Rational Unified Process for Systems Engineering

Part II: System architecture

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

Last month 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 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. This month, in Part II, we will focus on system architecture and introduce the RUP SE architecture framework, which describes the internals of the system from multiple viewpoints. Part III, to be published in October, will cover requirements analysis and flowdown, an introduction to the method for deriving requirements, and specifications for the 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. Part III will also include a discussion of RUP SE programmatics.

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 (authorization required).

Definitions

A clear understanding of RUP SE is impossible without a grounding in several terms and concepts. [Other standards may define these terms differently; what we strive for here is internal consistency.]

System decomposition: Successful system engineering relies on the ability to reason about many things at once. System-level decomposition is one powerful technique for accomplishing this.

A system may be decomposed in two ways:

- Into further systems using logical decomposition; this is the so-called "systems of systems" decomposition.
- Into system components that make up the delivered system.

System model dimensions: A RUP SE system model has two dimensions, which allow for separation of concerns by different teams involved in the design and construction of the system.

- **Viewpoint dimension:** the context for addressing a limited set of quality concerns.
- **Model level dimension:** UML diagrams that capture a specific level of design detail.

Model: A representation of a system, including views that capture all areas of concern, levels of specificity, and model entity relationships.

Model level: The level of abstraction at which each model may be constructed, from the more general -- hiding or encapsulating detail -- to the more specific -- exposing more detail and explicit design decisions.

Viewpoint: As the name implies, a viewpoint is a notional "position" from which some aspects or concerns about the system are made visible, implying the application of a set of concepts and rules to form a conceptual filter. To understand a system, it is usually not sufficient to examine the actual system itself, which is why models are constructed to represent the various viewpoints involved.

View: A projection of a model level that shows entities that are relevant from a particular viewpoint. These projections will typically be illustrated by diagrams of some kind. The intersection of viewpoint and model level (of abstraction) will contain (or at least identify) views of model(s) relevant to that viewpoint (concern) at that level of abstraction.

Viewpoints

The RUP SE framework provides a set of viewpoints, as expressed in Table 1.

Table 1: System model viewpoints

Viewpoint	Expresses	Concern
Worker	Roles and responsibilities of system workers	• Worker activities • Automation decisions • Human/system interaction • Human performance specifications
Logical	Logical decomposition of the system as a coherent set of UML subsystems that collaborate to provide the desired behavior	• Adequate system functionality to realize use cases • Extensibility and maintainability • Internal reuse • Good cohesion and connectivity
Physical	Physical decomposition of the system and specification of the physical components	• Adequate physical characteristics to host functionality and meet supplementary requirements
Information	Information stored and processed by the system	• Sufficient capacity to store data • Sufficient throughput to provide timely access to the data

Process	Threads of control, which carry out the computation elements	• Sufficient partitioning of processing to support concurrency and reliability needs

The viewpoints in Table 1 are some of the most common ones for software-intensive systems. Many system architectures require additional viewpoints that are domain-specific: safety, security, and mechanical viewpoints, for example.

Viewpoints represent different areas of concern that must be addressed in the system architecture and design. If there are system stakeholders or experts whose concerns are important to the overall architecture, there likely is a need for a set of views to capture their design decisions.

It is important to build a system architecture team with staff members whose skills will enable them to manage the various viewpoints. The team might include business analysts and users who take primary responsibility for the worker viewpoint, software architects who attend to the logical viewpoint, engineers who concern themselves with the physical viewpoint, and experts on domain-specific viewpoints.

Model levels

In addition to viewpoints, building a system architecture requires levels of specification. As the architecture is developed, it evolves from a general, abstract specification to a more specific, detailed specification. Consistent with RUP guidelines, there are four architectural model levels in RUP SE, as described in Table 2.

