

▶ **Transitioning from Requirements to Design**

by [Paul Reed](#)

President

Jackson-Reed, Inc.

One of the biggest challenges facing software projects is determining when and how to begin the transition from specifying requirements to working on a system design. Questions arise such as: When am I done with use cases? Do I detail all the use cases first and then jump into design? Which requirements should I tackle first in design?

Most team members seem to realize the benefits of making a correct decision as well as the downside of making an incorrect one. Leap too early, and the project runs the risk of formulating a design based on sketchy requirements. Leap too late, and you take on more risk by postponing high-risk architecture decisions until later in the project lifecycle. In addition, when preparing the artifacts to transition from requirements to design, you must take care not to lose the context in which they were originally captured (e.g., Who made the decisions? Were there unique relationships between certain requirements?).

This article lays the foundation for a smooth transition from requirements specification to design by focusing in on those items that present the most trouble. First I will discuss just how far a team should go with use cases before beginning design. Second, I will review a framework for identifying architecturally significant requirements. And finally, I will explain how to utilize use-case realizations as the pivotal artifact to bridge the transition from requirements specification to design.

When Exposing Risk Is a "Good Thing"

We know, based on history, that the traditional waterfall approach to software development does not mitigate the greatest risks early in the project's lifecycle. This stems from the fact that high-risk decisions, such as the architectural direction of the project, are not tested for validity until coding begins. This can lead to disastrous results; many waterfall projects



▶ [subscribe](#)

▶ [contact us](#)

▶ [submit an article](#)

▶ [rational.com](#)

▶ [issue contents](#)

▶ [archives](#)

▶ [mission statement](#)

▶ [editorial staff](#)

are either trimmed back or cancelled altogether after significant investments have been made.

The Rational Unified Process® (RUP®) stands in contrast to this approach because of two key characteristics: 1) RUP is risk-based; 2) RUP is architecture centric. RUP challenges the project team to, early on, identify requirements that will both expose the greatest potential risks and allow them to put plans in place to mitigate those risks. These so-called *architecturally significant* requirements are typically uncovered during the Inception phase of RUP and are the target of further analysis and design in the first iteration of the Elaboration phase.

Exposing this risk is a *good* thing for the project, because it demands that the team formulate mitigation plans as soon as possible. The challenge in moving forward is answering the all important question: How far does the team need to go with use cases before taking that initial leap into design?

A Mile Wide and Five Inches Deep

Before the project can identify architecturally significant use cases (see sidebar), it must flesh out the requirements enough to make intelligent decisions about where the greatest risk lies. The key to how much detail to provide lies in the mantra, "a mile wide and five inches deep."

In the Inception phase of a project, my standard script when dealing with project teams is to have them perform the following tasks:

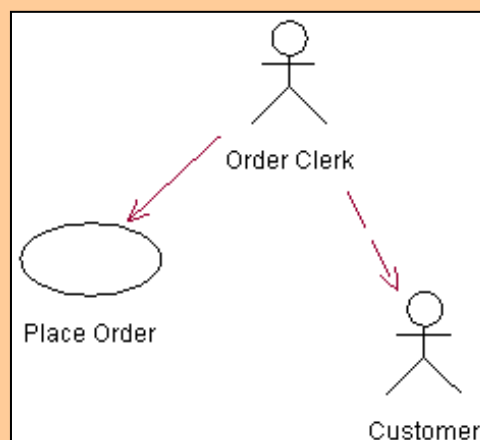
1. Identify the project stakeholders and key product features for the system under discussion (RUP's Vision artifact).
2. Brainstorm an event list by first identifying the actors (humans, other systems, hardware devices, and timers) and what events they stimulate the system with.
3. Brainstorm which use cases satisfy those

What's Up with Actors and Use Cases?

Projects have struggled for years to conceive the best approach to elicit, document, and trace functional requirements. Approaches range from mini-specifications -- a text narration of the requirements in paragraph form -- to diagrams that show each requirement's flow of control.

