times.
- Charts can easily be changed, and if different colors are used, can show some indication of a change history.
- Charts are easy and flexible to use, and thus do not impede the modeling process.
- Charts do not enforce UML or SysML modeling syntax rules, and allow work to proceed faster (this can also be a disadvantage by allowing modeling errors to persist undetected).

Flip charts also have some important disadvantages, including these:

- Charts are not easily copied and distributed for review.
- Charts are not automated and provide no traceability links or checking.
- Charts can become unattractive and tattered over time.

On balance, we find that flip charts are often the best way to begin a modeling effort and to do the initial drafting of model elements, use case flows, and diagrams. When the model has reached some level of stability, we find it best to put the model into a UML or SysML modeling tool, such as Rational Software Modeler, Rational System Developer, or similar, and maintain it there.

In other situations, where the engineers involved are experienced in using modeling languages and modeling tools, it might be better to proceed directly to using the modeling tool to capture the modeling work right from the start.

Organizing an MDSD model using tree-structured packages in a modeling tool can be confusing. The following sections detail an approach that we have found to work.

Level 0 model organization

In Figure 6-1 we show the main enterprise modeling elements. Blue boxes represent packages and yellow represent diagrams. The locations of individual modeling elements are shown in the next sections. At the top, a single Level 0 package contains the context and use case diagrams for the enterprise. Below that, a package is created to contain all of the use cases at this level. Within this package, a package for each use case contains the optional activity diagram for this use case, as well as the black box sequence diagrams for all documented scenarios of this use case.

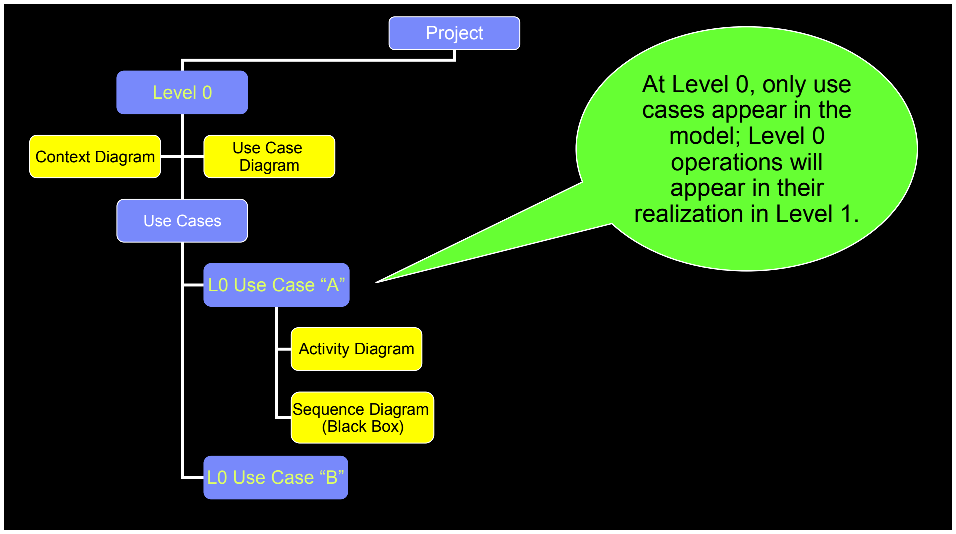

Project
Level 0
Context Diagram
Use Case Diagram
Use Cases
L0 Use Case "A"
Activity Diagram
Sequence Diagram (Black Box)
L0 Use Case "B"
At Level 0, only use cases appear in the model; Level 0 operations will appear in their realization in Level 1.

Figure 6-1 Level 0 model organization

At level 1, the structure becomes a little more complicated (Figure 6-2). At the top, there is a package for Level 1 and then immediately below that, any grand context diagrams or grand use case diagrams created at this level. These two diagrams show all or some of the level's logical elements and use cases, respectively, and are optionally created if they add clarity to the model. If a locality diagram is used at this level, it can be included here as well.

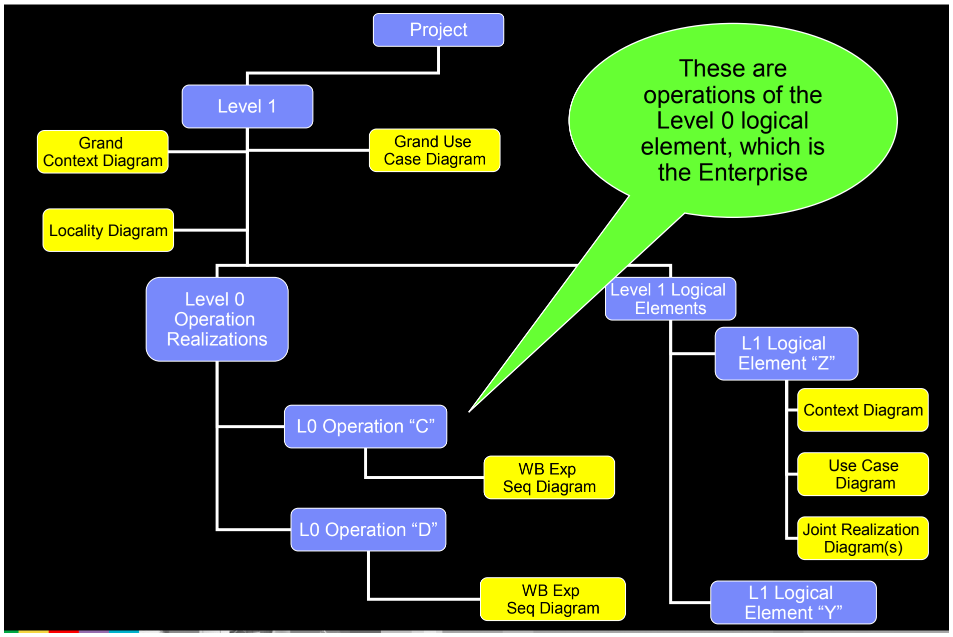


Figure 6-2 Level 1 model organization

At level 1 and beyond, we create two sets of packages for the remaining model elements. One holds the level 0 operation realizations, that is, the realization of each level 0 operation. There is a package for each level 0 operation, containing the white-box sequence diagrams for all scenarios of this operation. These items serve to expresses the realization of this operation.

The other category (level 1 logical elements) contains a package for each logical element at level 1. These elements were determined in the process of doing the realizations of the level 0 operations. In each element package, you will see a context diagram for this element, a corresponding optional use case diagram and any joint realization diagrams.

Moving on to level 2 and beyond, we add one more dimension (Figure 6-3). Looking under the level 1 operation realizations, we see an additional level of package for each level 1 logical element. This is because there are distinct sets of level 1 operations to be realized here at level 2-n set for each level 1 logical element. The remainder of the level 2 structure is the same as level 1. Levels below level 2 are identical to level 2 in structure.

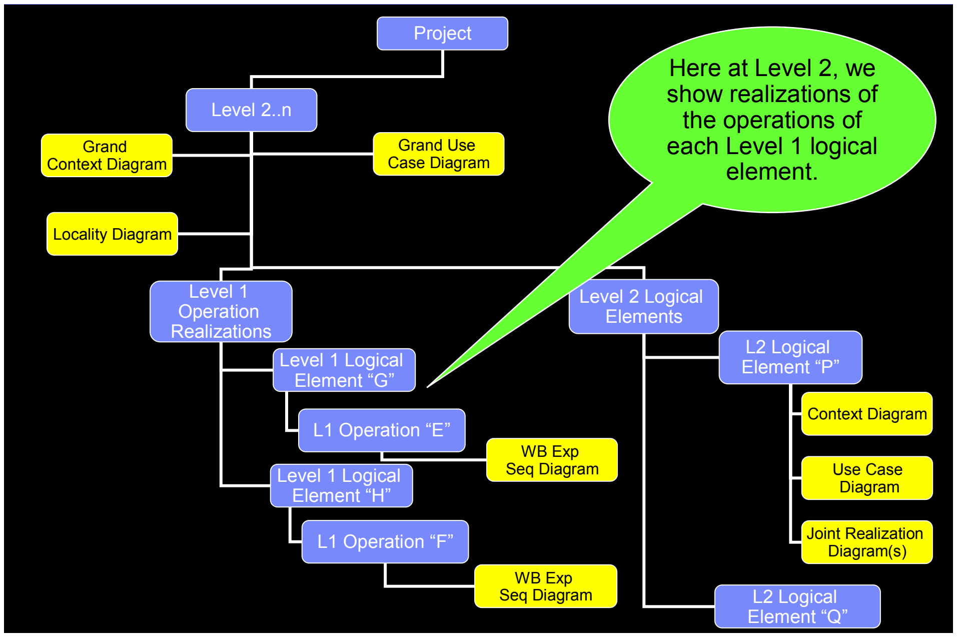
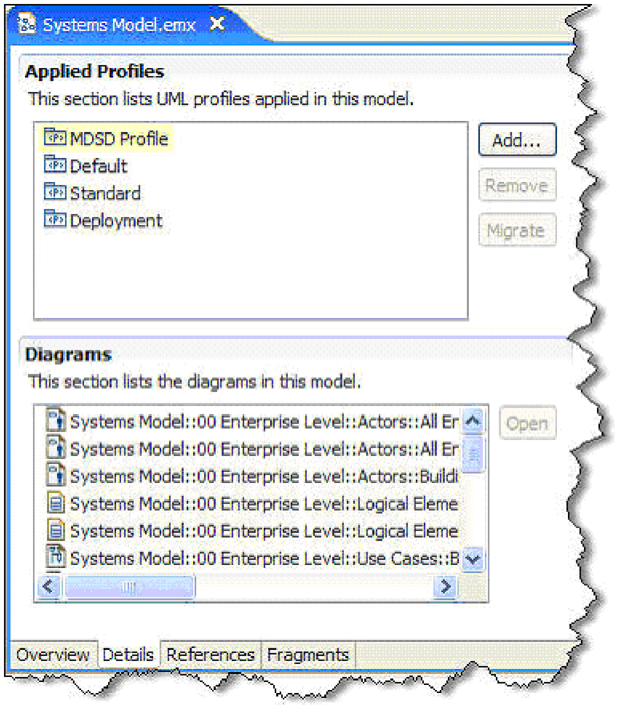


Figure 6-3   Level 2 and beyond model organization

MDSD UML Profile

A Rational Software Architect/Modeler plug-in has been created that, as of the time of this writing, contains a UML Profile for MDSD, as well as a model template with the structure described in the following sections. Once the profile has been applied, it should show up in the Applied Profiles section in the Details tab. Figure 6-4 shows an example with the profile highlighted.


Systems Model.emx

Applied Profiles

This section lists UML profiles applied in this model.

- MDSD Profile
- Default
- Standard
- Deployment

Diagrams

This section lists the diagrams in this model.

- Systems Model::00 Enterprise Level::Actors::All Er
- Systems Model::00 Enterprise Level::Actors::All Er
- Systems Model::00 Enterprise Level::Actors::Buildi
- Systems Model::00 Enterprise Level::Logical Eleme
- Systems Model::00 Enterprise Level::Logical Eleme
- Systems Model::00 Enterprise Level::Use Cases::B

Figure 6-4   MDSD Profile applied to the model

Stereotypes

Once applied to the model, the MDSD Profile adds stereotypes used for modeling MDSD concepts. Three of these stereotypes have shape icons associated with them. Figure 6-5 shows a domain model of the stereotypes and the UML elements that they are applied to. Figure 6-6 shows the shape icons associated with the three stereotypes.

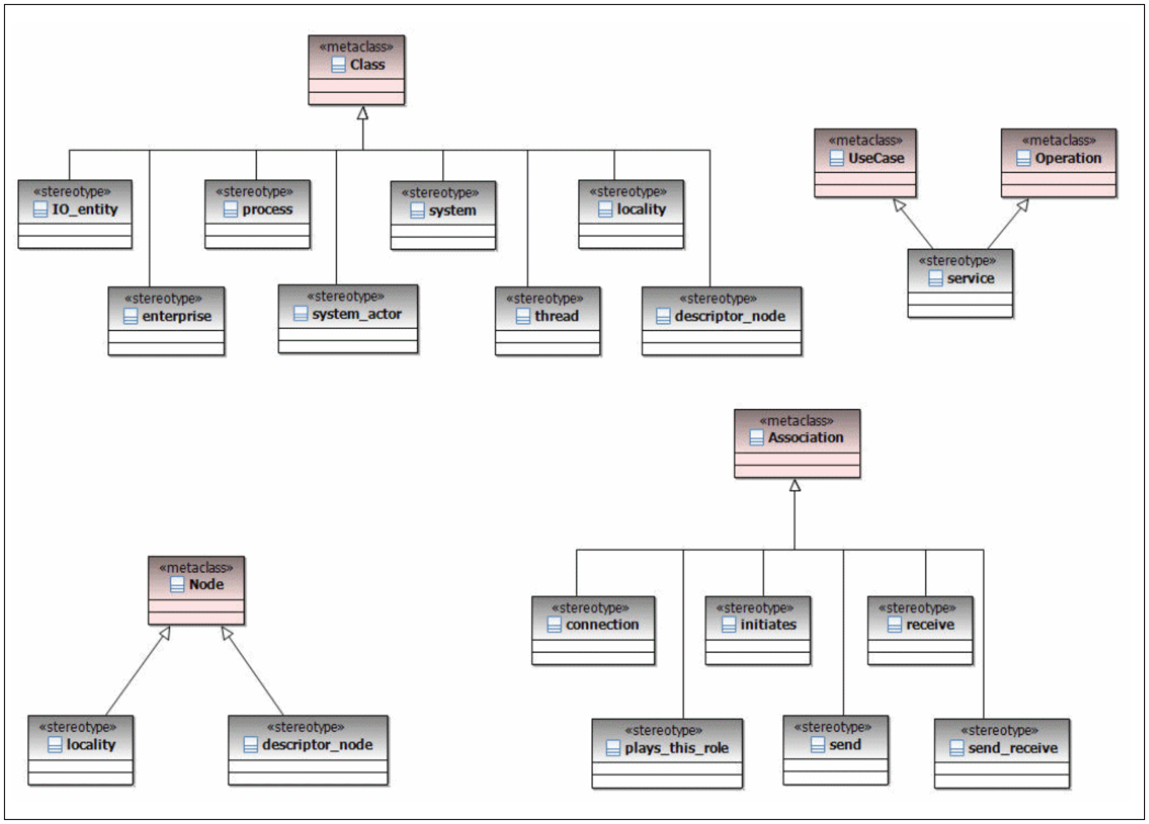


Figure 6-5 MDSD UML Profile stereotypes

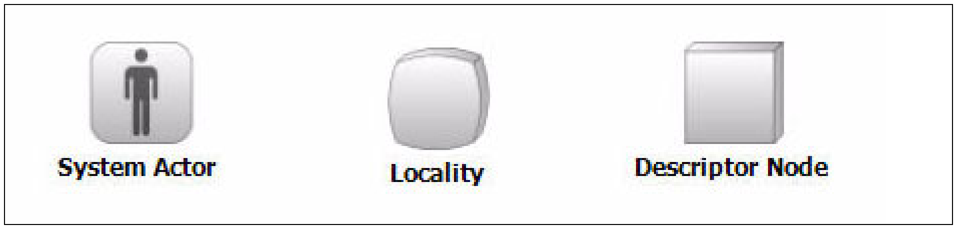


Figure 6-6 Shape Icons for three of the MDSD Profile stereotypes

Levels of decomposition

MDSD starts with system decomposition, that is, the division of a system into elements in order to improve comprehension of the system and the way in which it meets the needs of the user. In this approach, the system is decomposed into a comprehensible set of elements, each of which has a comprehensible set of requirements. Sometimes, to manage complexity in very large systems, system decomposition must be applied recursively. Effective application of system decomposition requires the means of modeling the system from a variety of viewpoints and at increasing levels of specificity.[1]

The model structure gives a means for deriving the next level of decomposition, and helps maintain traceability through the model through specifying the different system elements and their integration.

Figure 6-7 shows the beginning point for a system of systems (2 levels). In this instance the levels are named Enterprise Level and Level 1. In practice these names, as well as the names for any further levels, will be picked by the company or project doing the work. The names are not indigenous to MDSD and to be generic, we can call them Level 0, Level 1, Level 2, and so forth. (For everything at Level 1 and below the term Level 1+ will be used.) The term Enterprise for the top level seems to be well accepted though.

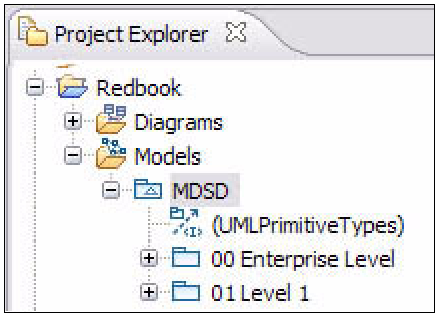
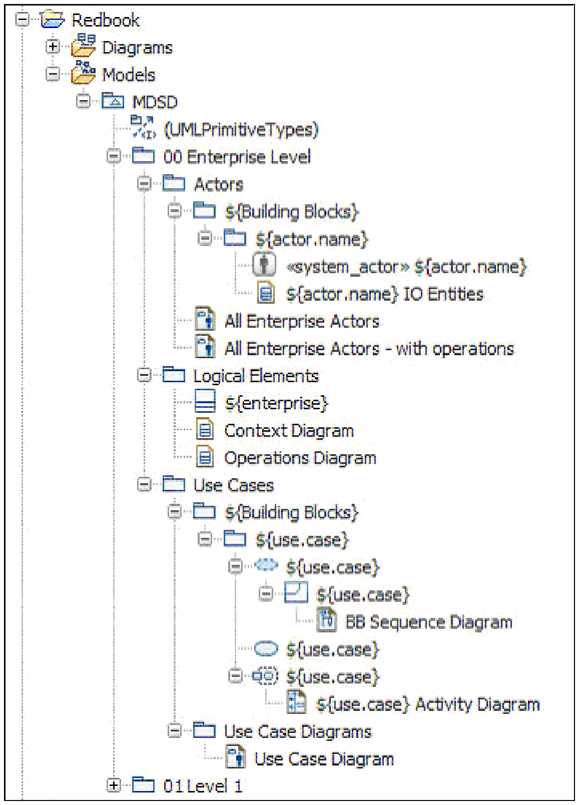


Figure 6-7 Two levels of a sample MDSD model

Within each level there are different artifacts that have to be grouped for organizational clarity. Here, there are three main groupings at the top level. There are Actors, Logical Elements, and Use Cases (Figure 6-8).

[1] Balmelli et al, *Model-driven systems development,* as cited in chapter 2, footnote 1 on page 17, `http://www.research.ibm.com/journal/sj/453/balmelli.html`



```
Redbook
  Diagrams
  Models
    MDSD
      (UMLPrimitiveTypes)
      00 Enterprise Level
        Actors
          ${Building Blocks}
            ${actor.name}
              «system_actor» ${actor.name}
              ${actor.name} IO Entities
          All Enterprise Actors
          All Enterprise Actors - with operations
        Logical Elements
          ${enterprise}
          Context Diagram
          Operations Diagram
        Use Cases
          ${Building Blocks}
            ${use.case}
              ${use.case}
                ${use.case}
                  BB Sequence Diagram
              ${use.case}
              ${use.case}
                ${use.case} Activity Diagram
          Use Case Diagrams
            Use Case Diagram
      01 Level 1
```

Figure 6-8   Model with first level expanded

Actors

In the Actors package, each actor has a package named for the actor. In this package is the actor itself along with a diagram showing the actor and all the I/O entities connected with that actor. Figure 6-9 shows an example of this kind of diagram:

- Notice that the associations between the Actor and I/O Entities are stereotyped with either <send>>, <<receive>>, or <<send_receive>>. These come from the MDSD UML profile contained in the MDSD plug-in.
- If the model template is used, there will be a Building Blocks folder in several places within the model structure. This is there to help create a consistent mini-structure. Just copy/paste the folder under the ${Building Blocks} folder, to the folder above.

- One example (see Figure 6-8 on page 101) would consist of copy/pasting the ${actor.name} package to the Actors package. Right-click on the pasted package and select *Find/Replace*. Change the placeholder name to the name of the actor. This will change the placeholder everywhere it exists under that package. (In the future this is being considered as an area for automation within the MDSD plug-in).
- Within the Actors package is a place for a diagram containing all of the actors. Also, if you want to show all the actors along with any operations they contain, this can be shown in another diagram.

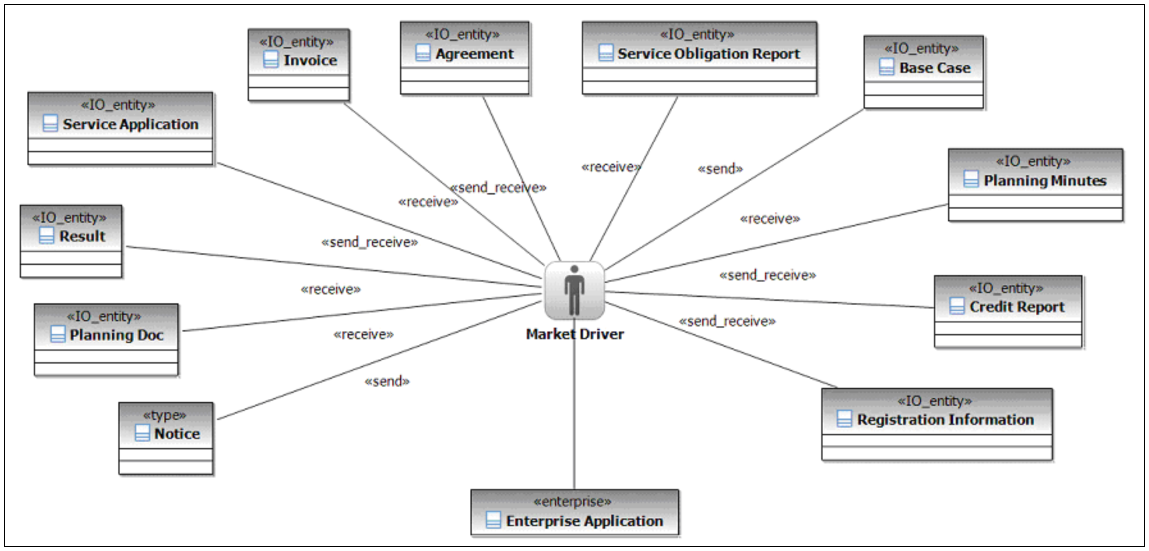


Figure 6-9 An actor with its connected I/O entities (the entity that it is connected to in any context diagram (the enterprise) is also shown)

Logical entities

At the top, or Enterprise, level there is only one entity, so it is a simple case. Under the Logical Elements folder is the element representing that entity, in this case a class, along with a context diagram. If the element has many operations as a result of use case analysis, then add an operations diagram, which is just the element with its operations displayed. Displaying them in the context diagram would make it cluttered.

Figure 6-10 shows an example context diagram.

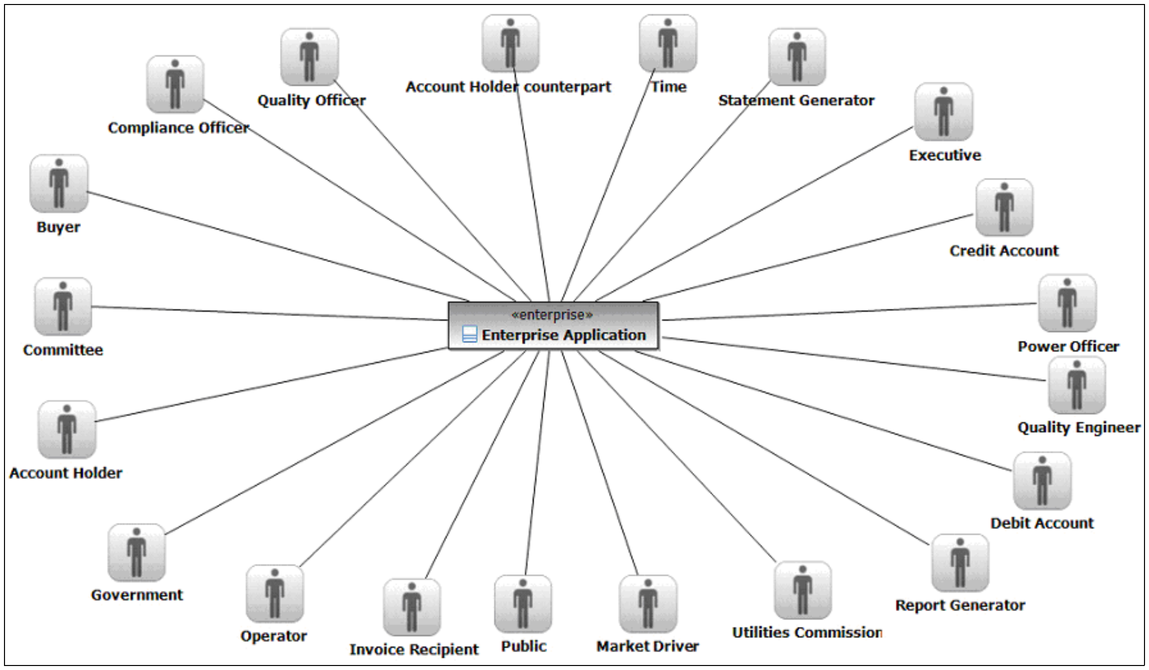


Figure 6-10 Example context diagram for the Enterprise level

For Level 1 (and further levels) there will be multiple elements, representing the architecture for that particular level of decomposition. For each of these there will be a package, under which will be an element, such as class (or possibly a block if using SysML) and a context diagram for that element. There is a Building Block template for this structure under the Logical Elements folder starting at Level 1. This is shown in Figure 6-11.

Figure 6-12 shows an example context diagram for a Level 1+ element. Notice that other, sibling, elements as well as actors can and will be a part of the context.

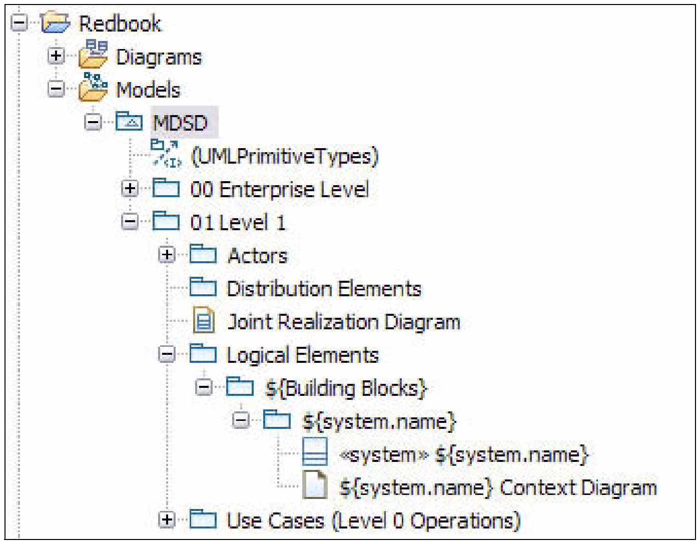


Figure 6-11 Level 1 model structure

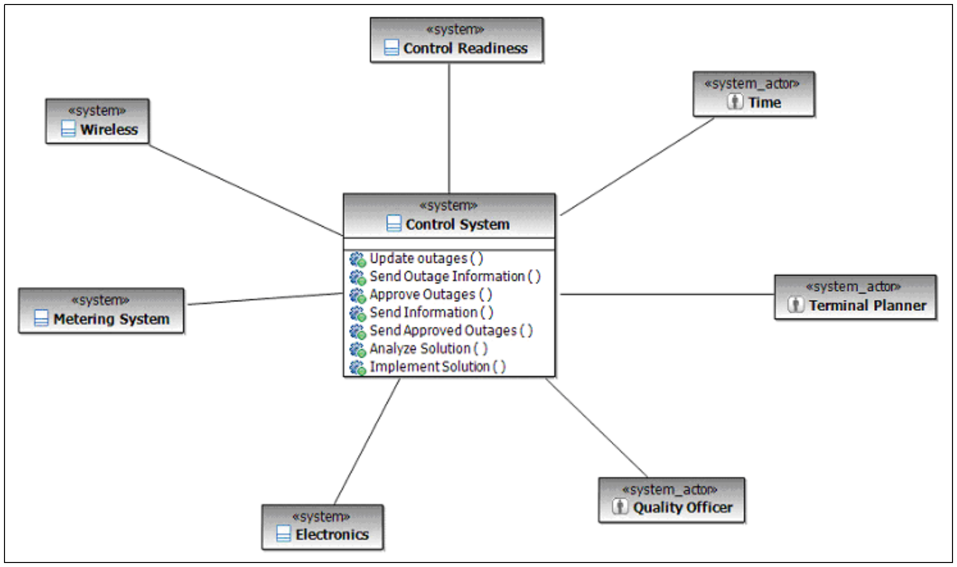


Figure 6-12 Example of a context diagram for a Level 1+ element

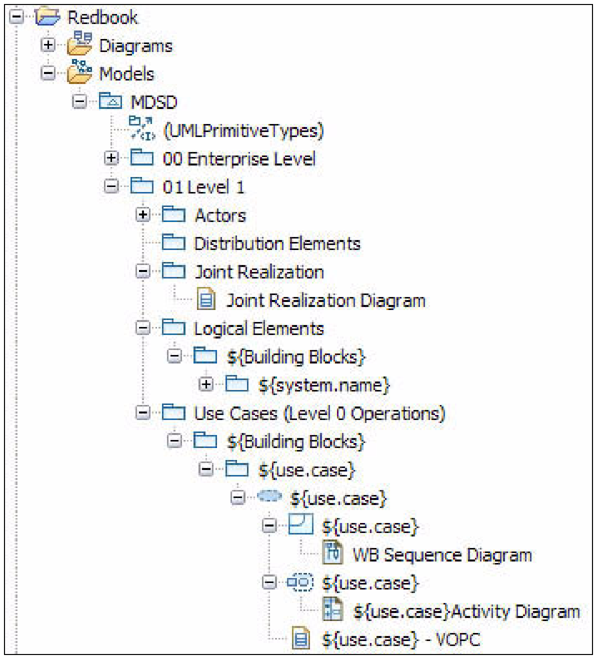
Use cases and operations

At the Enterprise level, use cases are discovered and analyzed, creating actors, use cases, sequence diagrams, and optionally activity diagrams:

- The use cases are organized under a Use Cases package.
- For each use case there is a package with the use case name, and within that is the use case itself, an activity diagram, and a collaboration containing an interaction containing the black-box sequence diagram for that use case.
- If you want to have multiple sequence diagrams, then there will have to be multiple interactions.
- Figure 6-13 shows this structure. There is a Building Block template for this in the Use Cases folder. Because there can be multiple use case diagrams, these are put in a Use Case Diagrams folder within the Use Cases folder.

