

Rational Unified Process for Systems Engineering

Part III: Requirements analysis and design

by **Murray Cantor**
Principal Engineer
Rational Brand Services
IBM Software Group

In the August issue of The Rational Edge*, we began a three-part series to provide an overview of the latest evolution of Rational Unified Process for Systems Engineering,® or RUP SE.® RUP SE is an application of the Rational Unified Process,® or RUP,® software engineering process framework. 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.*

Part I included a discussion of systems, the challenges facing the modern systems developer and how RUP SE addresses them, RUP SE Unified Modeling Language (UML)-based modeling and requirement specification techniques, and the use of UML semantics. Part II focused on system architecture and introduced the RUP SE architecture framework, which describes the internals of the system from multiple viewpoints. Now, in Part III, we will cover requirements analysis and flowdown, and specifications for elements of the RUP SE framework. This will include a description of the Joint Realization Method, a novel technique for jointly deriving the specification of architectural elements across multiple viewpoints. We will also include a brief discussion of system development with RUP SE.

Editor's note: The RUP SE v1 Plug-In was made generally available in 2002, and v2 of this plug-in was made available in June of 2003. Although the information in this series is consistent with v2, the articles do discuss a

few possible extensions to the process framework. Please note that the RUP SE Plug-In -- v1 and v2 -- is downloadable from IBM Rational Developer Network (http://www.rational.net; authorization required).

RUP SE and requirements

Following common system practice, RUP SE addresses two types of requirements:

1. **Behavioral requirements** -- What the system does to fulfill its role in the enterprise. In RUP SE, a system's behavior is captured by its use cases and their analyses into services. Use cases and services may have associated performance requirements.
2. **Supplementary requirements** -- Nonfunctional requirements, including design goals (e.g., reliability or cost of ownership), and system attributes (e.g., data capacity or total weight).

RUP SE also offers a process pattern for deriving requirements for architectural elements:

1. Determine the blackbox requirements or specifications for a given model element.
2. Decompose that model into whitebox elements, assigning roles and responsibilities to these elements.
3. Study how the elements collaborate to jointly meet the blackbox requirements. This usually involves some form of collaboration diagram.
4. Synthesize the analysis of the collaboration to determine the blackbox requirements for the elements.

This process pattern is well known,[1] and it is particularly interesting that Friedenthal et al. adopted it in their Object Oriented System Engineering Method (OOSEM).[2]

System specification, therefore, means defining use cases, system services, and supplementary requirements that, if met, would result in a system that meets its business purpose or mission. RUP SE distinguishes between *allocated* and *derived* requirements. A requirement is *allocated* if a blackbox requirement is assigned to a whitebox element. A whitebox requirement is *derived* if it is determined by studying how the whitebox element collaborates with others to meet a blackbox requirement. Both the behavioral and supplementary requirements may be derived. Note that derived requirements take into account the role of the architectural element in the system design.

Consider the following example. Most automobiles have differentials, the devices connecting the drive shaft to the axles that allow the driving wheels to go at different speeds whenever the automobile turns a curve. This function is necessary because the inner wheel must go more slowly

than the outer wheel in order for both wheels to maintain traction. If one of the wheels loses traction, the automobile may spin out and possibly flip over. Nevertheless, there is no automobile *system* requirement for a differential. The automobile system requirement is simply that the automobile maintain traction when it traverses a curve, which can be accomplished in a variety of ways that don't involve a differential. For instance, either of these alternatives might work:

- A single driving wheel (as in the three-wheeled vehicles that sometimes appear in science-fiction movies).
- Two motors, one per driving wheel, with some sort of drive-by-wire solution.

The convention of the differential prevails either because it is superior from an engineering standpoint (i.e., it does a better job of meeting a variety of requirements in addition to traction control, such as maintaining overall stability, optimizing interior volume, and managing cost of materials maintenance) or from a design constraint that the engineer must take advantage of previous excellent engineering.

Now, since the differential is not a required element of the automobile, there is no mechanism for assigning system requirements to the differential. Rather, the differential plays a role in collaboration with other elements of the automobile (steering, brakes, etc.) so that they jointly can meet the required behavior of the automobile to safely traverse a curve. The behavior of the differential, such as "adjust wheel velocity," is derived from the system requirement and the role the differential plays.This behavior is derived, not allocated.

Further, the differential must meet derived supplementary requirements in order to support the system's defined supplementary requirements. For instance, the differential will have a weight and volume budget as well as a reliability measure.