Table 2: RUP SE model levels

Model Level	Expresses
Context	The system and its actors
Analysis	Initial system partitioning in each of the viewpoints to establish the conceptual approach
Design	Realization of the analysis level to hardware, software, and people

Implementation	Realization of the design model into specific configurations

Through these levels, the design goes from the abstract to the physical. The context model level captures all of the external entities (actors) that interact with the system. These actors may be either external or internal to the enterprise that deploys the system. In either case, the actors may be human beings or other systems. At the analysis level, the partitions do not reflect choices of hardware, software, and people. Instead, they reflect design approaches for dividing up what the system needs to do and how the effort should be distributed. At the design level, decisions are made regarding the sorts of hardware and software components and worker roles that are needed. At the implementation level, specific choices of hardware and software technology are made to implement the design. For example, at the design level, a data server is specified. At the implementation level, the decision is made to use a specific platform running a specific database application.

It is important to maintain traceability among these levels. As the enterprise or mission changes, the context level views need to be amended, along with any affected lower-level views. As the underlying technology changes, the implementation level and possibly the design level can be affected. In brief, the impact of enterprise changes flows down, whereas the impact of technology changes flows up.

System architecture views

The next step is to capture the system architecture in a set of views that express the architecture from various viewpoints and model levels. Each of the cells in Table 3 provides a view of the system. Note that at the implementation level, a single diagram captures the realization of hardware and software components for each system configuration.

Table 3: RUP SE model framework

Model levels	**Model viewpoints**				
	Worker	***Logical***	***Information***	***Physical***	***Process***
Context	UML organization view	System context diagram	Enterprise data view	Enterprise locality (distribution of enterprise resources)	Business processes
Analysis	Generalized system worker view	Subsystem view	System data view	System locality view	System process view

Design	System worker view	Subsystem class views Software component views	System data schema	Descriptor node view	Detailed process view
Implementation	*Worker role specifications and instructions*	Configurations: deployment diagram with software and hardware system components			

The relationships among model levels, viewpoints, and views can be seen in Figure 1 below.

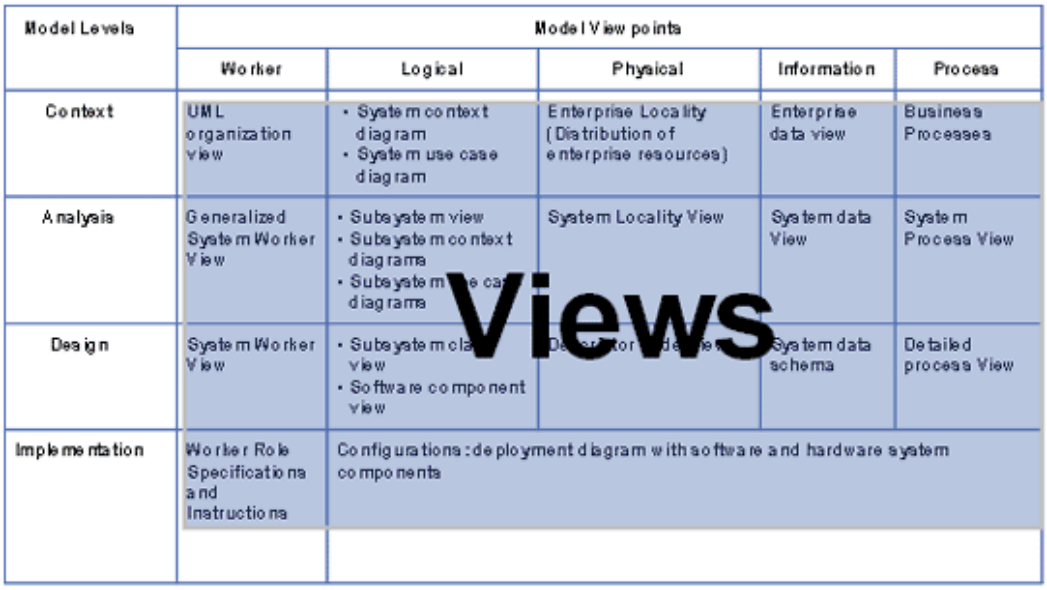

Figure 1: Model levels, viewpoints, and views

The domain-specific viewpoints also should have artifacts in place for one or more of the levels. The set of project artifacts within this framework should be a part of the project development case. Let us briefly investigate each of the viewpoints.

