# Iteration-Specific Requirements: More Control Where You Really Need It

by **Mike Taylor**
Software Engineering Specialist
Rational Software

*The Rational Unified Process (RUP®) is based on an iterative approach to software development and includes a number of workflows that are applied iteratively across four development lifecycle phases: inception, elaboration, construction, and transition. The RUP facilitates gradually building application functionality by implementing and testing requirements in a series of iterations across these lifecycle phases. In a RUP environment, it is primarily up to the software architect to consider technical risk along with other economic, teaming, and political factors, and then decide exactly what functionality to develop within each iteration.*

*The RUP is also based on six best practices. One of is to manage requirements effectively, which means using a tool to help maintain integrity and consistency throughout the lifecycle. Rational's requirements management solution is Rational RequisitePro (ReqPro). In this article I introduce a special type of requirement that software architects can use with ReqPro to fine-tune requirements implementation for each iteration within the development lifecycle. By tracking changes to these special requirements, project managers can better prepare for subsequent iterations and assess the effectiveness of initial project plans.*

## Mapping Requirements to Iterations

Two common traps in iterative development are 1) trying to do too much, and 2) not accurately defining the scope of work. It's easy to fall into these traps, particularly in elaboration iterations when the architecture is quickly evolving and constantly presenting new technical trade-offs. Mostly, we overestimate what it is really possible to do within the time constraints of an iteration. Additionally, we work with coarse-grained work statements that don't accurately capture what is -- or is not -- within the iteration's scope.

The standard strategy for mapping requirements to iterations within Rational RequisitePro is to use a text attribute such as "target iteration." Figure 1 shows how this attribute might be implemented in a simple ReqPro project with three RUP construction-phase iterations: C1, C2, and C3.

This mapping strategy is too general for the software architect to fine-tune the scope of the content with any precision. It assumes that each requirement is completely implemented in its target iteration. Although this may often be true for small projects, in larger ones, requirements are frequently implemented in stages across a number of iterations.

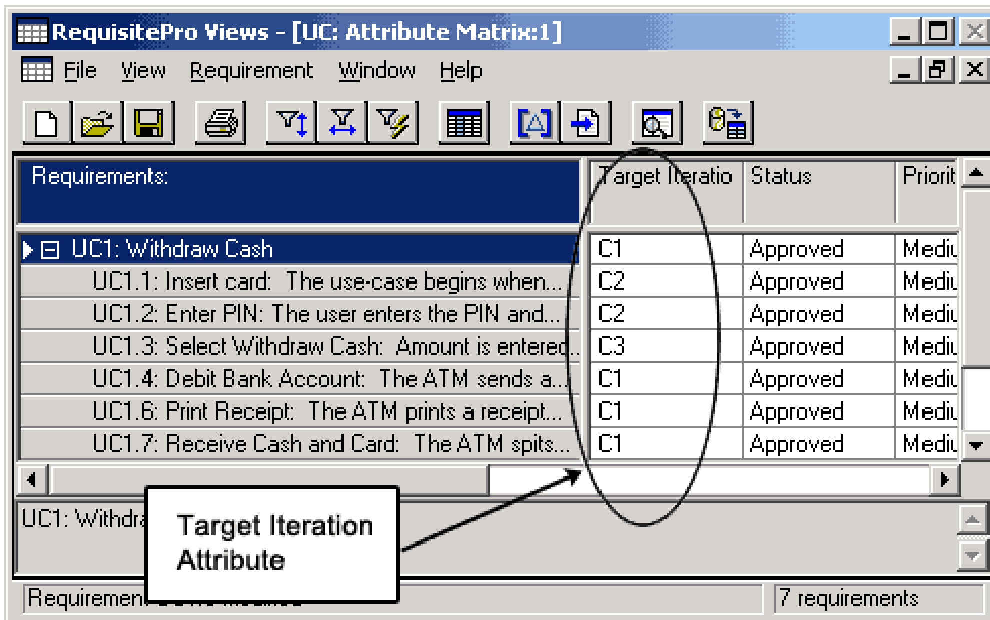



**Figure 1: Rational RequisitePro View Showing Target Iteration Attributes**

## Using BARs as Implementation Constraints

So what can an architect do to better define iteration content? An effective approach is to add more information to the Rational RequisitePro project with an extra requirement type that we'll call a BAR (for Build Additional Requirement). A BAR is an implementation constraint on one or more "real" requirements for a single iteration. It's also a favorite hangout for some software developers, so this may be an easy sell in your organization.