The use of derived requirements for subsystems collaborating to carry out use cases is called *logical decomposition*. Similarly, determining subsystem requirements by allocation is *called functional decomposition*. Generally, logical decomposition is essential for quality systems.[3]

It follows that the system requirements are derived from an understanding of the enterprise services and the role that the system plays in the enterprise. In the analysis model, the system architectural elements are subsystems, localities, and processes, as described in the **System architecture** section in Part II of this series. It is in the requirements analysis discipline that requirements for each of these types of architectural elements are determined. For example, with the business model in place, RUP SE suggests that you partition the enterprise into the system and its actors to derive system requirements. Then, to determine system requirements, an analyst may study how the system and its actors collaborate to meet the business requirements.

The following sections describe RUP SE's approach to deriving functional

requirements for systems and elements of the analysis model.

Deriving functional requirements through use-case flowdown

Use-case flowdown is an activity for deriving functional requirements for systems and their elements. Flowdown can be applied to add detail within a model level or to specify elements at a lower model level. For example, flowdown can be used to determine system services at the context level. Similarly, it can be used at the analysis level to identify subsystem services and to break subsystems into further subsystems.

It is important to note that *flowdown may be applied recursively* -- in other words, whitebox elements become blackbox for the next application. This allows the team to reason about large systems at the appropriate level of specificity. Repeated application of the flowdown activity allows teams to add detail while managing a consistent level of abstraction. It also permits concurrent design; that is, each whitebox entity can be specified sufficiently to be treated as a blackbox entity for further design by separate teams. You can use this approach not only to derive requirements for elements of the analysis model, but also, with little modification, to determine system requirements from business requirements.

Performing flowdown in the hierarchical manner we described above results in an interesting relationship between services and use cases: *blackbox services become whitebox use cases*. Use cases describe how an entity and elements in its context collaborate to fulfill some purpose. Here, the purpose of the use-case flowdown is to support delivery of a system service. The realization of the service consists of use-case scenarios. For each UML subsystem, you can build a context diagram showing system actors with which the subsystem collaborates, as well as the peer subsystems with which it shares a dependency relationship (these are akin to the enterprise and internal actors discussed under **System Specification** in Part I). From the subsystem point of view, the service realization is exactly how it collaborates with its actors to carry out its role. This is exactly a use-case scenario. Note that flowdown does change the common value heuristic of use-case analysis. The use cases in flowdown provide value to the blackbox entity, and not necessarily to any of the participating actors.

Simple realization

Use-case flowdown is an extension of *use-case realization*, an elemental practice of object analysis. Use-case realization consists of finding classes that participate in carrying out a use-case scenario, and discovering how the objects of the various classes collaborate. The realization includes specifying the order of objects' messages that are passed during the collaboration, and it is captured in a sequence or collaboration diagram. In fact, by building sequence diagrams, you can often discover the messages a class operation must provide so that its objects can participate in realizing its use cases.

In RUP SE, this notion of realization is extended in several ways. First, realization is applied to model levels higher than design. For example, the outcomes of flowdown applied between enterprise and system result in identification of system services. If applied between the system and its model elements, flowdown results in:

- A use-case survey for subsystems.
- Identification of subsystem services and interfaces.
- A survey of hosted subsystem services and/or supported interfaces for localities.

The idea of extending use-case realization to UML subsystems is not new. For example, realizations for UML subsystems are often referred to as *Architectural Interaction Diagrams*.

Here are the flowdown steps for building the system context diagram and identifying system services:

1. Model an enterprise whitebox as a set of collaborating systems.
2. Model how systems collaborate to realize enterprise services, mission, and so forth.
3. Create a context diagram for the system.
4. Determine actors (i.e., entities that collaborate with the system).
5. Identify I/O entities.
6. Aggregate similar collaborations between the system and its actors into use cases.
7. Add use-case detail: performance, pre- and post-conditions, and so forth.
8. Identify system services -- what the system does to support its use cases; aggregate similar whitebox steps.
9. Add system attributes from your analysis of enterprise needs.

When a realization consists of one type of whitebox element, such as classes or UML subsystems, we call this *simple realization*. An example is the flowdown from enterprise to system, as delineated below.

Procedure 1: Joint realization

In future versions of RUP SE, the simple realization described above is extended to *joint realization*: analyzing how the elements of multiple viewpoints collaborate in carrying out a service. For example, in joint realization, the flowdown might consist of simultaneously determining the collaboration of logical, physical, and informational elements.

Joint realization consists of the following procedure:

1. Choose the participating viewpoints. The logical viewpoint is mandatory.
2. For each whitebox step in realizing a blackbox service, you must:
    - Specify the logical element that executes it.
    - Model how the additional viewpoints participate. For example, you might include:

      -*Physical viewpoint* -- Specify hosting locality; if there are two localities, then decompose into two steps.
      -*Process viewpoint* -- Specify executing process; if there are two processes, then decompose into two steps.
      -*Information viewpoint* -- Specify which data schema element supports handling of any information that is used.

      Throughout this process, apply the following *joint realization rule*: If a given logical element whitebox step requires more than one element of the other viewpoints, divide that step into further steps so that each step requires exactly one whitebox element from each viewpoint.