For Level 1+ the structure can be slightly different:

- In the case where the operations at the level above are used for analyzing the behavior and distribution of behavior, there are no use cases or use case diagrams.
- Here the use case names are the same as the operations from the level above.
- The sub-structure has a folder for the use case, and within that is an activity diagram (as described above), and a collaboration containing an interaction that contains a white-box sequence diagram.
- An added optional diagram is a view of participating classes (VOPC) diagram. This diagram shows the actors and classes needed for the operation realization, and the associations needed for the messaging that is shown in the white-box sequence diagram to take place. Figure 6-14 shows an example of such a diagram.



```
Redbook
  Diagrams
  Models
    MDSD
      (UMLPrimitiveTypes)
      00 Enterprise Level
      01 Level 1
        Actors
        Distribution Elements
        Joint Realization
          Joint Realization Diagram
        Logical Elements
          ${Building Blocks}
            ${system.name}
        Use Cases (Level 0 Operations)
          ${Building Blocks}
            ${use.case}
              ${use.case}
                ${use.case}
                  WB Sequence Diagram
                ${use.case}
                  ${use.case}Activity Diagram
                ${use.case} - VOPC
```

Figure 6-13 Level 1 use case structure

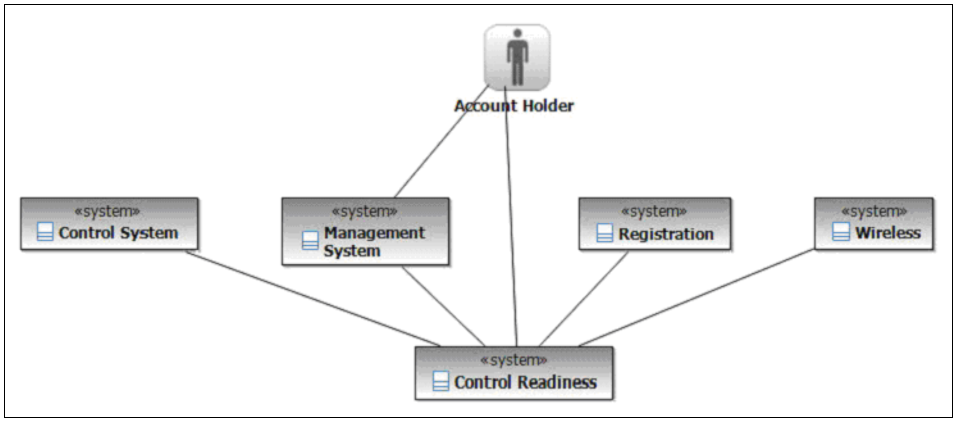


Figure 6-14 Example view of a participating classes (VOPC) diagram

Distribution entities

MDSD is described as both a *separation of concerns*, where designers can address each set of stakeholder concerns independently, as well as an *integration of concerns*, where there is enforcement of integration by requiring the use of a common set of design elements across multiple sets of concerns.

One of the concerns is the logical aspects of the system that have been described already. Another is the distribution aspects of the system. (There can be many more, such as process, security, and so forth.) The entities used to model this viewpoint are called *localities*. The distribution viewpoint describes how the functionality of the system is distributed.

Figure 6-15 shows an example of a diagram showing localities and their connections. The locality is represented using a stereotyped Class (or Block in SysML). In the MDSD UML plug-in, the locality stereotype uses the shape image shown in Figure 6-15. These go in a package named Distribution Elements. Localities can perform operations and have attributes appropriate to specify physical design. A connection is a generalized physical linkage. Connections are characterized by what they carry or transmit (data, power, fuel) and the necessary performance and quality attributes in order to specify their physical realization at the design level.

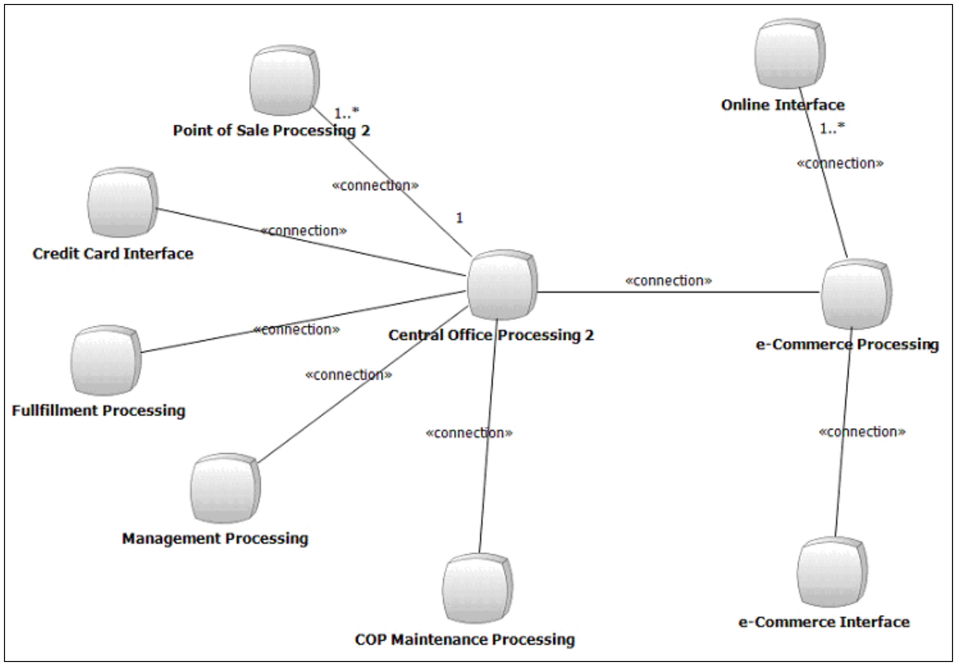


Figure 6-15 Locality diagram

The integration of concerns is accomplished by sharing interfaces with the logical entities. Figure 6-16 shows how operation signatures can be shared between the logical entities and the distribution entities. This ensures that the operations shared between them are the same operations. These interfaces can be put in the Joint Realization package (see Figure 6-13 on page 106).

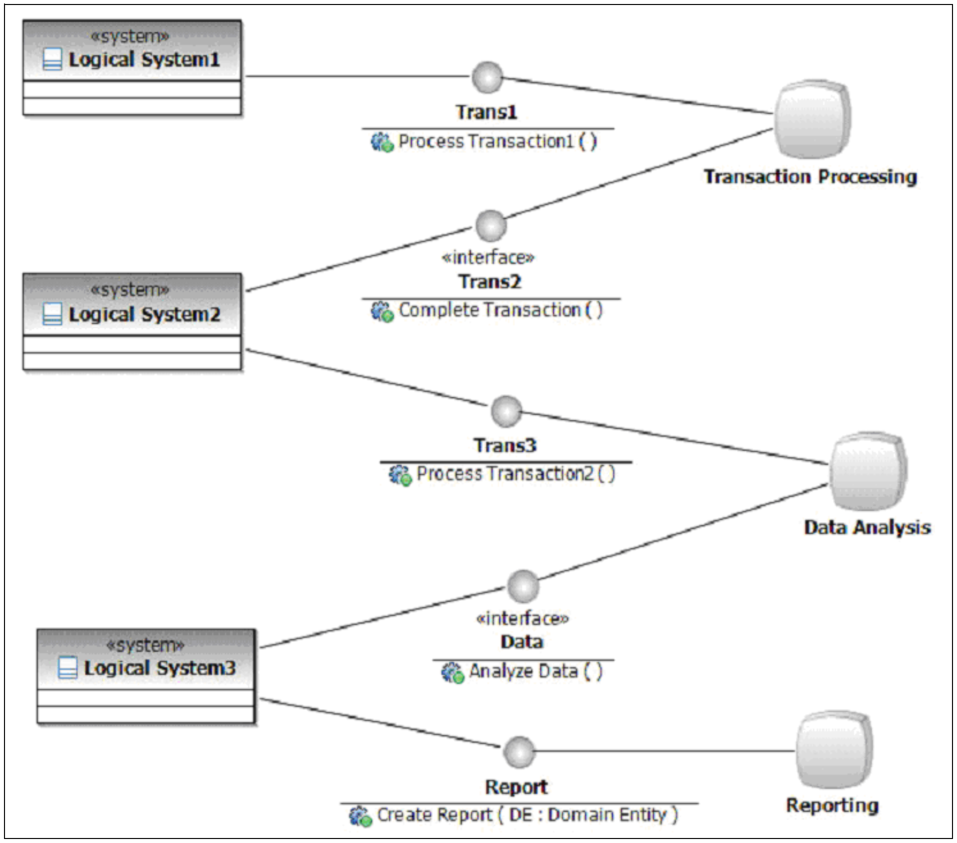


Figure 6-16 Joint realization diagram

Automation

Several parts of the model structure are manually created at this time. The Building Blocks template is an example of a pattern used multiple times to keep consistency in naming, as well as look and feel. A current effort is going on to discover such patterns and *automate* their creation programmatically within the MDSD plug-in. This might change some of the structure described above, but allow for better consistency and traceability.

Creating MDSD artifacts

Creating artifacts to capture the essence of the MDSD process involves a small number of diagrams. We include here instructions on how to draw them using IBM Rational Systems Developer.[2]

UML diagrams for systems modeling

There are only a few diagrams needed in UML to capture the essence of the MDSD process. The following sections assume that you have a Rational modeling tool and the MDSD profile. We guide you through the following tasks:

- Load the MDSD profile.
- Draw a context diagram.
- Draw two sequence diagrams for flowdown.
- Draw a Locality diagram.

Preparing the environment

IBM Rational Systems Developer is an Eclipse-based integration, design, and construction product that enables systems and software architects and developers to create applications that are optimized for C++ and Java™ SE. Rational Systems Developer also provides modeling capabilities supporting UML 2.0.

Rational Systems Developer is based on the Eclipse Workbench. If you are not already familiar with the Eclipse Workbench environment, take some time during this section to explore the environment.

You will configure the environment in preparation for this section. You will customize the way that UML connectors are displayed on diagrams to make the diagrams more readable. We do not have to see the multiplicity and roles information for this purpose, so we configure the environment so that they are not shown in the diagrams.

Because Rational Systems Developer is based on Eclipse, we have the ability to create and use plug-ins to provide additional features and functionality. In this section you are installing a plug-in that provides additional tools to support and enable model-driven systems development. When you install this plug-in, take some time to see what the effect was and consider how it can be a valuable capability to have.

[2] They can also be created using Rational Software Architect or Rational Software Modeler.

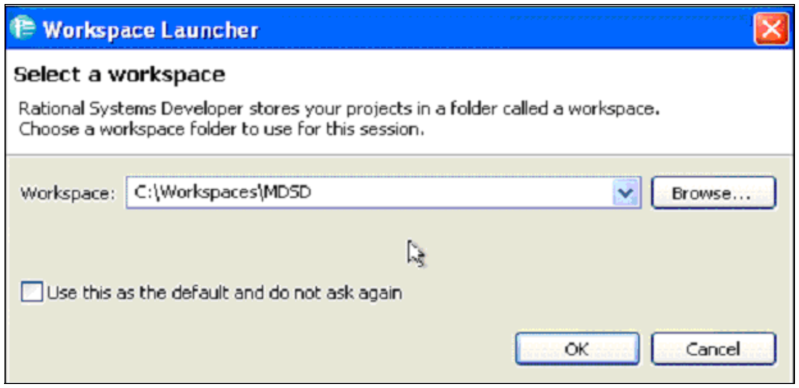
In this first section, you will:

- Explore and become familiar with the Workbench.
- Customize the Workbench to hide multiplicity and roles from UML diagrams.
- Install the MDSD plug-in to support model-driven systems development.

Preparing the Workbench

In this task you are customizing the environment so that multiplicity and role information is not displayed on the diagrams:

- Launch Rational Systems Developer[3] by selecting **Start** → **IBM Rational Systems Developer**.
- Create a new workspace by typing `C:\Workspaces\MDSD` into the Workspace field (replacing anything that might already be there) (Figure 6-17).



Rational Systems Developer stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Workspaces\MDSD

☐ Use this as the default and do not ask again

Figure 6-17 Workspace Launcher

- If it appears that RSD is hung, look for the Workspace Launcher dialog behind the RSD window.
- When the Workbench starts up, close the Welcome window if it is displayed (Figure 6-18).

[3] This example assumes Rational Systems Developer Version 7

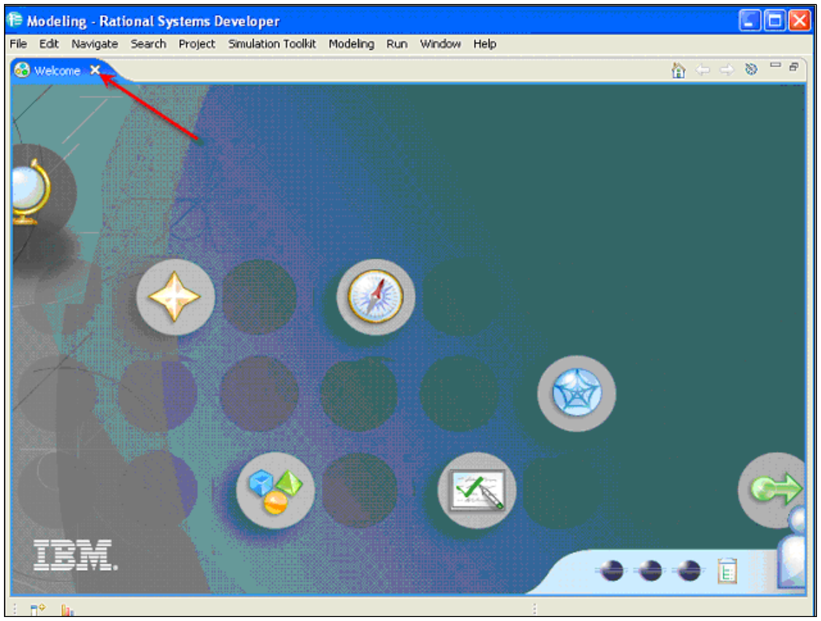
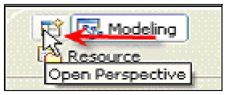

Modeling - Rational Systems Developer
File Edit Navigate Search Project Simulation Toolkit Modeling Run Window Help
Welcome
IBM.

Figure 6-18 Welcome window

Create a new UML Modeling Project

Follow these steps:

- Switch to the **Modeling** perspective using the **Open Perspective** icon:



- Select **File** → **New** → **Project**.
- Expand **Modeling**, select **UML Project**, and click **Next** (Figure 6-19).

Shortcut: To quickly get to the UML Project option in the Project Creation wizard, type `uml` in the filter field.

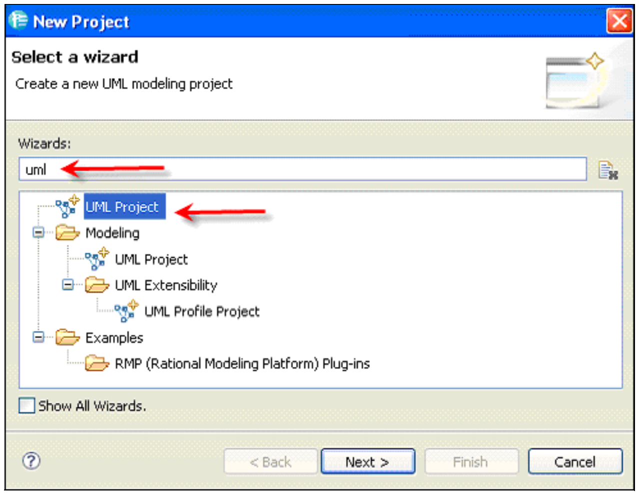


Figure 6-19 Create UML Project (1)

- Name the new project **Weather Tracking System**.
- Clear **Create new UML model in project** (you create a UML Model in the next section).
- Click **Finish** (Figure 6-20).

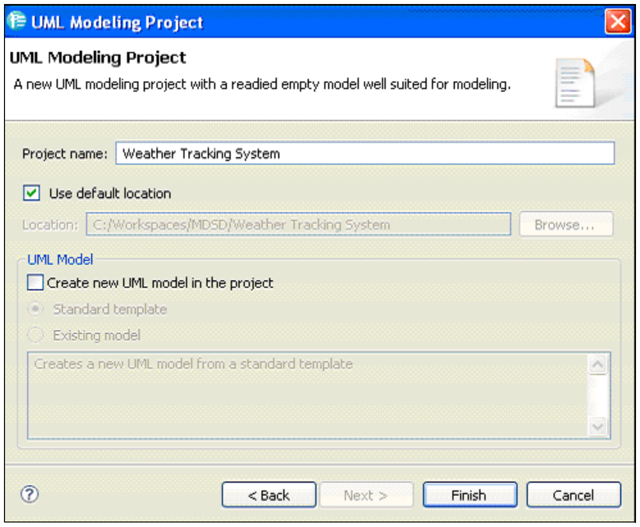

UML Modeling Project
UML Modeling Project
A new UML modeling project with a readied empty model well suited for modeling.
Project name: Weather Tracking System
Use default location
Location: C:/Workspaces/MDSD/Weather Tracking System
Browse...
UML Model
Create new UML model in the project
Standard template
Existing model
Creates a new UML model from a standard template
< Back
Next >
Finish
Cancel

Figure 6-20 Create UML Project (2)

- Turn off the display of multiplicity and roles:
  - Select **Window** → **Preferences**.
  - Expand **Modeling** → **Diagrams** → **Appearance**.
  - Select **Connectors**.
  - Clear **Show multiplicity** and **Show roles**.
  - Click **Apply** and **OK** to close the preferences window (Figure 6-21).

Shortcut: To locate the Connectors entry, type `connectors` in the filter field.

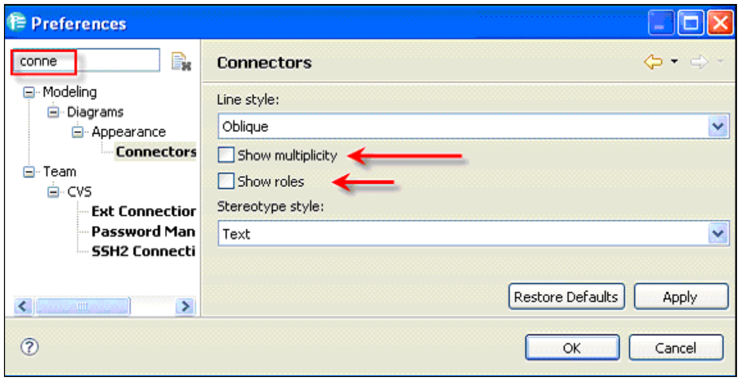
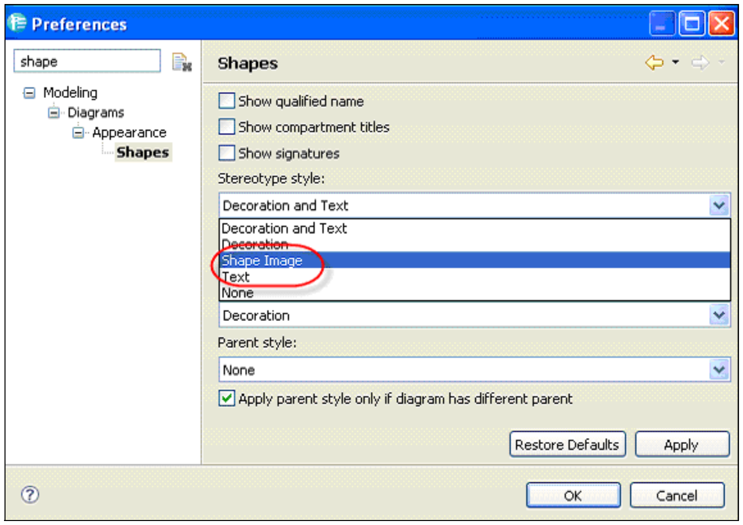


Figure 6-21 Preferences: Multiplicity and roles

- Turn on display of stereotype shapes:
  - Select **Window** → **Preferences**.
  - Type **shape** in the filter box.
  - Select **Stereotype Style** → **Shape Image**.
  - Click **OK** (Figure 6-22).



Show qualified name
Show compartment titles
Show signatures
Stereotype style:
Decoration and Text
Decoration and Text
Decoration
Shape Image
Text
None
Decoration
Parent style:
None
[✓] Apply parent style only if diagram has different parent

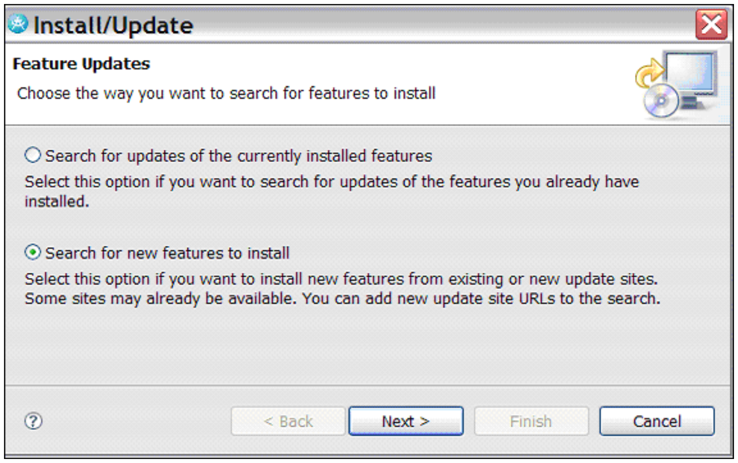
Figure 6-22 Preferences: Shape Image

Installing the MDSD plug-in

In this task you install a plug-in that enables model driven systems development in Rational Systems Developer:

- Obtain or locate the zip file **MDSD.zip**[4].
- Unzip the file to `C:\MDSD\InstallLocation`.
- From the Rational Systems Developer main menu, select **Help** → **Software Updates** → **Find and Install**.
- In the Install/Update dialog, select **Search for new features to install**, and click **Next** (Figure 6-23).

[4] Contact an IBM MDSD practitioner for the MDSD plug-in: Tim Bohn, `tbohn@us.ibm.com`


Install/Update

Feature Updates

Choose the way you want to search for features to install

( ) Search for updates of the currently installed features

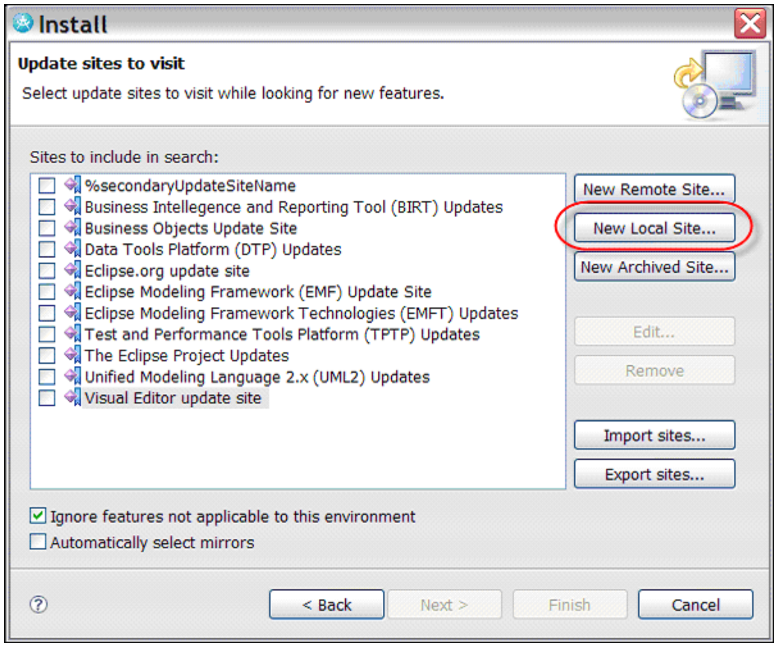
Select this option if you want to search for updates of the features you already have installed.

(●) Search for new features to install

Select this option if you want to install new features from existing or new update sites. Some sites may already be available. You can add new update site URLs to the search.

Figure 6-23 Install/Update dialog

- Click **New Local Site** (Figure 6-24).


Install

Update sites to visit

Select update sites to visit while looking for new features.

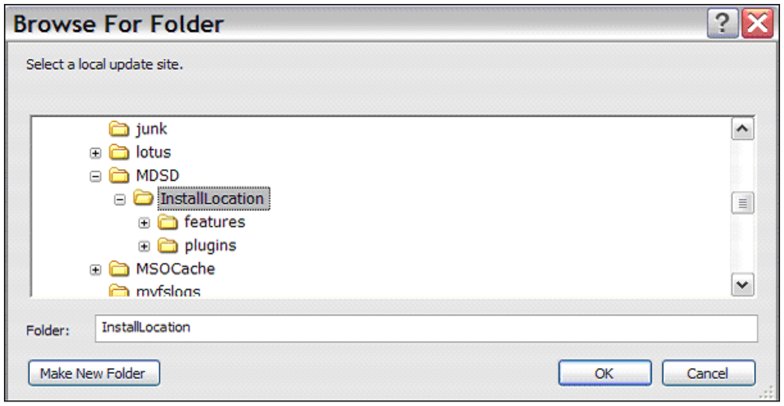
Sites to include in search:

- [ ] %secondaryUpdateSiteName
- [ ] Business Intellegence and Reporting Tool (BIRT) Updates
- [ ] Business Objects Update Site
- [ ] Data Tools Platform (DTP) Updates
- [ ] Eclipse.org update site
- [ ] Eclipse Modeling Framework (EMF) Update Site
- [ ] Eclipse Modeling Framework Technologies (EMFT) Updates
- [ ] Test and Performance Tools Platform (TPTP) Updates
- [ ] The Eclipse Project Updates
- [ ] Unified Modeling Language 2.x (UML2) Updates
- [ ] Visual Editor update site

- [x] Ignore features not applicable to this environment
- [ ] Automatically select mirrors

Figure 6-24 New local site

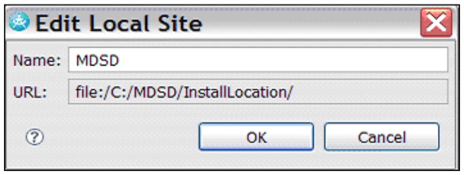
- Navigate to and select `C:\MDSD\InstallLocation`, then click **OK** (Figure 6-25).



Folder: InstallLocation

Figure 6-25 Browse to install location

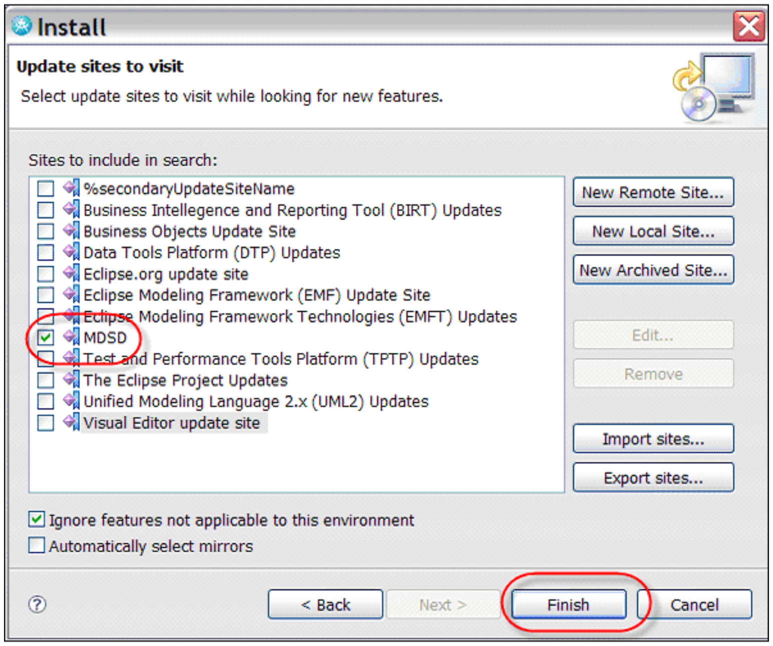
- Name the new site **MDSD** (Figure 6-26).



Name: MDSD
URL: file:/C:/MDSD/InstallLocation/

Figure 6-26 Edit Local Site

- Select **MDSD** as the only site, and click **Finish** (Figure 6-27).