Note: At this writing, UML 2.0 is being readied for adoption and OMG has released a Request for Proposal for a Systems Profile for UML. Once UML 2.0 and the Systems profile are adopted, the RUP SE view semantics will be updated to take full advantage of those standards.

Worker viewpoint

Workers are sufficiently unique to warrant their own viewpoint. Workers are both logical and physical entities. They are logical in that they can, when instructed, provide services and collaborate with other logical entities. They are physical in that they are limited in terms of performance, responsiveness, and capacity. Of course, workers are blackbox entities not subject to further subdivision in the model.

Reasoning about how workers interact with the automated portions of the system and each other is a system engineering specialty. The worker viewpoint provides the setting for this reasoning.

Also, note that *system* workers are not the same as *business* workers. System workers are human beings who are a part of the system. They are *not* system actors, as they are partially responsible for delivery of the system services. In the RUP SE framework, system workers are represented as stereotyped classes. They may be associated if they have dependencies on one another or some other relationship. In the generalized system view, generic system workers are expressed with low detail.

In some applications, it is useful to introduce an abstraction of the automated part of the system -- the *machine*, which differs from a general system in that its realization contains no workers. The generalized workers and the machine may be used in the flowdown workflow to determine worker specifications and to reason about automation decisions. Figure 2 shows an example of a worker diagram.

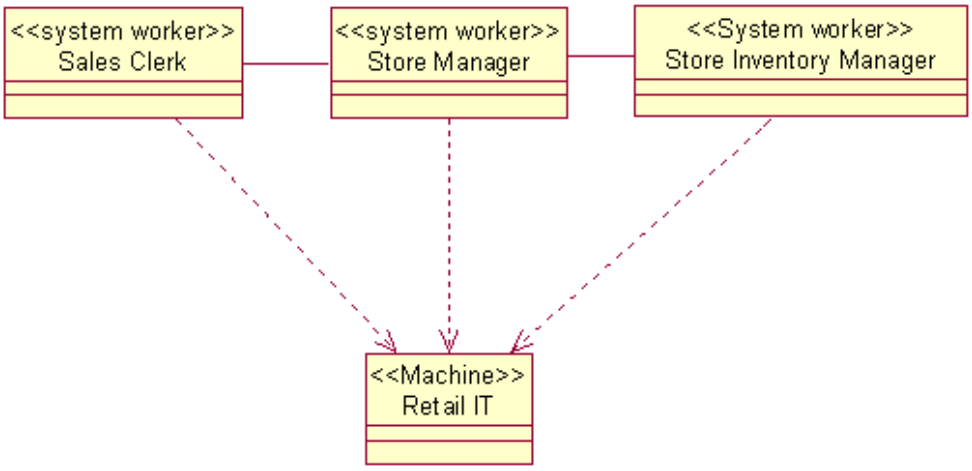

Figure 2: A RUP SE worker diagram

For example, if you were modeling a system for a ship, at the analysis model level you might represent a sailor as a general system worker. At the design level, however, you might define a multitude of specific sailor roles.

Note: You may want to include an additional stereotyped classifier in the worker diagram -- a *machine* -- to support automation decisions. In the joint realization method discussed below, the machine would perform whitebox logical steps to be supported through automation.

Logical viewpoint

The logical viewpoint is the most familiar to object analysts. It describes, at different levels of abstraction, the kind of objects that realize the system. The elements of the views in the logical viewpoint are classes and UML subsystems. In UML 1.4, systems and subsystems inherit from classifiers and packages; there is no UML syntax that captures both the classifier and package aspects of a subsystem. Normally in UML, subsystems are represented as packages with dependencies. In RUP and RUP SE, proxy classes are used to represent the classifier semantics. In RUP SE, we stereotype the proxies and the packages as *systems* or *subsystems* as

appropriate, and, as appropriate, add the system semantics described above to subsystems. Figure 3 shows a UML subsystem view for a click-and-mortar retail system using the common notation. One could choose to use subsystem classifiers in place of the packages in this figure.