3. Create interaction diagrams for each viewpoint:
    - Architecture interaction diagram
    - Locality interaction diagrams
    - Process interaction diagrams
4. Budget supplementary requirements for performance, accuracy, and so forth, to each step; evaluate/confirm with interaction diagrams.

Procedure 2: Specify resources with joint realization

Joint realization has a variety of applications. For example, it can be used for flowdown from system to logical and worker view to reason about automation decisions. Or it can be used for flowdown from system to logical, physical, and process elements (this application is described in more detail below). To uncover specifications for the system's physical resources, you must:

1. Develop initial analysis model-level views (system whitebox). To do this:
    - Use object-oriented analysis methods for the logical view.
    - Apply physical considerations for the locality view.
2. Use joint realization to model each (architecturally significant) system service specification, including:
    - Collaborating steps for UML subsystems.
    - Hosting localities.

- Executing process.

3. Capture the whitebox performance requirements -- in other words, the budgeting of the blackbox performance requirements to the whitebox steps. To do this:
   - Identify UML subsystem use cases; in other words, for each subsystem, identify system services involving that subsystem.
   - For each subsystem, identify its services from applying aggregation methods on messages in collaboration.
   - For each locality, create a survey of hosted subsystem services.
   - For each process, create a survey of executed subsystem services.
4. Document traceability between system and subsystem use cases, and/or system and subsystem services.

Procedure 3: Flowdown from context to analysis model level

The assignment of whitebox steps to subsystems, localities, and processes involves a set of design decisions. Each decision adds detail to the role that each analysis element plays in the overall system design. In the process of making assignments, the team may decide to refactor the design, shifting responsibilities from one element to another within a given view. Also, note that flowdown provides opportunity for adding an appropriate level of detail and refactoring subsystem, locality, and process roles and responsibilities.

Table 1 shows an example whitebox flowdown for the system service "Closing sale with credit card," using the subsystem (Figure 6) and locality model 1 (Figure 8) for a click-and-mortar retail system.

Table 1: Joint realization table

System Actor Action	Black Box Budgeted Requirements	Worker Action	Subsystem Action	White Box Budgeted Requirements	Locality	Process
THE CUSTOMER PROVIDES A CREDIT CARD.	30 seconds	Clerk swipes credit card.	The Point of Sale terminal requests that Credit Card Services provide validation.	.5	Point of Sale Terminal	Terminal
			Credit card services queries bank credit card system.	28 seconds	Store Server	Order Processing
			If valid, Point of Sale prints receipt.	.5 seconds	Point of Sale Terminal	Terminal
		Clerks requests that customer sign receipt.				

The next step is to determine the UML subsystem use cases and context. A UML subsystem context view, like a system context, consists of the subsystem, its actors, and any relevant I/O entities. For a subsystem, its actors can consist of its peer subsystems and, possibly, system actors. Figure 1 provides a subsystem context diagram example.