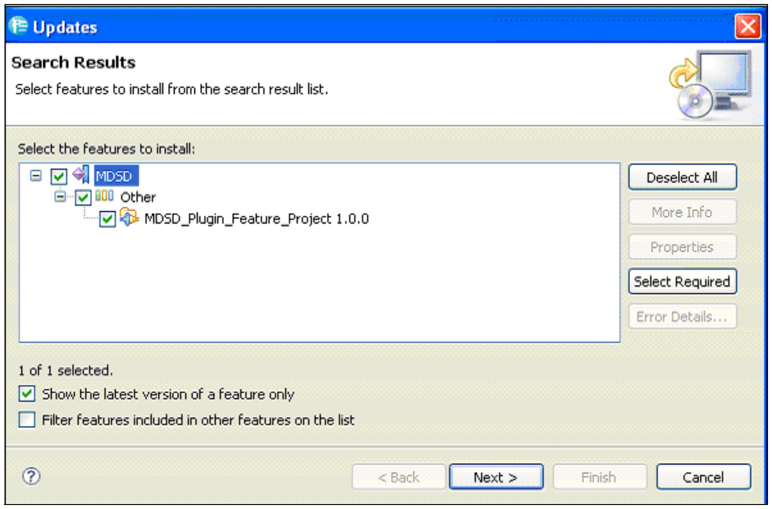


- [ ] %secondaryUpdateSiteName
- [ ] Business Intellegence and Reporting Tool (BIRT) Updates
- [ ] Business Objects Update Site
- [ ] Data Tools Platform (DTP) Updates
- [ ] Eclipse.org update site
- [ ] Eclipse Modeling Framework (EMF) Update Site
- [ ] Eclipse Modeling Framework Technologies (EMFT) Updates
- [✓] MDSD
- [ ] Test and Performance Tools Platform (TPTP) Updates
- [ ] The Eclipse Project Updates
- [ ] Unified Modeling Language 2.x (UML2) Updates
- [ ] Visual Editor update site

[✓] Ignore features not applicable to this environment
[ ] Automatically select mirrors

Figure 6-27 Select the MDSD update site

- Select **MDSD** and click **Next** (Figure 6-28).



[✓] MDSD
  [✓] Other
    [✓] MDSD_Plugin_Feature_Project 1.0.0

1 of 1 selected.
[✓] Show the latest version of a feature only
[ ] Filter features included in other features on the list

Figure 6-28 Select the MDSD feature to install

- Accept the license agreement and click **Next**.
- Accept the default installation location and click **Finish**.
- When prompted, restart Rational Systems Developer.

Modeling the system as a black box

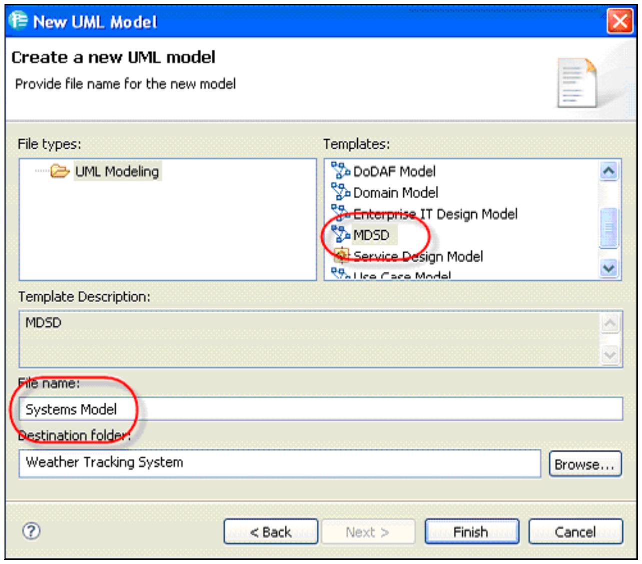
In this section you work at the highest level of abstraction, modeling the system as a black box. After completing this section, you will have created:

- A black-box context diagram identifying the system, I/O entities, and the actors
- A use case diagram identifying the various benefits (use cases) that the system provides to its stakeholders
- Sequence diagrams identifying the flow of events and operations required of the system

Create the system model

In this task you create the system model using the MDSD template:

- Create a new UML Model:
  - In the Project Explorer, right-click the **Weather Tracking System** project, and select **New** → **Other** → **UML Model**.
  - Click **Next**.
  - Select **Standard template**.
  - Select the **MDSD** template.
  - Name the model **Systems Model**.
  - Click **Finish** (Figure 6-29).


New UML Model

Create a new UML model

Provide file name for the new model

File types:

UML Modeling

Templates:

DoDAF Model
Domain Model
Enterprise IT Design Model
MDSD
Service Design Model
Use Case Model

Template Description:

MDSD

File name:

Systems Model

Destination folder:

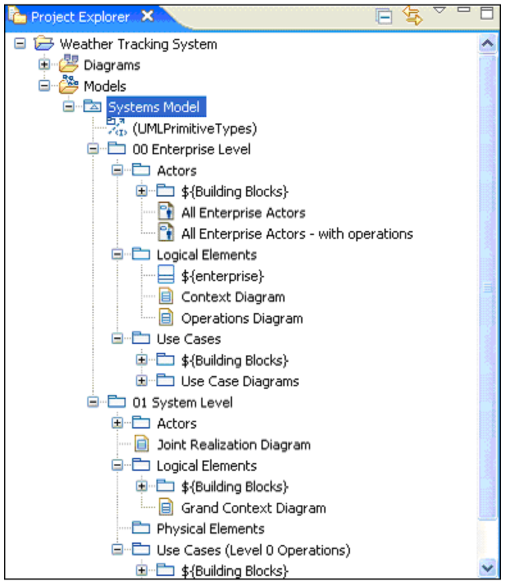
Weather Tracking System

Figure 6-29 Create the Systems Model

- Notice the model structure given by the template:
  - In the Project Explorer, select the **Systems Model**.
  - In the Properties view, select the **Profiles** tab.
  - Notice that the MDSD Profile has already been applied to the model.

    Note: You can also look at the `Systems Model.emx` tab in the editor under the Details tab.

  - In the Project Explorer, notice that several artifacts have already been created for you. These were all provided as part of the **MDSD model template** that became available to you when the **MDSD Plug-in** was installed (Figure 6-30).



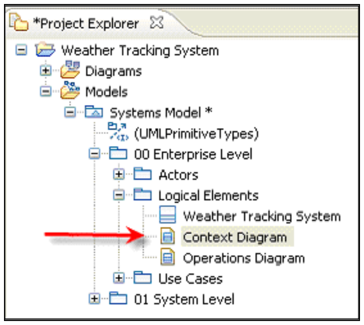
```
Project Explorer
Weather Tracking System
  Diagrams
  Models
    Systems Model
      (UMLPrimitiveTypes)
      00 Enterprise Level
        Actors
          ${Building Blocks}
          All Enterprise Actors
          All Enterprise Actors - with operations
        Logical Elements
          ${enterprise}
          Context Diagram
          Operations Diagram
        Use Cases
          ${Building Blocks}
          Use Case Diagrams
      01 System Level
        Actors
        Joint Realization Diagram
        Logical Elements
          ${Building Blocks}
          Grand Context Diagram
        Physical Elements
        Use Cases (Level 0 Operations)
          ${Building Blocks}
```

Figure 6-30 Project Explorer with the Systems Model expanded

Create the context diagram

In this task you create the system level black-box context diagram:

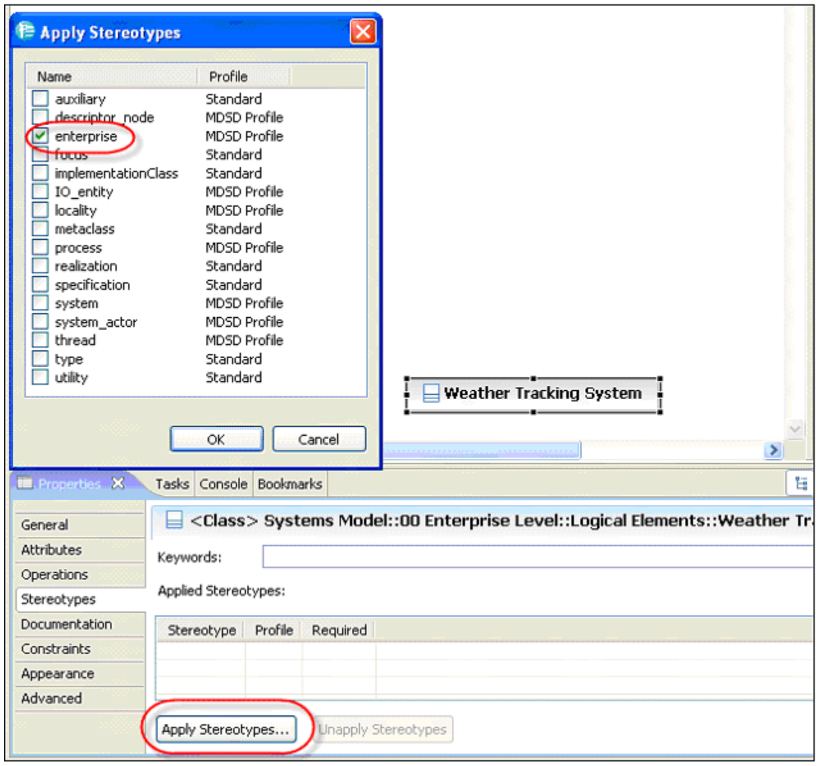
- Expand **Systems Model** → **00 Enterprise Level** → **Logical Elements**.
- Select the **${enterprise}** class and rename it to **Weather Tracking System**:
  - In the Project Explorer, right-click the **${enterprise}** class and select **Rename**.
  - Type **Weather Tracking System** and press **Enter**.
- Open the **Context Diagram** to see that this entity is there, to begin creating the context diagram (Figure 6-31).



```
*Project Explorer
Weather Tracking System
  Diagrams
  Models
    Systems Model *
      (UMLPrimitiveTypes)
      00 Enterprise Level
        Actors
        Logical Elements
          Weather Tracking System
          Context Diagram
          Operations Diagram
        Use Cases
      01 System Level
```

Figure 6-31 Open the context diagram

- Stereotype the Weather Tracking System as <<enterprise>>:
  - Select **Weather Tracking System** in the diagram.
  - In the Properties view, select the **Stereotypes** tab.
  - Click **Apply Stereotypes**.
  - In the Stereotypes dialog, select **enterprise** from the **MDSD Profile**.
  - Click **OK** (Figure 6-32).



Name	Profile
☐ auxiliary	Standard
☐ descriptor_node	MDSD Profile
✓ enterprise	MDSD Profile
☐ focus	Standard
☐ implementationClass	Standard
☐ IO_entity	MDSD Profile
☐ locality	MDSD Profile
☐ metaclass	Standard
☐ process	MDSD Profile
☐ realization	Standard
☐ specification	Standard
☐ system	MDSD Profile
☐ system_actor	MDSD Profile
☐ thread	MDSD Profile
☐ type	Standard
☐ utility	Standard

<Class> Systems Model::00 Enterprise Level::Logical Elements::Weather Tr

Keywords:

Applied Stereotypes:

Stereotype	Profile	Required

Figure 6-32 Apply stereotype

- Create actors:
  - Expand **00 Enterprise Level** → **Actors** → **${Building Blocks}**.
  - Right-click **${actor.name}**. and select **Copy**.
  - Right-click **Actors** and select **Paste**.
  - Right-click the new **${actor.name}** folder that you pasted and select **Edit** → **Find/Replace** (or press Ctrl-f).

– Put **${actor.name}** in the Search string field and click **Replace** (Figure 6-33).

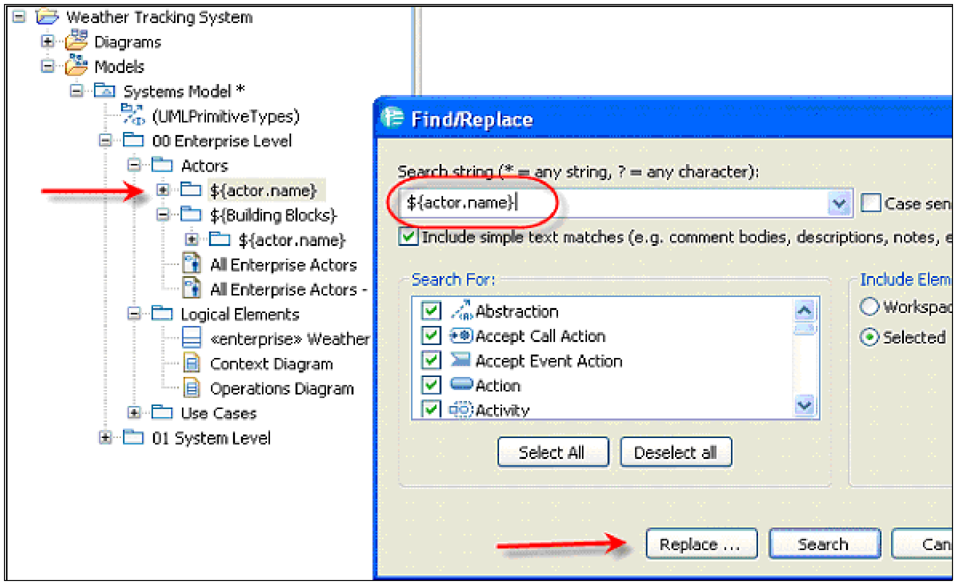


Figure 6-33 Copy $(actor.name) and find/replace

– Type **Local Forecaster** in the With field and click **Replace All** (Figure 6-34).

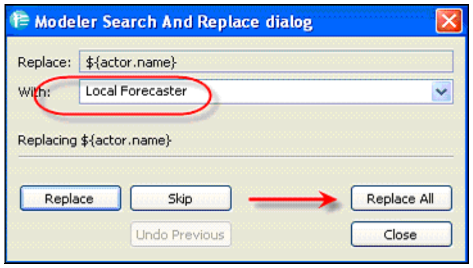


Figure 6-34 Replace the actor name

– Repeat these steps two more times to create actors named **Alert System** and **Online User**.

- Add the actors to the context diagram:
  – In the Project Explorer, expand the **Actors** package.
  – Expand the **Alert System** package.

- Drag and drop the **Alert System** actor into the context diagram (Figure 6-35).
- Repeat these steps for the **Local Forecaster** and **Online User** actors.

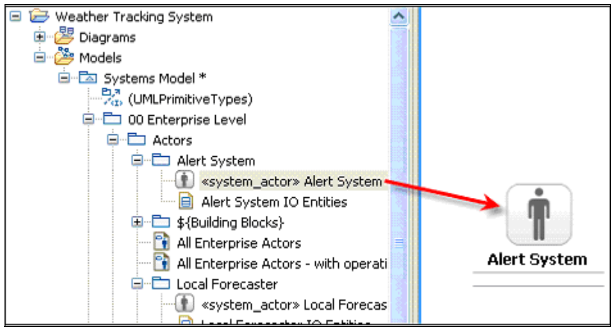


Figure 6-35 Drag an actor into the context diagram

- ► Create associations between the actors and the Weather Tracking System:
  - In the context diagram hover over the **Online User** actor.
  - Grab the handle and drag it on top of the **Weather Tracking System** class.
  - Drop it on the class and select **Create Bidirectional Association** from the pop-up dialog.
  - Perform the steps also for the **Alert System** and the **Local Forecaster** actors (Figure 6-36).

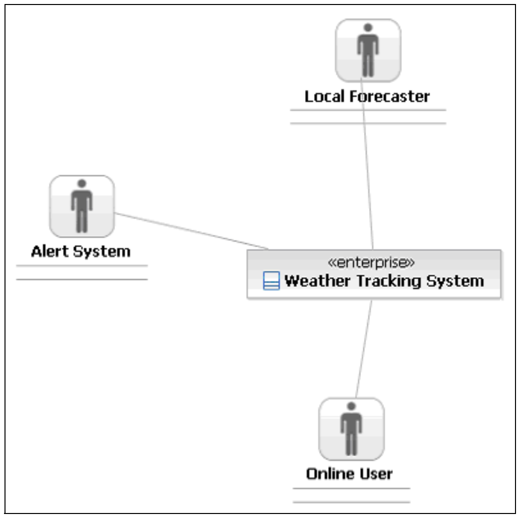


Figure 6-36 Context diagram: Weather Tracking System with actors

Note: You can also use the tool palette to create associations.

- Create use cases (this sequence is similar to creating actors):
  - Expand **00 Enterprise Level** → **Use Cases** → **${Building Blocks}**.
  - Right-click **${use.case}** and select **Copy**.
  - Right-click the **Use Cases** folder and select **Paste**.
  - Right-click the new **${use.case}** folder that you pasted and select **Edit** → **Find/Replace** (or press Ctrl-f).
  - Put **${use.case}** in the Search string field and click **Replace**.
  - Type **RegisterForAlert** in the With field and click **Replace All**.
  - Repeat these steps to create use cases named **GetLocalForecast** and **GetRawWeatherData**.

Note: Use cases are not placed into the context diagram. Relationships between actors and the use case are created in the use case diagram.

- Create associations between the use cases and the actors:
  - In the Project Explorer, expand the **Use Case Diagrams** folder.
  - Open (double-click) the **Use Case Diagram**.
  - In the Project Explorer, expand the **GetRawWeatherData** folder.
  - Select the **GetRawWeatherData** use case and drop it into the use case diagram (Figure 6-37).

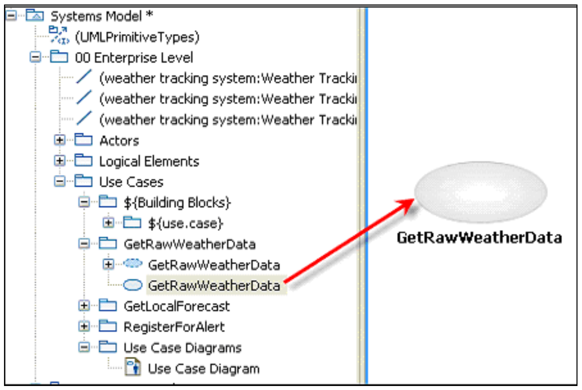


Figure 6-37 Drag use case into the use case diagram

- Repeat these steps for the use cases **RegisterForAlert** and **GetLocalForecast**.
- In the Project Explorer, expand the **Actors** folder.
- Expand the **Alert System** folder.
- Select the **Alert System** actor and drop it into the diagram.
- Repeat the last two steps for the actors **Online User** and **Local Forecaster**.
- In the Palette, select **Association**.
- Drag the mouse from **Online User** to **GetLocalForecast** to create an association (Figure 6-38).

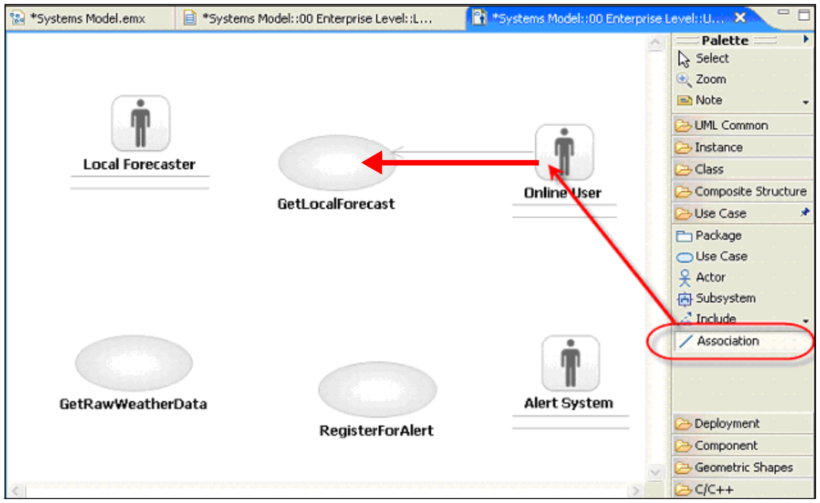


Figure 6-38 Create an association in the use case diagram

- Create an association between **Alert System** and **RegisterForAlert**.
- Create an association between **Local Forecaster** and **GetRawWeatherData**.
- The use case diagram is shown in Figure 6-39.

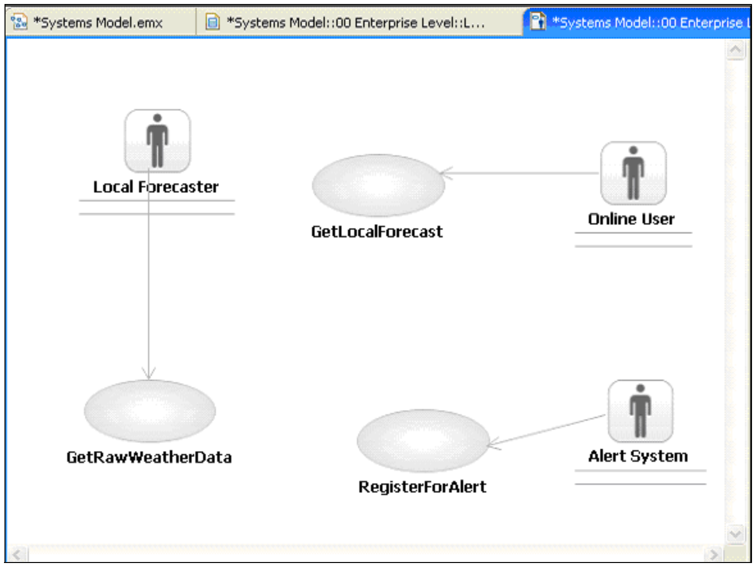


Figure 6-39 Use case diagram

- Create I/O entities:
  - Open the Context Diagram containing the **Weather Tracking System** enterprise and the actors, if it is not already opened.
  - Hover over any white space on the diagram and select the **Add Stereotyped Class** icon. In the popup dialog, select **Create <<IO_entity>> Class** (Figure 6-40).

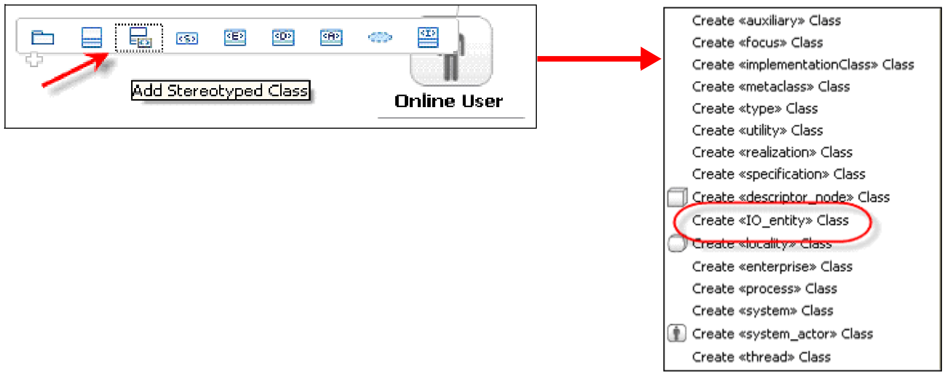


Figure 6-40 Create I/O entity class

– Name the class **Location**.
– Repeat these steps to create IO_entity classes **WeatherData** and **HazardousWeatherAlert**.
– In the Project Explorer, expand the **00 Enterprise Level** folder.
– Select all three IO_entity classes.
– Drag and drop the three classes into the **Logical Elements** package to relocate them.
– Figure 6-41 shows the context diagram.

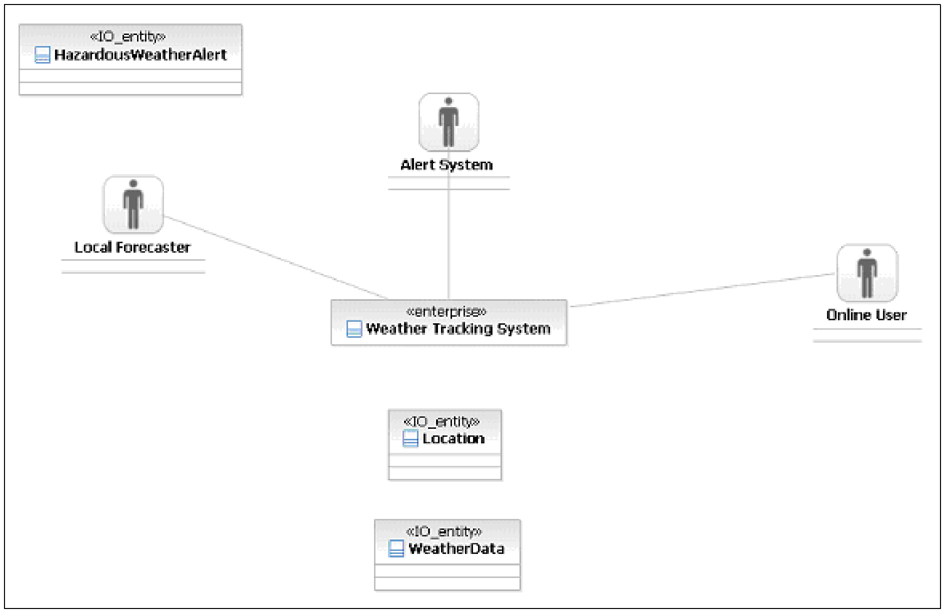
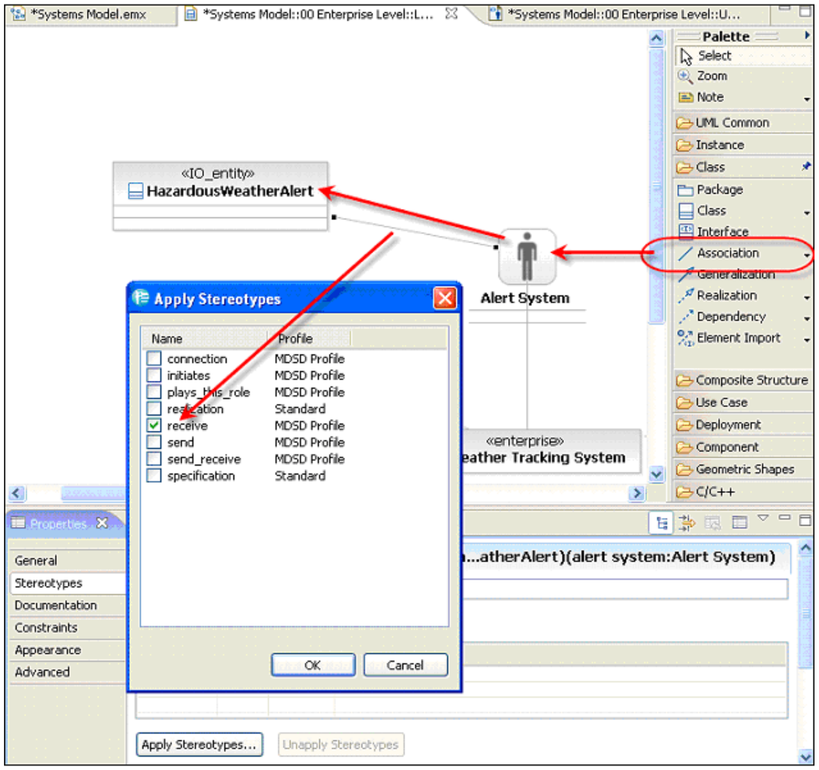


Figure 6-41 Context diagram with I/O entities

- Create associations for the I/O entity classes:
  – In the context diagram select the **Association** element from the Palette.
  – Drag the mouse from the **Alert System** actor to the **HazardousWeatherAlert** I/O_entity.
  – In the Properties view for the new association select the **Stereotypes** tab.
  – Click **Apply Stereotypes.**
  – Select **<<receive>>** in the Apply Stereotypes dialog.
  – Click **OK** (Figure 6-42).



	Name	Profile
☐	connection	MDSD Profile
☐	initiates	MDSD Profile
☐	plays_this_role	MDSD Profile
☐	realization	Standard
✓	receive	MDSD Profile
☐	send	MDSD Profile
☐	send_receive	MDSD Profile
☐	specification	Standard

Figure 6-42 Creating associations with I/O entities

– Repeat these steps to create the associations listed inTable 6-1.

Table 6-1 Associations between actors and I/O entities

Actor	I/O Entity	Association Stereotype
Local Forecaster	Location	<<send>>
Local Forecaster	WeatherData	<<receive>>
Online User	Location	<<send>>
Online User	WeatherData	<<receive>>

- Save the work (Ctrl+s).
- The context diagram is shown in Figure 6-43.

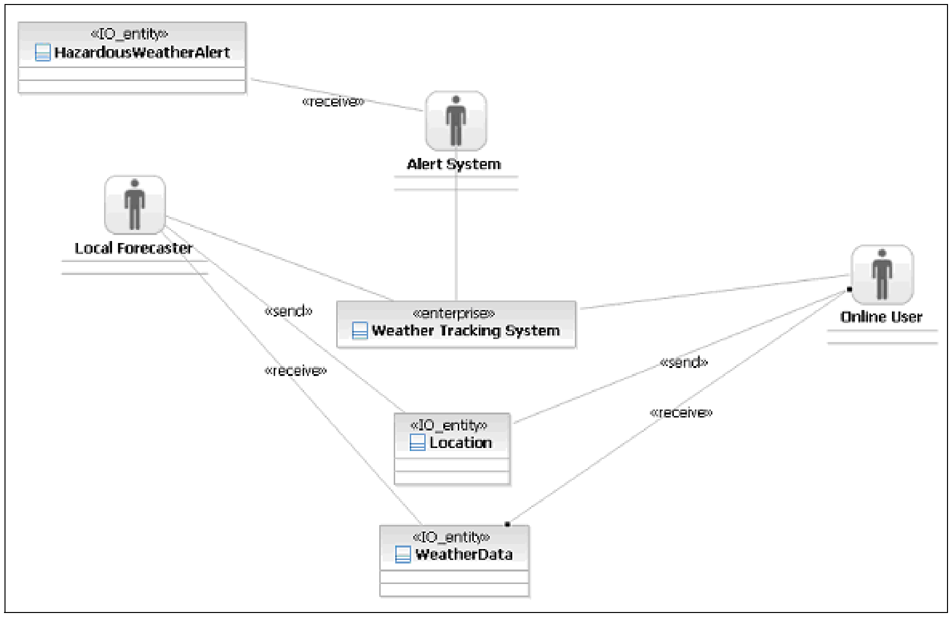


Figure 6-43 Context diagram with associations of I/O entities

Create black-box sequence diagram

In this task you create a black-box sequence diagram for the **GetLocalForecast** use case:

- Open the black-box sequence diagram for the **GetLocalForecast** use case.
  - In the Project Explorer, expand **00 Enterprise Level** → **Use Cases** → **GetLocalForecast**.
  - Expand the collaboration **GetLocalForecast**, then expand the interaction **GetLocalForecast** (Figure 6-44).

