Modeling in the Large with the Rational Unified Process—Part I: Concepts

By Peter Eeles & Maria Ericsson

Introduction

Developing a system is rarely about developing a single software application. The business problems that the system is intended to address requires a much broader perspective, one that considers the system as consisting of other systems and that involves, perhaps, many interdependent projects. This paper expands on the Rational Unified Process® or RUP® product, which, “out of the box,” focuses on executing a single project to develop a single software application. In particular, we consider the RUP as a process framework for initiatives such as:

- **Enterprise architecting**: defining an architecture\(^1\) that underpins a number of systems,

- **Strategic reuse**: developing reusable assets that are used within a number of systems,

- **Systems engineering**: developing a system that contains elements of hardware, software, workers, and data,

- **Enterprise Application Integration (EAI)**: developing a solution that includes the integration of a number of legacy systems,

- **Packaged application development**: developing a solution that includes the configuration of a packaged application, such as an Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) solution, and

- **Outsourced development**: defining an architecture that lends itself to the outsourced development of its constituent parts, while ensuring the quality and integrity of these parts.

\(^1\) In this paper, the term “architecture” is very broad and encompasses software architecture, hardware architecture, organizational structures, and so on.
Furthermore, organizations undertake a number of these types of initiatives in combination because of the business situation they are in, or because of technological factors. We explore the circumstances under which such initiatives take place.

Each of these initiatives is often considered as a “system of systems.” In other words, although the result of each initiative can be considered to be a “system,” the inherent complexity involved requires us to decompose this system into a number of “subsystems” that are implemented within a number of projects. The relationship between the system and its associated subsystems requires careful consideration of a number of areas including:

- Architecting
- Project management
- Requirements management
- Change management
- Testing

This paper focuses on the architecting aspects of a system of systems and has two main influences: the RUP and the System of Interconnected Systems pattern (discussed below). In the following sections we give a high-level description of the use of the RUP in supporting initiatives such as those described above. This paper does not provide an introduction to the RUP; such introductory information can be found in [Kruchten] and the RUP itself.

“System” is a key term in this paper and before proceeding we should be more precise about what we mean by it. Consider the following definitions:

“A system is a top-level subsystem in a model. A subsystem is a grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements.” [UML]

“A system is a collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints.” [RUP]

“A system provides a set of services that are used by an enterprise to carry out a business purpose. System components typically consist of hardware, software, data, and workers.” [RUP-SE]

---

Although these definitions provide different levels of detail, they are remarkably consistent. In particular, we can derive the following assumptions about the nature of a system:

- A system exhibits behavior.
- A system can contain other elements.
- A system fulfils a specific purpose.
- A system can be composed of hardware, software, data, and workers.

Another key term in this paper is “pattern,” which we use in a general sense as meaning a “common solution to a common problem in a known context.” Christopher Alexander has a more elaborated definition that (we find) fits well with the system of interconnected systems pattern:

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” [Alexander]

The impetus for this paper is that we have seen a growing need among our customers to understand how all their development efforts tie together. This paper will not give all the answers, but can act as a first step on the way to understanding how to scale up the process framework provided in the RUP to meet these needs.

**The System of Interconnected Systems Pattern**

The system of interconnected systems architectural pattern is used to help control the complexity inherent in a system of systems. One of these systems represents overall capability, and is referred to in the pattern as the **superordinate system**. The other systems represent a part of this overall capability, and each is referred to as a **subordinate system**. A superordinate system is clearly distinct from the subordinate systems that implement it, and the relationship can be formalized (as we shall explain later) in that, from the perspective of the superordinate system, the subordinate systems are subsystems of the superordinate system, as shown in Figure 1.
Another very important characteristic of the system of interconnected systems pattern is that it is **recursive**, meaning that a subordinate system may also have subsystems of its own and be **superordinate in relation to them**. This characteristic is particularly important to initiatives such as enterprise architecting and systems engineering, as discussed later.