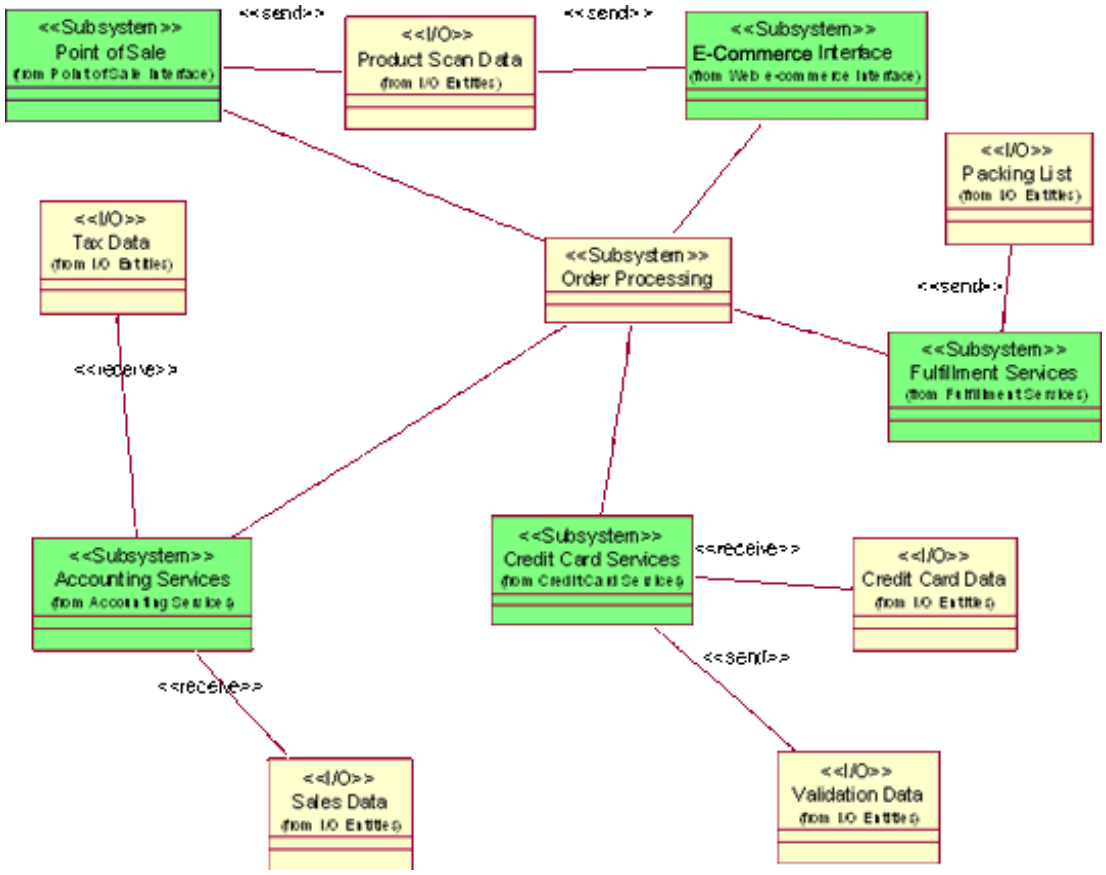

Figure 1: Subsystem context diagram

Recall that a use case describes how a system and its actors collaborate to provide a service of value. For a subsystem, the service of value is the system service itself. It follows that for each subsystem, its use cases are exactly the system services in which it collaborates. If you are going to partition the development effort along subsystem boundaries or as a basis for developing test cases, it is useful to keep a use-case survey for subsystems.

You can find subsystem services by sorting the service whitebox steps by subsystem. For each subsystem, sort the whitebox steps and aggregate similar steps, as shown in Table 2. This results in the specification of services provided by each subsystem.

Table 2: Example survey of locality hosted services

Locality Name: services Store Processing			
Locality Responsibility: This locality hosts central store sales transactions and accounting. It provides the interface to the central office and credit card processing.			
Subsystem Service	**Subsystem**	**System Service**	**Service Whitebox Text**

Initiate Credit Card Sale	Order Processing	Enter a sale	Order Processing starts a sales list
Add Product data	Order Processing	Enter a sale	The scanner data is sent to order processing, which retrieves name, price, and taxable status from an inventory and updates list
Compute Total	Order Processing	Enter a sale	Order Processing sums the price and computes the taxes.

After determining the subsystem services, you can sort the set of subsystem services by locality or by process. The survey of hosted services for each locality expresses what computing occurs at the locality as well as the associated performance requirements. This information provides input to the specification of physical components that will be deployed at the locality. Similarly, the survey of executed services for each process serves as input to the specification of software components. These activities add the following steps to the joint realization process:

- For each locality, create a survey of hosted services (such as those shown in Table 2).
- For each process, create a survey of executed services.

We'll describe component specification more fully below.

An alternate approach for associating subsystem services to localities is to define a subsystem interface comprising services that are hosted on the locality, and then associate that interface with the locality. This approach has the benefit of keeping the service-to-locality association fully contained in the UML model.

The textual description in the whitebox flow of events can also be expressed as a set of sequence or collaboration diagrams. These diagrams convey the traffic between analysis elements: Each diagram is a sequence diagram whose objects are proxy classes for the analysis elements. The messages are invocations of the subsystem services. Figures 2 and 3 show the subsystem and locality interaction diagrams for the flow of events described in Table 2.

