

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

▶ Rational Unified Process for Systems Engineering

Part 1: Introducing RUP SE Version 2.0

by [Murray Cantor](#)

Principal Engineer
Rational Brand Services
IBM Software Group

Editor's Note:

The IBM Rational Unified Process,[®] or RUP,[®] offers software development organizations a framework for streamlining all activities related to the development lifecycle. Since its formal debut in 1996, RUP has evolved to support a variety of development requirements, including "systems engineering," or SE. In 2001, the first RUP Plug-In to support systems engineering was proposed by Rational Software's strategic services organization. The RUP SE v1 Plug-In was made generally available in 2002, and it continues to be supported by the newly created IBM Rational brand services team.



Outlining a vision for the next generation of RUP SE, Murray Cantor begins a series of three articles this month; we will continue this series through the September and October issues of *The Rational Edge*. **While these articles are consistent with the current RUP SE Plug-In, they introduce a few extensions to the process framework. Please note that the currently available RUP SE Plug-In is RUP SE v1,** downloadable from [Rational Developer Network](#) (authorization required).

Introduction

This article provides an overview of the latest evolution of Rational Unified

Process for Systems Engineering,[®] or RUP SE.[®] RUP users should note that the currently available RUP Plug-In for SE is the RUP SE v1 Plug-In, which was made available in 2002.

RUP SE is an application of the Rational Unified Process, or RUP, software engineering process framework. RUP SE supports the development of large-scale systems composed of software, hardware, workers, and information components. RUP SE includes an architecture model framework that enables the consideration of a set of different perspectives (logical, physical, information, etc.) in order to deliver a solution that addresses the concerns of various development stakeholders. A distinguishing characteristic of RUP SE is that the requirements for these different sorts of components are jointly derived in increasing specificity from the overall system requirements.

RUP SE addresses projects that:

- *Are large enough to require multiple teams with concurrent development.*
 - *Have concurrent hardware and software development.*
 - *Have architecturally significant deployment issues.*
- OR
- *Include a redesign of the underlying information technology infrastructure to support evolving business processes.*

RUP SE is delivered as a RUP Plug-In. This article contains a few concepts that, at this writing, have not yet been included in the Plug-In. Nevertheless, these concepts are presented here in order to provide our latest and best understanding of how to meet the needs of systems development.

*We will begin with a discussion of systems and the challenges facing the modern systems developer. This discussion is followed by the design points of RUP SE -- that is, how it addresses the challenges of systems development. The next two sections introduce the RUP SE UML-based modeling and requirement specification techniques as well as the use of Unified Modeling Language (UML) semantics. The first section, **System Specification**, provides a complete blackbox description of the system. Part II, to be published next month, will focus on system architecture and introduce the RUP SE architecture framework, which describes the internals of the system from multiple viewpoints. Part III, in October, will cover requirements analysis and flowdown, an introduction to the method for deriving requirements and specification for the elements of the RUP SE framework. This will include a description of the Joint Realization Method, a novel technique for jointly deriving specification of architectural elements across multiple viewpoints. Part III will also include a discussion of RUP SE programmatics.*

Terminology and concepts in systems development

By a *system*, we mean a set of resources that provide services that are used by an enterprise to carry out a business purpose¹ or mission. System components typically consist of hardware, software, data, and workers. Systems are specified by the services they provide, along with other non-behavioral requirements such as reliability or cost of ownership. Designing a system consists of specifying components, their attributes, and their relationships.

System is one of those words that have a set of different, if related, meanings² that can cause confusion when used to discuss technical development. Generally, a system is a set or assemblage of elements that exhibit behavior collectively. All of the definitions found in the systems engineering literature build on this idea. In addition to the definition of system we mentioned above, here are some additional definitions of *systems* and *systems engineering*:

From the International Council on Systems Engineering³

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*
- *Performance*
- *Test*
- *Manufacturing*
- *Cost & Schedule*
- *Training & Support*
- *Disposal*

From Mil-STD-499A

Engineering Management -- The management of the engineering and technical effort required to transform a military requirement into an operational system. It includes the system engineering required to define the system performance parameters and preferred system configuration to satisfy the requirement, the planning and control of technical program tasks, integration of the engineering specialties, and the management of a totally integrated effort of design engineering, specialty engineering, test engineering, logistics engineering, and production engineering to meet cost, technical performance and schedule objectives.