Rational's Ivar Jacobson pioneered the notion of use cases while working on complex telecommunications projects at Ericsson. The use-case approach focuses first on identifying the Actors or users of the application. Actors typically take one of four forms: humans, other systems, hardware devices, or timers. They have a goal that needs to be satisfied by the system, and they rely on the use case to accomplish that.

Use cases represent major categories of functionality as perceived by the application's user; it is the use case that ultimately provides measurable value to the Actor. Use-case diagrams are complemented by a use-case template for each use case. Figure 1 shows a use case with Actors.



events.

4. For each use case, complete a use-case template that at a minimum will identify the key pathways. These pathways may be broken into categories: most common or happy path, variations to the happy path, and exception paths.
5. Time permitting, detail the steps of the happy path for each use case, or at a minimum the happy path for the project's central use cases.

6. Identify those requirements that represent the greatest amount of risk.

Figure 1: A Diagram for the Place Order Use Case

Want more information on Actors and Use Cases?
Try these sources:

Web

- <http://www.rational.com/uml/index.jsp>
- <http://www.usecases.org/>
- <http://www.therationaledge.com/admin/archives.jsp>
(search for Use Cases)

Books

- Alistair Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2001.
- Daryl Kulak and Eamonn Guiney, *Use Cases: Requirements in Context*. Addison-Wesley, 2000.

These tasks can be done in an iterative fashion, cycling through them as packages of related use cases for larger projects if necessary.

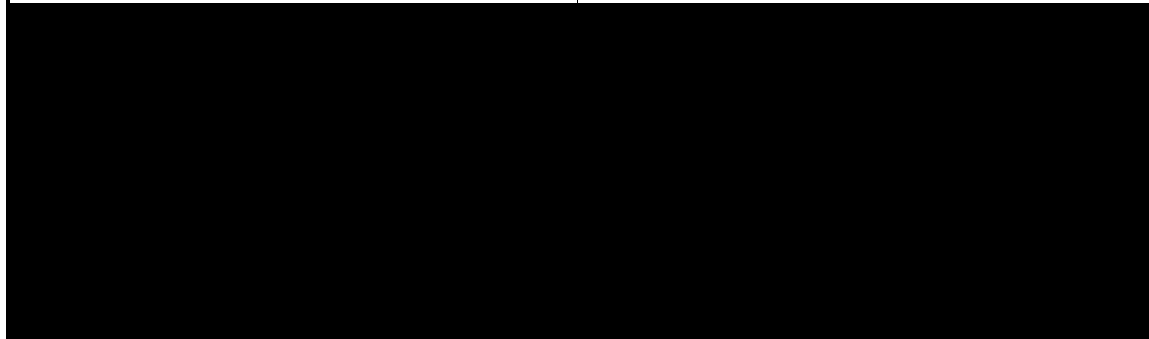
At this point in the project, the team has traveled a mile wide across the entire breadth of the project but only five inches into its depths. You cannot say at this point that all the functional requirements are known. However, you do know enough about the project to identify those requirements that will expose the greatest amount of architectural risk.

Architecturally Significant Requirements

When working with project teams, I focus on coverage areas that are architecturally significant (i.e., that carry a high risk; see Table 1).

Table 1: Architecturally Significant Coverage Areas

Coverage Area	Risk Factors
1. New technology or frameworks not presently employed in other projects within the organization	Identifies areas in which the organization is not yet adept at using a new technology.