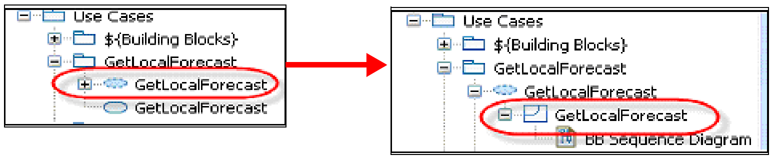


Figure 6-44 Expand a use case, collaboration, and interaction

  - Open (double-click) the **BB Sequence Diagram**.
- Add the participants to the sequence diagram:
  - Expand **00 Enterprise Level** → **Actors** → **LocalForecaster**.

– Drag the **LocalForecaster** system actor into the **BB Sequence Diagram** (Figure 6-45).

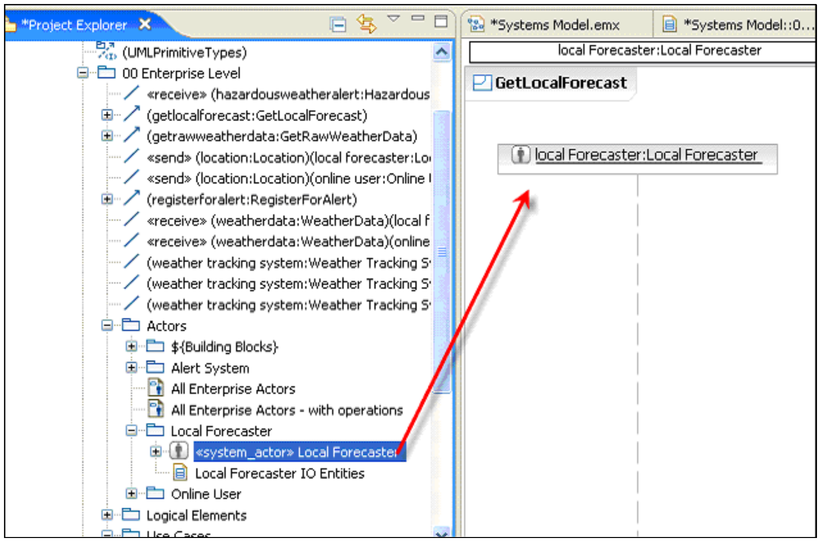

*Project Explorer
*Systems Model.emx
*Systems Model::0...
(UMLPrimitiveTypes)
00 Enterprise Level
«receive» (hazardousweatheralert:Hazardous
(getlocalforecast:GetLocalForecast)
(getrawweatherdata:GetRawWeatherData)
«send» (location:Location)(local forecaster:Lo
«send» (location:Location)(online user:Online
(registerforalert:RegisterForAlert)
«receive» (weatherdata:WeatherData)(local f
«receive» (weatherdata:WeatherData)(online
(weather tracking system:Weather Tracking S
(weather tracking system:Weather Tracking S
(weather tracking system:Weather Tracking S
Actors
${Building Blocks}
Alert System
All Enterprise Actors
All Enterprise Actors - with operations
Local Forecaster
«system_actor» Local Forecaster
Local Forecaster IO Entities
Online User
Logical Elements
local Forecaster:Local Forecaster
GetLocalForecast
local Forecaster:Local Forecaster

Figure 6-45   Drag actor into sequence diagram

– Expand **00 Enterprise Level** → **Logical Elements**.

– Drag **Weather Tracking System** into the **BB Sequence Diagram** (Figure 6-46).

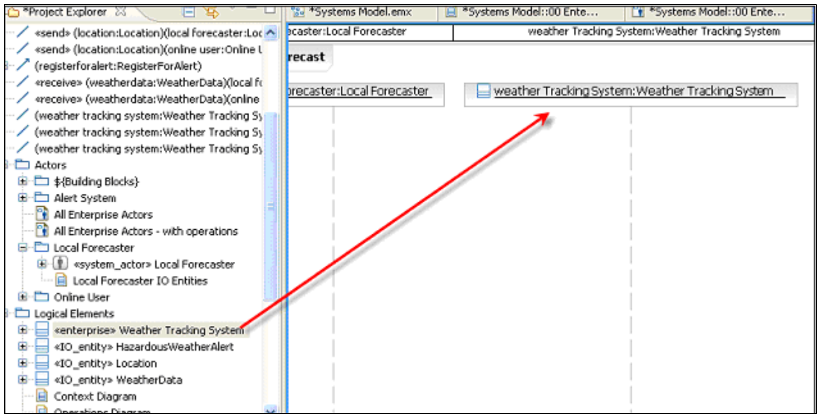

*Project Explorer
*Systems Model.emx
*Systems Model::00 Ente...
*Systems Model::00 Ente...
«send» (location:Location)(local forecaster:Loc
«send» (location:Location)(online user:Online L
(registerforalert:RegisterForAlert)
«receive» (weatherdata:WeatherData)(local fo
«receive» (weatherdata:WeatherData)(online
(weather tracking system:Weather Tracking Sy
(weather tracking system:Weather Tracking Sy
(weather tracking system:Weather Tracking Sy
Actors
${Building Blocks}
Alert System
All Enterprise Actors
All Enterprise Actors - with operations
Local Forecaster
«system_actor» Local Forecaster
Local Forecaster IO Entities
Online User
Logical Elements
«enterprise» Weather Tracking System
«IO_entity» HazardousWeatherAlert
«IO_entity» Location
«IO_entity» WeatherData
Context Diagram
caster:Local Forecaster
weather Tracking System:Weather Tracking System
recast
recaster:Local Forecaster
weather Tracking System:Weather Tracking System

Figure 6-46   Drag system into the sequence diagram

- Create a message from the LocalForecaster actor to the Weather Tracking System:
  - In the sequence diagram drawing surface, hover over the lifeline of the **LocalForecaster** actor.
  - Grab the displayed handle and drop it on the lifeline of the **Weather Tracking System** (Figure 6-47).

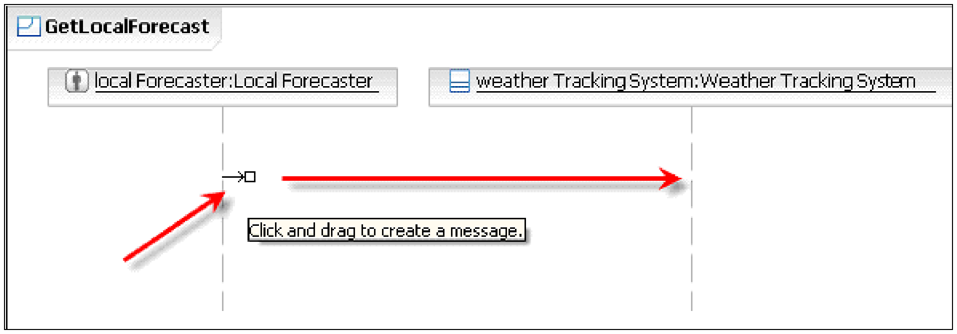

GetLocalForecast
local Forecaster:Local Forecaster
weather Tracking System:Weather Tracking System
Click and drag to create a message.

Figure 6-47 Create a message in the sequence diagram (1)

  - Select **Create Message** (Figure 6-48).

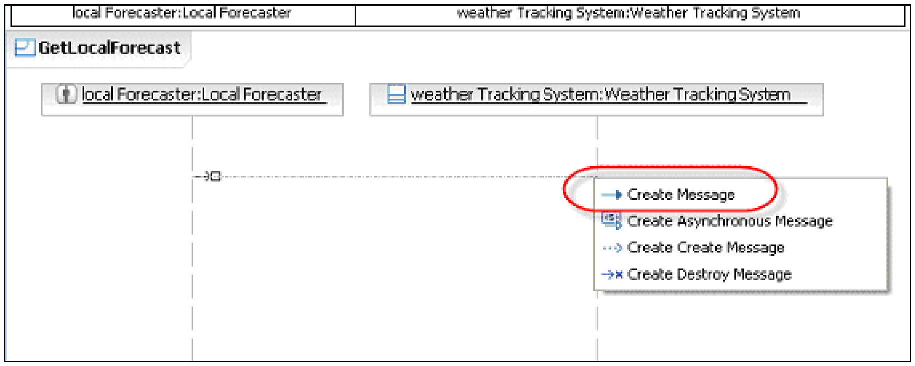
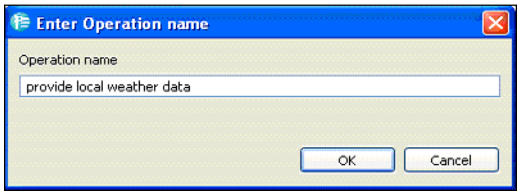

local Forecaster:Local Forecaster
weather Tracking System:Weather Tracking System
GetLocalForecast
local Forecaster:Local Forecaster
weather Tracking System:Weather Tracking System
Create Message
Create Asynchronous Message
Create Create Message
Create Destroy Message

Figure 6-48 Create a message in the sequence diagram (2)

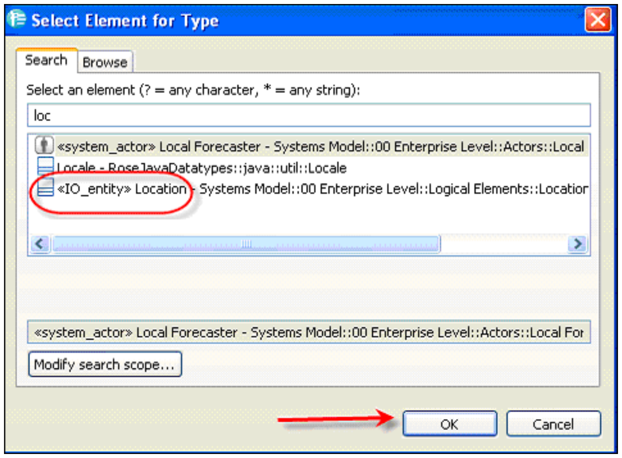
– Label the operation **provide local weather data** (Figure 6-49).



provide local weather data

Figure 6-49 Name the operation

- Add parameters to the message:
  - In the Project Explorer, expand **00 Enterprise Level** → **Logical Elements** → **Weather Tracking System**.
  - Right-click **provide local weather data** and select **Add UML** → **Parameter**.
  - In the Properties view change the name of the parameter to **location**.
  - Click **Select type** and select the **I/O Entity Location** as the type (Figure 6-50).



loc

«system_actor» Local Forecaster - Systems Model::00 Enterprise Level::Actors::Local
Locale - RoseJavaDatatypes::java::util::Locale
«IO_entity» Location - Systems Model::00 Enterprise Level::Logical Elements::Location

«system_actor» Local Forecaster - Systems Model::00 Enterprise Level::Actors::Local For

Figure 6-50 Specify the parameter type

  - Repeat these steps to add another parameter called **weatherData** of type **WeatherData**.
  - In the Properties view for the **weatherData** parameter, select the **General** tab, and change the direction to **Out**.
  - Save the work (Ctrl+s).

– The sequence diagram is shown in Figure 6-51.

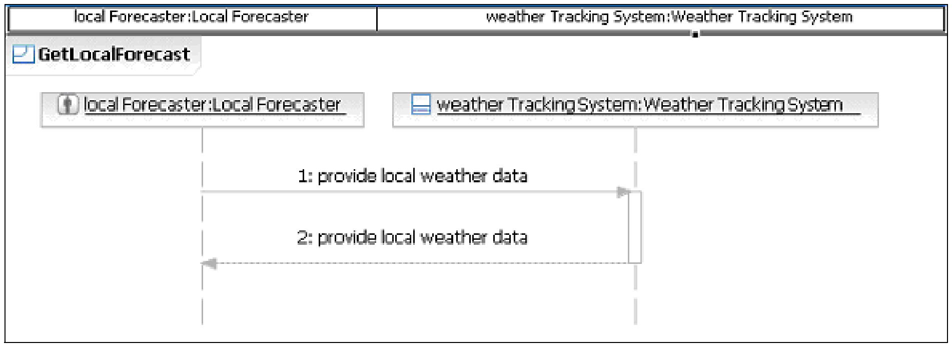

local Forecaster:Local Forecaster
weather Tracking System:Weather Tracking System
GetLocalForecast
local Forecaster:Local Forecaster
weather Tracking System:Weather Tracking System
1: provide local weather data
2: provide local weather data

Figure 6-51 Black-box sequence diagram

Summary

In this section you worked at the highest level of abstraction, modeling the system as a black box. Through the course of this section, you have created:

- A black-box context diagram identifying the system, I/O entities, and the actors
- A use case diagram identifying the various benefits (use cases) that the system provides to its stakeholders
- A black-box sequence diagram identifying the flow of events and operations required of the system

Modeling the system at level 1

In this section you work at the next level of abstraction, modeling the system as a white box. When you have completed this section, you have created:

- A white-box sequence diagram identifying the flow of events for the provide local weather data operation.
- New systems as identified during this white-box analysis.

Identify systems that will collaborate at L1

In this task you identify the systems that have to collaborate to realize the level 1 use cases:

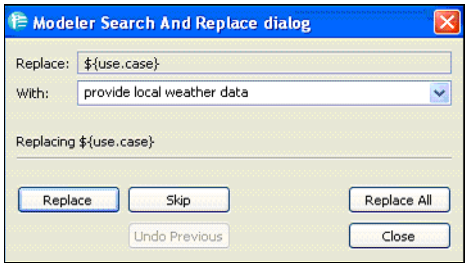
- In the Project Explorer, expand the **01 System Level** → **Logical Elements** → **${Building Blocks}**.
- Right-click the **${system.name}** folder and select **Copy**.
- Right-click the **Logical Elements** folder and select **Paste**.

- Right-click the new **${system.name}** folder that you pasted and select **Edit** → **Find/Replace** (or press Ctrl-f).
- Type **${system.name}** in the Search string field and click **Replace**.
- Type **DopplerController** in the With field and click **Replace All**.
- Repeat these steps to create the systems **GroundStation**, **WeatherTrackingController**, **WeatherTrackingProcessor** and **WeatherTrackingUI**.

Realize a system operation

In this task you take one of the candidate system operations (provide local weather data operation) identified in the last task and realize it as a use case at the system level:

- In the Project Explorer, expand **01 System Level** → **Use Cases (Level 0 Operations)**.
- Copy the **{$use.case}** building block and paste it in the **Use Cases (Level 0 Operations)** package
- Use the Find/Replace feature to change the name to **provide local weather data** (Figure 6-52).



Replace: ${use.case}
With: provide local weather data

Replacing ${use.case}

Figure 6-52 Rename the use case

- In the Project Explorer, expand the **provide local weather data** package, then expand the **provide local weather data collaboration**, and then expand the **provide local weather data interaction** (Figure 6-53).

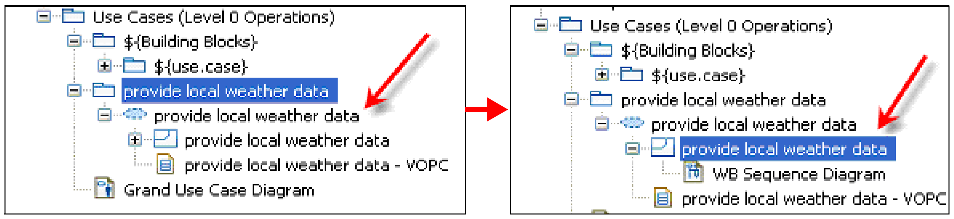
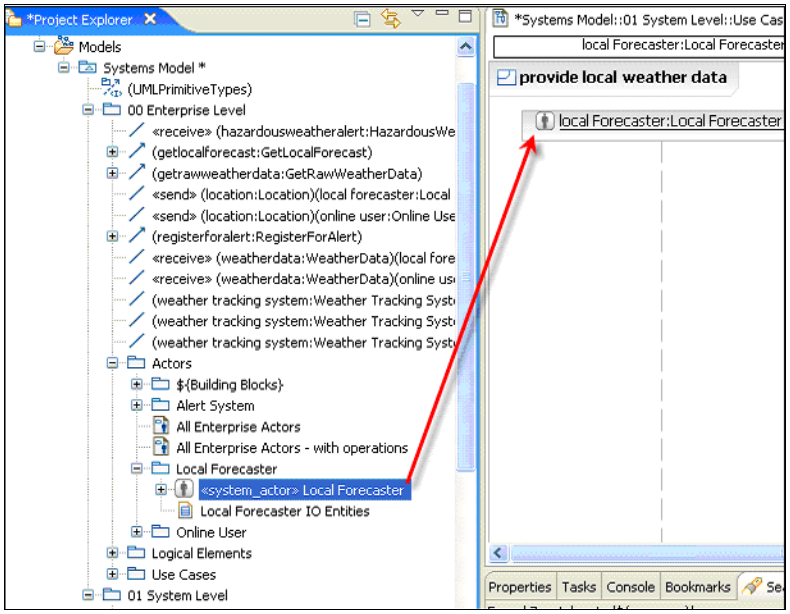

Use Cases (Level 0 Operations)
${Building Blocks}
${use.case}
provide local weather data
provide local weather data
provide local weather data
provide local weather data - VOPC
Grand Use Case Diagram
Use Cases (Level 0 Operations)
${Building Blocks}
${use.case}
provide local weather data
provide local weather data
provide local weather data
WB Sequence Diagram
provide local weather data - VOPC

Figure 6-53 Expand the use case, collaboration, and interaction

- Open the **WB Sequence Diagram**.
- Add the participants to the sequence diagram:
  - Expand **00 Enterprise Level** → **Actors** → **LocalForecaster**, and drag and drop the **LocalForecaster** system actor into this sequence diagram (Figure 6-54).



*Project Explorer
Models
Systems Model *
(UMLPrimitiveTypes)
00 Enterprise Level
«receive» (hazardousweatheralert:HazardousWe
(getlocalforecast:GetLocalForecast)
(getrawweatherdata:GetRawWeatherData)
«send» (location:Location)(local forecaster:Local
«send» (location:Location)(online user:Online Use
(registerforalert:RegisterForAlert)
«receive» (weatherdata:WeatherData)(local fore
«receive» (weatherdata:WeatherData)(online us
(weather tracking system:Weather Tracking Syst
(weather tracking system:Weather Tracking Syst
(weather tracking system:Weather Tracking Syst
Actors
${Building Blocks}
Alert System
All Enterprise Actors
All Enterprise Actors - with operations
Local Forecaster
«system_actor» Local Forecaster
Local Forecaster IO Entities
Online User
Logical Elements
Use Cases
01 System Level
*Systems Model::01 System Level::Use Cas
local Forecaster:Local Forecaster
provide local weather data
local Forecaster:Local Forecaster
Properties
Tasks
Console
Bookmarks
Se

Figure 6-54 Drag actor into the sequence diagram

  - Expand **01 System Level** → **Logical Elements**:
    - Drag and drop the **WeatherTrackingUI** system into the sequence diagram.
    - Drag and drop the **WeatherTrackingController** system into the sequence diagram.
    - Drag and drop the **DopplerController** system into the sequence diagram.
    - Drag and drop the **GroundStation** system into the sequence diagram
    - Drag and drop the **WeatherTrackingProcessor** system into the sequence diagram.
  - Save the work (Ctrl+s).

- Add messages to the sequence diagram:
  - In the sequence diagram hover over the **LocalForecaster** lifeline to grab the handle and drop it on the **WeatherTrackingUI** lifeline.
  - Select **Create Message** (Figure 6-55).

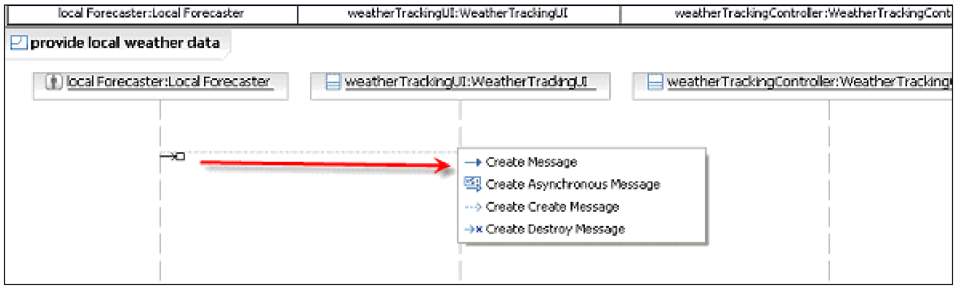

local Forecaster:Local Forecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController:WeatherTrackingCont
provide local weather data
local Forecaster:Local Forecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController:WeatherTracking
Create Message
Create Asynchronous Message
Create Create Message
Create Destroy Message

Figure 6-55 Create a message in the sequence diagram (1)

  - Name the operation **provide local weather data** (Figure 6-56).

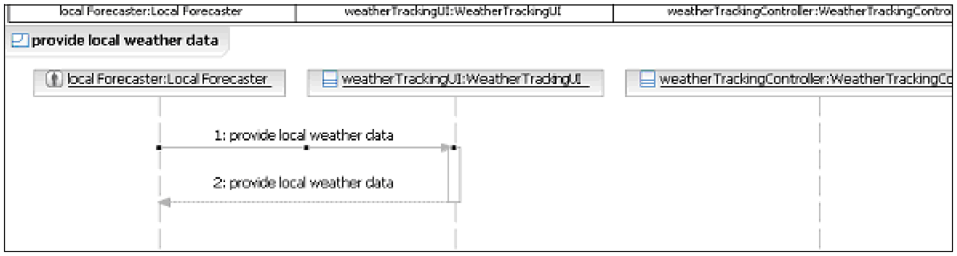

local Forecaster:Local Forecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController:WeatherTrackingContro
provide local weather data
local Forecaster:Local Forecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController:WeatherTrackingCo
1: provide local weather data
2: provide local weather data

Figure 6-56 Create a message in the sequence diagram (2)

  - Hover over the **WeatherTrackingUI** lifeline inside of the area spanned by the message just added.
  - Drag the handle from the **WeatherTrackingUI** lifeline to the **WeatherTrackingController** lifeline to create a message.
  - Name the operation **provide local weather data** (Figure 6-57).

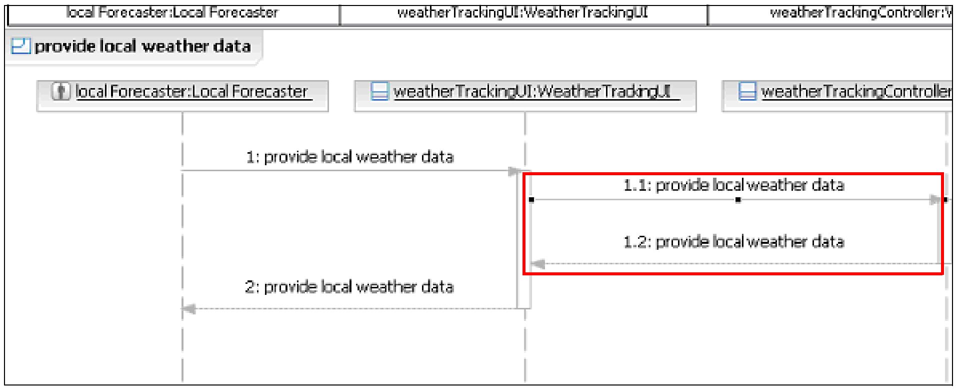

localForecaster:LocalForecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController:W
provide local weather data
localForecaster:LocalForecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController
1: provide local weather data
1.1: provide local weather data
1.2: provide local weather data
2: provide local weather data

Figure 6-57 Create another message in the sequence diagram

– Repeat these steps to create the messages listed in Table 6-2.

Table 6-2 Messages in the sequence diagram

From	To	Message
LocalForecaster	WeatherTrackingUI	provide local weather data
WeatherTrackingUI	WeatherTrackingController	provide local weather data
WeatherTrackingController	DopplerController	provide doppler data
WeatherTrackingController	GroundStation	provide satellite data
WeatherTrackingController	WeatherTrackingProcessor	combine doppler and satellite data

► Save the work (Ctrl+s).

- The resulting sequence diagram is shown in Figure 6-58.

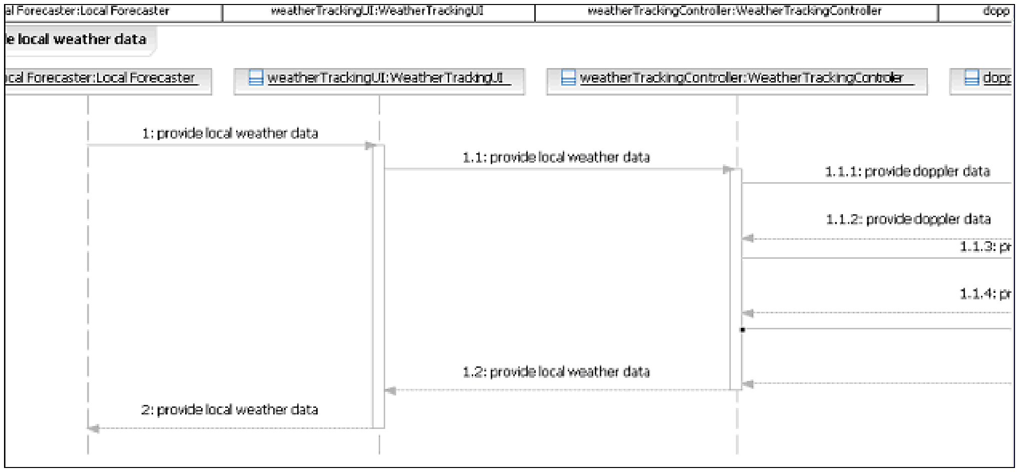

al Forecaster:Local Forecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController:WeatherTrackingController
dopp
e local weather data
cal Forecaster:Local Forecaster
weatherTrackingUI:WeatherTrackingUI
weatherTrackingController:WeatherTrackingController
dopp
1: provide local weather data
1.1: provide local weather data
1.1.1: provide doppler data
1.1.2: provide doppler data
1.1.3: pr
1.1.4: pr
1.2: provide local weather data
2: provide local weather data

Figure 6-58 White-box sequence diagram with messages

Creating a localities diagram

In this task you create a localities diagram:

- In the Project Explorer, expand **Systems Model** → **01 System Level**.
- Right-click on **Physical Elements** and select **Add UML** → **Package**.
- Name the package **Localities**.
- In the Project Explorer, change the name of the diagram created in the **Localities** package from **Main** to **Localities**.
- Hover over the drawing surface of the **Localities** diagram (it should have opened when you created the **Localities** package) and select the **Add stereotyped class** icon. Select **Create <<locality>> Class** (Figure 6-59).

  Use this method to create the following classes:

  – DopplerControlCenter1
  – DopplerRadarStation1
  – DopplerRadarStation2
  – DopplerRadarStation3
  – GroundStation1
  – WeatherTrackingSystem1

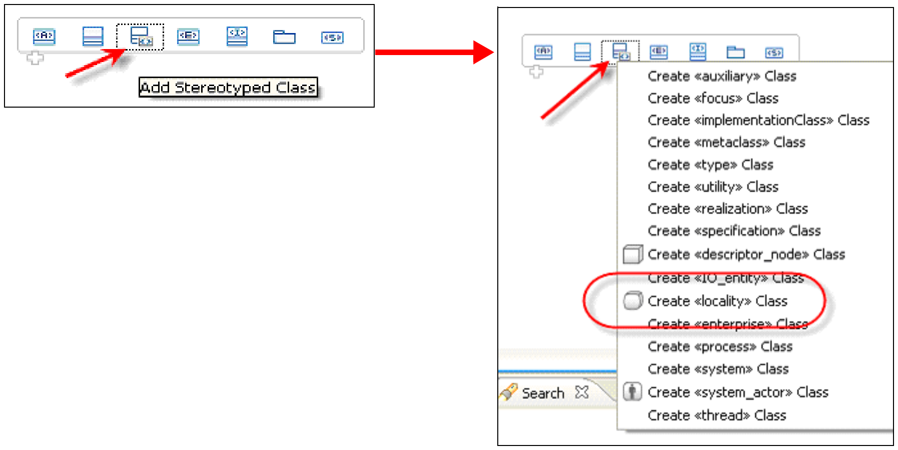


Figure 6-59 Create a locality class

- Create associations:
  - In the **Localities** diagram, hover over the **WeatherTrackingSystem1** locality.
  - Grab the handle and drop it onto the **GroundStation1** locality.
  - Select **Create Association** (Figure 6-60).

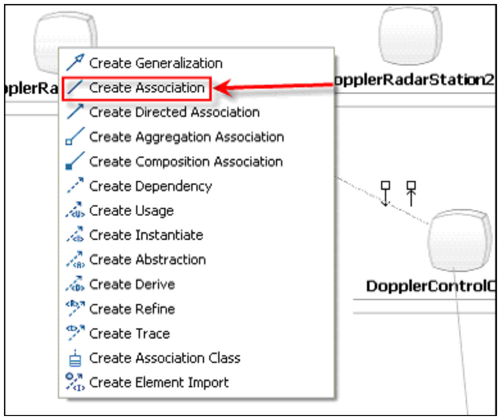


Figure 6-60 Create locality association

Repeat these steps to create the associations listed in Table 6-3.