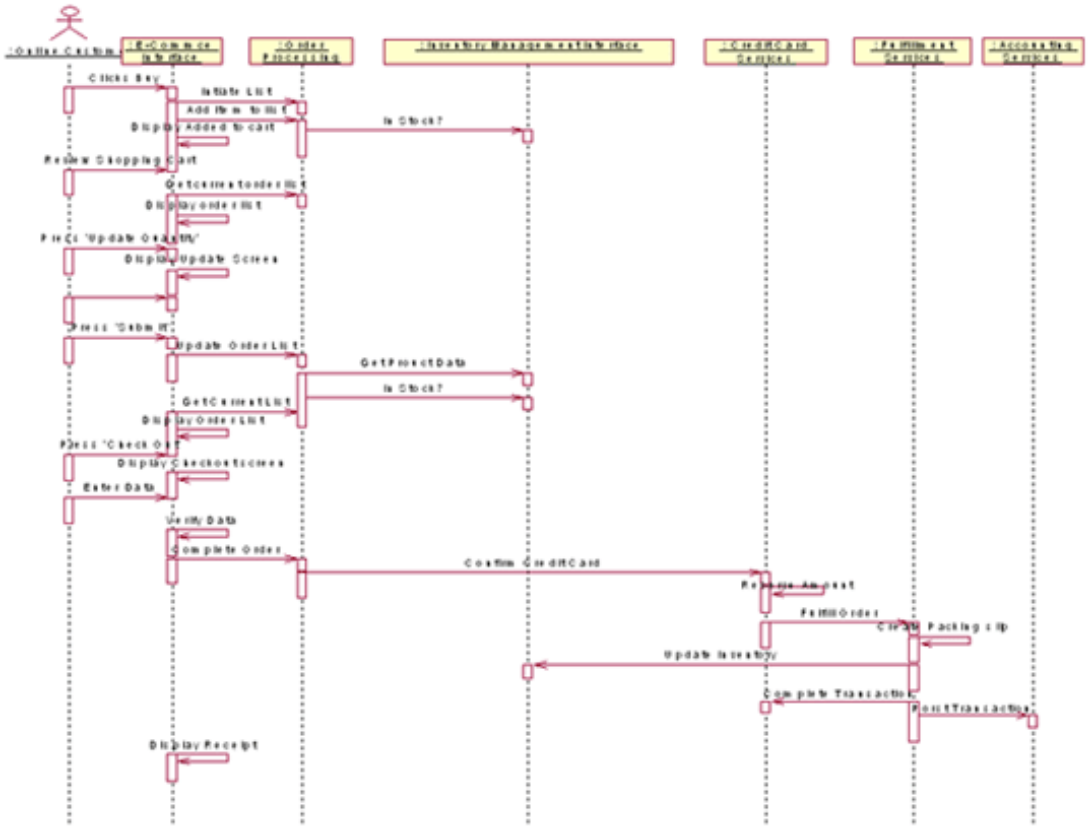

Figure 2: Example subsystem interaction diagram
Click to enlarge

Figure 2 provides insight into the coupling and cohesion of the subsystems. This insight may be used to refactor the subsystem design; if there is a lot of traffic between a pair of subsystems, for example, it may make sense to combine them. Figure 3 is an example locality interaction diagram.

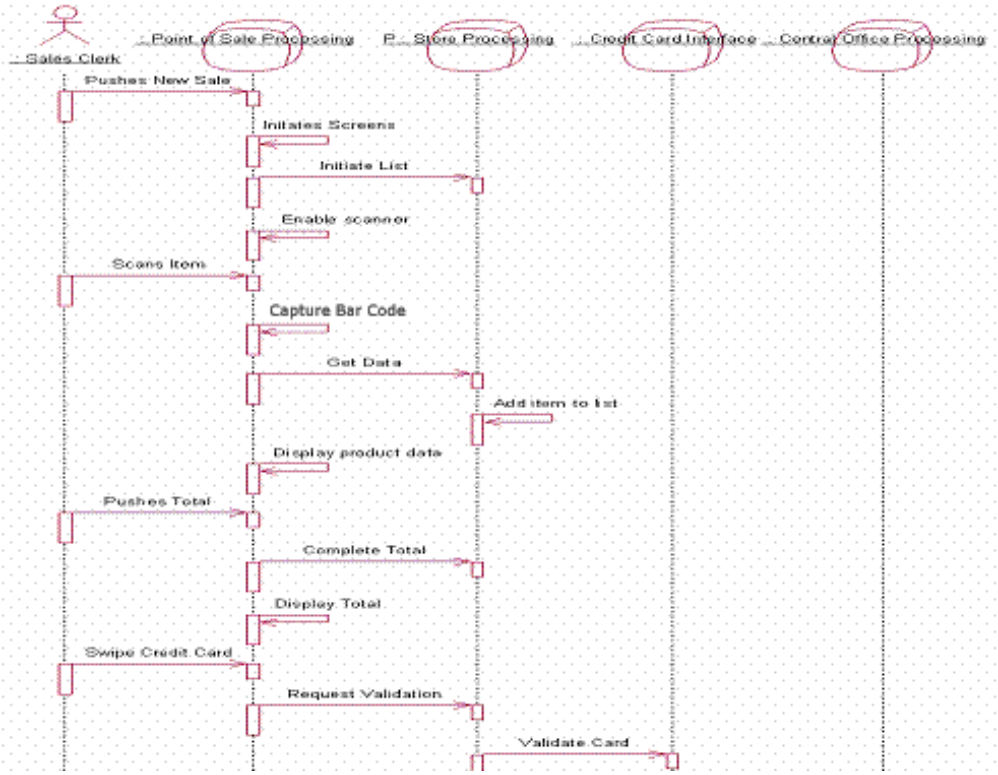

Figure 3: Example locality interaction diagram

The traffic in Figure 3 shows what data must flow between the localities. This information is used to specify associations between localities.