<p>2. New or revised business processes that the customer is introducing via new technology</p>	<p>Exposes expectations the customer may have about workflows that have not been tested on a new technology. For example, if a branch bank is switching its account data entry and retrieval processes to a thin client, Web-based application, then that application might not be nearly as flexible as an earlier, client-centric solution when it comes to workflow and user interface possibilities.</p>
<p>3. Time based processing</p>	<p>There are very few robust, off-the-shelf products that facilitate time-based event processing. Many applications require either a customized solution or a combination of pre-purchased components and custom code.</p>
<p>4. Batch processing</p>	<p>Don't believe the myth that batch-oriented tasks have disappeared with newer generation technology choices. This is just plain wrong; batch processing can be a delicate part of the architecture. Sharing business logic between batch and online can be a tricky proposition.</p>
<p>5. Multi-panel interactions requiring state management throughout the process (workflow management)</p>	<p>This is targeted primarily at Web-based solutions, given the stateless nature of the Web. The way in which software vendors manage state when dealing with multi-page interactions ranges in complexity and affects availability.</p>
<p>6. Security, logging, and archiving demands</p>	<p>Given most customers' desire for single sign-on (SSO) and integration of security and archiving capability with pre-purchased solutions, this area alone can consume tremendous amounts of effort to integrate into the overall solution.</p>
<p>7. Persistence management</p>	<p>If the solution will be based on object-oriented techniques, then care must be given to the impedance mismatch when mapping objects to relational stores.</p>

8. Quality of service	Performance is always a consideration. Although actual volume testing may not be feasible in the early stages of Elaboration, simulation tools can be applied to provide meaningful approximation of potential throughput.
------------------------------	--

Keep in mind that the use cases must be assessed for their ability to exercise one or more of the coverage areas noted in Table 1. All too often, a project's first misstep is to identify requirements that are easy to implement but have a relatively small impact on lessening architectural risk.

The use-case pathways, versus the entire use case, should be the focus of the search for architecturally significant requirements. These should be drawn from the entire pool of use cases. In the first iteration of the Elaboration phase, I try to avoid an excessive number of CRUD (Create, Read, Update, Delete) paths. Although these may be good for tweaking the coverage areas, doing too many of them can be a way of avoiding other, more important areas.

The Most Important Iteration You Will Ever Undertake

Without a doubt, the first iteration in the Elaboration phase is the most important iteration the project will ever undertake (see Figure 1 and RUP sidebar). The inputs for this iteration's deliverable, the Architectural Prototype, are the architecturally significant requirements selected at the end of the Inception phase.