From *The Art of Systems Architecting* by Maier and Richtin⁴

...systems are collections of different things which together produce results unachievable by the elements alone.

Examples of systems include

- **Information technology** systems consisting mainly of software, computer hardware and peripherals, and workers.
- **Products** such as airplanes, satellites, automobiles, and telephone switches consisting of hardware and software.
- **Enterprises** that provide services or carry out a mission. These enterprises may consist of workers enabled by hardware and software. Often these enterprises cross organizational boundaries.

All of these definitions suggest that systems can be viewed from two different perspectives:

- **Blackbox** perspective: The system as a whole: the services it provides and the requirements it meets.
- **Whitebox** perspective: The elements or parts that make up the system.

Which of these two perspectives we use in viewing, understanding, and using a system depends on the needs involved. The problem of systems engineering is to design and implement a system that meets the needs of all system stakeholders, including

- **Users** who are concerned with functionality and performance.
- **Owners** who are concerned with cost of deployment and ownership.
- **Investors** who are concerned with competitive advantage in the market or mission space.

RUP SE provides mechanisms and a UML-based model framework to support teams of systems engineers as they determine the blackbox view of the system and specify an optimal whitebox system design that meets all stakeholder needs.

Modern system challenges

Systems engineering⁵ was identified as a discipline in the mid-twentieth century.⁶ In the early days, systems engineers were challenged to design and specify complex, standalone entities that provided unprecedented capabilities. Some early successes of the systems approach include the Athena Rocket, NORAD, and the Apollo program. These days, systems are not only expected to provide the right set of features; they are also expected to meet a new set of challenges.

Of course, early systems engineering projects presented significant

challenges of their own. For example, the launch of the first imaging satellite represented a great technical accomplishment. First, the development team for that project had to analyze what capabilities were required for this unprecedented system to meet user needs. Then, they had to figure out how to use existing technology to image a structure on the ground, store the image long enough to download it to a ground station, and then make it accessible from an analyst's workstation. At that time, it was of little concern that the solution required dedicated resources. Providing the capability itself was enough to meet stakeholder needs.

Today, those designing satellite imaging systems are concerned less with providing the basic capability and more with optimizing the system to meet a broader set of stakeholder needs. For example, in addition to wanting information that the system gathers as soon as possible, analysts want the ability to integrate that information with data from other sources. System owners want to minimize dedicated hardware (and software) use, because they cannot afford to maintain one system per capability, and they want greater reuse of existing resources. Further, they are very concerned with lowering ongoing operation and maintenance costs.

On top of this, it is certain that the mission and enabling technologies for the system will change several times over the program's lifespan. In addition, those who invest in the system want it to evolve in response to these changes at minimal cost and with minimal disruption. They also expect their investment to result in reusable intellectual property.

Many requirement types

To meet stakeholder needs, systems engineers need to consider a broad set of requirements. Here is a partial list of considerations:

- **Functionality:** The capability provided to users and other systems for meeting the business need. Functional requirements should include the behavior the system exhibits as it provides the functionality.
- **Usability:** Ease of access to system function.
- **Maintainability:** Ease of discovery, isolation, and removal of defects.
- **Extendibility:** Ease of adding functionality.
- **Scalability:** The ability to support increasing numbers of users, data items, and so forth, as requirements grow over time.
- **Reliability:** The probability of a correct system response, possibly including safety concerns.
- **Performance:** The expected response time of the system to a step in a use case under capacity loads.
- **Capacity:** The expected number of users and data items.
- **Supportability:** The ease of service in the field, including

acceptable down time.