Supplementary requirements flowdown

Supplementary requirements are initially captured as system class attributes or tagged values. As part of the analysis process, the system architects develop an initial locality diagram. The locality view is a synthesis of the nonfunctional considerations and provides a context for addressing how nonfunctional requirements such as reliability and capacity will be addressed.

Standard engineering practice allows for the budgeting of capacity, permitted failure rates, and so forth. This results in a set of derived supplementary requirements for each locality element. Locality characteristics are determined from these requirements.

Component specification

Moving from the analysis to the design level of an architecture entails determining the hardware and software component design. This design-level specification consists of the components to be deployed: hardware, software, and workers.

Hardware components are determined by analyzing the localities, along with their derived characteristics and hosted subsystem services. With this information, you can do descriptor-level realizations of the localities. Descriptor node diagrams specify the components, servers, workstations,

workers, and so forth, without showing specific technology choices. Figure 4 is an example descriptor node diagram that realizes the locality diagram shown in Figure 8. The fulfillment locality is realized as four components: a warehouse gateway, a mailing/postage system, and two workers.

The descriptor nodes inherit characteristics from their localities through an allocation or budgeting process.

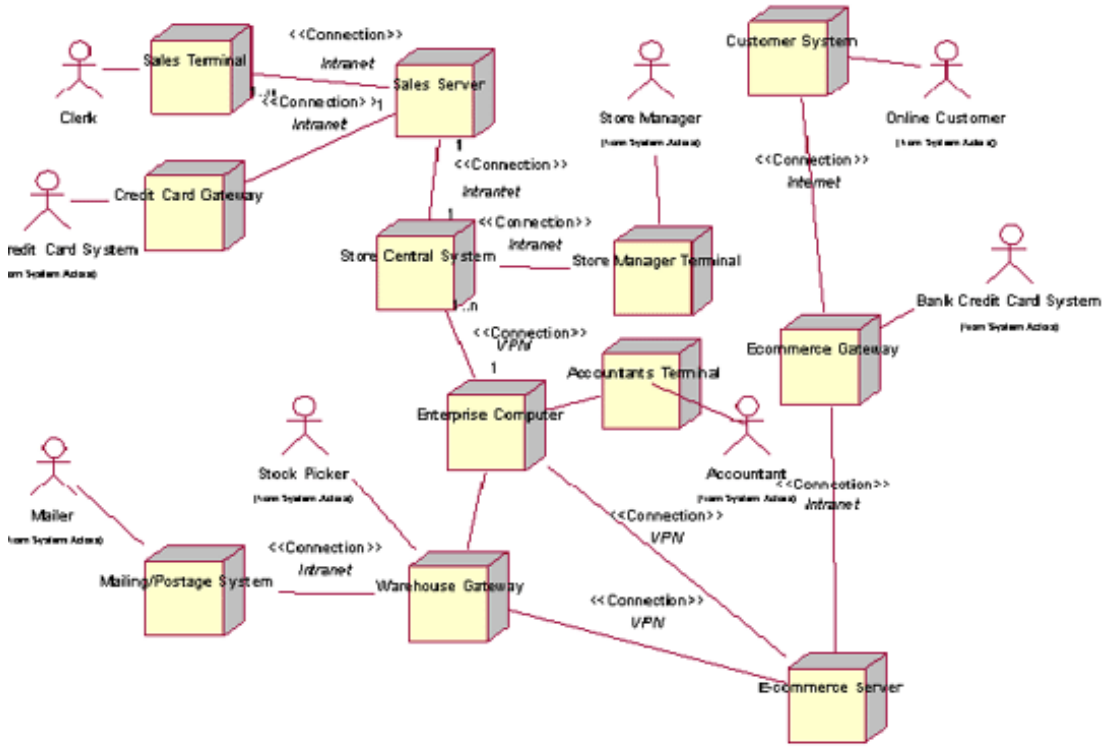

Figure 4: Example descriptor node diagram

The implementation hardware components -- in other words, the actual deployed set of hardware -- are determined by making cost/performance/capacity trades from the descriptor view. In fact, a system may have more than one hardware configuration, each meeting different price/performance points.

Components are determined by specifying a set of object classes, and then compiling and assembling the code associated with those classes into executable files. A fully considered software component design must reflect a variety of concerns:

- **Locality** -- where the components need to run.
- **Hosting** -- processor instruction set and memory restrictions for the executing code.
- **Concurrency** -- separation of processing into different hosts or memory spaces to address reliability and related concerns.

It follows that the information needed to specify components includes the surveys of hosted subsystem services for localities and their realized

hardware components, surveys of executed services for processes, and the view of participating classes (VOPC) for the subsystem services.