The system of interconnected systems pattern is documented in more detail in a number of publications, including *Software Reuse—Architecture, Process and Organization for Business Success* [Jacobson] and “Developing Large-scale Systems with the Rational Unified Process” [Ericsson] (the principles of which are consistent with the approach described in [Cantor] and applied in [RUP-SE]). [Jacobson] uses the pattern primarily with respect to its value in defining strategic reuse initiatives; [Ericsson] describes, at a high level, the use of the pattern within the context of the RUP. In this paper, we describe a more detailed alignment of the pattern with RUP.

**The System of Interconnected Systems Pattern and the RUP**

Before discussing the use of the system of interconnected systems pattern within the context of RUP, we should first be clear about the distinction between systems and projects. In particular, the system of interconnected systems pattern is primarily concerned with the **architectural decomposition of a large and complex system into a number of subsystems**. The RUP, on the other hand, is primarily concerned with the **execution of a project**. These two facets—architecture and projects—should not be confused.

It is true to say that it is often the case that a particular system (superordinate or subordinate) is best implemented as a single project. However, this paper acknowledges that this is a simplification in that a single project may implement a number of systems, and a single system may be implemented as a number of projects. On a related note, it is sometimes beneficial to have the mindset that a superordinate system is a “living entity” that, to a large extent, is never “finished.” As a result, a superordinate system typically undergoes a series of sequential “development cycles” (one “pass” through the RUP phases of inception, elaboration,
construction, and transition) that contribute to the ongoing evolution of the superordinate system. Each of these development cycles are considered within a separate project.

We have found that applying the RUP to the various initiatives mentioned in this paper can effectively be described in terms of RUP artifacts, and this is the approach we take below. Related concerns, such as architectural views, are also touched on, but only lightly. For a discussion of the detailed process that can be applied to specific initiatives, we refer the reader to [Cantor] and [RUP-SE] for systems engineering and to [Jacobson] for strategic reuse. Figure 2 shows the relationship between a superordinate system and a number of subordinate systems in terms of some key RUP artifacts, and provides a framework for thinking about specific artifacts of a superordinate project, specific artifacts of a subordinate project, and the relationships between them (additional relationships are considered in our upcoming paper “Modeling in the Large with the Rational Unified Process–Part 2: Example”). The artifacts are aligned with the RUP disciplines in which they are produced. A description of each of these artifacts can be found in [RUP] and [Kruchten]. Although the RUP is typically applied to projects involving software elements only, the concepts and best practices underpinning the RUP (such as requirements management) also apply to nonsoftware elements.

Figure 2 shows that, in general, a particular subordinate artifact is constrained in two dimensions. The first dimension is the relationship with artifacts associated with the superordinate system. For example, a subordinate design model is constrained by the superordinate design model. The second dimension is the relationship with artifacts associated with the same subordinate system. For example, a subordinate design model is constrained by
the subordinate analysis model that it refines. Each of the relationships shown in Figure 2 is described in “Modeling in the Large with the Rational Unified Process–Part 2: Example.”

It is also worth noting that the superordinate system is concerned primarily with providing a “broad brush” perspective, concentrating only on elements that are considered to be architecturally significant. However, it is the obligation of each subordinate system to provide any detail required. In this respect, each subordinate system can be perceived as “populating” aspects of the overall superordinate system. It is therefore also true to say that the development of a system of systems is both top down (in that the superordinate system provides a context for each subordinate system) and also bottom up (in that each subordinate system populates aspects of the superordinate system). RUP for Systems Engineering [RUP-SE] provides some prescription of how to proceed—in particular, top-down modeling of use cases (use-case flowdown). The discussion in this paper focuses on dependencies from subordinate artifacts to superordinate artifacts. However, we acknowledge that, in more general terms, the superordinate system is also dependent on each of the subordinate systems that implement it.