### *Example #1*

As a simple example, suppose requirement UC 1.4 in Figure 1 is implemented in stages across a number of iterations. The full requirements text is:

| **UC 1.4: Debit Bank Account** |
| --- |
| The ATM sends the card ID to the bank consortium for verification. The card ID is verified and sufficient funds are available for withdrawal. The consortium replies that the transaction is accepted and the customer is notified. |

A third-party Database Management System (DBMS) is planned for persistent and recoverable tracking of encryption details for each card identification in

the Automated Teller Machine (ATM), and this technology will be used when implementing UC 1.4. When the Database Administration (DBA) team sees this requirement, it forges ahead and implements a complete schema with all the queries, views, and stored procedures. Iteration performance testing shows that their DBMS doesn't scale, however, so the implementation effort was a waste.

Experience and common sense in iterative development would have kept the DBA team from going down this path. The architect could have told the DBA team not to develop the entire database all at once. Better, a BAR could have been added to Rational RequisitePro to specify the extent of the implementation. For example:

| **BAR1: Database Implementation** |
| --- |
| A maximum of three database tables will be implemented. The number of attributes (columns) for each table will be minimized. A maximum of one example of each implementation technique (stored procedure, query, etc.) will be developed. |

The BAR constrains the implementation and retires the technical implementation risk while keeping focus on the functionality provided.

***Example #2***

As another example, let's say that it's impossible to physically contact the bank consortium in early iterations, but we want an end-to-end scenario implemented. The architect can use another BAR specifying that a loopback be used to simulate the bank consortium's behavior.

| **BAR2: Bank Consortium Loopback** |
| --- |
| A simple loopback generator will randomize responses to bank account lookups for success and failure. |

Figure 2 shows a Rational RequisitePro attribute view of these two sample BARs. It works well for BARs to reside in the "database only" view and be extracted into the *Software Architecture Document* or *Iteration Plan* using Rational SoDA. Figure 3 shows one such example.

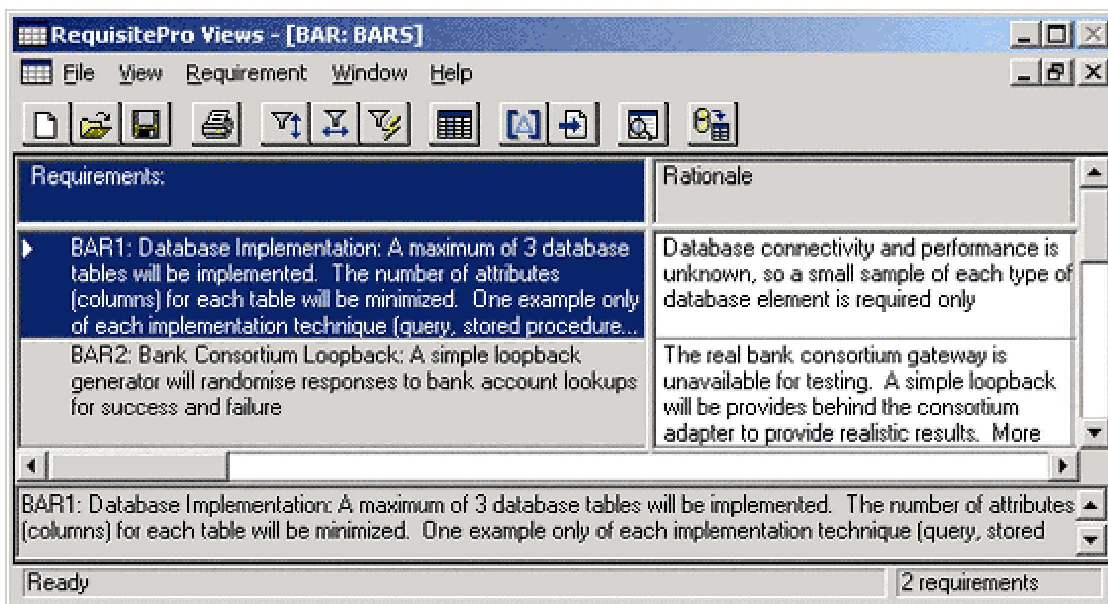



**Figure 2: Rational RequisitePro Attribute View with BARs**

Note that BARs may apply to several requirements; so many-to-many mapping between requirements and BARs is required.

## Applying BARs Across Multiple Iterations

Examples #1 and #2 above illustrate the technique, but how can you apply BARs across a number of iterations?