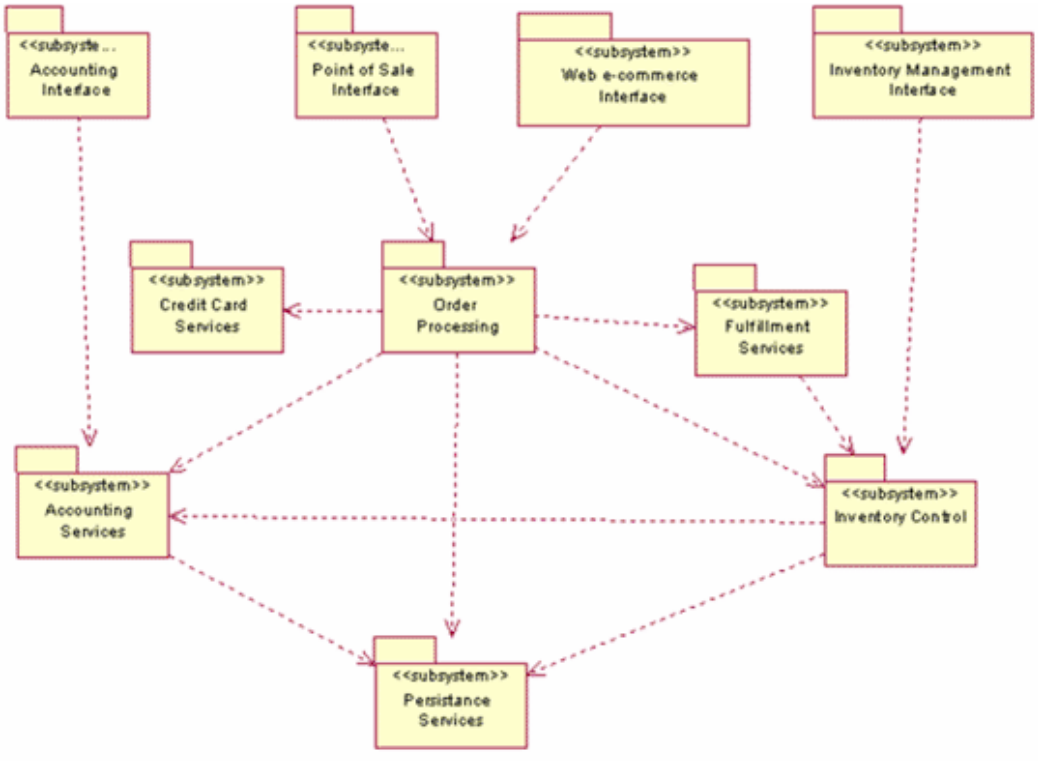

Figure 3: Click and mortar subsystem diagram
Click to enlarge

Physical viewpoint

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 environment 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 shown in Table 3, RUP SE uses an analysis level, physical viewpoint diagram called *System locality view*. In the physical 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.

As shown in Figure 4, locality diagrams consist of two elements:

- **Localities:** groupings of physical resources that enable a conceptual, physical partitioning of the system. Their icon is a rounded cube.
- **Connections:** linkages between the localities that may be used to pass data, service requests, or I/O entities. Connections are represented in UML as stereotyped associations.