So what comes first—the superordinate system or the subordinate system? It is actually a classic “chicken and egg” problem—one that some call a “design paradox.” The whole cannot be defined without understanding the technicalities of the parts, and the parts cannot be defined in detail without understanding the whole. This tells us that they are interdependent, and their development should go hand in hand. There is often a tendency to solve complex problems top down, just because it is easier to break a large problem down into a number of smaller and more manageable problems than attacking the whole beast in one go. However, the risk is that we forget to consider how the solution of each of the smaller problems impact the overall solution and get bogged down in building parts that in the end don’t fit together. Another common situation is when we have many “parts” (e.g., when integrating legacy systems) that we’d like to fit together; we then need to create the whole based on what the given parts are able to provide.

**How to Apply the Pattern**

In this section we discuss the application of the pattern in the context of each of the initiatives discussed earlier: enterprise architecting, strategic reuse, systems engineering, EAI, packaged application development, and outsourced development.

**Enterprise Architecting**

Enterprise architecting is concerned with providing a “platform” for developing all systems that comprise an enterprise and typically has concerns within a number of areas, such as data, functionality, geographic distribution, and people.

The decomposition of an enterprise into its respective elements can be expressed in terms of a “system of systems” where the enterprise itself is aligned with the concept of a superordinate

---

3 EAI initiatives are often considered to be “bottom up” since they entail the integration of already implemented legacy systems.
system, as described in this paper, and the elements of the enterprise aligned (in this context) with the concept of a subordinate system. It is especially important to note that the system of interconnected systems pattern is recursive since enterprise architecting initiatives are often described at a number of levels.

**Strategic Reuse**

There are many dimensions to a strategic reuse initiative, including business concerns (such as return on investment decisions) as well as technical concerns. In this paper, we are concerned primarily with the technical aspects of “architecting for reuse.” Also, although reusable assets can take many forms (including documents and models), this paper focuses primarily on elements of the final system, such as software, or reusable hardware.

Reusable assets (whether a simple component or an entire system) do not, by definition, exist in isolation (since they are reused within a number of contexts). A fundamental premise of any strategic reuse initiative, therefore, is to define the services that each asset provides and any services that the asset requires.

These assets and their relationships can be described in terms of a system of systems. For example, [Jacobson] considers the concepts of an “application system” and also a “component system.” In simple terms, an application system represents an application that provides value to an end user, whereas a component system represents a set of components that are used by a number of application systems. The system of systems that shows the relationships between the application systems and component systems is described in terms of an overall layered architecture—the product of “Application Family Engineering,” which other texts refer to as “Product Line Engineering.”

**Systems Engineering**

As discussed earlier, systems engineering is concerned with hardware, software, workers (people), and data. Two aspects of the process of systems engineering are identifying the elements that comprise the system and understanding the relationships between them. In identifying the elements, nonfunctional requirements in particular, such as performance and cost, are taken into account. For example, we may choose to implement a system element in hardware rather than software for performance reasons, or in terms of workers (people) rather than software for usability reasons (where an end user may have a better “user experience” of the system when interacting with a human being!).

The system as a whole can be expressed in terms of a superordinate system, which identifies the elements that comprise the system and the relationships between them. Each of these elements would be expressed in terms of a subordinate system. Again, it should be borne in mind that the system of interconnected systems pattern is recursive since most systems engineering efforts require more than two levels of decomposition. Note that various system boundaries (and subsystem boundaries, and potentially subsubsystem boundaries) in system engineering actually enclose people, hardware (computational and noncomputational), and software.

Enterprise Application Integration

EAI is concerned with developing a solution that includes the integration of a number of legacy systems. Such efforts are, to a large extent, driven from the “bottom up” since elements of the solution already exist. However, there is still some “top down” effort required to understand the context within which such legacy systems will “fit.” Also, if the legacy system represents software, it is often the case that techniques such as “wrapping” interfaces to the legacy software applications is required, together with a good understanding of the available middleware technologies that can be used to interact with such applications.