For each hardware configuration, the RUP SE method requires creation of a component from the classes participating in all the subsystem services hosted on each node. If those services need to be executed in more than one process, we divide the components further by assigning the participating classes of the subsystem services executed by each of the processes. Note that some subsystem services may be executed by more than one process, and therefore their classes may be in more than one component. We complete the process by dividing the components further to account for memory constraints (such as .exe and .dll trade-offs), shipping media limitations, and so forth.

These activities result in a set of specific hardware and software components that make up the system.

System development

RUP SE projects are managed much the same as any RUP project. However, because of the size of, and additional activities required for, most systems engineering efforts, there are some differences, which we will discuss briefly in this section.

Project organization

Moving from a traditional serialized process ("waterfall" process) to an iterative process has profound implications with respect to how a project must be organized. In a serialized process, staff members are often assigned to a project until their artifacts are complete. For example, the engineering staff completes the specifications, hands them off to the development staff, and moves on to the next project. In any RUP-based project, no such handoff occurs. Rather, the artifacts evolve iteratively throughout the development process. This requires that the staff members responsible for project artifacts such as the requirements database and UML architecture, must remain assigned to the development project throughout its lifecycle.

Figure 5 shows the organization for a typical RUP SE project. It consists of a collection of development teams, each with a project manager and a technical lead. There are also teams that deal with overall system architecture and project management.

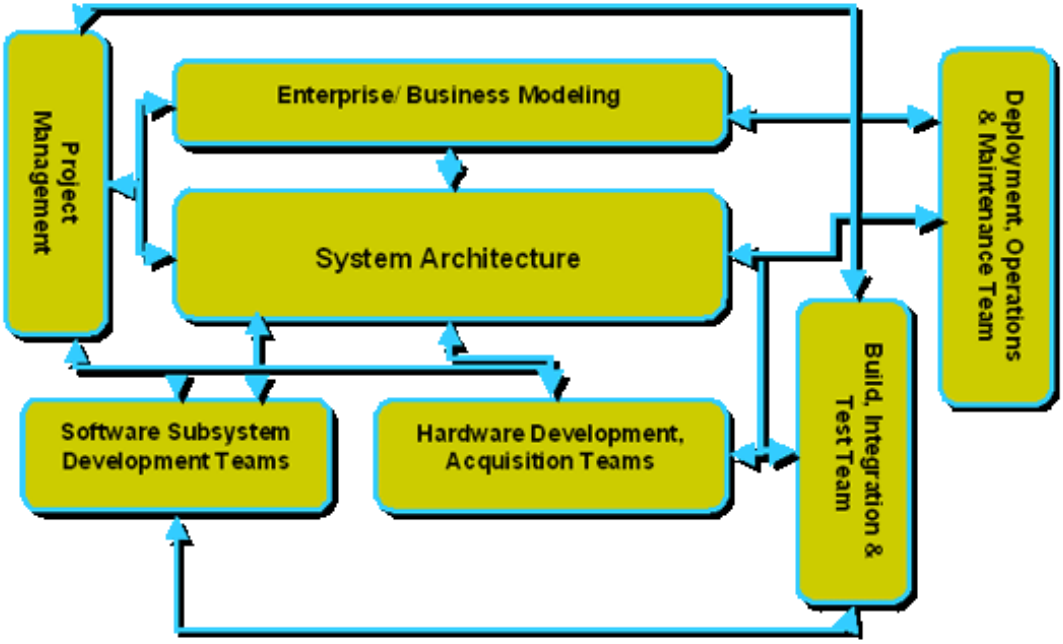

Figure 5: A RUP SE organization chart

The teams in this figure have the following functions:

- The **Enterprise Modeling Team** analyzes the business need and generates business models and/or related artifacts, such as concept of operations documents.
- The **System Architecture Team** works with the Enterprise Modeling Team to create the system context and derive system requirements. This team develops the subsystem and locality views as well as their derived requirements. Throughout the development process, the System Architecture Team serves as a technical escalation point to resolve architectural and engineering issues. This team also works with the development teams to specify the software component architecture. Team members include the technical leads of the development teams.
- The **Project Management Team** looks after the standard project issues such as project reviews, resource planning, budget tracking, earned value and variances, and coordinated iteration planning.
- For each iteration, the **Integration and Test Team** receives the code and hardware components from the development teams, builds the software components, and installs the hardware and software components in a laboratory setting. The team also plans, executes, and reports on the system tests for each iteration.
- The **Subsystem Development Teams** are responsible for the design and implementation of the software realization of one or more subsystems. These teams base their work on the derived use cases discovered during flowdown activity. Depending on the size and complexity of the system, the subsystem use cases may be realized as class design and associated code modules, or the

subsystems may be further decomposed into subsystems. In the latter case, a subsystem team may be further decomposed into sub-subsystem teams, and a subsystem architecture team may be created. This process enables scalability of the RUP SE approach.