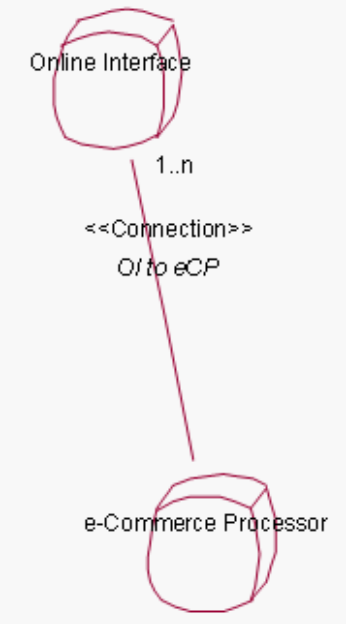

Figure 4: Locality diagram elements

Locality semantics

Localities are used to realize the physical characteristics of the system class, and their semantics derive from those associated with the physical nature of the system. In particular, localities have class and instance attributes, and measures of effectiveness captured as tagged values. Localities have two default sets of tags:

- **Quality:** reliability, availability, performance, capacity, and so on
- **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.

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 the section on **Localities, services, and interfaces** below, we will 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

Since 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 may 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.

Localities and nodes

Recall that UML nodes are classifiers that have processing ability and memory.[1] 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.

The 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 physical 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."[1]

In RUP SE, this intent is preserved. A node, then, is a special sort of locality that is used at the design and implementation model levels to specify physical resources that execute software. However, as a kind of locality, RUP SE nodes can be stereotyped to include all of the locality semantics. Note that these semantics differ from standard nodes in UML. Localities are not so much stereotyped nodes as nodes are stereotyped localities. UML 2.0 will provide better means for dealing with

physical partitioning.

Localities, services, and interfaces

A locality specifies the physical resources that provide logical services. 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 workflow we will describe below.

The set of hosted subsystem services for a given locality can be captured in a couple of ways:

- Survey of hosted subsystem services document
- Associated subsystem interfaces

The first method is simpler, associating a requirements document with a locality. The second requires a more sophisticated use of the UML. Subsystems are classifiers, and their services are classifier operations. In addition, the UML allows operations, and therefore subsystem services, to be organized into interfaces. That is, an interface is a subset of subsystem services. In this second approach, one defines the needed interfaces for each of the subsystems and then assigns them to the appropriate localities. Generally, there will be more than one interface associated to a locality.

Design trades

"Design trades" is the name of a common system 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 of the system.

Figures 5 and 6 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) depicts processing capability in the stores. The second solution (Figure 6) 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. Today, most people would agree that Figure 5 represents a better design; however, the solution in Figure 6 may be considered superior in a few years.