- **Manufacture**
- **Deployment cost**
- **Operational cost**

Depending on circumstances, there might be other system requirements such as logistics support, security, and remote training needs.

Some of these requirements are familiar to software development teams. Some cannot be addressed without hardware, software, and worker considerations, all three of which must be specified concurrently in a systems design discipline.

Another feature of systems development is the possible need to maintain a number of system configurations. For example, one might want to maintain system specifications for products that have common architectures but different hardware and/or software deployments that meet different cost/performance points.

Rapid change

Much of the original systems engineering methodology was developed in the early days of the technology revolution (~ 1955-1980), when today's rate and impact of technological change were not anticipated. The original systems engineering methods focused on ensuring that requirements were carefully specified, then met to the extent possible.

Today, we know that a system can have a useful life of thirty years. Therefore, because no engineer can know exactly what the system will need to do in five years -- let alone thirty -- there is a higher premium placed on systems that can adapt to evolving needs. Today's system designer must also consider that the environment or context in which the system operates will evolve. Over its lifespan, any system fielded today is likely to interact with unanticipated systems. For example, commercial systems will continue to become more integrated, which will place new requirements on legacy systems; new defense systems will come online, placing new requirements on existing systems.

We also know that the enabling technology for any system will change. Over time, existing systems will fail to be competitive or will not be cost effective to maintain. Modern systems development architecture frameworks need to provide a technology-independent means to reason about rehosting or redeploying.

A larger solution space

Today's technologies can help designers meet these complex challenges because they provide more ways to approach system design. For example, whereas the designers of the first satellite image system had limited choices and had to be clever about overcoming technical constraints, today's system designers actually have excess processing capacity and

can reallocate subsystem responsibilities in order to optimize the system. So sometimes they need to make difficult choices. This phenomenon is true for a number of domains, such as telecommunications, avionics, information technology, and so on.

Balancing logical and physical considerations

There are many ways to conceptualize a system. Two of them are

- **As a physical entity**, governed by the laws of physics and classical disciplines such as mechanical, electrical, and civil engineering.
- **As a large state machine**, governed by the insights of computer science and software engineering.

Some systems, such as rocket engines, are perhaps best thought of as a physical entity. Others, such as most information technology systems, are large state machines. In fact, all systems fit some aspects of both concepts. Rocket engines have software-driven embedded controllers. Information systems are governed by the physical laws that constrain their performance, latency, and reliability.

In the past, project leaders would typically adopt one of these points of view, depending on the kind of system, and manage accordingly. If they thought of the system primarily from a hardware perspective -- as a physical thing -- they viewed the software as a "necessary evil" that enabled the hardware to do its job. If they thought of the system from a software perspective -- the hardware as a hosting mechanism -- they usually treated the hardware as an afterthought.

Increased software size and complexity

With the advent of object technology, component frameworks, and software development automation, software development productivity has increased threefold since 1970,⁷ and the size and complexity of software applications continues to grow. At the same time, there have been large gains in processing power, computer memory, available data storage, and network bandwidth. These changes, in turn, have led to ever more sophisticated operating environments. The software and systems industries have used all of this improved capability to develop increasingly large and more complex programs. In fact, competitive pressures have led to a tenfold increase in the size of an average software application, as measured by function points.⁸ And, presumably, as technology continues to improve, this trend will continue.

Many disciplines

In modern systems development, a typical development team consists of workers playing roles such as architects, developers, designer, testers, and others. In both RUP and RUP SE, all of these people work concurrently to evolve their particular artifacts throughout the lifecycle. They workers do not hand off work to each other in serial fashion. Instead, they work together throughout the effort, evolving levels of detail to address their

areas of concern. One of the chief goals -- and challenges -- of an engineering process and an architecture framework is to provide a means for the various development stakeholders to communicate and align their design decisions. Given the complexity of modern system design, the system development team must address a wide variety of concerns:

- Builds and integration
- Business modeling
- Data modeling
- Domain issues
- Hardware development
- Human factors
- Information technology
- Logistics and field support
- Software development
- Overall system specification and design

Different development team members take responsibility for different, but overlapping, sets of concerns. For example, in addition to ensuring the adequacy of the software architecture to meet functional requirements, *software architects* are generally concerned with

- **Usability:** ease of accessing the system functionality.
- **Maintainability:** ease of isolating and removing defects without introducing others.
- **Extendibility:** ease of adding new functionality to an existing software product.