Table 6-3 Associations in the localities diagram

From	To	Stereotype
WeatherTrackingSystem1	GroundStation1	<<connection>>
WeatherTrackingSystem1	DopplerControlCenter1	<<connection>>
DopplerControlCenter1	DopplerRadarStation1	<<connection>>
DopplerControlCenter1	DopplerRadarStation2	<<connection>>
DopplerControlCenter1	DopplerRadarStation3	<<connection>>

- Multi-select each association on the diagram.
- In the Properties view select the **Stereotypes** tab.
- Click **Apply Stereotypes**.
- Select **<<connection>>**.
- Click **OK**.
- The resulting diagram is shown in Figure 6-61.

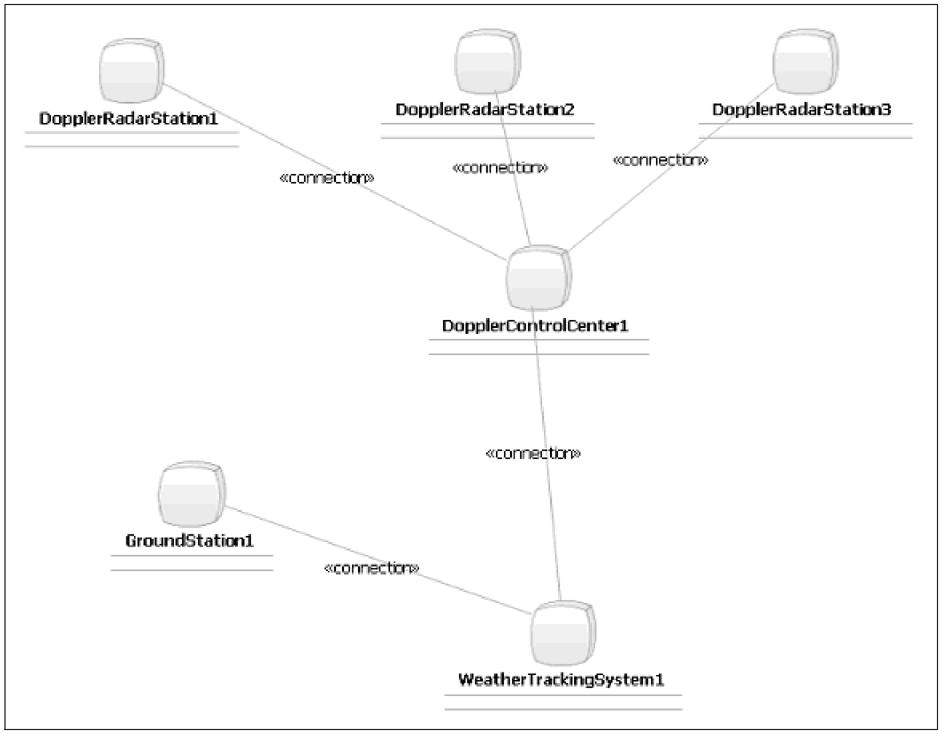


Figure 6-61 Localities diagram

Summary

In this section, you worked at the next level of abstraction modeling the system as a white box. After completing this section, you have created:

- A white-box sequence diagram identifying the flow of events for the provide local weather data operation
- A localities diagram
- New systems as identified during this white-box analysis

7

MDSD and SysML

This chapter discusses the use of SysML with MDSD. We have referred to SysML throughout this document; this chapter attempts to pull together all the important points needed to do MDSD with SysML.[1]

The screen captures in this chapter were taken from the EmbeddedPlus SysML Toolkit.

[1] Parts of this chapter are adapted from an article by Laurent Balmelli: *An Overview of the Systems Modeling Language for Products and Systems Development*, in Journal of Object Technology, vol. 6, no. 6, July-August 2007, pp. 149-177, http://www.jot.fm/issues/issue_2007_07/article2

Introduction

SysML was developed in response to the same issues that MDSD addresses—the need to be able to promote shared understanding across a wide set of stakeholders and participants in the systems development process, the need to manage complexity through separation of concerns with multiple views of a system, and the need to provide traceability through a hierarchy of models, among other things.[2]

MDSD (RUP SE) as contributor to SysML

MDSD, like RUP SE and object-oriented software engineering (OOSE), predates SysML. In fact, MDSD was developed in response to the same kinds of pressures that Rational and then IBM clients were feeling as they developed large, complex, systems of systems. SysML was developed by a consortium of industry participants. IBM and their SysML partner EmbeddedPlus Engineering played an active role in its development (especially IBM participants Murray Cantor and Laurent Balmelli, and EmbeddedPlus participants Salah Obeid, Cory Bialowas, Jim Hummell, and Kumar Marimuthu) contributing concepts and writing parts of the specification. Concepts from RUP SE influenced the development of SysML, for instance, the need for means to express semantics of localities, distribution of responsibilities, and ability to reason about non-functional requirements and a wide variety of stakeholder concerns.

MDSD with SysML

Because SysML was developed in response to the same kinds of issues that MDSD wants to address, it makes sense to use SysML to do MDSD. In essence, SysML is optimized to address the very concerns of MDSD, as noted before. In particular, the use of SysML makes reasoning about parametrics much more effective than trying to do the same in UML. Likewise, traceability between requirements and design elements can be done in SysML, whereas there are no explicit semantics in UML for handling the relationship between requirements and design elements. Finally, the concept of a block transcends the software domain and is intended to express multiple kinds of system elements—while classes can be used to express many of the semantics expressed by blocks, they have a software flavor to them which seems to be antithetical to systems engineers. Furthermore, classes cannot express the kinds of semantics that blocks can, especially in the area of parametrics.

[2] S. Friedenthal, A. Moore, and R. Steiner, *OMG SysML Tutorial*, pg. 8, http://www.omgsysml.org/INCOSE-2007-OMG-SysML-Tutorial.pdf

Basics of SysML

SysML is based on the standard for software engineering, the Unified Modeling Language (UML) developed within the Object Management Group (OMG) consortium. SysML was developed as a response to the request for proposal (RFP) issued by the OMG in March 2003.

Figure 7-1 compares SysML with UML. The text in the figure summarizes the various diagrams available in SysML. Requirements, parametrics and allocations are new diagrams available only in SysML. Activity and block diagrams are reused from UML2.0 and extended in SysML. Lastly, state machines, interactions, and use cases are reused from UML2.0 without modifications.

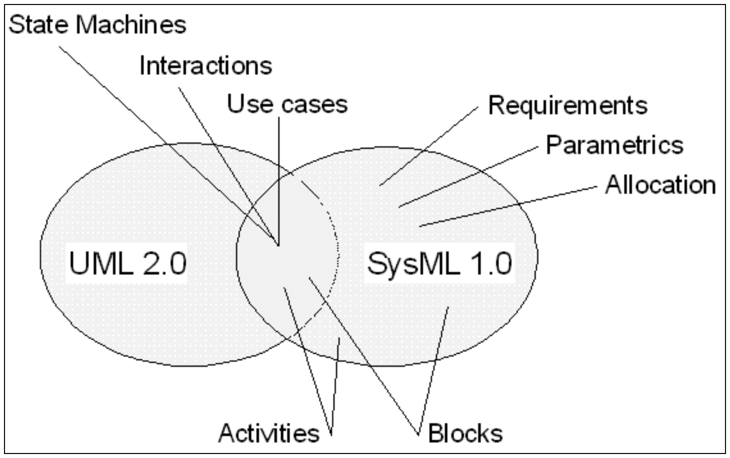


Figure 7-1 Comparison of SysML1.0 with UML2.0

SysML is a modeling language for representing systems and product architectures, as well as their behavior and functionalities. It builds on the experience gained in the software engineering discipline of building software architectures in UML. SysML allows modelers to represent elements realizing the functional aspect of their product. The physical aspect can be represented as well, for example when the architecture represents how the software is deployed on a set of computing resources. As we have seen, this is a key aspect of MDSD.

As noted in Figure 7-1 and its caption, SysML makes use of some UML constructs and concepts without modification, extends some UML constructs, and adds some of its own. Also note that SysML does not use all of the UML 2.0 semantics.

Areas of focus of SysML

The constructs, diagrams, and semantics of SysML are grouped around four areas of focus:

- Requirements modeling
- Improved behavior modeling
- Blocks (improved structure modeling/semantics with blocks)
- Parametrics

Requirements modeling: SysML allows the representation of requirements as model elements. Hence requirements become an integral part of the product architecture. The language offers a flexible means to represent text-based requirements of any nature (for example, functional or non-functional) as well as the relationships between them.

Improved behavior modeling: SysML uses UML constructs for interaction diagrams and state machines and enhances activity diagram semantics, including the addition of semantics to enable modeling of continuous behavior.

Improved structure modeling: SysML provides a basic structural element called a *Block*, whose aim is to provide a discipline-agnostic building block for systems. Blocks can be used to represent any type of components of the system, for example, functional, physical, and human. Blocks assemble to form architectures that represent how different elements in the system co-exist.

Parametrics: SysML provides semantics for reasoning about properties of blocks and their relationships, and allows the integration of engineering analysis with design models. Parametrics in SysML are based on constraint equations--sets of constraints can be depicted graphically, along with their parameters. More specifically, constraints are properties in blocks named *ConstraintProperty* and are typed by *ConstaintBlocks*. A constraint block defines an expression and the attributes that represent its parameters. SysML does not prescribe any language to represent the expressions or provide a solver for it. This is typically offered within the usage of a tool optimized for constraint solving.

We will discuss requirements modeling, blocks, and parametrics in turn, but not changes to activity diagrams, state machines, or other material pertaining to behavior as changed in SysML.

Requirements modeling

Requirements have been traditionally represented as text (accompanied with figures and drawings) and stored in files or databases. The requirements describe all the product functions and the constraints under which these functions should be realized:

- SysML allows the representation of requirements as model elements, and can be related to other model elements. Hence requirements become an integral part of the product architecture. The language offers a flexible means to represent text-based requirements of any nature (for example, functional or non-functional) as well as the relationships between them.
- Figure 7-2 shows a requirement diagram for the Rain Sensing Wiper (RSW) system. Note that it contains both functional and non-functional requirements. **Requirements in SysML are abstract classifiers (that is, they cannot be instantiated) without operations or attributes**. They cannot participate in associations or generalizations, however, a set of predefined relationships help to characterize the relationships between the requirements and other model elements. We review these relationships next.
- Sub-requirements are related to their parent requirement using the cross-hair relationship (that denotes namespace embedding). For example, in Figure 7-2 some of the sub-requirements of the requirement *Automatic Wiping* are connected to it using the cross-hair. The parent requirement is a package for the embedded requirements. In that sense, deleting the parent requirement will automatically delete all the embedded ones. Another example of a requirement acting as a package for other requirements is the requirement *Core Functions*, which contains two sub-requirements. For readability in the model, a user-defined keyword *package* is rendered next to the Requirement stereotype.
- During requirements analysis (system decomposition and operational analysis) new requirements are created by derivation. These new requirements can be connected to the initial ones with the <<deriveRqt>> dependency. For example, in Figure 7-2 a set of core functions for the product are derived from the set of requirements under *Automatic Wiping.* The name <<deriveRqt>> was chosen to avoid any confusion with the standard <<Derive>> dependency in UML 2.0.
- Other examples of derived requirements are the technical choices for each function (see the box Technical choices in Figure 7-2). Note that in Figure 7-2 the designer captures a <<rationale>> comment to explain his choice for using a sensor fixed on the windshield.
- A last example of derived requirement is the quality requirement *System Calibration* stating that the system should be calibrated. This is the requirement added to the product after the RSW failure was identified.[3] The satisfaction of this requirement insures that the product will be resilient to changes in the sensor and windshield characteristics.

[3] See Balmelli, *An overview of the systems modeling language for product and systems development -- Appendix A,* http://www.ibm.com/developerworks/rational/library/aug06/balmelli/appendixa.html

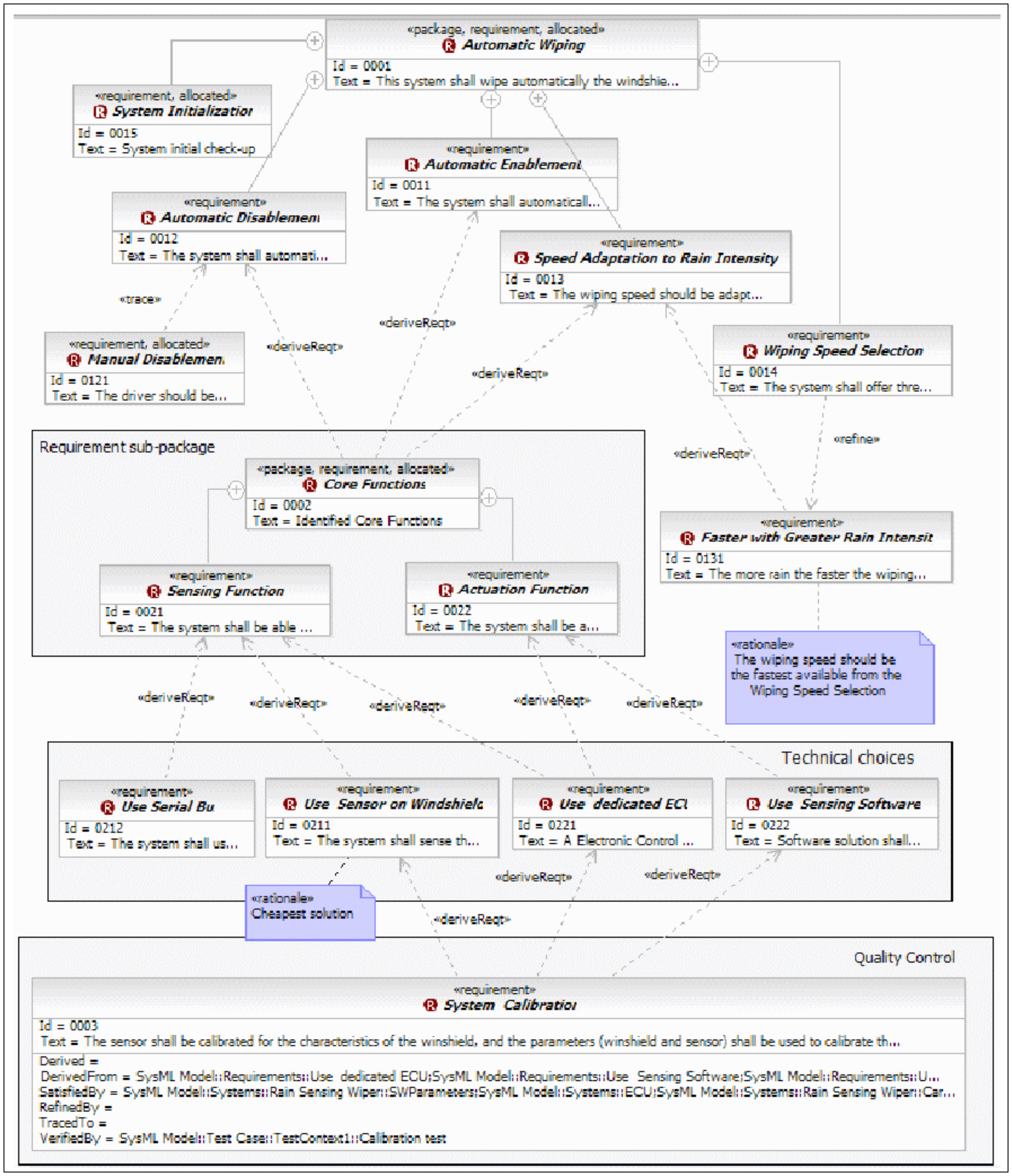

«package, requirement, allocated» Automatic Wiping
Id = 0001
Text = This system shall wipe automatically the windshie...
«requirement, allocated» System Initializatior
Id = 0015
Text = System initial check-up
«requirement» Automatic Enablement
Id = 0011
Text = The system shall automaticall...
«requirement» Automatic Disablemen
Id = 0012
Text = The system shall automati...
«requirement» Speed Adaptation to Rain Intensity
Id = 0013
Text = The wiping speed should be adapt...
«trace»
«requirement, allocated» Manual Disablemen
Id = 0121
Text = The driver should be...
«deriveReqt»
«requirement» Wiping Speed Selection
Id = 0014
Text = The system shall offer thre...
«refine»
Requirement sub-package
«package, requirement, allocated» Core Functions
Id = 0002
Text = Identified Core Functions
«requirement» Sensing Function
Id = 0021
Text = The system shall be able ...
«requirement» Actuation Function
Id = 0022
Text = The system shall be a...
«requirement» Faster with Greater Rain Intensit
Id = 0131
Text = The more rain the faster the wiping...
«rationale» The wiping speed should be the fastest available from the Wiping Speed Selection
Technical choices
«requirement» Use Serial Bu
Id = 0212
Text = The system shall us...
«requirement» Use Sensor on Windshielc
Id = 0211
Text = The system shall sense th...
«requirement» Use dedicated ECl
Id = 0221
Text = A Electronic Control ...
«requirement» Use Sensing Software
Id = 0222
Text = Software solution shall...
«rationale» Cheapest solution
Quality Control
«requirement» System Calibratio
Id = 0003
Text = The sensor shall be calibrated for the characteristics of the winshield, and the parameters (winshield and sensor) shall be used to calibrate th...
Derived =
DerivedFrom = SysML Model::Requirements::Use dedicated ECU;SysML Model::Requirements::Use Sensing Software;SysML Model::Requirements::U...
SatisfiedBy = SysML Model::Systems::Rain Sensing Wiper::SWParameters;SysML Model::Systems::ECU;SysML Model::Systems::Rain Sensing Wiper::Car...
RefinedBy =
TracedTo =
VerifiedBy = SysML Model::Test Case::TestContext1::Calibration test

Figure 7-2 SysML requirements diagram for the Rain Sensing Wiper (RSW) system

- Another relationship between requirements is <<refine>>. An example of requirement refinement is shown in Figure 7-2 on page 148. The requirement on speed actuation is refined by the possible selection for speed (slow, medium or fast.) Lastly, a generic <<trace>> dependency can be used to emphasize that a pair of requirements are related in some way or another. In Figure 7-2 the requirement for *Manual Disablement* is traced to the one about *Automatic Disablement*.
- Requirements have a number of derived attributes to store the status of the relationships reviewed in the above paragraphs. We will see later in this chapter that these attributes become particularly handy when requirement relationships are represented outside requirements diagrams.
- Often requirements are elicited during the whole product life cycle and additional requirement diagrams are used to represent them. Hence the product requirements are typically laid out on a set of requirement diagrams.
- SysML provides a generic model for requirements that allows the modeling of both functional and non-functional requirements. A non-normative set of requirement types are proposed in the appendix of the OMG SysML specification.[4] Specific types of requirements (for example related to timing or scheduling) are handled by language extensions. SysML (like UML) supports a profile mechanism to extend the language. The Object Management Group (OMG) has released a series of modeling standards that address specific needs: for the modeling of non-functional requirements related to performance and quality [quality of service (QoS), software test plan (STP)], and for the modeling of test cases [testing profile]. These profiles can be used in SysML without restriction.

It should be noted that while SysML allows for requirements decomposition and allocation of requirements to design elements, MDSD does not encourage this. In general, MDSD promotes the derivation of requirements (as opposed to their allocation) through system decomposition and operations analysis.[5]

Additionally, much of the manual labor of creating requirements related diagrams can be automated, and should be, based on the artifacts resulting from following the MDSD process. For example, each use case or operation realization in a model represents the derivation of requirements on the participating collaborators. The functional requirements (operations) on each of the collaborators are derived from the use case or operation being realized.

[4] SysML 1.0 Specification (ptc/06-05-04), OMG final adopted specification, available at http://www.omgsysml.org/

[5] See discussion in Chapter 2 concerning requirements decomposition and Cantor's article on Functional Decomposition: *Thoughts on Functional Decomposition*, Rational Edge, June 2003, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/apr03/FunctionalDecomposition_TheRationalEdge_Apr2003.pdf

Traceability relationships can be deduced from the model using the structures created in operations analysis. Requirements information can also be deduced from joint realization tables.

Block semantics

As noted above, SysML provides a basic structural element called a *Block*, whose aim is to provide a discipline-agnostic building block for systems. Blocks can be used to represent any type of component of the system, for example, functional, physical, and human. Blocks assemble to form architectures that represent how different elements in the system co-exist.

Block definition diagram

The *SysML Block Definition Diagram* (BDD) is the simplest way to describe the structure of the system. It is the equivalent to the class diagram in UML. It is used to represent the system decomposition using for example associations and composition relationships. The BDD is ideal to display the features of a block, such as its properties, and operations. SysML allows blocks to own special types of properties: Block properties and distributed properties.

- Block properties impose additional constraints on classic UML properties, and can for instance own a SysML *value type*. Value types are designed to hold units (for example, physical units) and dimensions.
- Distributed properties let the user apply a probability distribution to the values of the property. SysML proposes model libraries for possible values of units, dimensions, and probability distributions.

In Figure 7-3 we show a BDD for the RSW. For the sake of readability of the diagram, we do not render the associations between the sub-systems and the *Rain Sensing Wiper* element, although these associations exist in the model. Instead we use an illustrative box around each set of components (composite and external) and a black diamond shape over the composite component as a visual clue for composition. The main components of the RSW are an interface to actuate the wiper, an electronic control unit, a sensor and the windshield element. Both the interface and the windshield can exist in the car with or without the RSW (In SysML they are so-called *reference properties*).

- The properties and the operations for each block are visible in Figure 7-3. Properties (more precisely SysML block properties, shown using the stereotype <<blockProperty>>) are used to model the physical characteristics of the components. The operations (called sometimes services) represent the functional aspects of the system.

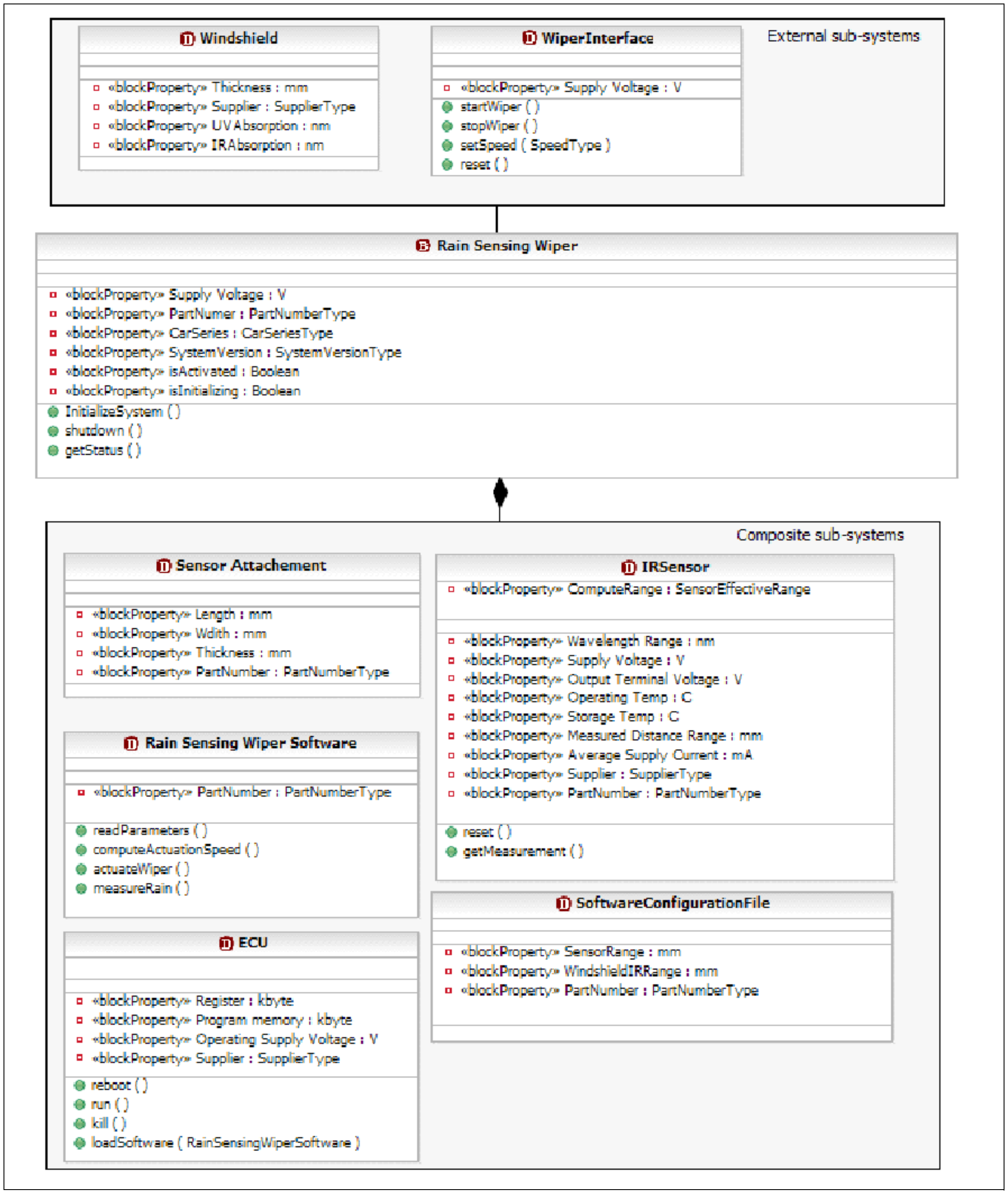
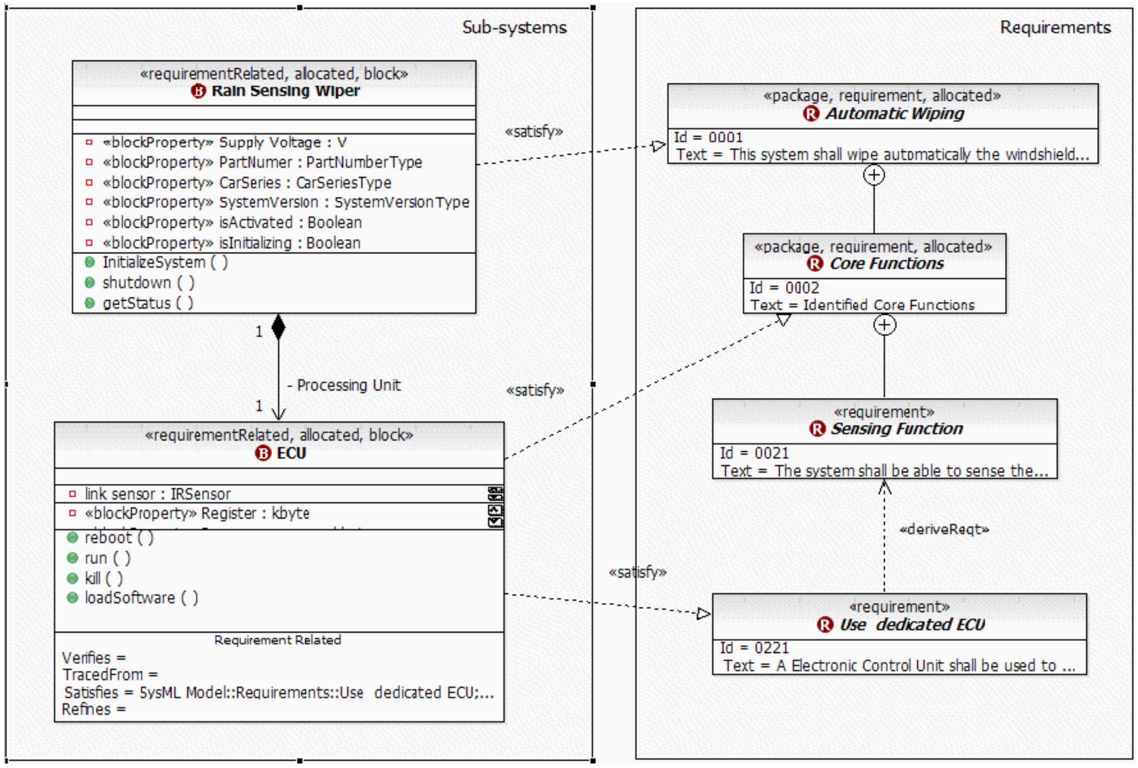

External sub-systems
Windshield
«blockProperty» Thickness : mm
«blockProperty» Supplier : SupplierType
«blockProperty» UV Absorption : nm
«blockProperty» IR Absorption : nm
WiperInterface
«blockProperty» Supply Voltage : V
startWiper ( )
stopWiper ( )
setSpeed (SpeedType)
reset ( )
Rain Sensing Wiper
«blockProperty» Supply Voltage : V
«blockProperty» PartNumer : PartNumberType
«blockProperty» CarSeries : CarSeriesType
«blockProperty» SystemVersion : SystemVersionType
«blockProperty» isActivated : Boolean
«blockProperty» isInitializing : Boolean
InitializeSystem ( )
shutdown ( )
getStatus ( )
Composite sub-systems
Sensor Attachement
«blockProperty» Length : mm
«blockProperty» Width : mm
«blockProperty» Thickness : mm
«blockProperty» PartNumber : PartNumberType
IRSensor
«blockProperty» ComputeRange : SensorEffectiveRange
«blockProperty» Wavelength Range : nm
«blockProperty» Supply Voltage : V
«blockProperty» Output Terminal Voltage : V
«blockProperty» Operating Temp : C
«blockProperty» Storage Temp : C
«blockProperty» Measured Distance Range : mm
«blockProperty» Average Supply Current : mA
«blockProperty» Supplier : SupplierType
«blockProperty» PartNumber : PartNumberType
reset ( )
getMeasurement ( )
Rain Sensing Wiper Software
«blockProperty» PartNumber : PartNumberType
readParameters ( )
computeActuationSpeed ( )
actuateWiper ( )
measureRain ( )
SoftwareConfigurationFile
«blockProperty» SensorRange : mm
«blockProperty» WindshieldIRRange : mm
«blockProperty» PartNumber : PartNumberType
ECU
«blockProperty» Register : kbyte
«blockProperty» Program memory : kbyte
«blockProperty» Operating Supply Voltage : V
«blockProperty» Supplier : SupplierType
reboot ( )
run ( )
kill ( )
loadSoftware (RainSensingWiperSoftware)

Figure 7-3 SysML Block Definition Diagram for the Rain Sensing Wiper system

- We now examine how the product structure and the product requirements can be related: One of the important consequence of having requirements as model elements is that it allows the designer to specify which components in the system satisfy a given set of requirements. This is called the *allocation process*. We show an example of requirement allocation in Figure 7-4, where the part on the left hand side represents some elements of the RSW, and the part on the right hand side is a hierarchy of requirements. One way to perform allocation is to use the <<satisfy>> dependency. In the figure, the *Rain Sensing Wiper* model element is allocated to the requirement named *Automatic Wiping*. Any element in SysML can be used to satisfy a requirement.
- Another way to display allocation is to use a dedicated compartment named *requirement related*. This compartment displays the status of a set of derived properties related to requirements. In Figure 7-4 the element ECU displays this compartment: The ECU element is allocated to the requirement named Use dedicated ECU.