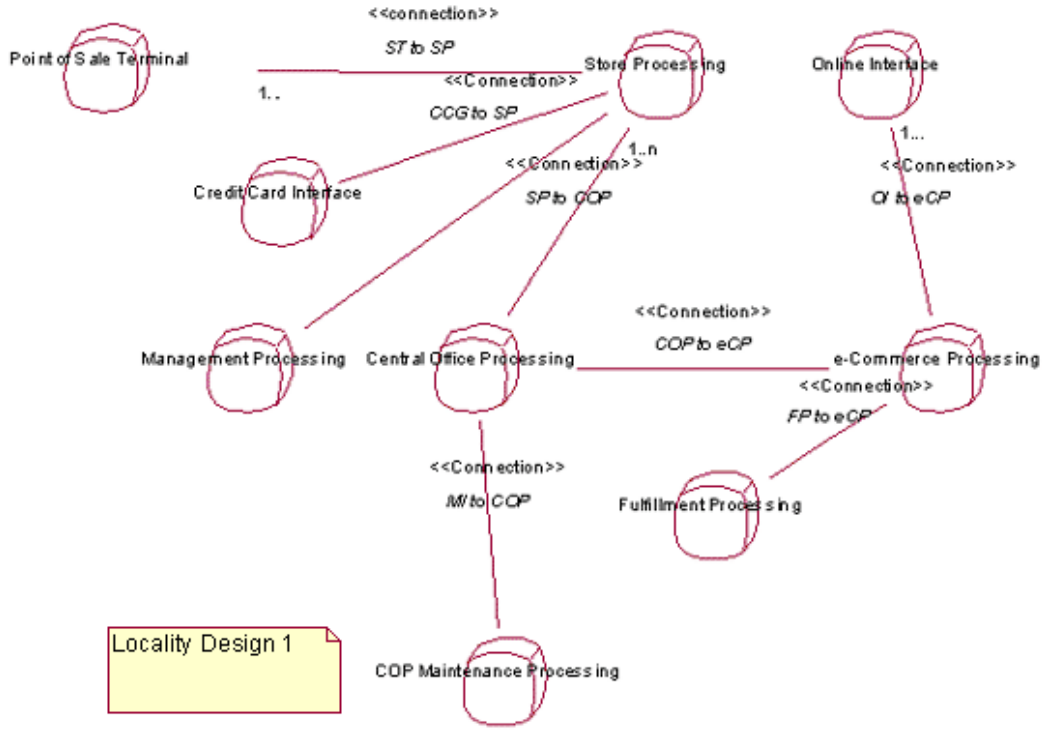

Figure 5: System locality view -- Example 1

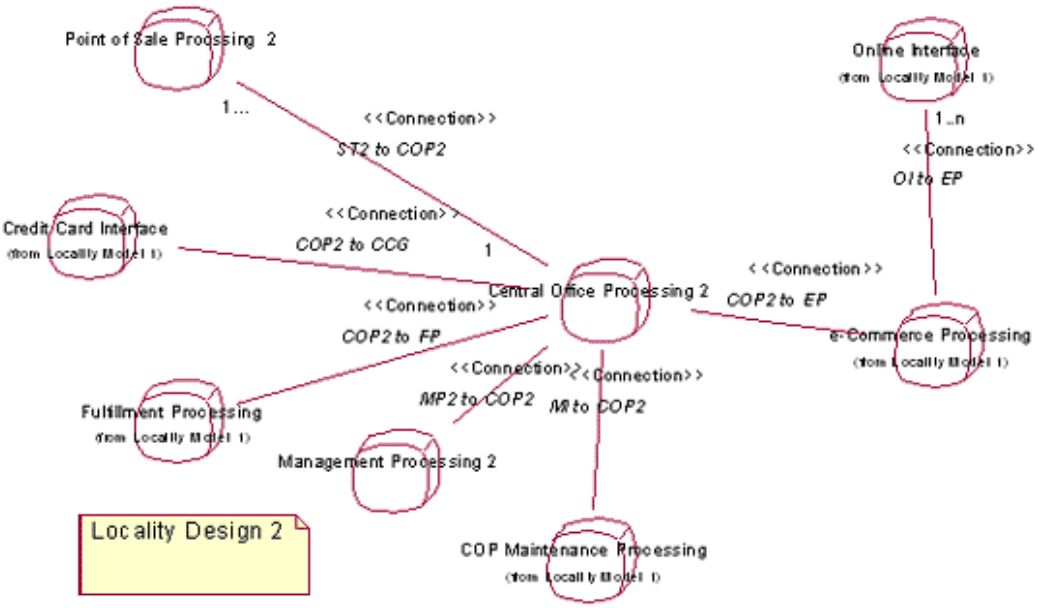

Figure 6: System locality view -- Example 2

Locality decomposition and realization

Like subsystems, localities can be decomposed hierarchically into further localities. It is tempting to use aggregation to associate the localities with the sublocalities. However, there is a critical difference between the whole-part relationship in a physical decomposition and the relationship normally expressed with class aggregation: In common usage, when a class object (whole) is aggregated from

other class objects (parts), the whole's attributes include the attributes of the parts. The attributes of the whole in a system locality are *functions* of the attributes of the parts. A simple example is that the weight of the whole is the sum of the weight of the parts. Often the relationship between a whole attribute and a part attribute is much more complex, yet there are no current semantics in the UML to express functional relationships between attributes. A workaround for capturing the relationship can be inserted in the model, using private attributes and operations that carry out the functions.

When realizing localities as physical components, we suggest that the realization be hierarchical. That is, each component is part of the realization of no more than one locality. Otherwise, it is difficult to maintain the traceability of derived nonfunctional requirements between localities and the components. However, you need not follow this suggestion if you want to have reusable components across localities. In this case, of course, the components will have to meet the most stringent requirements, discovered through flowdown of the system requirements across localities.