The context within which a legacy system fits can be described in terms of a superordinate system, with the legacy system itself represented as a subordinate system.

Packaged Application Development

Packaged applications can be considered customizable frameworks that allow you to build what could be considered a family of applications that support a certain aspect of a business such as CRM or HR (Human Resources). These frameworks can be considered at two levels.

First, such frameworks often implement a piece of a larger system. In this context, the packaged application (or a piece of the packaged application) represents a subordinate system. Second, such frameworks are often large and complex, and thinking of them as a system of systems in their own right can help us understand them, especially when it comes to understanding how the packaged application will be applied in terms of what pieces will be used as-is, what pieces will be used after modification or configuration, and what pieces will not be used at all.

Outsourced Development

An organization may outsource development in order to supplement development capacity or reduce costs. In such circumstances, it is vital that architectural integrity is maintained between the system under consideration and the constituent parts, some of which may be outsourced. The constraints imposed on the development of the “parts” may vary. For example, an outsourced part may be described in terms of the requirements it must fulfill, where there is complete flexibility in terms of the solution. On the other hand, an outsourced part may be described in terms of a detailed set of services that it must provide and even some specification of how it will be implemented.

Whatever the scope of the outsourcing, such circumstances can benefit from describing the overall architecture in terms of a superordinate system with the constituent parts described in terms of subordinate systems.

When to Apply the Pattern

This section discusses the circumstances under which the pattern is applied. These circumstances are either business driven or technology driven.
Merging Organizations

Organizations merge to (among other things) save costs and then find that they struggle with merging their systems due to the complexity of this task. There are many difficult decisions to make: what future platform to choose, which systems to keep and which to replace, and how to tackle the business impact of changing systems, to name a few. A merger is a system of systems problem involving people, hardware, and software. It may involve the following kinds of initiatives:

- **Enterprise architecting**: to get an overall understanding of the problem
- **Enterprise application integration**: to integrate existing and new systems
- **Strategic reuse and outsourced development**: may be part of the effort, but not always

Modernizing Legacy Systems

Organizations find it necessary to move to new technologies to modernize their systems. This is often the case when an organization finds that, over time, it becomes more and more costly (and time consuming) to add capability. In such cases, re-architecting and replacing legacy systems makes business sense. A typical approach is one in which one or more legacy systems are gradually replaced and integrated on a schedule that ensures any business or technical risks are managed appropriately. Understanding the overall business impact of evolving legacy systems is a system of systems problem involving people, software, and hardware. This type of effort may involve the following initiatives:

- **Enterprise architecting**: to understand the impact of introducing modern technology
- **Enterprise application integration**: to integrate legacy systems with new systems and also legacy systems with other legacy systems
- **Packaged application development**: to introduce new technology such as an ERP solution

Building Technically Complex Products

Building technically complex products, such as telecom network products or air-traffic control systems, has always been considered a system of systems problem involving both hardware and software. As technologies are now changing more rapidly than ever before, there is a stronger need for a thorough treatment of such problems. It is no longer sufficient to break down the problem into smaller pieces and resolve them independently while handling dependencies along the way—you need a well-defined approach that helps you proactively handle dependencies and
be flexible. Architectural decisions are more difficult to make, and such decisions can no longer rely only on peoples’ experience. More efficient methods to assess the impact (cost, resources, architectural, or technology impact) of these decisions are required. This type of effort may involve the following initiatives:

- **Systems engineering**: the entire initiative is typically treated as a systems engineering effort
- **Outsourced development and reuse**: this may be part of the effort

### Hardware Development Going Soft

Many organizations that traditionally considered themselves as builders of hardware are becoming increasingly dependent on building quality software. Consumer products such as mobile phones, TVs, and cars contain more software than ever before. Designing the hardware and software to work optimally together is a system of systems problem. This type of effort may involve the following initiatives:

- **Systems engineering**
- **Strategic reuse**: there is very often a focus on reuse

### Using the Pattern Effectively

As we have seen, the system of interconnected systems pattern has many uses and can also be applied in different ways. In this section, we look at the practicalities of applying the pattern.