Sub-systems
«requirementRelated, allocated, block»
Rain Sensing Wiper
«blockProperty» Supply Voltage : V
«blockProperty» PartNumer : PartNumberType
«blockProperty» CarSeries : CarSeriesType
«blockProperty» SystemVersion : SystemVersionType
«blockProperty» isActivated : Boolean
«blockProperty» isInitializing : Boolean
InitializeSystem ( )
shutdown ( )
getStatus ( )
1
- Processing Unit
1
«requirementRelated, allocated, block»
ECU
link sensor : IRSensor
«blockProperty» Register : kbyte
reboot ( )
run ( )
kill ( )
loadSoftware ( )
Requirement Related
Verifies =
TracedFrom =
Satisfies = SysML Model::Requirements::Use dedicated ECU;...
Refines =
«satisfy»
«satisfy»
«satisfy»
Requirements
«package, requirement, allocated»
Automatic Wiping
Id = 0001
Text = This system shall wipe automatically the windshield...
«package, requirement, allocated»
Core Functions
Id = 0002
Text = Identified Core Functions
«requirement»
Sensing Function
Id = 0021
Text = The system shall be able to sense the...
«deriveReqt»
«requirement»
Use dedicated ECU
Id = 0221
Text = A Electronic Control Unit shall be used to ...

Figure 7-4 Example of requirement allocation

Internal block diagram

The *SysML Internal Block Diagram* (IBD) allows the designer to refine the structural aspect of the model. The IBD is the equivalent of the composite structure in UML. In the IBD properties (or parts) are assembled to define how they collaborate to realize the behavior of the block. A part represents the usage of another other block.

The most important aspect of the IBD is that it allows the designer to refine the definition of the interaction between the usages of blocks by defining *Ports*, as explained below.

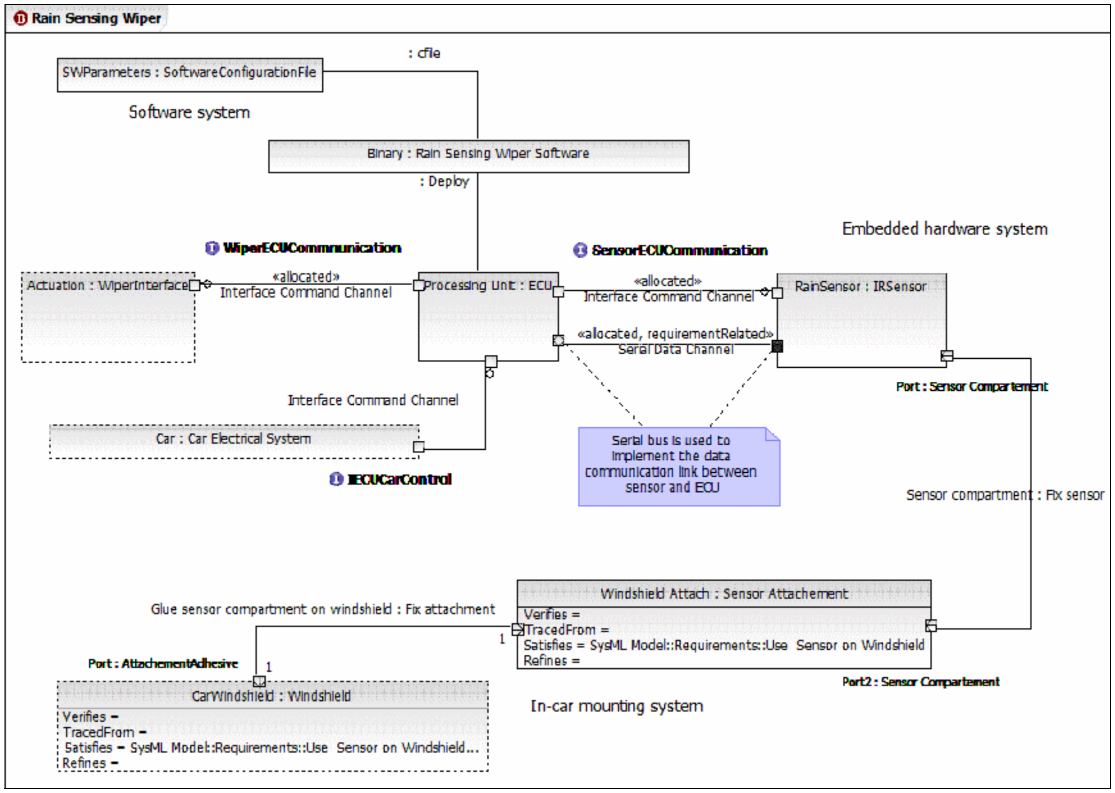
Ports

Ports are parts available for connection from the outside of the owing block. Ports are typed by interfaces or blocks that define what can be exchanged through them. Ports are connected using *connectors* that represent the use of an association in the IBD.

Two types of ports are available in SysML: *Standard ports* handle the requests and invocations of services (function calls) with other blocks, and *flow ports* let blocks exchange flows of information or material.

For standard ports, an *interface* class is used to list the services offered by the block. For flow ports, a *Flow Specification* is created to list the type of data that can flow through the port. When only a single type of object can flow through a port, then the type is used as type for the port directly. Such a port is named *Atomic Port*. The class Item Flow is used to represent what does actually flow between blocks in a particular usage context. We refer the interested reader to the standard specification for more details on item flows.[6] The IBD is shown in Figure 7-5.

[6] SysML 1.0 Specification (ptc/06-05-04), OMG final adopted specification, available at http://www.omgsysml.org/



Rain Sensing Wiper
: cfile
SWParameters : SoftwareConfigurationFile
Software system
Binary : Rain Sensing Wiper Software
: Deploy
Embedded hardware system
WiperECUCommunication
SensorECUCommunication
Actuation : WiperInterface
«allocated»
Interface Command Channel
Processing Unit : ECU
«allocated»
Interface Command Channel
RainSensor : IRSensor
«allocated, requirementRelated»
Serial Data Channel
Port : Sensor Compartement
Interface Command Channel
Car : Car Electrical System
IECUCarControl
Serial bus is used to implement the data communication link between sensor and ECU
Sensor compartment : Fix sensor
Windshield Attach : Sensor Attachement
Verifies =
TracedFrom =
Satisfies = SysML Model::Requirements::Use Sensor on Windshield
Refines =
Glue sensor compartment on windshield : Fix attachment
1
Port : AttachementAdhesive
1
CarWindshield : Windshield
Verifies =
TracedFrom =
Satisfies = SysML Model:Requirements::Use Sensor on Windshield...
Refines =
Port2 : Sensor Compartement
In-car mounting system

Figure 7-5 SysML Internal Block Diagram of the Rain Sensing Wiper system

- In Figure 7-5 we refine our initial description of the RSW by showing how parts are interacting inside the block named Rain Sensing Wiper. Previously to constructing the IBD, we have to define a model for the associations characterizing the relationships between the different blocks. Also, additional blocks are defined for example to type the ports. We show this model in another BDD that can be found in Figure 7-6.
- The central part of Figure 7-5 consists of the parts of the system that represent the embedded hardware. The parts underneath are used for mounting the system in the car. The parts above represent the software. A set of standard ports and interfaces are defined to represent the functional aspect of the communication between the parts. For example, the Processing Unit (ECU) accesses the Actuation (interface) of the wiper through the interface `WiperECUCommunication`. Details about the interfaces used in this IBD are found in Figure 7-6.

- The Processing Unit communicates with the RainSensor using a flow port. The data exchanged is two bitstreams, one containing the measurements from the sensor and another containing synchronization data. The port is typed with a specification of these flows using the element `SensorECUCommunication` (see Figure 7-6). Notice the direction of the flows in the definition.
- For convenience a flow port can be conjugated in the sense that its input and outputs are inversed (flows declared as *in* becomes *out* and vice-versa) with respect to the definition of the interface. This is useful when connecting two systems whose flow ports are conjugated with respect to each other. This is the case for instance between the Processing Unit and the RainSensor in Figure 7-5. A conjugated flow port is represented in black. Because the synchronization data flow is declared as *inout*, the conjugation of the port has no effect on it.
- Note that in Figure 7-5 connectors between ports link parts defined within the block. SysML actually allows direct connection between ports defined at different levels of granularity, for example between a port and another one defined *inside* a part. This type of connector are called *nested connectors*. We refer readers to the standard specification [OMG SysML] for more details about these connectors.
- Flow ports are also useful to define physical contact between parts: For example the Sensor Attachement unit is fixed to the Windshield using an adhesive. The block representing the adhesive material `AttachementAdhesive` (Figure 7-6) is used to type the flow port connecting these parts.

The addition of flow ports to SysML allows us to reason more effectively about physical or electrical design issues. UML does not do this without inventing a stereotype or extension which would provide the equivalent semantics.

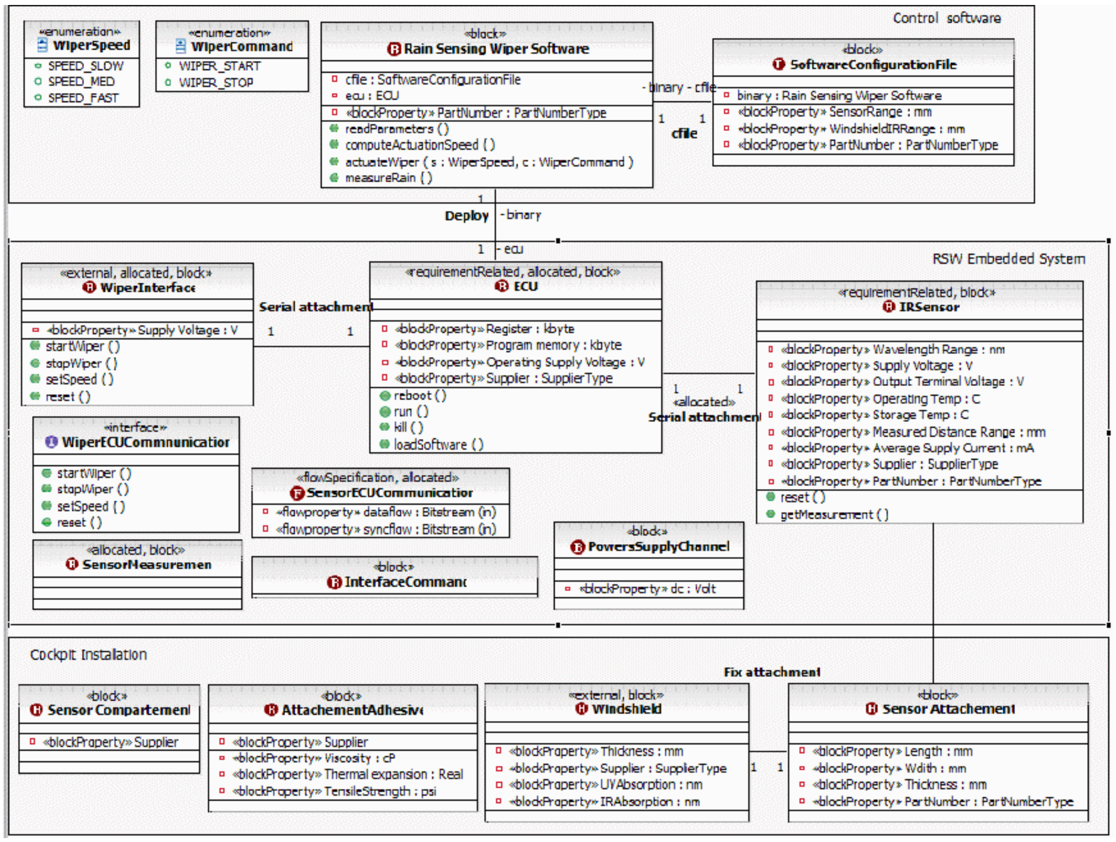
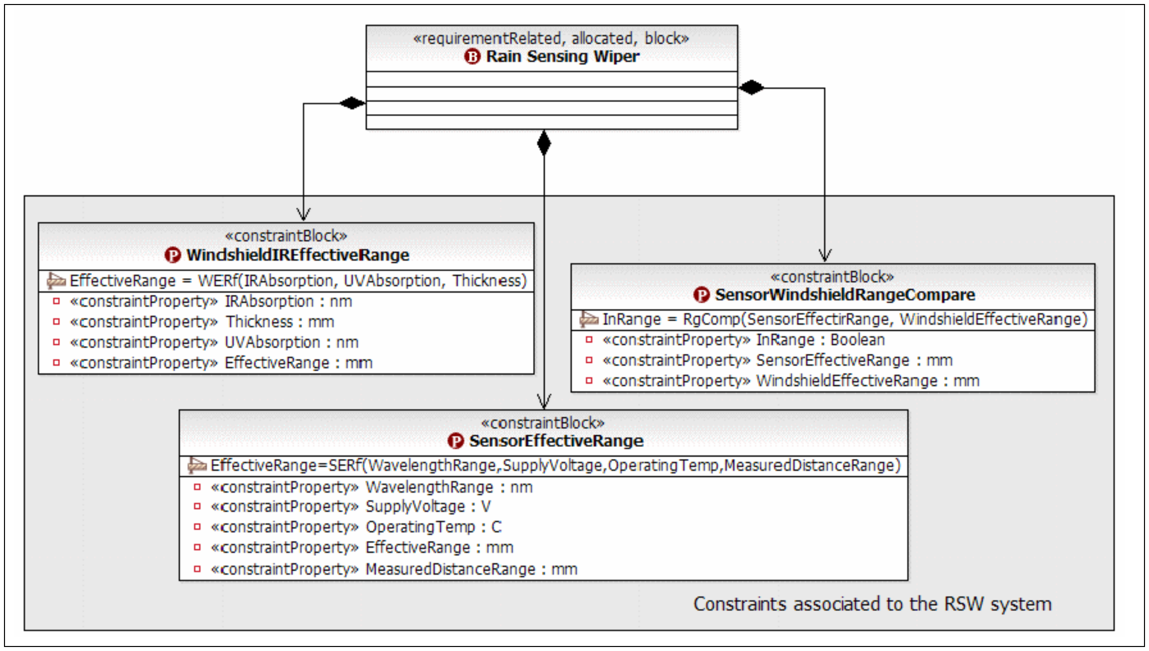

Control software
«enumeration» WiperSpeed
SPEED_SLOW
SPEED_MED
SPEED_FAST
«enumeration» WiperCommand
WIPER_START
WIPER_STOP
«block» Rain Sensing Wiper Software
cfile : SoftwareConfigurationFile
ecu : ECU
«blockProperty» PartNumber : PartNumberType
readParameters ()
computeActuationSpeed ()
actuateWiper (s : WiperSpeed, c : WiperCommand)
measureRain ()
«block» SoftwareConfigurationFile
binary : Rain Sensing Wiper Software
«blockProperty» SensorRange : mm
«blockProperty» WindshieldIRRange : mm
«blockProperty» PartNumber : PartNumberType
- binary - cfile
cfile
Deploy
- binary
- ecu
RSW Embedded System
«external, allocated, block» WiperInterface
«blockProperty» Supply Voltage : V
startWiper ()
stopWiper ()
setSpeed ()
reset ()
Serial attachment
«requirementRelated, allocated, block» ECU
«blockProperty» Register : kbyte
«blockProperty» Program memory : kbyte
«blockProperty» Operating Supply Voltage : V
«blockProperty» Supplier : SupplierType
reboot ()
run ()
kill ()
loadSoftware ()
«requirementRelated, block» IRSensor
«blockProperty» Wavelength Range : nm
«blockProperty» Supply Voltage : V
«blockProperty» Output Terminal Voltage : V
«blockProperty» Operating Temp : C
«blockProperty» Storage Temp : C
«blockProperty» Measured Distance Range : mm
«blockProperty» Average Supply Current : mA
«blockProperty» Supplier : SupplierType
«blockProperty» PartNumber : PartNumberType
reset ()
getMeasurement ()
«allocated»
Serial attachment
«interface» WiperECUCommunication
startWiper ()
stopWiper ()
setSpeed ()
reset ()
«flowSpecification, allocated» SensorECUCommunication
«flowproperty» dataflow : Bitstream (in)
«flowproperty» syncflow : Bitstream (in)
«block» PowersSupplyChannel
«blockProperty» dc : Volt
«allocated, block» SensorMeasuremen
«block» InterfaceCommand
Cockpit Installation
Fix attachment
«block» Sensor Compartement
«blockProperty» Supplier
«block» AttachementAdhesive
«blockProperty» Supplier
«blockProperty» Viscosity : cP
«blockProperty» Thermal expansion : Real
«blockProperty» TensileStrength : psi
«external, block» Windshield
«blockProperty» Thickness : mm
«blockProperty» Supplier : SupplierType
«blockProperty» UVAbsorption : nm
«blockProperty» IRAbsorption : nm
«block» Sensor Attachement
«blockProperty» Length : mm
«blockProperty» Width : mm
«blockProperty» Thickness : mm
«blockProperty» PartNumber : PartNumberType

Figure 7-6 SysML Block Definition Diagram to type ports

Constraints

We have seen so far how attributes are defined for blocks in order to represent their physical characteristics. Often, attributes of a set of systems are not independent. Consider two sub-systems A and B having attributes a and b, respectively, and that the constraint {A.a greater than B.b} must hold true. SysML *ConstraintBlocks* allows the engineer to define any relationships (for example, analytical) between the system attributes. These constraints form networks of expressions that are typically leveraged in simulations, for example, for requirements verification. Note that constraint blocks are not instantiated as runtime objects, but rather used to type special properties of blocks, as explained below.

- Constraints are properties in sub-systems (that is, blocks) named *ConstraintProperty* and are typed by <<constraintBlock>>. A constraint block defines an expression and the attributes that represent its parameters. SysML does not prescribe any language to represent the expressions or provide a solver for it. This setting is typically offered within the usage of a particular tool.
- The RSW uses a set of analytical constraints to verify that the system is properly calibrated (requirement *System Calibration* in Figure 7-2 on page 148). Three constraints are shown in Figure 7-7:
  - The constraint `SensorEffectiveRange` computes an operational range for the sensor, based on some of its parameters.
  - Similarly, the constraint `WindshieldIREffectiveRange` computes an operating range for infrared sensor that can be compared with the one computed for the sensor.
  - Finally the constraint `SensorWindshieldRangeCompare` is used to compare the above values.


«requirementRelated, allocated, block»
Rain Sensing Wiper
«constraintBlock»
WindshieldIREffectiveRange
EffectiveRange = WERf(IRAbsorption, UVAbsorption, Thickness)
«constraintProperty» IRAbsorption : nm
«constraintProperty» Thickness : mm
«constraintProperty» UVAbsorption : nm
«constraintProperty» EffectiveRange : mm
«constraintBlock»
SensorWindshieldRangeCompare
InRange = RgComp(SensorEffectirRange, WindshieldEffectiveRange)
«constraintProperty» InRange : Boolean
«constraintProperty» SensorEffectiveRange : mm
«constraintProperty» WindshieldEffectiveRange : mm
«constraintBlock»
SensorEffectiveRange
EffectiveRange=SERf(WavelengthRange,SupplyVoltage,OperatingTemp,MeasuredDistanceRange)
«constraintProperty» WavelengthRange : nm
«constraintProperty» SupplyVoltage : V
«constraintProperty» OperatingTemp : C
«constraintProperty» EffectiveRange : mm
«constraintProperty» MeasuredDistanceRange : mm
Constraints associated to the RSW system
Figure 7-7 Definition of constraint blocks for the Rain Sensing Wiper system

Parametrics

The *SysML Parametric Diagram* (PD) is used to represent the usage of constraint blocks as constraint properties. Syntactically the PD is actually is similar to IBD. In a PD, constraint properties are connected to each other through the parameters defined by their constraint block. In turn they connect to other properties in the context of their owning block. These other properties must be directly bound to parameters of the constraint properties because they can only play a "feeding role" to the constraints parameters in a PD.

- An example of a PD is shown in Figure 7-10 on page 160. Constraint properties are represented by boxes with rounded corners. In this diagram, both the sensor and windshield parts compute an operational range that is compared by the property named *compare*. These values are also fed to the part representing the configuration file (bottom of the figure). If the sensor and the windshield are compatible, the flag *IsCalibrated* (exposed as a port) is set to true. The verification of the calibration requirement is hence reduced to testing the value of this port. The system is therefore resilient to changes in windshield and sensor characteristics.
- The usage of the constraint blocks `WindshieldIREffectiveRange` and `SensorEffectiveRange` can be seen in the diagrams of Figure 7-8 and Figure 7-9, respectively. They are nested in the parts named RainSensor and CarWindshield (see comments in the figure).
- An attractive aspect of constraint blocks is that they provide a reusable mechanism to define types of constraints. Hence the same constraint can be used several times in the model. It is important to note that a constraint does not specify which variable is an input or an output. Values are assigned by the context and a numerical solver will provide results for the variables of the system.[7]

[7] See the work by Peak et al. on constraints for more details: Peak RS, Friedenthal S, Moore A, Burkhart R, Waterbury SC, Bajaj M, Kim I, *Experiences Using SysML Parametrics to Represent Constrained Objectbased Analysis Templates*. 2005. 7th NASA-ESA Workshop on Product Data Exchange (PDE): *The Workshop for Open Product & System Lifecycle Management (PLM/SLiM)*, Atlanta. See also `http://www.pslm.gatech.edu/topics/sysml/`

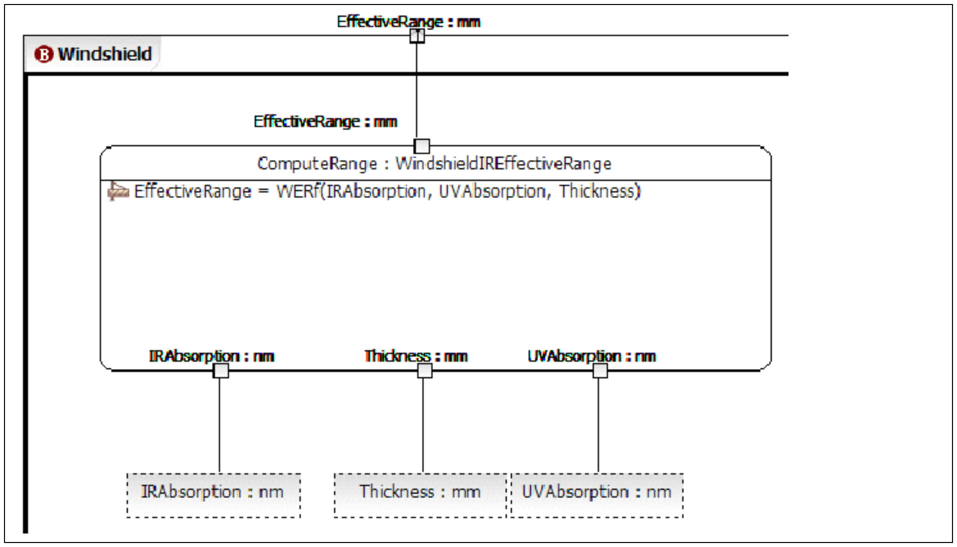


Figure 7-8 Parametric diagram for the windshield

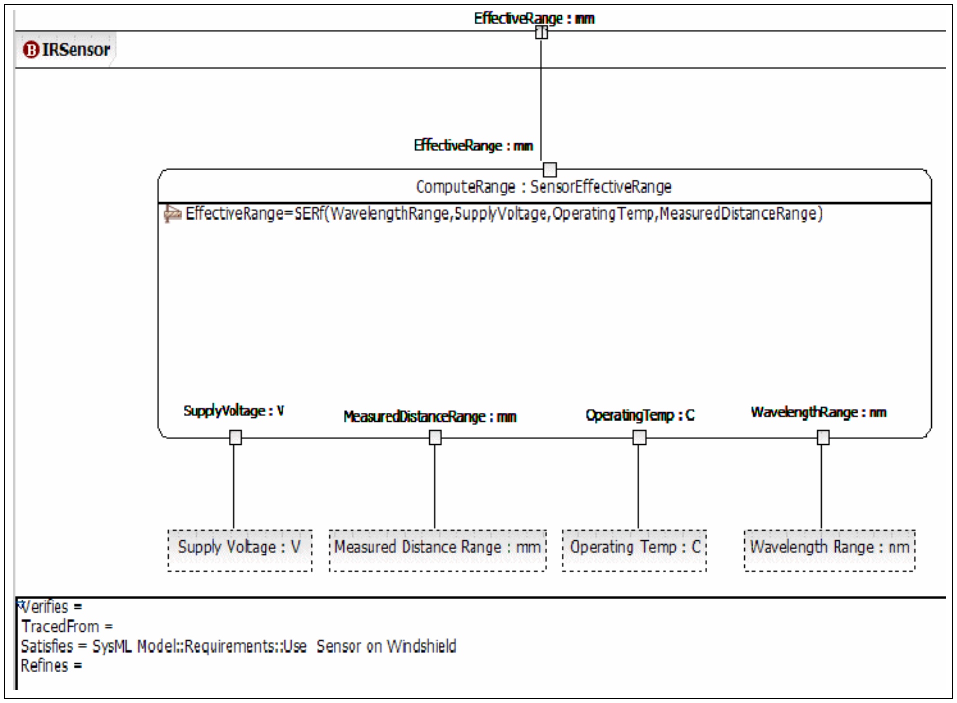

EffectiveRange : mm
IRSensor
EffectiveRange : mm
ComputeRange : SensorEffectiveRange
EffectiveRange=SERf(WavelengthRange,SupplyVoltage,OperatingTemp,MeasuredDistanceRange)
SupplyVoltage : V
MeasuredDistanceRange : mm
OperatingTemp : C
WavelengthRange : nm
Supply Voltage : V
Measured Distance Range : mm
Operating Temp : C
Wavelength Range : nm
Verifies =
TracedFrom =
Satisfies = SysML Model::Requirements::Use Sensor on Windshield
Refines =

Figure 7-9 Parametric diagram for the sensor

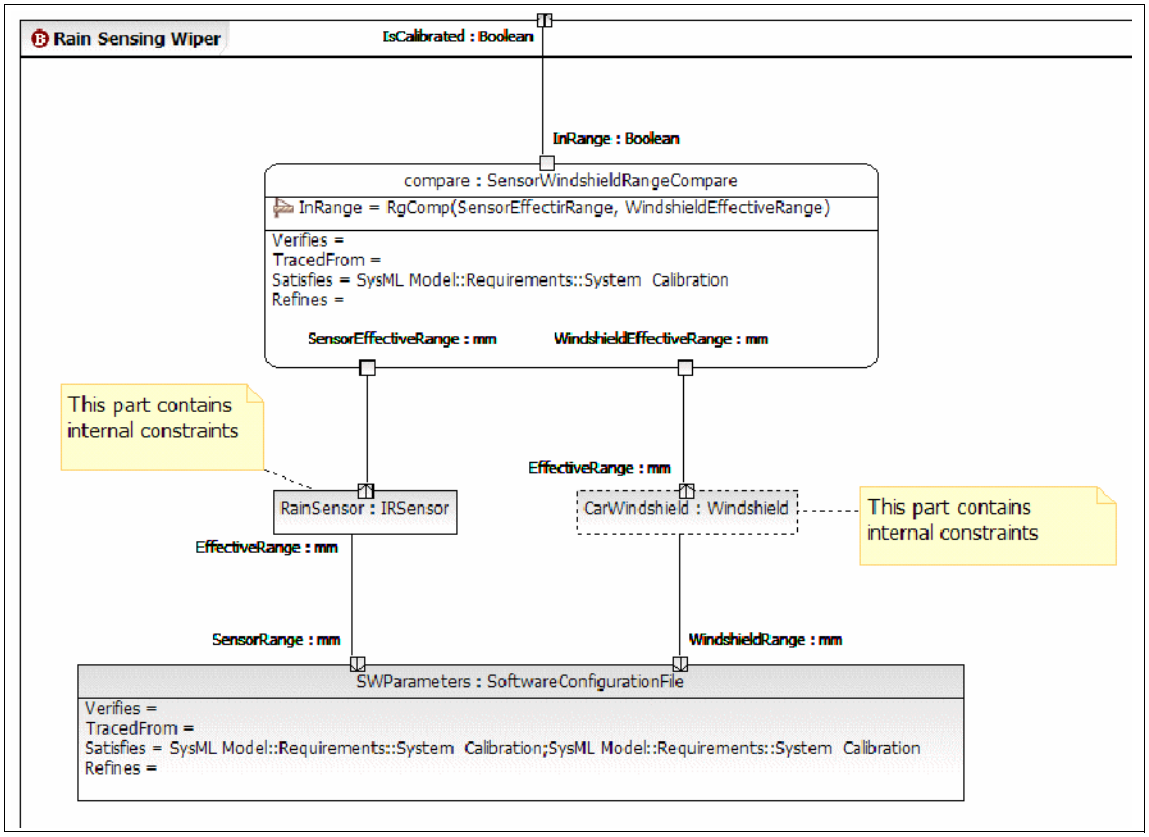

Rain Sensing Wiper
IsCalibrated : Boolean
InRange : Boolean
compare : SensorWindshieldRangeCompare
InRange = RgComp(SensorEffectirRange, WindshieldEffectiveRange)
Verifies =
TracedFrom =
Satisfies = SysML Model::Requirements::System Calibration
Refines =
SensorEffectiveRange : mm
WindshieldEffectiveRange : mm
This part contains internal constraints
EffectiveRange : mm
RainSensor : IRSensor
CarWindshield : Windshield
This part contains internal constraints
EffectiveRange : mm
SensorRange : mm
WindshieldRange : mm
SWParameters : SoftwareConfigurationFile
Verifies =
TracedFrom =
Satisfies = SysML Model::Requirements::System Calibration;SysML Model::Requirements::System Calibration
Refines =

Figure 7-10 SysML Parametric Diagram for the Rain Sensing Wiper system

- Requirement allocation is shown in PDs using compartments: In Figure 7-10 the requirement allocation compartment is displayed in both the constraint used for comparison and the part representing the configuration file. These elements satisfy the requirement named *System Calibration*.