Whereas *systems engineers* usually address issues of

- **Availability/reliability:** the likelihood that the system will be available and respond correctly to input.
- **Performance:** responsiveness of the system to some input.
- **Capacity:** the number of items such as users or data records that the system can handle.
- **Scalability:** the ease of increasing capacity.
- **Supportability:** the ease of providing support in the field. Supportability can include installing the system and applying patches.

Other domain-specific systems engineering concerns include security, ease of training, and logistics support.

Meeting the challenges of complexity

RUP SE provides the artifacts for addressing the concerns we have described, and the workflows for evolving their detailed specification.

Although this article series does not address all concerns of all stakeholders, it does describe an architecture framework that provides model viewpoints to enable a separation of concerns at all levels of system specification. The flowdown mechanism we will describe provides a way to maintain model consistency across the viewpoints. Use of these mechanisms enables the team to jointly follow the RUP practice of continual evolution of artifacts throughout the development lifecycle.

RUP SE design points

The systems development problem differs from the software-only development problem in that systems development addresses a broader set of requirements than are normally addressed in software efforts. Even so, it is important to note that almost all software development efforts contain some elements of the systems problem. Examples of software development efforts that present system concerns include Web-based applications, business applications, information technology integrations, and embedded software, as well as defense and intelligence systems.

The important point here is that systems development projects must address the many challenges today's systems engineers face, as we discussed in the previous section. In order to address these problems, RUP SE embraces the following design points:

- Follow industry (de facto) standard definition of systems.
- Apply the RUP framework to systems development.
- Extend the RUP 4+ 1 architectural model into the RUP SE model framework, and extend or modify the RUP roles, activities, artifacts, and disciplines to account for new views.
- Employ UML as the modeling language.
- Provide tool assets.
- Maintain all model levels as a program asset.

Let's explore each of these design points in more detail.

Follow industry (de facto) standard definition of systems

As we noted in the **Terminology and concepts** section above, systems can be viewed from both blackbox and whitebox perspectives. RUP SE follows this principle. The blackbox perspective is described in the **System specification** section (see below). The whitebox perspective is described in the **System architecture section**, included in next month's installment. Note that the elements described by RUP SE include

hardware, software, workers, and data.

Apply the RUP framework to systems development

The RUP lifecycle and disciplines are shown in Figure 1. RUP SE follows the RUP in these ways:

- **Lifecycle:** Focusing on removing risks, RUP SE follows RUP's four phases by leveraging the team's evolving understanding of the project details.
- **Iterations:** RUP SE 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:** RUP SE follows the focus areas, or "disciplines" shown in Figure 1, 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' workflows and activities are modified to address broader system problems. These modifications are described in the following sections.