As a project proceeds, successive iterations will have fewer and fewer implementation constraints (BARs) until, at the end, none are left. Table 1 shows requirements from Figure 1 in a spreadsheet along with some BAR statements. The BAR Map at upper right matches BARs and iterations against requirements; it shows elaboration-phase iterations E1 and E2, and construction-phase iterations C1, C2, and C3. A solid black cell indicates a fully implemented requirement to which BARs no longer apply.

For example, consider UC 1.1: "Insertcard." Reading left to right, this requirement has two BARs (2.1 and 2.2) applied in E2. An additional BAR constraint (3.1) is applied at C1. The requirement is fully implemented in C2 and remains fully implemented for the C3 iteration.

**Table 1: Requirements and BARs/Iterations Map**

| | | | BAR Map | | | | |
|---|---|---|---|---|---|---|---|
| UC Tag | **Requirement text** | **target iteration** | **E1** | **E2** | **C1** | **C2** | **C3** |
| UC1 | Withdraw Cash | C3 | | | | | |
| UC1.1 | insert card: The use-case begins | C2 | | 2.1,2.2 | 3.1 | | |
| UC1.2 | Enter PIN: The user enters the PI | C2 | 1.1 | 2.2 | 3.2 | | |
| UC1.3 | Select Withdraw Cash: Amount r | C3 | 1.2 | 2.3 | 3.2 | 4.1,4.2,4.3 | |
| UC1.4 | Debit Bank Account: The ATM s | C1 | 1.3 | 2.3 | | | |
| UC1.5 | Print receipt: The ATM prints a re | C1 | 1.4 | 2.1 | | | |
| UC1.6 | Receive cash and card: The ATM | C1 | 1.4 | 2.3 | | | |

Time →

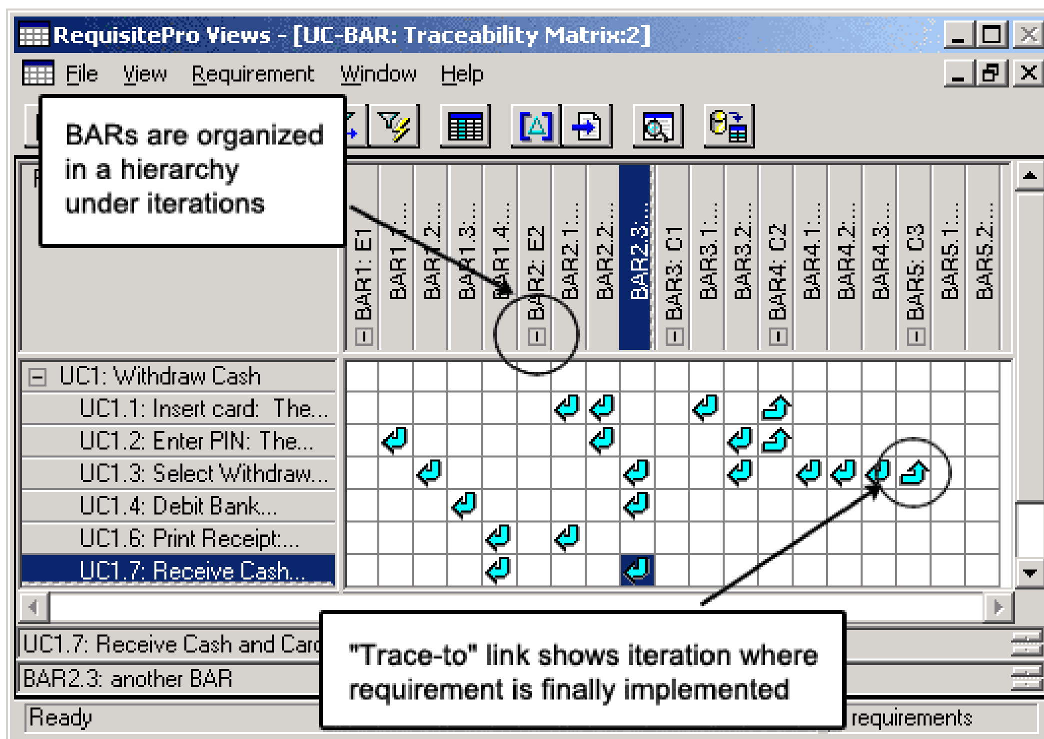
| BAR Tag | **Requirement text** | **Rationale** |
|---|---|---|
| BAR1 | E1 BARs | |
| BAR1.1 | Database Implementation: A max | Database connectivity and perform ance is unkonwn, so a sma |
| BAR1.2 | Bank Consortium Loopback: A si | The real bank consortium gateway is unavailable for testing. A |
| BAR1.3 | Another BAR | |
| BAR1.4 | Another BAR... | |
| BAR2 | E2 BARs | |
| BAR2.1 | Another BAR | |
| BAR2.2 | Another BAR... | |
| BAR2.3 | Another BAR | |
| BAR3 | C1 BARs | |
| BAR3.1 | Another BAR | |
| BAR3.2 | Another BAR | |
| BAR4 | C2 BARs | |
| BAR4.1 | Another BAR | |
| BAR4.2 | Another BAR | |
| BAR4.3 | Another BAR | |