Behavior modeling

For behavior and activity modeling, see Balmelli's article,[8] and reference within it to Bock.[9] As noted above, the major difference between UML and SysML in this area is that SysML has improved semantics to handle continuous behavior.

[8] Laurent Balmelli, *An Overview of the Systems Modeling Language for Products and Systems Development*, in Journal of Object Technology, vol. 6, no. 6, July-August 2007, pp. 149-177 http://www.jot.fm/issues/issue_2007_07/article2

[9] Conrad Bock, *SysML and UML 2 Support for Activity Modeling*, Wiley InterScience, DOI 10.1002/sys, http://www.mel.nist.gov/msidlibrary/doc/sysmlactivity.pdf

MDSD with SysML

Let us focus now on using SysML for MDSD. How can we best use it to accomplish the goals of MDSD? We want to build upon the strengths of both MDSD and SysML; we want to use SysML to optimally express what we are trying to do with MDSD.

Blocks as basic structural units

Blocks will be our basic structural units. They can stand for software, hardware, or workers within the system or systems under consideration. They are ideal to represent system decomposition—we can have blocks within blocks.

Understanding context

Let us begin with understanding context. One of the first, if not the very first, artifacts we build in MDSD is a context diagram.

Using blocks to stand for systems

The first, fairly obvious decision is to use blocks to represent systems in our context diagrams. We can show or hide compartments, attributes, operations, and so on, depending on the level of detail we want to show.[10]

Next, we need to consider the relationship between actors, the system under consideration in the context diagram, and I/O entities.

The simplest option here is to use basically the same semantics we would use in UML to represent these concepts and relationships, that is, to create associations between the actors and the system under consideration, and to relate the actors to the I/O entities with associations as well (Figure 7-11).

[10] Exactly how to do this is tool dependent. Any reasonable modeling tool will have this capability.

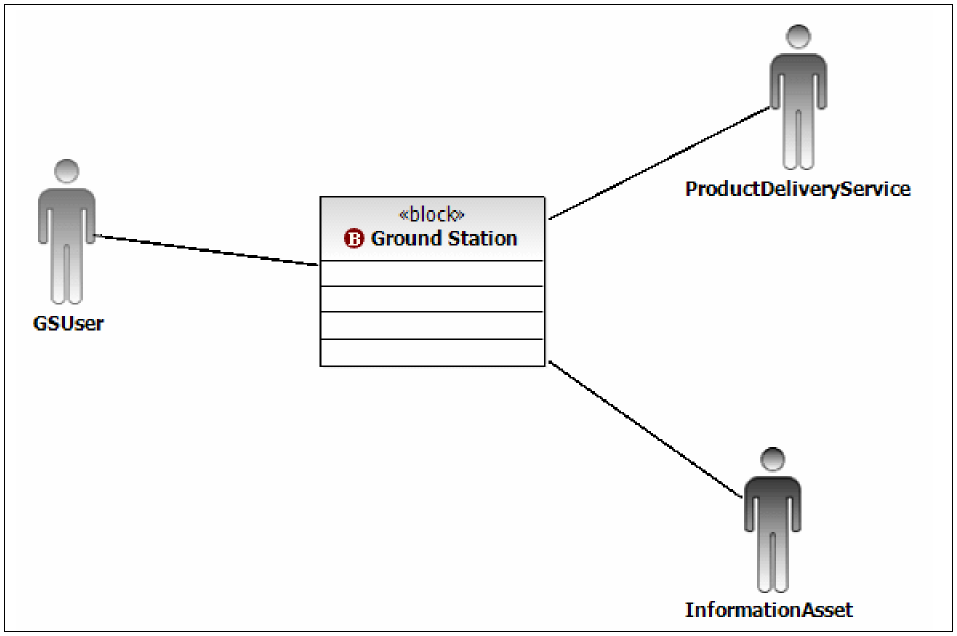
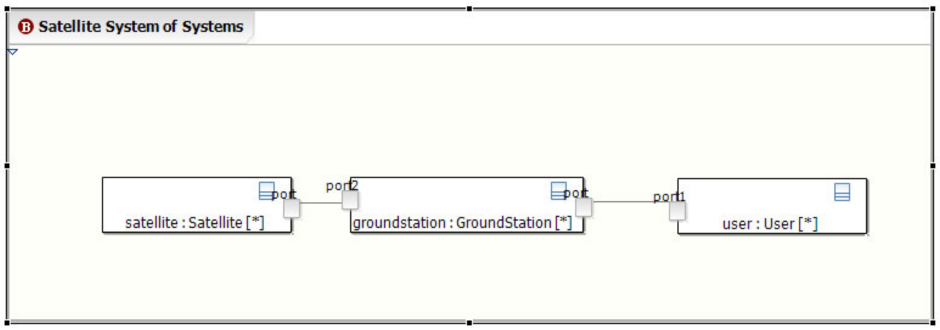


Figure 7-11 Context diagram with blocks and associations

A more complex option, but one that will likely carry more information specified with more precise semantics, would be to use ports and connectors between blocks in an enclosing context. In this case, I/O entities will be the information that gets exchanged through the ports and connectors. This will allow for greater specificity. The danger here is that specificity often comes at a price—perhaps it is too early in our analysis process to be at this level of detail.

This is a judgement call—we must remember why we are modeling (to manage complexity and to communicate effectively, among other things) and what we have to accomplish at any given point in our development process. It is often better to begin with less specificity (because we really do not know enough yet) and to refine and get more specific as we progress through our process.

In any case, here is an example of a context diagram using blocks, ports, and connectors (Figure 7-12).


Satellite System of Systems
port
satellite : Satellite [*]
port2
port
groundstation : GroundStation [*]
port1
user : User [*]
Figure 7-12 Context diagram with blocks and ports

Requirements and understanding context

Requirements on the system at this level can be represented either as system attributes or as requirements that are related to the system and depicted on a requirements diagram. We do not want to try to represent all system requirements on a diagram or as attributes; that would produce a very complicated unreadable diagram because of the possibly large number of requirements. But if there is a small set of requirements that constrain the system in such a way that the architecture is likely to be influenced by them, it would be good to represent this visually.

For example, the range desired for a radar will influence its size and weight, due to power needs. Also, if we want to provide automated reasoning or simulation capabilities, we will want to include as much information as needed to drive our reasoning or simulation engines.

Figure 7-13 shows the sample diagrams for range of radar as attribute, and Figure 7-14 shows the same diagram with block and associated requirements.

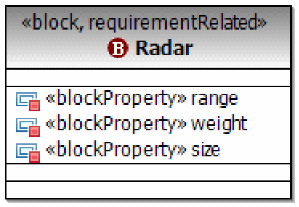


Figure 7-13 Example diagram for range of radar as an attribute

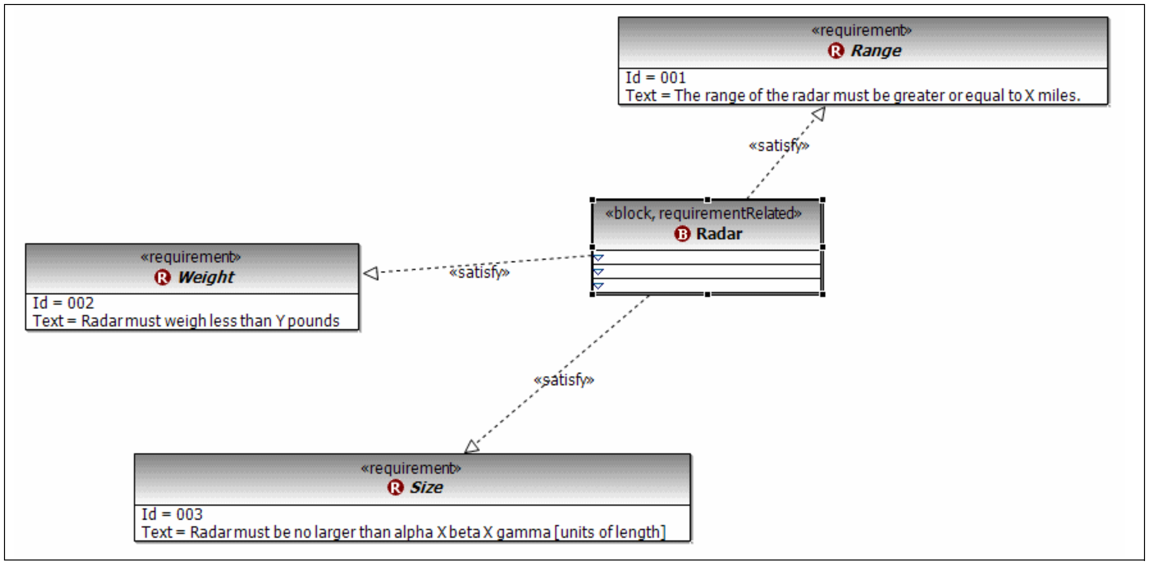

«requirement»
Range
Id = 001
Text = The range of the radar must be greater or equal to X miles.
«satisfy»
«block, requirementRelated»
Radar
«requirement»
Weight
Id = 002
Text = Radar must weigh less than Y pounds
«satisfy»
«satisfy»
«requirement»
Size
Id = 003
Text = Radar must be no larger than alpha X beta X gamma [units of length]

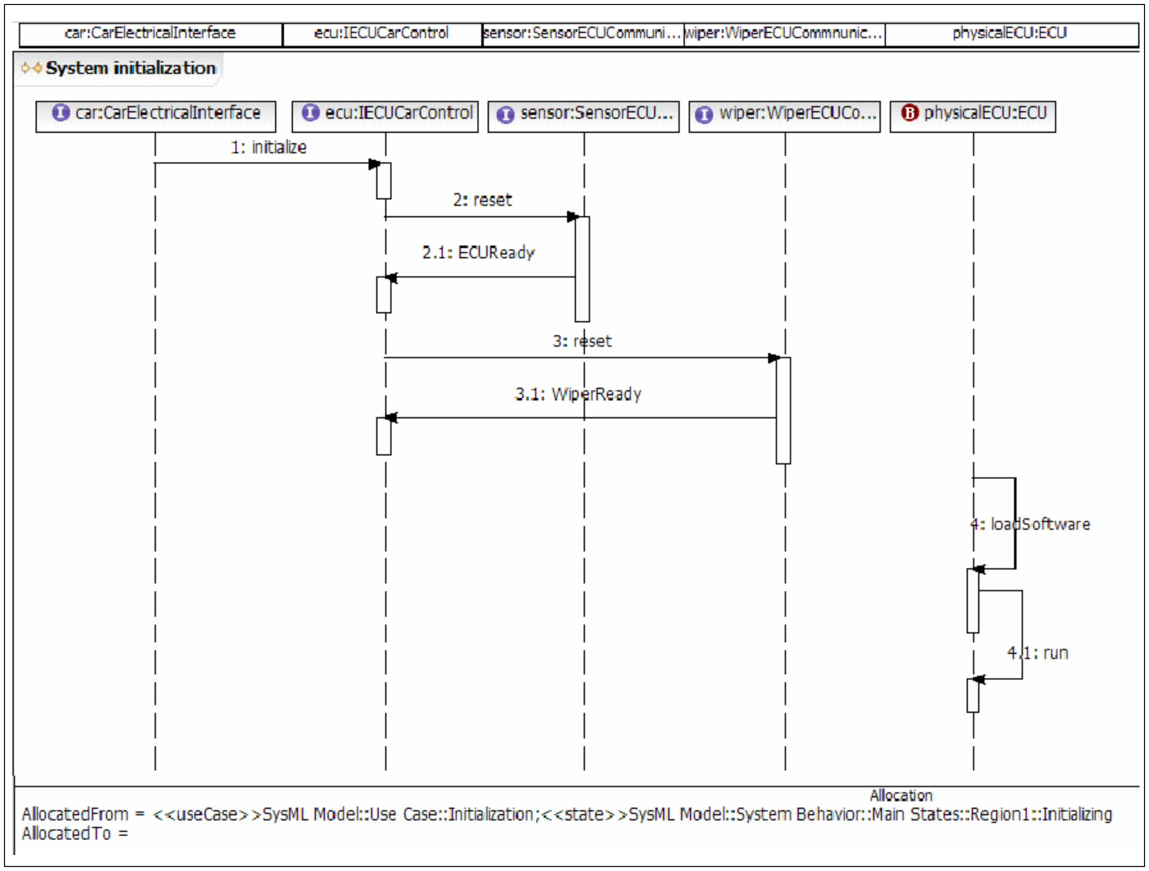
Figure 7-14 Example diagram for range of radar with block and associated requirements

Understanding collaborations

Understanding collaborations basically the same as with UML—in our practice at IBM, we have found sequence diagrams to be most useful for a variety of reasons. We will still use them with SysML, but blocks will play the part of roles along the top of the sequence diagram. Because blocks are classifiers, the result will look the same as with UML.[11]

Figure 7-15 shows an example of an interaction diagram using blocks as roles in the interaction.

[11] For a more detailed discussion of this interaction diagram, see Balmelli's article, cited in footnote 8 on page 160, p. 166



car:CarElectricalInterface
ecu:IECUCarControl
sensor:SensorECUCommuni...
wiper:WiperECUCommnunic...
physicalECU:ECU
System initialization
car:CarElectricalInterface
ecu:IECUCarControl
sensor:SensorECU...
wiper:WiperECUCo...
physicalECU:ECU
1: initialize
2: reset
2.1: ECUReady
3: reset
3.1: WiperReady
4: loadSoftware
4.1: run
Allocation
AllocatedFrom = <<useCase>>SysML Model::Use Case::Initialization;<<state>>SysML Model::System Behavior::Main States::Region1::Initializing
AllocatedTo =

Figure 7-15 SysML interaction diagram

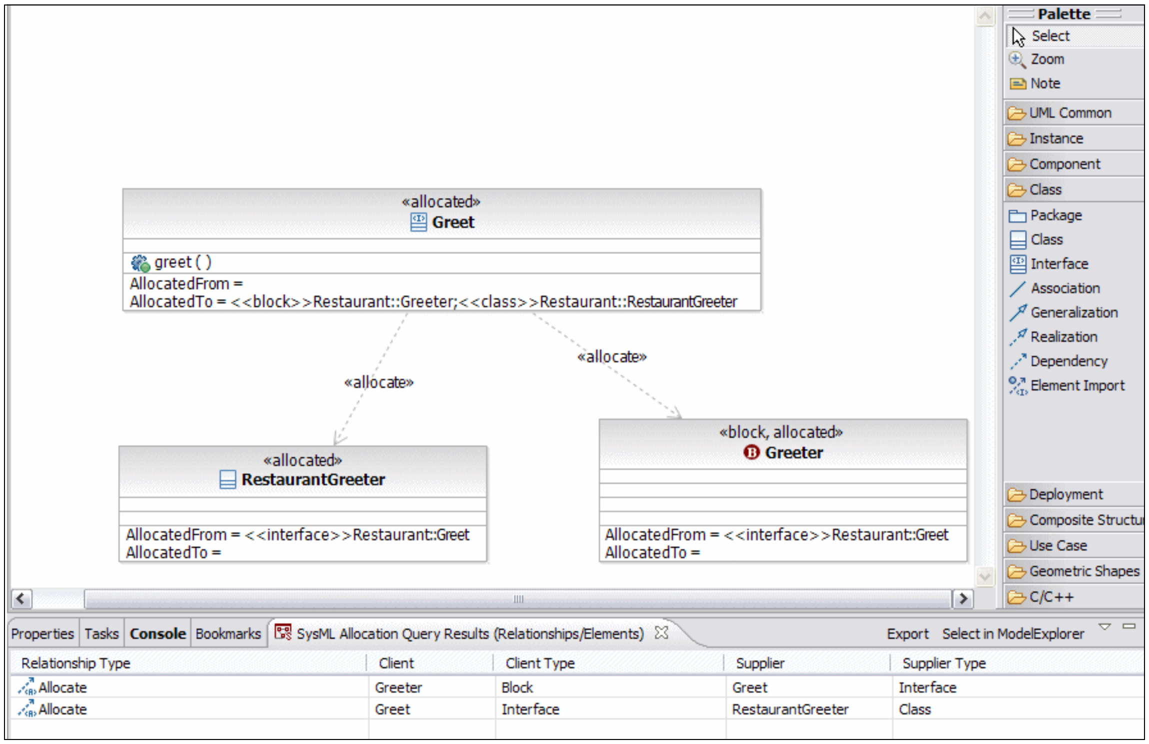
Activity diagrams in SysML provide the ability to represent continuous flow; this could not be done [as well?] in UML. However, we have found that interaction diagrams better express the semantics of collaboration. Additionally, we have found it simpler to extract information automatically from interaction diagrams; nevertheless, we know that activity diagrams are used extensively by major practitioners.[12]

[12] For a more detailed discussion of activity diagram semantics and usage in SysML, see Balmelli's article, cited in footnote 8 on page 160, pp. 167-71

Understanding distribution of responsibilities

Functionality can be distributed to logical elements by discovering what operations a block will provide. This is the same as discovering operations on classes; blocks after all are classifiers.

However, we can also depict the distribution of logical functionality to blocks depicting physical entities as well, or allocate tasks to workers. In Figure 7-16 we show the allocation of a `Greet` interface to both a software class `RestaurantGreet` and a worker block `Greeter`.



Relationship Type	Client	Client Type	Supplier	Supplier Type
Allocate	Greeter	Block	Greet	Interface
Allocate	Greet	Interface	RestaurantGreeter	Class

Figure 7-16 SysML joint allocation diagram

This conveys the semantics of joint realization we discussed in Chapter 5, “Understanding distribution of responsibility” on page 79. As we refine the model, we might discover that we need a realization relationship between at least the interface and the class, if not between the interface and the block as well, as depicted previously (Figure 5-4 on page 87).

Parametrics

Perhaps the most important addition of SysML is the capability it gives us to reason about systems concerns through parametrics, and through its more accurate semantics regarding non-functional and other concerns. This topic needs a book in its own right; we will limit ourselves here to a few illustrative examples that will hopefully demonstrate its power.[13]

Let us look at two examples: restaurant profitability, and radar range.

In “Restaurant ownership” on page 7 we used a restaurant as an example of some of the issues you can reason about with MDSD. Let us take a simplified profitability equation, and diagram it in SysML.

Clearly, profit is generally the difference between revenue and expenses (of all types, including taxes). Let us assume the revenue from the restaurant comes the price of the meals (for simplicity, we consider only meals and not drinks) times the number of meals. Costs are the cost of ingredients, salaries, and rent (or mortgage). See Figure 7-17.

```
profit = revenue - expenses
revenue = number of meals * price of meal
expenses = ((number of meals)*cost of meal (ingredients))
           + salaries + rent
rent = square footage * location factor
```

[13] These examples are drastically over-simplified for pedagogical purposes. For a more detailed discussion of parametrics with examples, see RS Peak, RM Burkhart, SA Friedenthal, MW Wilson, M Bajaj, I Kim (2007) *Simulation-Based Design Using SysML*—Part 1: A Parametrics Primer. International Council on Systems Engineering (INCOSE) Intl. Symposium, San Diego, and RS Peak, RM Burkhart, SA Friedenthal, MW Wilson, M Bajaj, I Kim (2007) *Simulation-Based Design Using SysML*—Part 2: Celebrating Diversity by Example. INCOSE Intl. Symposium, San Diego.

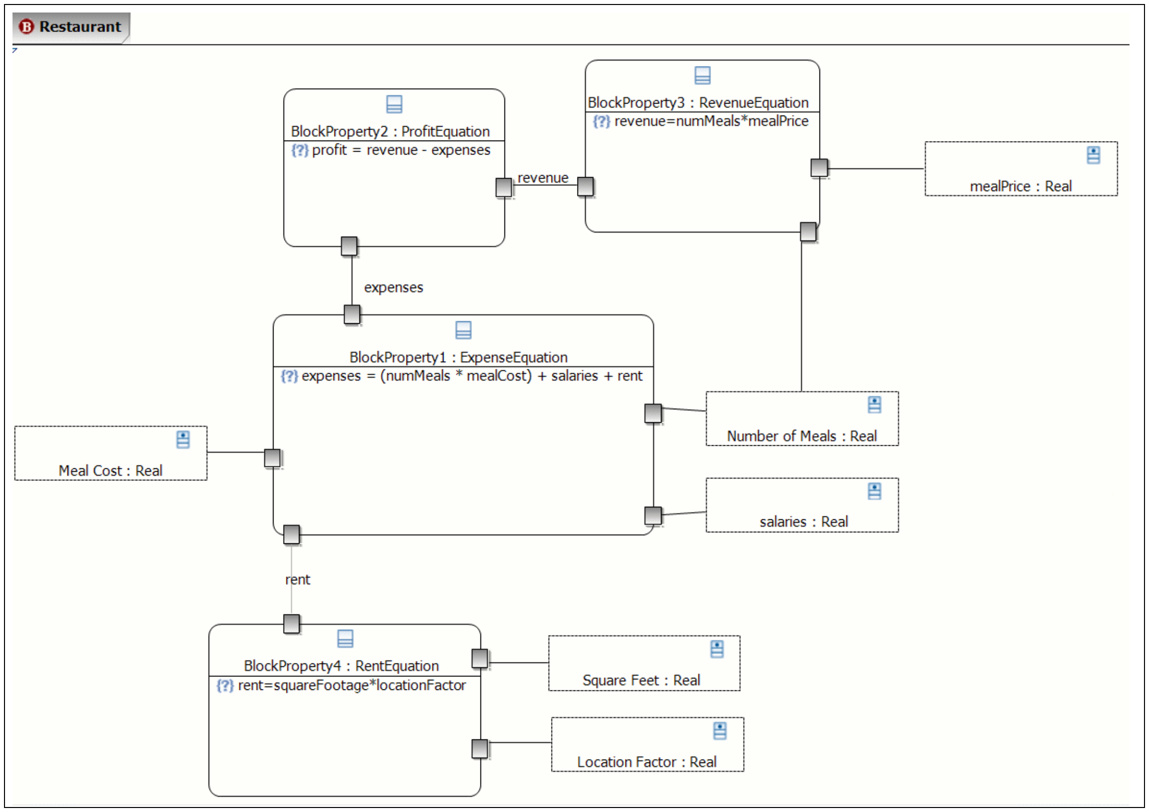

Restaurant
BlockProperty2 : ProfitEquation
{?} profit = revenue - expenses
BlockProperty3 : RevenueEquation
{?} revenue=numMeals*mealPrice
revenue
mealPrice : Real
expenses
BlockProperty1 : ExpenseEquation
{?} expenses = (numMeals * mealCost) + salaries + rent
Number of Meals : Real
Meal Cost : Real
salaries : Real
rent
BlockProperty4 : RentEquation
{?} rent=squareFootage*locationFactor
Square Feet : Real
Location Factor : Real

Figure 7-17 Diagram with restaurant with constraint equation: Profit = Revenue – Costs

We can see from both the equations and the diagram that the number of meals served plays a significant role in the profitability of the restaurant. We want to do further analysis and simulation on how we might increase the number of meals served. We will, however, be constrained by factors such as the size of the restaurant—after a certain point, increasing profit might mean creating new restaurants; you can only constrain salaries by so much; you can only charge what market in your area will bear, if you decrease the quality of your ingredients to reduce meal costs, you risk losing customers, and so on. We can express these in further equations and diagrams, associate data with them, and since the model now is populated with data, we will be able to hook it to a simulation engine.

Let us take another example. If we are building a radar, we will most likely need to consider its range as a requirement as well as its size and weight (Figure 7-18).

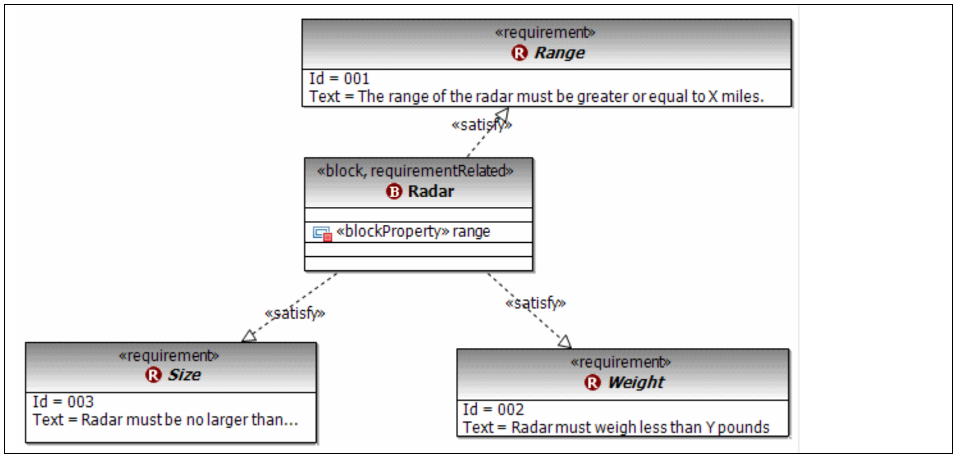

«requirement»
Range
Id = 001
Text = The range of the radar must be greater or equal to X miles.
«satisfy»
«block, requirementRelated»
Radar
«blockProperty» range
«satisfy»
«satisfy»
«requirement»
Size
Id = 003
Text = Radar must be no larger than...
«requirement»
Weight
Id = 002
Text = Radar must weigh less than Y pounds

Figure 7-18 Requirements diagram for radar

A radar consists of both physical and logical components. A generic set of physical components are diagrammed in Figure 7-19.[14]

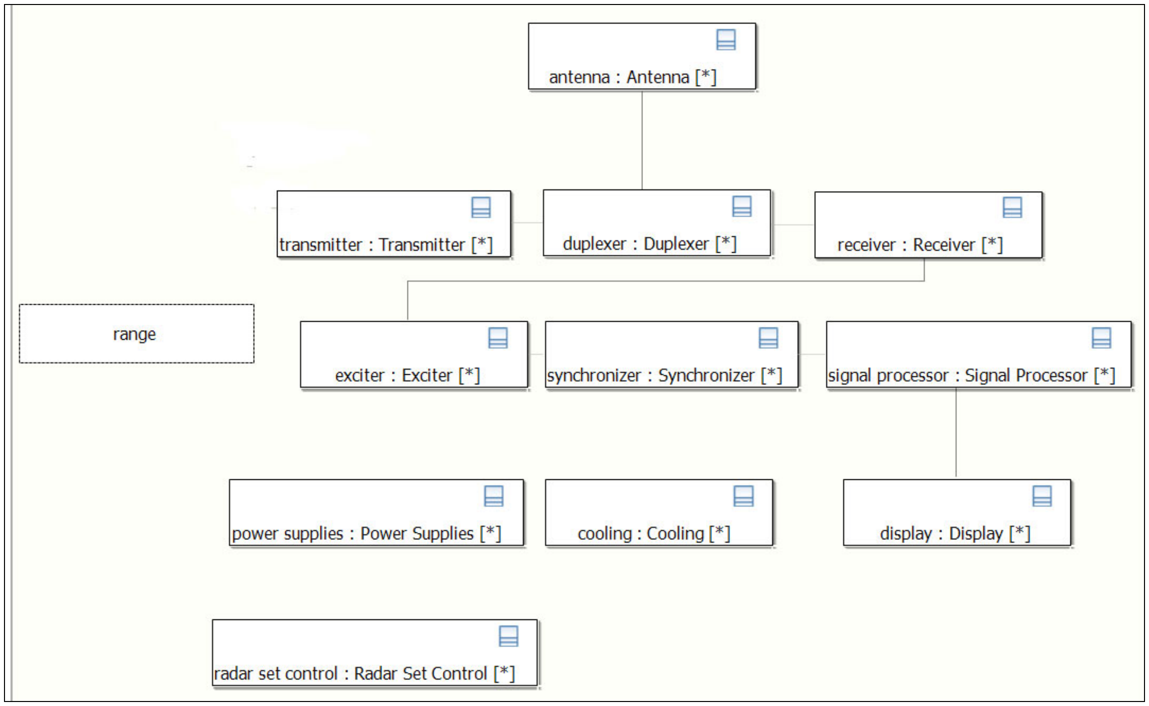

antenna : Antenna [*]
transmitter : Transmitter [*]
duplexer : Duplexer [*]
receiver : Receiver [*]
range
exciter : Exciter [*]
synchronizer : Synchronizer [*]
signal processor : Signal Processor [*]
power supplies : Power Supplies [*]
cooling : Cooling [*]
display : Display [*]
radar set control : Radar Set Control [*]

Figure 7-19 Radar components

[14] Adapted from T.A. Weil, *Transmitters*, in M. Skolnik, *Radar Handbook*, 2nd edition, 1990, pg 4.1

Using a gross simplification of the many factors actually involved, if we increase the power of the signal transmitted, we will increase the range of the radar:

- The useful range of a search radar varies as the fourth root of the product of average radio frequency (RF) power, antenna aperture area (which determines antenna gain), and the time allowed to scan the required solid angle of coverage (which limits how long the signal in each direction can be collected and integrated to improve signal-to-noise ratio):

  ```
  R^4 [varies by] P x A x T
  ```

- The range varies as the fourth root of power because both the outgoing transmitted power density and the returning echo energy density from the target become diluted as the square of the distance traveled. Trying to increase range by increasing transmitter power is costly: A 16-fold increase in power is needed to double the range. Conversely, negotiating a reduced range requirement can produce remarkable savings in system cost.[15]

So an increase in power will almost certainly mean an increase in the size, weight, and cost of that which produces the power of the signal—the transmitter, and ultimately the power supply:

- The transmitter is usually a large fraction of radar system cost, size, weight, and design effort, and it typically requires a major share of system prime power and maintenance. It generally ends up being a big box that sits in the corner of the radar equipment room, hums to itself, and has a big sign on it that says Danger, High Voltage; so most people prefer to keep away from it.[16]

Most of this is clear from the equations and the text just cited, but perhaps a diagram can reinforce these conclusions. So if we create a diagram that illustrates the relationships, we can obtain a better understanding of the design issues (Figure 7-20).