In addition, in some cases it makes sense to create multiple realizations of the same locality, flowing down to multiple system implementations. For example, one might have a product line with different implementations to support a range of price/performance points.

Information viewpoint

The use of UML for both object and relational database modeling is a well-developed practice that RUP SE makes use of in the information viewpoint. Note that maintaining the database modeling in the system model permits overall system coherency by supporting associations between data and functional classes, and by assigning database components to localities.

Process viewpoint

The process viewpoint is also represented using standard UML.[3] Figure 7 shows an example of a system process view.

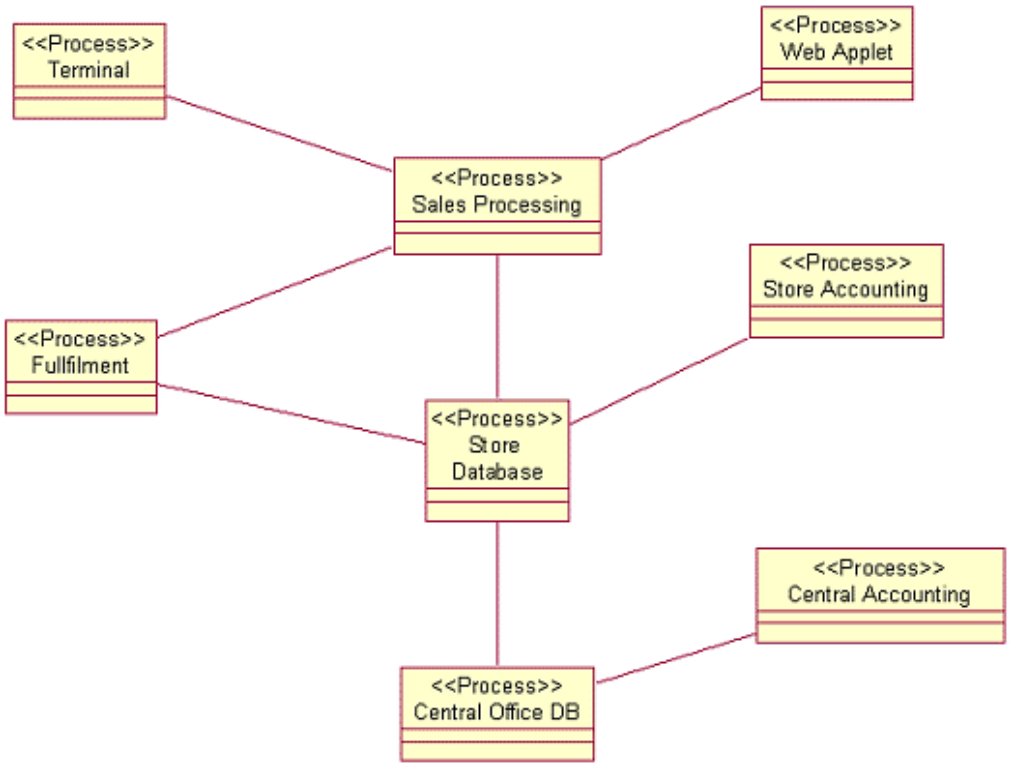

Figure 7: Sample system process view

Moving between model levels

Moving down model levels adds specificity, not accuracy, to the models. At each level, you need to be as accurate as possible in specifying model elements, because accuracy at each level adds to the understanding of the system and discipline of the process. As you move down the levels, each view is a more specific decision, resulting in configuration items at the implementation level. It is important to note *that the model elements at one level establish the requirements at the next level.* Or, as indicated in Figure 8, we can say that each model level *realizes* requirements discovered at a higher level. For example,

- The **analysis model level** shows how requirements specified in the context model level are met.
- The **design model level** shows how requirements arising from the system analysis model level are met.
- The **implementation model level** meets design specifications.