## Mapping with BARs in Rational RequisitePro

Rational RequisitePro includes a traceability matrix view for mapping. As Table 2 shows, BARs are organized hierarchically under the iterations to which they apply. This allows three elements -- BAR, requirement, and iteration -- to be captured in a two-dimensional matrix.

The "Trace-from" ReqPro link is used to trace a BAR to its applicable requirement. Although not strictly necessary, it's also useful to show the iteration in which the requirement is fully implemented. Table 1 indicates this with solid black cells. When you apply a "Trace-to" link in ReqPro to the iteration BAR where the requirement is complete, you'll also see a visual clue: a link icon that points upward, instead of downward like the "Trace-from" icon. For example, in Table 2, read across the line UC1.3. You'll see BAR 2.3 applied in E2, then BARs 3.2 applied in C1 and BARS 4.1, 4.2, and 4.2 in C2. Finally, when you get to C3, you'll see a "Trace-to" link showing that the requirement is fully functional in that iteration. Using different trace directions also works nicely with Rational SoDA documents; you can draw on the power of the relationship direction to show where a requirement is fully implemented. See Figure 3.

**Table 2: Relationships Between BARs and Requirements**



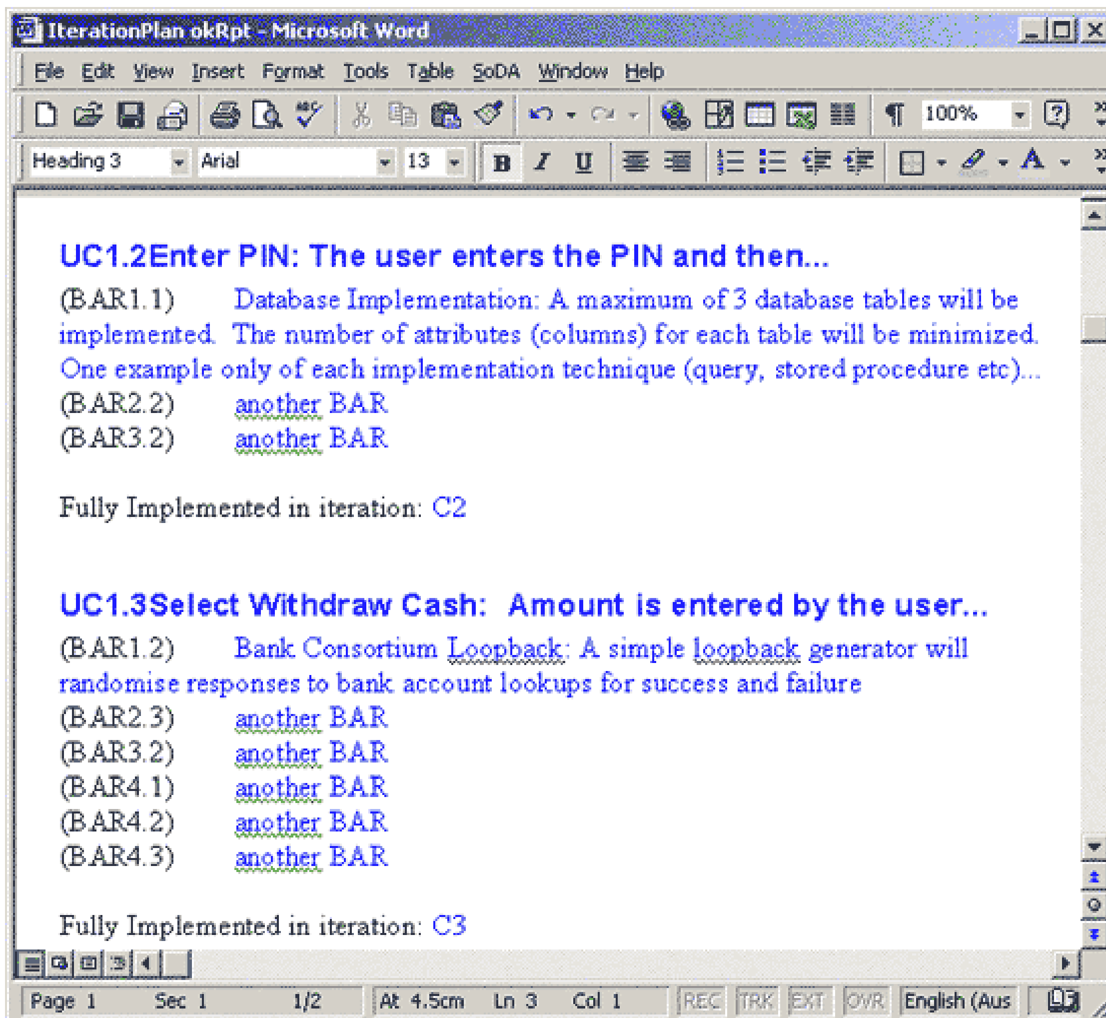




**Figure 3: Rational SoDA Report Showing BARs Applied to Each Requirement (Text in blue is generated directly from the Rational RequisitePro Database using SoDA.)**

**Direct Benefits: Controlling Schedule and Scope Creep**

In the Rational Unified Process, the task of assigning BARs logically fits into the activity *Prioritize Use Cases*, which the software architect performs. These BARs give the architect greater control over the iteration content.

Specifying BARs also helps to fine-tune the application team's scope of work for each iteration. Philippe Kruchten, one of Rational's RUP gurus, has identified a number of "tricks and traps" in the iterative approach, and I can add "failure to close an iteration" to the list. I have often seen developers continue to tinker well beyond the iteration deadline in an effort to get everything right. Closing an iteration requires ruthlessness: If you must leave in known bugs and deficiencies in order to meet your schedule, then grit your teeth and do it.

It's especially important to stick to your schedule if you're just starting out in introducing iterative development to your organization. Others outside the team will be quick to lay blame if there are delays; a shorter development cycle is one of the benefits that iterative development supposedly offers.

Another major benefit that BARs offer project managers is help with controlling scope creep. As a result they can:

- Reduce potential for wasted effort,
- Assist in timely closure of an iteration and thereby help to meet overall project deadlines.