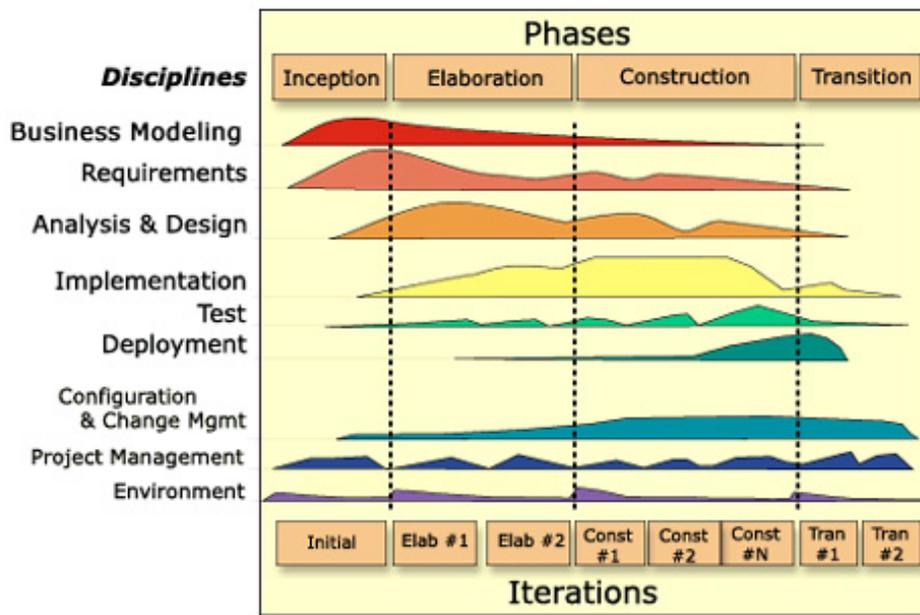


Figure 1: The RUP Process Framework (adopted by RUP SE)

As described below, RUP SE supplements RUP with additional artifacts, along with activities and roles to support the creation of those artifacts. These are described in more detail in the RUP SE Plug-In.

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

- Results-based management
- Architecture-centric development

Extend the RUP 4+1 architecture model into the RUP SE model framework

Architecture frameworks allow developers to reason about different specification and design concerns, and then document the results of that reasoning in a standard and consistent manner. The original RUP is a software development process; its 4 + 1 architecture framework is inadequate to address all of the concerns of systems development, because more stakeholders require more views. We will describe the new framework in Part II of this series, on architecture.

Employ UML as the modeling language

The RUP SE model framework uses UML to express the various diagrams that comprise the architecture model framework views. The current version adopts the UML 1.4 semantics, including stereotypes. In the next version, we will move to UML 2.0 semantics, along with the in-process systems engineering profile when adopted.

Provide tool assets

To support RUP SE, IBM Rational Software provides a RUP Plug-In that describes the RUP extension in detail, along with IBM Rational Rose® and IBM Rational RequisitePro® tool add-ins. The currently available Plug-In was released in 2002.

Maintain all model levels as a program asset

As mentioned earlier, systems lifespan often outlasts the initial requirements and enabling technologies, which leads, over time, to either outdated or otherwise insufficient functionality, or unacceptable 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. RUP SE provides the needed model levels and traceability.

System specification

System specification is the process of designating the blackbox features of the system: its externally visible functionality, what services it provides, and what measures of effectiveness it is expected to meet. In RUP SE, system specification consists of studying how the system is expected to perform in context. That is, the system is considered a participant in a broader enterprise. The system specification follows from an analysis of the enterprise and the role the system plays in enabling the broader enterprise to meet its business purpose or mission. This process is described below.

Enterprises

Note that an enterprise, like a system, is also a set of resources (workers, hardware, software, and data) used to meet a business purpose or fulfill a mission. Like a system, an enterprise has *actors* -- entities that use or collaborate with the enterprise. In fact, it is often very useful to consider an enterprise as a system of systems.⁹

Even a product such as an automobile or an airplane is a part of a broader enterprise. The airplane must interact with the pilot and all of the air traffic control systems.

Context analysis

The system being developed is always part of a larger enterprise. To specify the system, one needs to understand the activities of the broader enterprise, partition that enterprise into the system and other entities, analyze how the system participates in helping the enterprise provide its services, and then capture the results of that analysis as the system specification. In this section, we explore this workflow in more detail.

Note that systems have characteristics of both physical and logical entities. They are logical entities in that they provide services, pass messages, and interact with other logical entities; systems are physical in that they have finite limits on their capabilities, and these limits must be considered. Examples of these limitations are responsiveness and capacity. Currently, the UML has no semantics for such an entity. As discussed below, in RUP SE we model systems as stereotyped classifiers but add the relevant physical characteristics as class attributes.