- The **Hardware Development and Acquisition Teams** are responsible for the design, specification, and delivery of the hardware components.
- The **Deployment Operations and Maintenance Team** handles operational issues and serves as a liaison with the users. This team might install and maintain the system in the field. In other cases, this team might handle user defect reporting and provide patches to the field.

Concurrent design and implementation

An attractive feature of the RUP SE organization approach is that it scales to very large programs. Once you decompose the system into subsystems and localities with their derived requirements, each of these analysis model elements is suitable for concurrent design and development. As we noted, you can assign UML subsystems to separate development teams, and assign localities to hardware development or acquisition teams. Each team works from its derived survey of hosted services or assigned interfaces to develop its portion of the design model and implementation models. That means the design and implementation of the design elements can proceed in parallel.

For very large systems, a system-of-systems approach can be adopted. In such cases, each UML subsystem has its own locality mode, and you need only address logical concerns. This permits application of the organization structure shown in Figure 5 at the subsystem level, providing even more scalability.

Iterative development, integration, and testing

One central feature of the RUP approach is that the system is developed in a series of iterations, each of which produces a working prototype with incrementally new functionality. The system is integrated and tested at each iteration, and the iteration testing is a subset of the system tests. Consequently, the final iteration results in a fully tested system ready for transition to the operational setting.

The timing and content of iterations are captured in an iteration plan early in the project. However, like any RUP artifact, the iteration plan is updated continually to reflect the emerging understanding of the system as it comes together.

The content of an iteration, captured in a *system iteration plan*, is specified by what use cases and supplementary requirements are realized by the components developed in the iteration. Each iteration is tested by the subset of applicable system test cases.

Recall that subsystems have derived services that trace from system services. This tracing provides a basis for derived iteration plans for the subsystems and localities. That is, the content of each system iteration is traceable to the functionality that needs to be provided by the subsystems and localities to support the iteration. In practice, the development teams will negotiate the iteration content to reflect their development practicalities. For example, an early system iteration cannot require full functionality of a subsystem. Compromises must be made.

A good system iteration plan provides the opportunity to identify and resolve system technical risks early, in contrast to the typical late-stage panic of the waterfall-based integration and testing phase. The technical risks can involve both functional and nonfunctional requirements. For example, an early integration can shake out system bring-up and fail-over issues that cannot be fully understood with detailed design and interface specifications alone. In practice, the early iterations should validate that the architecture is sufficient to meet these sorts of nonfunctional requirements.

Iterative system development may seem more expensive because it requires more testing, as well as scaffolded or simulated hardware environments to support the early iterations. Coordination of content for each iteration across development teams also takes more project management effort. However, these apparent costs are offset by the savings you realize through early identification and mitigation of risks associated with the system architecture. It is a standard engineering principle that removing architectural defects late in a project is much more expensive than removing them early. In addition to added expense, removing defects late in the process also adds uncertainty, as well as schedule and budget risks, late in a project.

The role of the testing organization within an iterative project is different from the testing role within a serialized, waterfall-based project. Rather than allocating a large amount of time for overall system integration following development, an iterative-based testing organization spends time integrating, testing, and reporting defects throughout the project lifecycle.

In summary

RUP SE, delivered as a Rational Unified Process® (RUP) Plug-In, is an application of the RUP framework to support the development of large-scale systems that are composed of software, hardware, workers, and information components. RUP SE includes an architecture model framework that enables you to consider different formal perspectives (logical, physical, information, etc.) in order to deliver a solution that addresses the concerns of the various development stakeholders. A distinguishing characteristic of RUP SE is that the requirements for system components are jointly derived in increasing specificity from the overall system requirements.

RUP SE is ideally suited for projects that:

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

RUP SE provides the system development team with the advantages of RUP best practices while providing a framework for addressing overall system issues. Some of the benefits of RUP SE include:

- **System team support** -- Provides for ongoing collaboration among business analysts, architects, system engineers, software developers, hardware developers, and testers.
- **System quality** -- Provides views that enable teams to address system quality issues in an architecture-driven process.
- **System visual modeling** -- Provides UML support for systems architecture.
- **Scalability** --Scales upward to very large systems.
- **Component development** -- Provides workflows for determining hardware and software components.
- **System iterative design and development** -- Supports concurrent design, and iterative development of hardware and software components.

Notes

1 See Maria Ericsson's IBM Rational whitepaper, *"Developing Large Scale Systems Using the Rational Unified Process*" at http://www.rational.com/products/whitepapers/sis.jsp.

2 Sanford Friedenthal et al., "Adapting UML for an Object-Oriented Systems Engineering Method." Proceedings of the 2000 INCOSE Symposium.

3 Murray Cantor. "Thoughts on Functional Decomposition," *The Rational Edge*, April, 2003.

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