[15] T.A. Weil, *Transmitters*, in M. Skolnik, *Radar Handbook*, 2nd edition, 1990, pg 4.2. His equation is a simplification of the radar range equations discussed in chapter 2 of the handbook.
[16] T.A. Weil, *Transmitters*, in M. Skolnik, *Radar Handbook*, 2nd edition, 1990, pg 4.3

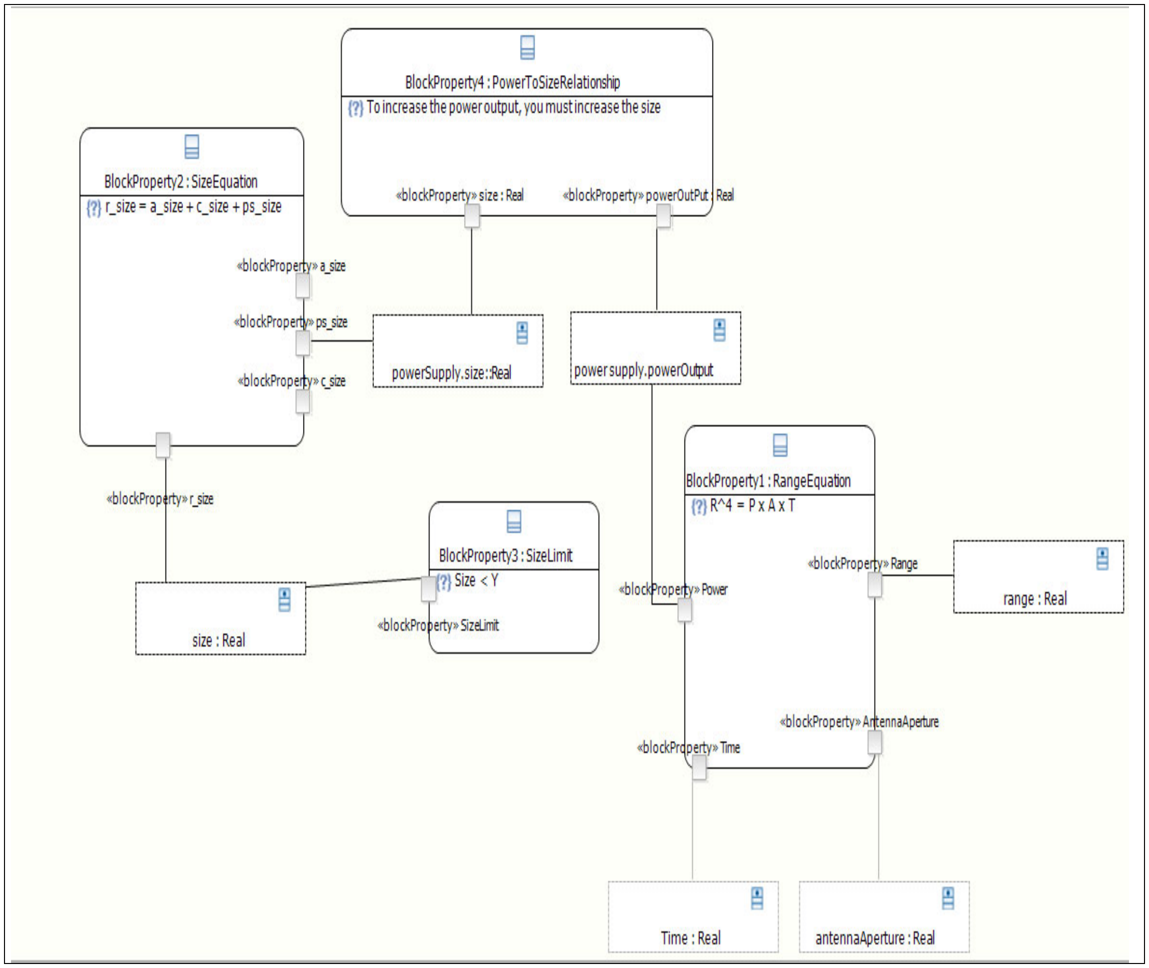

BlockProperty4 : PowerToSizeRelationship
{?} To increase the power output, you must increase the size
«blockProperty» size : Real
«blockProperty» powerOutPut : Real
BlockProperty2 : SizeEquation
{?} r_size = a_size + c_size + ps_size
«blockProperty» a_size
«blockProperty» ps_size
«blockProperty» c_size
powerSupply.size::Real
power supply.powerOutput
«blockProperty» r_size
BlockProperty1 : RangeEquation
{?} R^4 = P x A x T
BlockProperty3 : SizeLimit
{?} Size < Y
«blockProperty» SizeLimit
«blockProperty» Power
«blockProperty» Range
range : Real
size : Real
«blockProperty» AntennaAperture
«blockProperty» Time
Time : Real
antennaAperture : Real

Figure 7-20 Simplified Radar Power and Range parametric diagram

This diagram focuses only on power from the power supply and its relationship to overall radar size and range. It is simplified, but that is one of the things we should do with modeling—emphasize salient data to illustrate a point. We could draw other parametric diagrams focussing on other aspects of the pertinent equations.

Furthermore, by providing a means for including this information explicitly in the model, we open the possibility of hooking the model to other reasoning/analytical tools at our disposal. So if we are constrained by weight, size, or amount of radiation we can produce, we can include these constraints in the model, instantiate some values, and be warned if we violate constraints. In such a simplified example as this, such warning might not be all that useful; after all, we can do the math simply enough and see when the power/size relationship violates a constraint; but in a more complex set of constraints we might want to set up the constraint network and allow a constraint solver to warn us when one of the constraints is violated.[17]

We can see then, that parametrics in SysML provide us powerful capabilities for reasoning about non-functional requirements and systems concerns that are not available in UML, and provide us with semantics for modeling systems engineering concerns.

Summary of SysML basics

In this chapter we have discussed some of the different capabilities that SysML offers to system engineers and product designers. SysML is aimed at supporting the conceptual stage of the life cycle of the product. This stage is preceded by the decomposition of the customer needs into product features. We have seen that SysML allows the representation of these features as requirements in the model. In turn, these requirements can be allocated to the use cases, to the sub-systems and components (whether functional or physical) identified for the product.

- The conceptual stage requires the specification of the various sub-systems and the need for details depends on their level of integration. SysML provides a set of constructs to support the description of the structure of the product. Blocks are used to model sub-systems and components, and ports support the description of their interfaces. Dependencies (for example, analytical) between structural properties are expressed using constraints and represented using the parametric diagram.
- In addition to structure, the conceptual stage should clarify how the product behavior is expressed through the interaction of its components. For example, behavior modeling gives a detailed description of the product use cases. SysML provides three means for explicating the product behavior, namely interactions, state machine and activities. These three mechanisms are built as a unified behavior concept and can consequently be orchestrated in a single, uniform and complex behavior model for the whole product.
- A complex product model is form by several sub-models of different nature (for example, requirements, blocks, constraints, activities). SysML provides a mechanism to relate different aspects of the model and to enforce traceability across it.

[17] A simple example of how this can be done is provided by S.V. Hovater in *Implementing a domain-specific constraint in IBM Rational Systems Developer*, IBM developerWorks, `http://www.ibm.com/developerworks/rational/education/dw-rt-rsdconstraint/`

- The conceptual stage precedes the detailed elaboration of the components within the different engineering disciplines. Therefore, the conceptual design plays many central roles in the product life cycle, Next, we emphasize some of the most important ones, in our opinion.
- The formal description of the product at an early stage of the life cycle improves the understanding of the product requirements and how they answer the customer needs. The allocation of requirements to the model elements ensures that these needs are covered and provides a rationale for the engineer in charge of fulfilling these requirements. The rationalization of the design is therefore a **communication tool** spanning organizational levels and life cycle stages. It improves communication across teams, between teams (think of the different engineering disciplines) and between teams and decision makers. It uses a generic language (in the sense that it is not specific to any engineering discipline) that accommodates the incremental detailing of the product representation. That last aspect allows coping with organizational levels. Note that such a formal description is well suited to methodologies.
- The SysML model provides an electronic representation of the product that is leveraged as a **decision tool**. Trade-off studies are performed by evaluating functions on the model (cost function, estimation of the integration effort). At an early stage in the life cycle, often rough estimations are used, hence the model need not necessarily have a great amount of detail in order to be used efficiently. When details are added, or artifacts (for example, sub-system simulations) are produced by detailed engineering, the model is used to orchestrate the various simulations and perform requirement verification. Hence the SysML model is an evolving decision tool available throughout the whole life cycle of the product, and not only at the conceptual stage.
- The product model represents abstractions of artifacts that are progressively elaborated throughout the life cycle. These artifacts are distributed across the engineering disciplines participating to the design. Hence the model forms a traceability scaffold that provides a means to measure the development progress, perform change impact analysis, and manage dependencies between processes and the produced artifacts. The SysML model is therefore a **management and integration tool** for the stakeholders.

8

Conclusion

In this chapter we recapitulate why we build systems and how systems engineering and MDSD fit into this process.

Why we build systems

Building systems is a huge, complex, expensive and risky proposition. But, when we take a risk, manage it well, and overcome it, the rewards can be great.

There are a broad set of concerns that drive the development of any system. In the end, we want to improve our situation in the world—we want to transform the world for the better—however we define better. In essence, we want to gain something from our investments—we take risks for precisely the same reason—we hope we will gain something (or perhaps, be able to give something) from having taken the risk.

We want systems to do something for us, with a return that justifies the risk and expense we take to build the system. We want the system to perform, within a set of cost and risk constraints, that is, we want it to provide value that exceeds the cost and risk of building and maintaining it.

Systems engineering

The job of the systems engineer, and that of systems engineering, is to ensure that we are successful in this endeavor. Consider the International Council on Systems Engineering (INCOSE) definition of systems engineering:

What is systems engineering?

Systems engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem:

Operations	Cost & Schedule
Performance	Training & Support
Test	Disposal
Manufacturing	

Systems engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems Engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs.

In this meaning, system engineering consists of understanding as much as possible the stakeholder concerns, capturing those concerns into a consistent set of requirements, and then specifying a set of system components (hardware, software, worker instructions) that, when integrated meet the requirements. These stakeholder concerns are usually broader than those than can be met by hardware or software alone, for example, total cost of ownership, or mean time to recovery. System engineering requires the ability to address a very wide set of concerns with an elegant system design.

MDSD is meant to provide the means to achieve this elegant design.

Systems concerns

As is clear from INCOSE's definition, there is a wide variety of concerns that must be met to ensure the success of a system.

It is useful to make a distinction between *concerns* and *requirements*. Briefly:

- Concerns are issues that matter to the stakeholders.
- Requirements are a transformation of the concerns into a specification that can serve as a basis for architecting the system.

Let us briefly consider concerns. As stated above, there are many of them, and different kinds of them. Consider these items as a starting set (to be added to, or merged with the set implied in INCOSE's definition), as shown in Table 8-1.

Table 8-1 System concerns

Main concern	Subordinate concern	
Domain concerns	Security	► Data integrity
	Safety	► Physical ► Predators [?]
Cost concerns	Development	
	Fielding	► Serviceability (patches, repairs, hot swap ► Operating (see also Operational) ► Maintainability, extensibility ► Training, adoption
	Retirement/Disposal	

Main concern	Subordinate concern	
Value	Operational	► Availability ► Throughput ► Capacity ► Reliability
	Usability, human factors	
	Responsiveness	
	Functionality	

Each of these is worth discussing in detail; we will not, however, do so in this context.

Given this broader set of concerns, we need to transform them into requirements, and then transform the requirements into an architecture.

How do we do this? With MDSD.

How does MDSD fit in?

As we have discussed, MDSD consists of a set of transformations that progressively refine our knowledge, requirements, and design.

Following the MDSD process, we move from concerns to requirements to architecture. Hopefully this architecture allows us to provide value that can be measured against the cost and risk of producing it; a value that meets the concerns of the stakeholders.

We start with concerns—sometimes vague, amorphous, and likely contradictory. We start with understanding the context of the system. From concerns, we derive a set of black box systems requirements, both functional and nonfunctional.

Black box system requirements drive the architecture of the system. We find these requirements by understanding the system in its context, and by transforming concerns into requirements.

A black-box representation of the system has a set of functional requirements, constraints on those functional requirements, and constraints on the system itself.

We measure the effectiveness of the behaviors or functional requirements against the goals and larger concerns of the system, at the same time ensuring that the constraints are met.

The goals, constraints, and desired behaviors drive the system architecture. We postulate an architecture (or set of architectures) and then design and test against the goals.

Next, we analyze collaborations.

MDSD suggests that a breadth-first collaboration based approach across multiple viewpoints will be more effective than a traditional depth-first functional decomposition in creating an architecture that will not only meet the requirements, but will prove to be more resilient in the face of inevitable change.

We can analyze collaboration both from a black-box and a white-box perspective. Having gained an understanding of the system's context, we postulate an architecture, a structure or set of structures, that will realize the system's requirements. We break open the black box, and look at the system as a white box (yet another transformation). We decompose the system into pieces, understand how the pieces work together to meet the black-box requirements, thereby deriving requirements on the pieces. Through all of this, we integrate, refine, and refactor as we go, seeking to provide resiliency and avoid brittleness. The collaboration seeks to realize requirements, which have been formulated from the larger set of concerns.

MDSD also seeks to provide an effective distribution of responsibilities across resources—joint realization and abstractions such as localities provide an effective and elegant way of accomplishing this.

Finally, the ability to attach attributes and values to modeling entities and the parametric capabilities of SysML allow us to provide a basis for doing simulations or other models to meet cost, risk, and other concerns. While we have only touched upon this concept in this publication, it is clearly a future direction that we look forward to developing.

A

MDSD use case specification template

This appendix provides a use case specification template. The actual template is a Microsoft® Word document, available through the **Additional Material** download associated with this document. Refer to Appendix B, "Additional material" on page 193 for instructions on how to access the additional material.

Use Case Specification

<Project Name>

<Sub-Project Name>

Version <0.4>

20-Oct-07

[Note: The following template is provided for use with the Rational Unified Process. Text enclosed in square brackets and displayed in blue italics (style=InfoBlue) is included to provide guidance to the author and should be deleted before publishing the document. A paragraph entered following this style will automatically be set to normal (style=Normal Spaced).]

*[To customize automatic fields in Microsoft Word (which display a gray background when selected), select **File** → **Properties** and supply the Project Name for the Title Property on the Summary tab. Then replace the Project Name, Sub-Project Name and Document Version fields on the Custom tab with the appropriate information for this document. After closing the dialog box, automatic fields can be updated throughout the document by selecting **Edit** → **Select All** (or Ctrl-A) and pressing F9, or simply click on the field and press F9. This update action must be done separately for Headers and Footers. Alt-F9 will toggle between displaying the field names and the field contents. See Word help for more information on working with fields.]*

[The document version number should start at 0.1 for a given product version. Each update of a draft version will increment the minor version number (decimal place). The first baseline (signed) version of the document should be numbered 1.0, then draft updates to it 1.1, and so forth. Subsequent baselines or signed updates will increment the major version number (integer).]

<Project Name> <Sub-Project Name>	Document Version <0.1>
Use Case Specification	Date: 20-Oct-07
Template Name: UseCaseSpecification	Template Version: 0.1

Revision History

Date	**Version**	**Description**	**Author(s)**
			<Project Team Members>
		<add additional rows as necessary>	

Document Approval

Date	**Approved/ Rejected**	**Approved/Rejected By**	**Signature (indicate if electronic approval)**
		<Name> *<Role>*	

<*indicate if* Confidential>	copyright <COMPANY>, 2007	Page 1 of 9

<Project Name> <Sub-Project Name>	Document Version <0.1>
Use Case Specification	Date: 20-Oct-07
Template Name: UseCaseSpecification	Template Version: 0.1

Table of Contents

<*indicate if* Confidential>	copyright <COMPANY>, 2007	Page 2 of 9

Use-Case Specification: <Use-Case Name>

[The following template is for a Use-Case Specification, which is a verbal description of the use case. This document is used with a requirements management tool, such as Rational RequisitePro, for specifying and marking the requirements within the use-case properties.

The use-case diagrams can be developed in a visual modeling tool, such as Rational Rose®. A use-case report, with all properties, can be generated with Rational SoDA®. For more information, see the tool mentors in the Rational Unified Process.

Name Use Cases with an active voice verb phrase.]

1 Brief Description

[The description briefly conveys the role and purpose of the use case. A single paragraph will suffice for this description.]

2 Actor Catalog

[This section lists the actors involved in this use case and briefly notes their role in the use case. Note that these actors are also shown in the Use Case diagram.]

#	Actor Name	Brief Description of Actor

3 Preconditions

[A precondition of a use case is the state of the system that must be present prior to a use case being performed.]

3.1 < Precondition One >

4 Postconditions

[A postcondition of a use case is a list of possible states the system can be in immediately after a use case has finished.]

4.1 < Postcondition One >

5 Basic Flow of Events

[This use case starts when an actor requests that the system do something. An actor always initiates use cases. The use case describes what the actor does and what the system does in response. It is phrased in the form of a dialog between the actor and the system.

The use case describes the interaction between the system and the actors. If information is exchanged, be specific about what is passed back and forth. For example, it is not very illuminating to say that the actor enters customer information if it is not defined. It is better to say the actor enters the customer's name and address. The Domain Model is essential to keep the complexity of the use case manageable—things like customer information are described there to keep the use case from drowning in details.

Alternate flows must be described in the Alternative Flow subsection. Alternate flows must end with either "the use case ends" or "return to [a step in a flow]."

Complex flows of events should be further structured into sub-flows. In doing this, the main goal should be improving the readability of the text. Subflows can be re-used in many places. Remember that the use case can perform subflows in optional sequences or in loops or even several at the same time.

A picture is sometimes worth a thousand words, though there is no substitute for clean, clear prose. If it improves clarity, feel free to paste flow charts, activity diagrams or other figures into the use case. If a flow chart is useful to present a complex decision process, by all means use it! Similarly for state-dependent behavior, a state-transition diagram often clarifies the behavior of a system better than pages upon pages of text. Use the right presentation medium for your problem, but be wary of using terminology, notations or figures that your audience might not understand. Remember that your purpose is to clarify, not obscure.

Flow of Event Formats

There are two possible formats for flows of events. A basic numbered list can be used, such as:

1. *The use case begins when…*
2. *…*
3. *…*
4. *… and the use case ends.*

If the use case comes from a system-of-systems operation handed down from the level above, for example, and enterprise operation handed down to become a system level use case, then the flow of events should be expressed as an operation specification, including both white and black box perspectives, using the following table format:

<Project Name> <Sub-Project Name>	Document Version <0.1>
Use Case Specification	Date: 20-Oct-07
Template Name: UseCaseSpecification	Template Version: 0.1

Main Flow

Actor Action	*Black Box Step*	*White Box Step*	*White Box Budgeted Requirements*	*Locality*	*Process*

]

6 Alternative Flows

[Alternatives are described in this section, referred to in the Basic Flow subsection of Flow of Events section. Think of the Alternative Flow subsections like alternative behavior— each alternative flow represents alternative behavior usually due to exceptions that occur in the main flow. They can be as long as necessary to describe the events associated with the alternative behavior.

Start each alternative flow with an initial line clearly stating where the alternative flow can occur and the conditions under which it is performed.

End each alternative flow with a line that clearly states where the events of the main flow of events are resumed. This must be explicitly stated.

Using alternative flows improves the readability of the use case. Keep in mind that use cases are just textual descriptions, and their main purpose is to document the behavior of a system in a clear, concise, and understandable way.

Be sure to find and describe ALL of the alternate flows.]

<*indicate if* Confidential>	copyright <COMPANY>, 2007	Page 6 of 9

<Project Name> <Sub-Project Name>	Document Version <0.1>
Use Case Specification	Date: 20-Oct-07
Template Name: UseCaseSpecification	Template Version: 0.1

6.1 <Area of Functionality>

[Often there are multiple alternative flows related to a single area of functionality (for example specialist withdrawal facilities, card handling or receipt handling for the Withdraw Cash use case of an Automated Teller Machine). It improves readability if these conceptually related sets of flows are grouped into their own clearly named sub-section.]

6.1.1 < <n><a> First Alternative Flow >

[Describe the alternative flow, just like any other flow of events. Alternates are numbered according to guidelines in the Use Case Checklist.

Like the main flow, alternate flows can be expressed in a numbered list of steps:

1.If <condition> then …
2.…
3.…
4.… and the use case returns to … <or ends>.

Or, if using the operation specification style, a table can be used. Note the guard condition at the top of the table specifying the condition under which the alternate occurs.]

Alternate Flow
[guard condition]

Actor Action	*Black Box Step*	*White Box Step*	*White Box Budgeted Requirements*	*Locality*	*Process*

<*indicate if* Confidential>	copyright <COMPANY>, 2007	Page 7 of 9

<Project Name> <Sub-Project Name>	Document Version <0.1>
Use Case Specification	Date: 20-Oct-07
Template Name: UseCaseSpecification	Template Version: 0.1

6.1.1.1 < An Alternative Subflow >

[Alternative flows can, in turn, be divided into subsections if it improves clarity. Only place subflows here if they are only applicable to a single alternative flow.]

6.1.2 < <n><b> Second Alternative Flow >

[There can be, and most likely will be, a number of alternative flows in each area of functionality. Keep each alternative flow separate to improve clarity.]

6.2 <Another Area of Functionality>

[There can be, and most likely will be, a number of areas of functionality giving rise to sets of alternative flows. Keep each set of alternative flow separate to improve clarity.]

6.2.1 < <nn><x> Another Alternative Flow >

7 Subflows

7.1 <S1 First Subflow >

[A subflow should be a segment of behavior within the use case that has a clear purpose, and is "atomic" in the sense that you do either all or none of the actions described. You might need to have several levels of sub-flows, but if you can you should avoid this as it makes the text more complex and harder to understand.]

7.2 < S2 Second Subflow >

[There can be, and most likely will be, a number of subflows in a use case. Keep each sub flow separate to improve clarity. Using sub flows improves the readability of the use case, as well as preventing use cases from being decomposed into hierarchies of use cases. Keep in mind that use cases are just textual descriptions, and their main purpose is to document the behavior of a system in a clear, concise, and understandable way.]

<*indicate if* Confidential>	copyright <COMPANY>, 2007	Page 8 of 9

8 Extension Points

[Extension points of the use case.]

8.1 <Name of Extension Point>

[Definition of the location of the extension point in the flow of events.]

9 Special Requirements

[A special requirement is typically a nonfunctional requirement that is specific to a use case, but is not easily or naturally specified in the text of the use case's event flow. Examples of special requirements include legal and regulatory requirements, application standards, and quality attributes of the system to be built including usability, reliability, performance or supportability requirements. Additionally, other requirements—such as operating systems and environments, compatibility requirements, and design constraints—should be captured in this section.

Requirements listed in this section should also be stored in RequisitePro using the requirement type SSR (Software Supplementary Requirement), and should be traced to another project or program-level SSR in the Supplementary Specification.)]

9.1 < First Special Requirement >

10 Additional Information

[Include, or provide references to, any additional information required to clarify the use case. This could include overview diagrams, examples or any thing else you fancy.]

B

Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG247368`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the Redbooks form number, SG247368.

Using the Web material

The additional Web material that accompanies this book includes the file **MDSDUseCaseSpecification.doc**, which is the use case specification template described in Appendix A, “MDSD use case specification template” on page 181.

The MDSD plug-in, mentioned in “Installing the MDSD plug-in” on page 114, can be obtained from Tim Bohn, `tbohn@us.ibm.com`.

Abbreviations and acronyms

BB	black-box
BDD	block definition diagram
CONOPS	concept of operations
ECU	electronic control unit
GPS	global positioning system
I/O	input/output
IBD	internal block diagram
IBM	International Business Machines Corporation
INCOSE	International Council on Systems Engineering
IT	information technology
ITSO	International Technical Support Organization
JRT	joint realization table
MDD	model-driven development
MDSD	model-driven systems development
NFR	nonfunctional requirements
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
OOSE	object-oriented software engineering
PD	parametric diagram
PDE	product data exchange
RF	radio frequency
RFP	request for proposal
RMC	Rational Method Composer
RSW	rain sensing wiper
RUP	Rational Unified Process
SE	systems engineering
SKU	stock-keeping unit
SSR	software supplementary requirement
STP	software test plan
UML	Unified Modeling Language
WB	white-box

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks publications

For information about ordering these publications, see “How to get IBM Redbooks” on page 198. Note that some of the documents referenced here might be available in softcopy only.

- *Building SOA Solutions Using the Rational SDP*, SG24-7356
- *Patterns: Model-Driven Development Using IBM Rational Software Architect*, SG24-7105
- *Rational Application Developer V7 Programming Guide*, SG24-7501
- *Building Service-Oriented Banking Solutions with IBM Banking Industry Models and Rational SDP*, REDP-4232
- *Rational Business Driven Development for Compliance*, SG24-7244
- *Software Configuration Management: A Clear Case for IBM Rational ClearCase and ClearQuest UCM*, SG24-6399
- *The IBM Rational Unified Process for System z,* SG24-7362-00

Other publications

These publications are also relevant as further information sources:

- *Object-Oriented Design and Analysis with Applications*, Booch et al, 3rd Edition, Addison-Wesley, 2007, ISBN 020189551X
- *Software Project Management: A Unified Framework,* Walker Royce, Addison Wesley, 1998, ISBN 0201309580
- *Managing Iterative Software Development Projects*, Kurt Bittner and Ian Spence, Addison-Wesley, 2006, ISBN 032126889X
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Martin Fowler, Addison-Wesley, 2003, ISBN 0321193687
- *The Unified Modeling Language Reference Manual*, James Rumbaugh, Ivar Jacobson, Grady Booch, 2nd edition, Pearson, 2004, ISBN 0321245625

- *The Unified Modeling Language User Guide*, Grady Booch, James Rumbaugh, Ivar Jacobson, 2nd edition, Addison-Wesley, 2005, ISBN 0321267974
- *Systems Engineering and Analysis*, Benjamin S. Blanchard and Wolter J. Fabrycky, Prentice Hall, 1998, ISBN 0131350471

Online resources

These Web sites are also relevant as further information sources:

- IBM Rational Web site:

  `http://www.ibm.com/software/rational/`
- IBM developerWorks Rational:

  `http://www.ibm.com/developerworks/rational`
- OMG and OMG SysML:

  `http://www.omg.org`
  `http://www.omgsysml.org/`

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

IBM Redbooks

Model Driven Systems Development with Rational Products

(0.2”spine)
0.17”<->0.473”
90<->249 pages

Model Driven Systems Development with Rational Products

Understanding context

Understanding collaborations

Understanding distribution of responsibilities

This IBM Redbooks publication describes the basic principles of the Rational Unified Process for Systems Engineering, which is IBM Rational's instantiation of model-driven systems development (MDSD).

MDSD consists of a set of transformations that progressively refine knowledge, requirements, and design of complex systems. MDSD begins with activities and artifacts meant to promote an understanding of the system's context.

Requirements problems often arise from a lack of understanding of context, which, in MDSD, means understanding the interaction of the system with entities external to it (actors), understanding the services required of the system, and understanding what gets exchanged between the system and its actors. Managing context explicitly means being aware of the shifts in context as you go from one model or decomposition level to the next.

MDSD suggests that a breadth-first collaboration based approach across multiple viewpoints is more effective than a traditional depth-first functional decomposition in creating an architecture that will not only meet requirements, but will prove to be more resilient in the face of inevitable change. MDSD also seeks to provide an effective distribution of responsibilities across resources. Joint realization and abstractions such as localities provide an effective and elegant way of accomplishing this.

Finally, the ability to attach attributes and values to modeling entities and the parametric capabilities of SysML provide a basis for doing simulations or other models to meet cost, risk, and other concerns.

SG24-7368-00 ISBN 0738485683