First, consider whether or not the pattern should be applied at all. There are many instances where a system can be considered in its entirety, without the need to manage complexity by considering a number of separate subordinate systems. *The pattern is most applicable when the benefit of managing the complexity of the problem outweighs the overhead of defining subordinate systems* (and the production of additional artifacts). For example, if we were to build an air-traffic control system, we might face complexities such as hardware as well as software development; the need to clearly separate the responsibilities of (and boundaries between) system elements so that they can be outsourced; and geographically distributed teams. The different models of the pattern give you the communication mechanism (interfaces and abstraction) needed to manage the team dependencies. The benefit of treating the system as a system of systems would, in this case, outweigh the cost of defining a number of separate subordinate systems.

Should we decide to apply the pattern then we must identify the artifacts that should be produced for both the superordinate system and subordinate system (since we don’t necessarily need to produce all the artifacts). Although this paper discusses all of the key RUP artifacts (and the relationships between them), there is a cost involved in their creation and maintenance. We
therefore need to pragmatically think through the value that a particular artifact adds, the cost of creating and maintaining it, and the risk of not creating and maintaining it. Below we list some examples of choices that are often made with respect to certain artifacts:

- **Business Use-Case Model and Business Object Model.** Business modeling is often performed for a superordinate system, but not always for a subordinate system. This is the case when the business modeling effort associated with the superordinate system provides sufficient input to the development of the respective subordinate systems. However, business modeling may be performed for the subordinate system when the superordinate system is very large and complexity needs to be managed. In particular, you see business modeling associated with the subordinate system if there is a need for more detailed understanding of a particular aspect of the organization.

- **Use-Case Model.** It is common for requirements artifacts to be produced for both a superordinate system and a subordinate system. The trick is to find the right level of detail at the superordinate level—ensuring that enough detail is provided to those defining the subordinate level requirements, without doing their work for them. For example, we may see superordinate use cases identified and briefly outlined (with an emphasis on defining architecturally significant elements), but without detailed specifications of the flow of events. Such detail will be provided at the subordinate level.

- **Analysis Model and Design Model.** The analysis and design artifacts are critical for defining architecture. However, similar to requirements artifacts, the emphasis is different between the superordinate system and the subordinate systems. The development of the superordinate system is focused on defining an architecture within which the subordinate systems will be constrained, and also for identifying the subordinate systems. However, this effort may only extend as far as identifying the architecturally significant responsibilities (functional characteristics) and quality attributes (nonfunctional characteristics) of each subordinate system. The focus of the development of a subordinate system is to add the detail necessary to ensure that the subordinate system meets these responsibilities and quality attributes, in terms of the elements that implement the subordinate system. In some cases, an analysis model may only be created for the superordinate system since it can greatly assist in the decision making process with respect to the architecture at the superordinate level. However, such decision making may not be necessary at the subordinate level, and so the analysis model can be omitted for the subordinate system.

- **Implementation Model.** The development of a superordinate system may not undertake any implementation at all (other than that implied by the implementation of each of the subordinate systems that comprise the superordinate system). However, some implementation may be undertaken to prove aspects of the architecture of the superordinate system, or to ensure that some common components (for example) are available to each subordinate system. The development of subordinate systems...
typically includes an element of implementation in order to deliver the subordinate system (unless it is, itself, further decomposed).

- **Test Artifacts.** In a similar manner, some testing may be undertaken within the development of a superordinate system to validate the superordinate system, and is always required to test the superordinate system in terms of the integration of each of the subordinate systems. Testing undertaken within the development of a subordinate system is performed to validate the implementation of the subordinate system.