## More Benefits: Measuring Change Rates to Gauge Project Success

Inevitably, as a project progresses, there will be changes to BARs. Technology decisions based on what the architect thought was possible at the start of an iteration often change once development gets underway and new discoveries are made.

During requirements planning, the architect can enter a Rational RequisitePro text attribute "rationale" statement for each BAR, defining the reasons they are put in place. As the project progresses, he or she can then monitor these statements and remove or modify a BAR if the rationale for it is no longer valid.

By monitoring these changes to BARs, managers can learn a great deal about each iteration as well as the project overall. Tracking the number of changes to BARs during an iteration, for example, yields not only a count of technology changes but also a gauge of corresponding scope re-evaluations.

It's also useful for managers to measure the change rate for BARs over time. A declining rate indicates a healthy project: Changes are typically heavier at the outset to mitigate technology risks and adjust the scope. An accelerating change rate, in contrast, indicates that important technology decisions were delayed too long.

Changes to BARs, it's important to note, are different from changes in the functional requirement(s) to which they relate. The requirements themselves typically remain stable, varying only according to changes in user needs. BARs, however, change in accordance with shifts in technology-related decisions or fine-tuning of the iteration's scope. Comparing the change rate for "real" requirements with the rate for BAR requirements indicates the volatility of user requirements vs. technology/architectural volatility.

## Providing More Control Where It's Really Needed

The BAR technique gives the entire project team extra control in situations where it's really needed for the small cost of an extra requirement type and some up-front planning:

- For the software architect, it provides better scope control for iteration content.
- For the testing and development teams, it provides better communication about what will be delivered.
- For project managers, it affords more insight into changing requirements via metrics that differentiate between changes to user requirements (use-cases), and changes that indicate architectural alterations (BARs).

Politically, another advantage of the BAR technique is that it provides a comfort factor for managers who are hesitant about moving to an iterative process. It gives them more visibility into the "hedged bets" that software architects must make concerning technology decisions in early iterations as well as a way to measure their outcomes.

---

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

**Copyright Rational Software 2001 | Privacy/Legal Information**