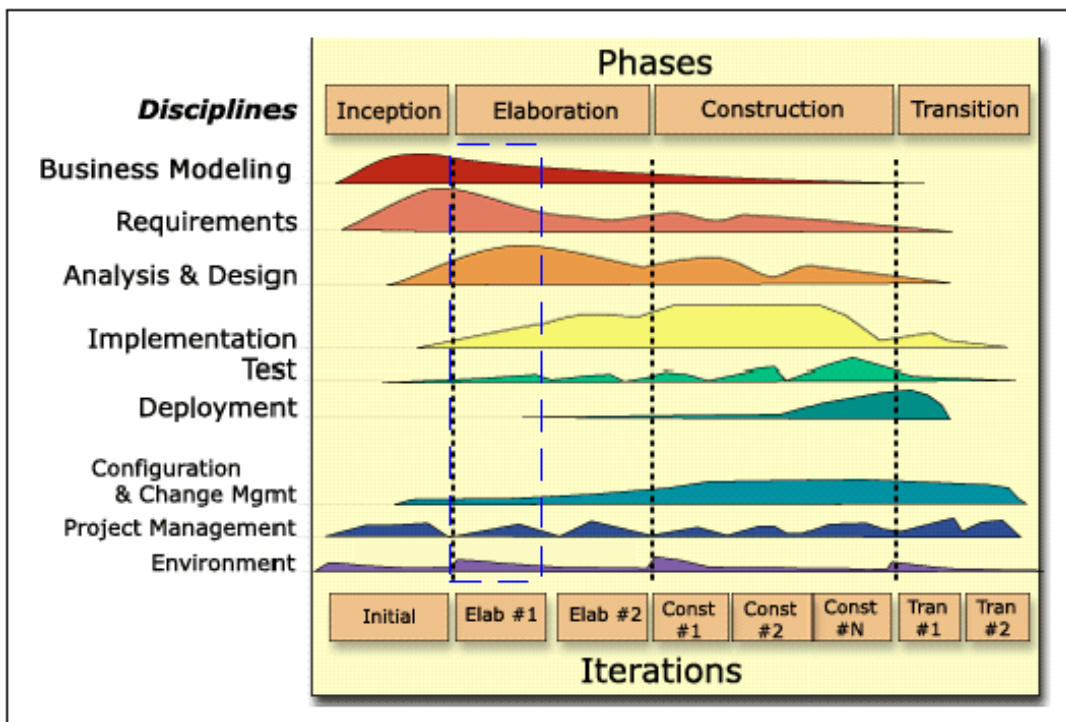


Figure 2: Phases and Iterations of the RUP

It is also during the first iteration in Elaboration that the team explores architecturally significant use-case pathways in depth. All the business rules and dependencies are completely fleshed out. This not only adds more functional knowledge to the project but also further tests the soundness of the architecture. Once the Elaboration phase is complete, there should be absolutely no more hard architecture decisions to make. Remaining use-case pathways should be picked up and detailed in subsequent iterations during Elaboration, Construction, and even Transition.

Use-Case Realizations: The Bridge between Requirements and Design

A key part of smoothing the transition from requirements to design is having an artifact that directly ties the two workflows together. The project team uses the use-case realization as their transitional artifact. This Design activity takes place initially in the first iteration of the Elaboration phase.

A use-case realization is a Design View of a use case. Initially, the use case only identifies "what" the user wants. The use-case realization is the transitional element that specifies "how" the use case will be implemented. However, the use-case realization is actually a composite artifact, containing other design models to represent the actual realization. The most commonly used models contained within the realization are the UML sequence and/or collaboration diagrams.

The project can graphically represent the use-case realization using Rational Rose® (see Figure 3). A stereotyped dependency relationship is

The Rational Unified Process (RUP)

As Figure 2 depicts, RUP consists of four phases and nine disciplines or workflows. An iteration is a vertical project that slices through the nine workflows, drawing from the available activities in each workflow. Each iteration usually incorporates elements from all nine disciplines. The resulting set of tasks comprises what is known as the iteration plan. A given phase may have multiple iterations, and the number is usually a factor of the technical complexity and size of the project. Sample iteration plans for each of the four phases are provided with the RUP product.

The end of each phase is marked by the completion of a milestone. The milestones for the four phases of RUP are: Lifecycle Objective, Lifecycle Architecture, Initial Operational Capability, and Product Release.

Unlike waterfall-based process models, the RUP's iterative model acknowledges that activities from the broad spectrum of workflows (i.e., requirements, design) actually take place concurrently throughout the lifecycle of the project.

Get more information on RUP from these sources:

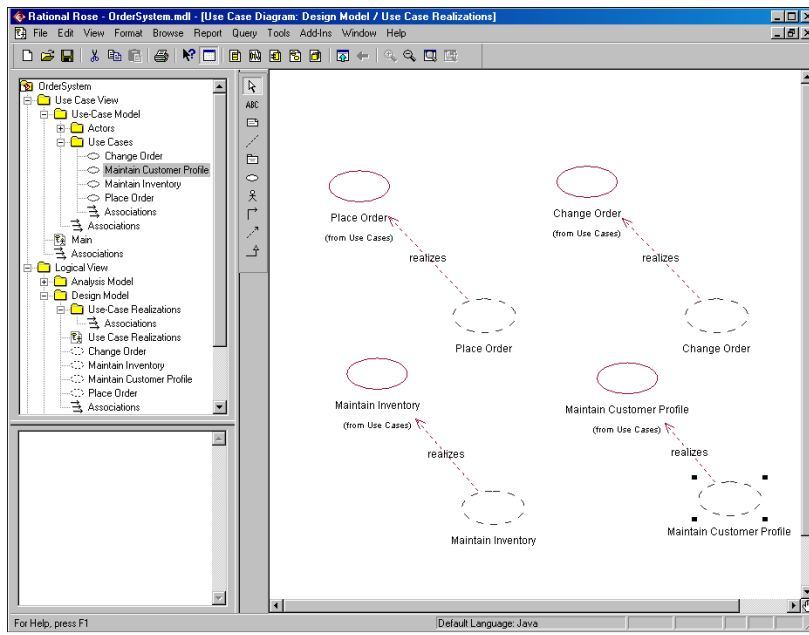
Web:

<http://www.rational.com/products/rup/index.jsp>
<http://www.therationaledge.com/admin/archives.jsp>
(search for RUP)

Books:

Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd Edition. Addison-Wesley, 2000.

created between the use case from the Use-Case View and a use case created in the Logical View stereotyped as a realization.

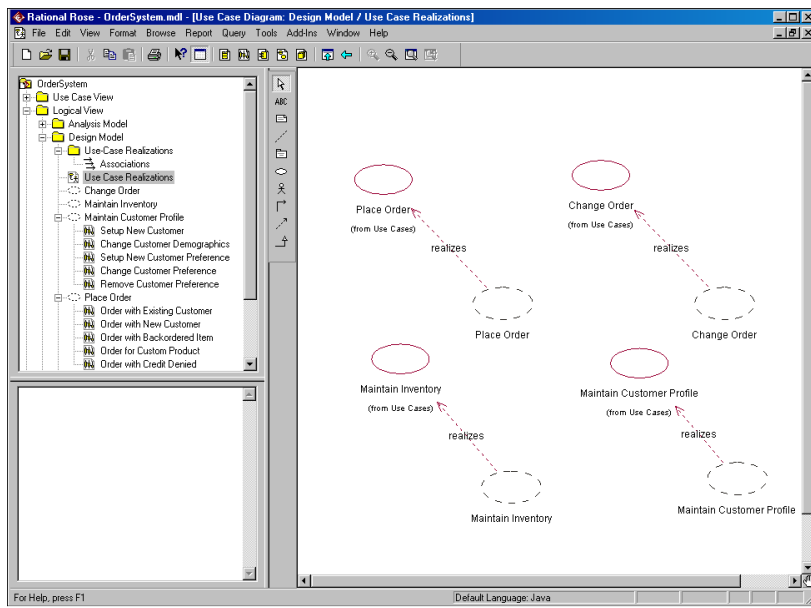


[Click to enlarge](#)

Figure 3: Use-Case Realization in Rational Rose

The tie back to the actual use case isn't just for looks. Remember that a use case in the Use-Case View contains pathways. These pathways are articulated as a sequence of steps with a myriad of business rules that enforce the structure of the pathway, all stated in terms of the user. It is these steps that we now model in Design as a collection of objects messaging with one another to satisfy the goal of the actor(s). This messaging is modeled with one of two interaction diagrams: Sequence or Collaboration. (*Note: I find in practice that projects prefer either the Sequence diagram or the Collaboration diagram and don't usually commingle the two.*)

The Tree View in Figure 4 shows the explosion of Sequence diagrams within each use-case realization. An interaction diagram is created for every key pathway through the use cases. Initially, during the first iteration of Elaboration, this means only those pathways that are selected for their ability to flush out high-risk project areas.