Rationalizing about what information should be specified at what level (superordinate or subordinate) can be overwhelming in and of itself. There is a tendency to treat the superordinate level very lightly and focus the majority of the work at the subordinate level. The risk is that the overall architecture of the superordinate system is not as coherent as desired since many decisions are made at the subordinate level, and the result is perhaps not quite as consistent as would be the ideal. On the other hand, there is the risk that the work at the superordinate level goes into too much detail that needs to be redone at the subordinate level. *The challenge is to find the right balance between providing sufficient detail to ensure consistency between subordinate systems, while allowing enough flexibility at the subordinate level without providing artificial constraints.* There are no simple rules for how to do this; it needs to be decided based on the characteristics of the system of systems being developed.

**Architectural Representation**

In this paper we have focused on the models that are used to describe different aspects of a system. However, it is common practice to also define an architectural representation of a system—one that omits elements that are not deemed to be architecturally significant. This representation is often expressed in the form of “architectural views,” where each view provides a particular perspective of a subset of one or more models.
There are a number of standard architectural representations available, and each is appropriate depending on the nature of the system being described. Examples include the following:

- The 4 + 1 Views of Software Architecture, defined by Philippe Kruchten, is the architectural representation advocated in the RUP. For more information see [Kruchten] and [RUP].

- The C4ISR (Command, Control, Computers, Communication, Intelligence, Surveillance, and Reconnaissance) Architecture Framework, defined by the U.S. Department of Defense (DoD), is the standard used in military domains. Background rationale for its use can be found in [West].

- The IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (ANSI/IEEE-1471-2000) standard provides a conceptual framework for architectural description and defines what is meant by a 1471-compliant architectural description. For more information see [IEEE 1471].

- The Reference Model for Open Distributed Processing (RM-ODP) is an ISO standard. For more information see [Putman].

- The Zachman Framework, defined by John Zachman, is most often associated with enterprise architecting. For more information see [Zachman]. An alignment of the Zachman Framework with RUP artifacts can be found in [De Villiers].

- The RUP SE model framework introduced in the Systems Engineering plug-in to the Rational Unified Process is available from the Rational Developer Network. For more information see [Cantor] and [RUP-SE].

Iterative Development

There is one particular characteristic of the RUP that should not be overlooked: iterative development. Although this paper does not emphasize this particular process, we do acknowledge that taking an iterative approach to the application of the pattern is critical if we are to ensure success. This is true of all of the system elements discussed—software, hardware, data, and workers.

The decision space associated with the initiatives discussed in this paper is huge. The likelihood of defining a successful architecture up front is extremely small and so the only effective means to converge on a suitable architecture is to design a little, implement a little, test a little, incorporate lessons learned, design a little, implement a little, test a little, ... and so on. Iterative development is discussed in detail in a number of sources, including [RUP] and [Kruchten].
Particular consideration of iterative development with respect to a system of systems can be found in [RUP-SE] and [Jacobson].

**Summary**

In summary, the RUP can be applied to a diverse set of initiatives, ranging from enterprise architecting to systems engineering. The system of interconnected systems pattern provides a means of managing complexity within such initiatives and is complementary to the best practices underpinning the RUP.

In our upcoming paper, “Modeling in the Large with the Rational Unified Process–Part 2: Example,” we will describe in detail various RUP disciplines as applied to the development of both a superordinate system and a subordinate system. This discussion also exemplifies a number of the artifacts discussed in here in Part 1.

**Acknowledgements**

The authors would like to thank the following people for their help and guidance in writing this paper: Roger Bowser (Rational), Dave Brown (Rational), Murray Cantor (Rational), Christina Cooper-Bland (BACS), Ivar Jacobson (Rational), Jon Pidgeon (Lloyds TSB), John Smith (Rational), Dave West (Rational), Alan Whitfield (UK Inland Revenue) and Gary Willcocks (EDS).

**References**

The following references have been used in the preparation of this paper.


white paper.


[Rational] Business Modeling with the UML. Rational University training course.