A system specification captures a blackbox description of the system. It establishes the scope and boundaries of the system, the services it provides, its other attributes, and the things it exchanges with other entities. This information is captured in a system context diagram¹⁰ as shown in Figure 2.

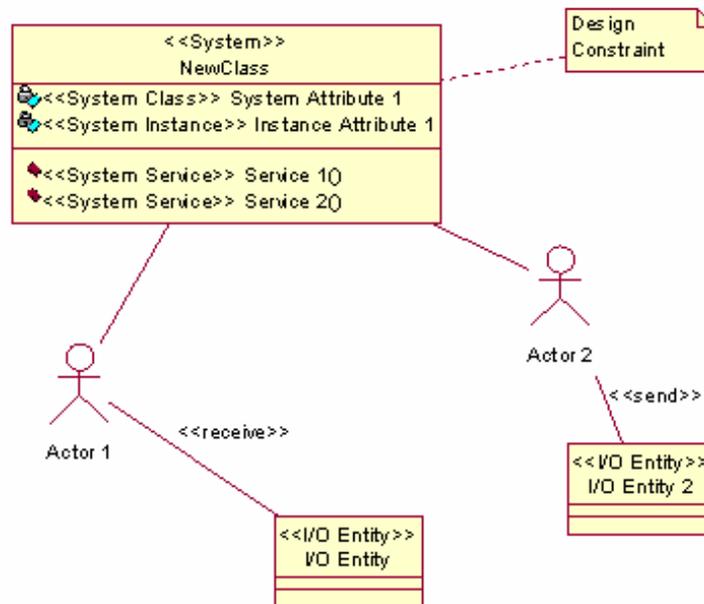


Figure 2: RUP SE system context diagram

As shown in Figure 2, a context diagram consists of

- System classifiers
- System actors (external systems and users)
- System actor relations
- Input/output entities
- Input/output entity relations
- Design constraints

Let's explore each of these elements.

System classifiers

Systems are entities. A set of systems, each of which meets the same specification, is an instance of a system class. For example, the Saab '93 automobile forms a class and the specific Saab '93 with VIN 12345678 is an instance or object in that class. Hence, systems may be described by UML object and class semantics. Even though in some cases, such as IT systems, it is expected that there will be only one instance of the system, these semantics are still useful.

Following the UML semantics, system objects may have three types of attributes:

- **Class attributes** that have the same value for every system object -- for example, total fuel capacity in the Saab '93 or total number of simultaneous users in an IT system.
- **Instance attributes** whose value can vary between system objects

-- for example, current fuel load in the VIN 123456789 or current number of users in an IT system.

- **Measures of merit** that are more general design goals such as mean time to failure or technical risk of development. These measures can be captured through tagged values.

Note that the instance attributes provide parameters for test-case development and simulations. The class attributes provide ranges for the test cases and simulations, whereas the measures of merit are used to determine adequacy of the system design and set the decision criteria for design trades.

System class operations are called *system services*. In UML terms, these are operations. Recall that, in UML, operations are classes, and instances of these classes are messages found in sequence diagrams or interaction diagrams that are part of the use-case realization. In systems development, they are found by studying how the system interacts with its actors to meet the needs or mission of the enterprise.

In addition, following the usual UML semantics, services may be aggregated into interfaces.

System actors

Recall that in UML, actors are entities that interact with the system, typically users or other systems. Many systems developers have found it useful to include environment elements such as time or atmospheric conditions as kinds of actors.

There are generally two kinds of system actors:

- **Enterprise actors** are external to the enterprise that interact directly with the system.
- **Internal actors** are part of the enterprise, not the system. These actors are usually business workers or other enterprise systems.

In addition, system actors should not be confused with system workers. Workers that are part of the system are not actors. As we will see below, they may be actors to some of the system subsystems.