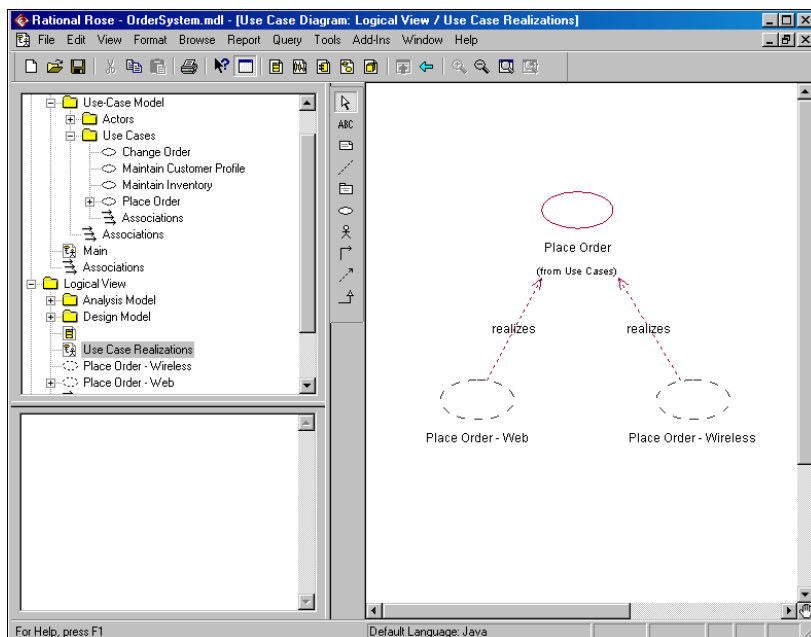


[Click to enlarge](#)

Figure 4: Sequence Diagrams Within the Use-Case Realizations (As Seen in the Tree View)

Use-Case Realizations: When One Isn't Enough

On recent projects I have been involved with, there was a need to have more than one use-case realization per use case within the Use-Case View. This usually indicates that the use case requires a multi-technology solution. For example, for the Place Order use case, you might have an additional non-functional requirement that orders can be placed either online through the Web or via a wireless interface for PDAs and the like. In this case, it would be appropriate to have one use-case realization for the Web implementation and one for the wireless implementation (see Figure 5).



[Click to enlarge](#)

Figure 5: Two Realizations for the Same Use Case

The reason behind having two realizations is that, technically, the solutions are quite different. In the case of the Web solution, assuming we are using Java, there will be Servlet classes plus any number of support classes that won't be used at all for the wireless solution. However, at the same time we can leverage the messaging that is common between the two different technical solutions. Let's face it: When all is said and done, the messaging that goes on between the entity classes (i.e., Order, Customer) to actually get the order in the system and to enforce the business rules that govern that process are identical for the two approaches.

In this case, the interaction diagrams in the Place Order-Web realization and the interaction diagrams in the Place Order-Wireless realization would both point to a common interaction diagram that deals with the common entity class messaging.

It is the use-case realization that provides the context when transitioning from requirements to design. I had a seminar attendee once ask if it wasn't dangerous to have realizations tied directly to how the requirements were structured. My response was that there could be nothing more natural. Just as object-oriented thought brought us the concept of real-world entities that represent both structure and behavior, so do use-case realizations represent the natural, real-world transition to the Design View of the use case.

Birds of a Feather

In past lifecycle approaches, there was an aura of mystery surrounding the transition from requirements gathering activities to design activities. Requirements were usually described in paragraphs within a textual format, and the visual design documents were completely untraceable to those requirements. This process of recording static requirements and then translating them into design artifacts promoted loss of context and often obscured the user's original intentions for the system.

In contrast, an iterative process like RUP allows you to select requirements early on that expose the high risks any software development project entails. The use-case realization provides a real-world Design View of those cursory requirements captured during Inception. Targeting those use-case pathways that expose the project to the greatest risk during the first iteration of Elaboration greatly enhances the team's chances for successful future iterations.

Don't settle for artifacts that don't transition well across project phases. Every artifact should be traceable, both forward and backward through the life of the project. Transitioning from requirements to design shouldn't be a mysterious or a miraculous event. It should be a natural process that is easy to explain and easily understood by all project team members.

References

Philippe Kruchten, *The Rational Unified Process: An Introduction*, 2nd Edition. Addison-Wesley, 2000.

Paul R. Reed, Jr., "Object-Oriented Analysis and Design using the UML." Seminar Material, 1993-2002.

Paul R. Reed, Jr., *Developing Applications with Java and UML*. Addison-Wesley, 2002.

Notes

¹ Ivar Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, 1992.



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