Model Levels	Model Viewpoints				
	Worker	Logical	Physical	Information	Process
Context	UML organization view	• System context diagram • System use case diagram	Enterprise Locality (Distribution of enterprise resources)	Enterprise data view	Business Process
Analysis	Generalized System Worker View	• Subsystem view • Subsystem context diagrams • Subsystem use case diagrams	System Locality View	System data View	System Process View
Design	System Worker View	• Subsystem class view • Software component view	Descriptor node View	System data schema	Detailed process View
Implementation	Worker Role Specifications and Instructions	Configurations: deployment diagram with software and hardware system components			

Realization

Figure 8: Lower model levels realize requirements established at upper model levels

Click to enlarge

Figure 9 shows an example of how the physical viewpoint at the design level contains a descriptor node diagram, which shows a physical design that realizes each locality.

Because each model level establishes requirements or specifications to be realized at the next lower level, you can maintain traceability between levels by capturing how design elements meet those specifications.

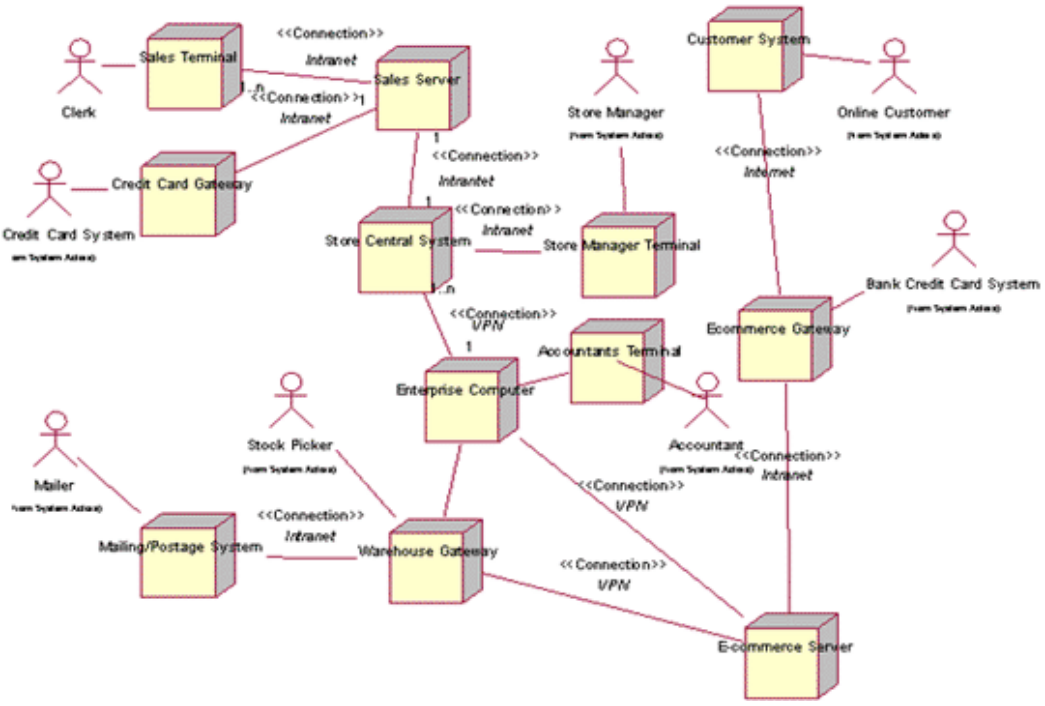

Figure 9: A realization of the click-and-mortar locality view

Click to enlarge

In practice, as a team develops a model level, they may likely discover that the upper level should be revised because, for example, one or more elements it

specified cannot be realized. Hence, as development proceeds, no level is ever really "frozen"; each is maintained throughout development. However, as development proceeds, the focus of the effort typically moves down, level by level.

Notes

[1] Grady Booch, James Rumbaugh, and James Jacobson, *The Unified Modeling Language User Guide*. Addison Wesley, 1999, p.358.

[2] Ibid, p.252ff.

[3] Ibid, p.455.

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