System actor relations

Actor and system collaborations are similar to UML1.x system and subsystem collaborations: The actor instances invoke system services to fulfill their role in the collaboration, and vice versa. It follows that the semantics specifying that an actor uses a system is a UML dependency. The directionality of the dependency denotes whether the system invokes the actor services, vice versa, or both.

Sometimes the actor and the system may be more tightly coupled so that the attributes of one are affected by the attributes of the other, and the

relationship of the actor and the system is promoted to an association. For example, consider a driver (the actor), and an automobile (the system). When the automobile is moving, presumably the driver is in a higher state of readiness than when the automobile is parked.

The system services can be aggregated into interfaces. The interfaces can be used to specify which set of operations is used by which actors or to simply create categories of services. Note that a given service may be included in more than one interface.

Input/output entities

Input/output (I/O) entities include anything generated and/or received by the system, such as information or physical items. A retail system may provide credit card information to another credit card information system. An air conditioning system may exchange hot and cold air with the atmosphere. Typical input entities include database queries, file updates, sensor inputs, and control inputs. Typical output entities include query results and sensor outputs.

I/O entities are UML classes with attributes but no operations.

Input/output entity relations

Note, as shown in Figure 2, that I/O entities have special stereotyped actor relationships. The association captures whether the entity is sent or received by the actor. The semantics maintain the actor-system relationship in the model and do not suggest that a context diagram is also a data flow diagram.

Design constraints

Frequently, system specifications are not strictly blackbox considerations. The specifications include constraints on the internals of the system. Examples of such design constraints include components that must be used and algorithms that must be followed. As the design is evolved and the whitebox elements are specified, these constraints will find a natural expression in one of the RUP SE views described in the following section. There are no specific UML semantics for expressing design constraints. In RUP SE, these may be captured in one of two ways: either as notes in the context diagram or, preferably, as a supplementary requirements document associated with the system class.

System use cases and services

RUP SE uses both use cases and services¹¹ to capture system behavior. In both cases, standard UML semantics¹² are used:

- **Use cases** describe how the system is used by its actors -- in other words, the set of anticipated collaborations/scenarios between the actors and the system. Use cases convey how the system is used to meet the broader enterprise business purpose or mission. In RUP

SE, Use-Case Descriptions are identical to those used in the standard RUP. In particular, they only describe blackbox interactions between the system and its actors.

- **Services** are the (possibly abstract) operations provided by the system so it can fulfill its role in the use-case scenarios. Instances of services are the messages found in the UML sequence diagrams that realize the use-case scenarios. Generally, there is an n-to-m mapping between use-case and services.

Figure 3 is an example of a partial context diagram for a retail system.

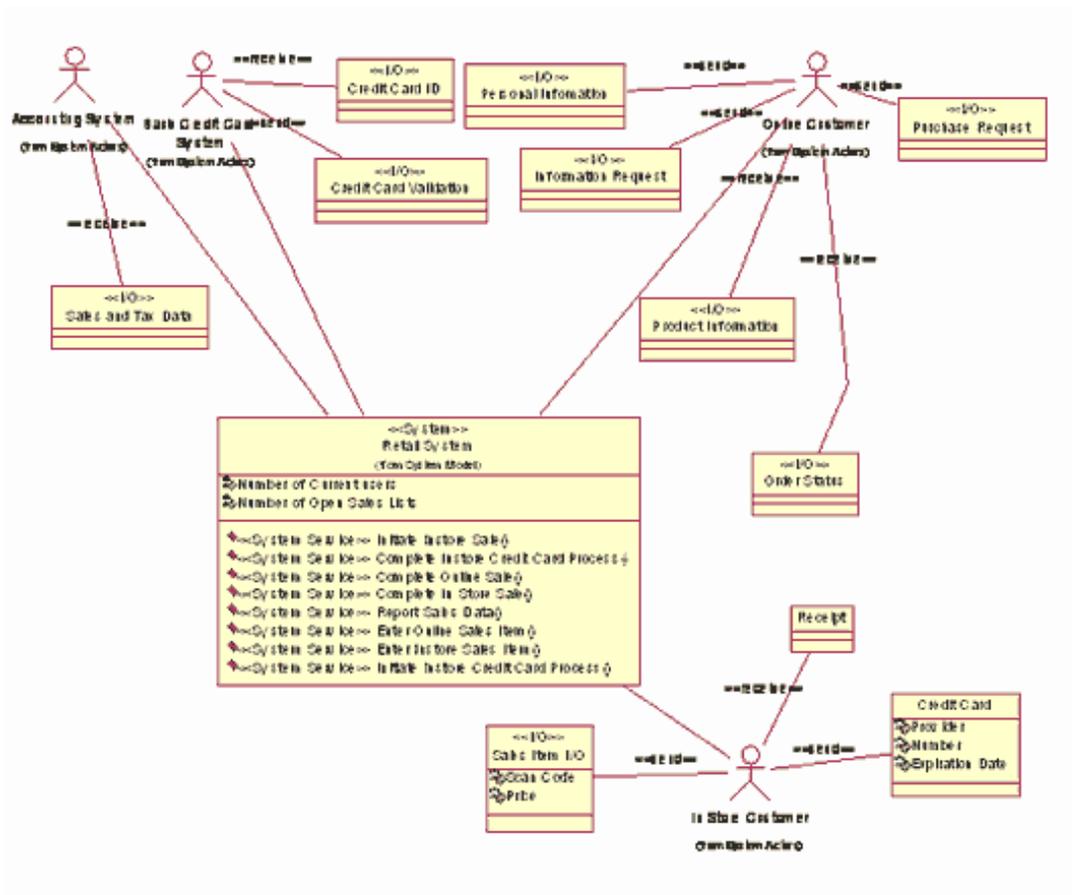


Figure 3: A retail system context diagram
[Click to enlarge](#)

Next month:

This series of articles on RUP SE will continue in the September issue of *The Rational Edge*, with a discussion of System Architecture as it pertains to systems development. The third and final installment will appear in the October issue, with a discussion of requirements analysis -- including flowdown of functional and supplementary requirements to architectural elements -- project organization, iterative development issues, and integration.

Notes

¹ Blanchard and Fabrycky, *Systems Engineering and Analysis* (third edition), Prentice Hall, 1998.

² For example, the *Oxford English Dictionary* lists eleven definitions.

³ <http://www.incose.org/whatis.html>

⁴ Mark W. Maier and Eberhardt Rechtin, *The Art of Systems Architecting* (second edition), CRC Press, 2000, p 8.

⁵ Much of this sections is taken from Murray Cantor, "Leading Modern System Development ," published in *The Rational Edge*, January, 2003, http://www.therationaledge.com/content/jan_03/f_modernSystemsDevelopment_mc.jsp

⁶ Thomas P. Hughes, *Rescuing Prometheus*. Pantheon Books, 1998.

⁷ David Longstreet, "Software Productivity Since 1970." <http://www.ifpug.com/Articles/history.htm>, 2002.

⁸ *Ibid.*

⁹ Paul Carlock and Robert Fenton, "System of System (SoS) Enterprise Systems Engineering for Information-Intensive Organizations," *Systems Engineering*, Vol. 4, No. 4, 2001.

¹⁰ The RUP SE context diagram is semantically equivalent to the Elaborated Context Diagram first described by Sanford Friedenthal Howard Lykins, and Abe Meilich. See <http://www.omg.org/cgi-bin/doc?syseng/2001-09-05>

¹² This distinction between use cases and services was is a feature of OOSEM, *loc cit.*

¹³ For a discussion of the use of classifier semantics for subsystems in UML, see the article by Fredrick Ferm, "The Why, What, and How of a Subsystem," *The Rational Edge*, June 2003. http://www.therationaledge.com/content/jun_03/t_subsystem_ff.jsp



**For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided.
Thank you!**

.

.