Capturing Architectural Requirements

Article · November 2001

CITATIONS
34

READS
4,904

1 author:

Peter Eeles

36 PUBLICATIONS **278** CITATIONS

SEE PROFILE

Capturing Architectural Requirements

by **Peter Eeles**
Rational Services Organization, UK

Capturing requirements is difficult. Capturing architecturally significant requirements is particularly *difficult. This article discusses the root causes of this difficulty, and suggests a systematic approach to capturing architectural requirements to ensure that these elusive, and yet extremely important, system specifications are not overlooked.*

What Is an Architectural Requirement?

Because this article focuses on an approach to gathering requirements of particular significance to the architecture of a system[1], let's start with the definition of an architectural requirement.

The Rational Unified Process® (RUP®) gives the following definition for *any* requirement:

> *A requirement describes a condition or capability to which a system must conform; either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document.*

An *architectural* requirement, in turn, is any requirement that is architecturally significant, whether this significance be implicit or explicit. Implicit architectural requirements are those requirements that have particular attributes. For example, any high-risk, high-priority, or low-stability requirement could be considered to be architecturally significant. However, this article will focus primarily on explicit requirements, which are often technical in nature. The following are examples of explicit architectural requirements:

- The product will be localized (support multiple human languages).
- The persistence will be handled by a relational database.
- The database will be Oracle 8i.
- The system will run seven days a week, twenty-four hours per day.
- An online help system is required.
- All presentation logic will be written in Visual Basic.

As you may notice, these requirements are extremely mixed. Some are functional, some non-functional; some are independent of technical mechanisms, others are not. What we need is a systematic approach that provides a framework for classifying architectural requirements, which ensures that valuable statements such as those listed above are not overlooked.

The FURPS+ System for Classifying Requirements

One such classification system was devised by Robert Grady at Hewlett-Packard.[2] It goes by the acronym FURPS+ which represents:

- Functionality
- Usability
- Reliability
- Performance
- Supportability

The "+" in FURPS+ also helps us to remember concerns such as:

- Design requirements
- Implementation requirements
- Interface requirements
- Physical requirements

Let's look at each category in detail.

Functional Requirements

These requirements generally represent the main product features. In a warehouse application, we might have requirements pertaining to order processing or stock control, for example. However, functional requirements are not always domain-specific. Providing printing capability is a functional requirement of particular significance to architecture, for example.

Table 1 lists additional functional requirements that might be considered.

Table 1: Architecturally Significant Functional Requirements

Function	Description
Auditing	Provide audit trails of system execution.
Licensing	Provide services for tracking, acquiring, installing, and monitoring license usage.
Localization	Provide facilities for supporting multiple human languages.
Mail	Provide services that allow applications to send and receive mail.
Online help	Provide online help capability.
Printing	Provide facilities for printing.
Reporting	Provide reporting facilities.
Security	Provide services to protect access to certain resources or information.
System management	Provide services that facilitate management of applications in a distributed environment.
Workflow	Provide support for moving documents and other work items, including review and approval cycles.

Usability, Reliability, Performance, and Supportability Requirements

The remaining "URPS" categories describe non-functional requirements that are generally architecturally significant.

- *Usability* is concerned with characteristics such as aesthetics and consistency in the user interface.
- *Reliability* is concerned with characteristics such as availability (the amount of system "up time"), accuracy of system calculations, and the system's ability to recover from failure.
- *Performance* is concerned with characteristics such as throughput, response time, recovery time, start-up time, and shutdown time.
- *Supportability* is concerned with characteristics such as testability, adaptability, maintainability, compatibility, configurability, installability, scalability, and localizability.

Design, Implementation, Interface, and Physical Requirements

The "+" in the FURPS+ acronym is used to identify additional categories that generally represent constraints.

- A *design requirement*, often called a design constraint, specifies or constrains the options for designing a system. For example, if you specify that a relational database is required, that's a design

constraint.

- An *implementation requirement* specifies or constrains the coding or construction of a system. Examples might include required standards, implementation languages, and resource limits.
- An *interface requirement* specifies an external item with which a system must interact, or constraints on formats or other factors used within such an interaction.
- A *physical requirement* specifies a physical constraint imposed on the hardware used to house the system -- shape, size, or weight, for example.

Realizing Requirements

From the descriptions above, we can easily see that some functional requirements, and most requirements in the other FURPS+ categories, are architecturally significant. Now let's look at how we might classify the seemingly unrelated architectural requirements we listed earlier. Using the FURPS+ classification we can see that:

- "The product will be localized (support multiple human languages)" is a *supportability* requirement.
- "The persistence will be handled by a relational database" is a *design* requirement.
- "The database will be Oracle 8i" is an *implementation* requirement.
- "The system will run seven days a week, twenty-four hours per day" is a *reliability* requirement.
- "An online help system is required" is a *functional* requirement.
- "All presentation logic will be written in Visual Basic" is an *implementation* requirement.

Knowing how such requirements are realized will help us ask the right questions of our stakeholders. There is also value in understanding the relationships between categories of requirement that, at first glance, appear very disparate. Considering architectural mechanisms can assist us on both counts.

Architectural Mechanisms

In simple terms, an architectural mechanism represents a common solution to a frequently encountered problem. Architectural mechanisms are often used to realize architectural requirements. Table 2 shows three categories of architectural mechanisms and shows how architectural mechanisms are expressed in each of these categories.

Table 2: Three Categories of Architectural Mechanisms

Analysis Mechanism	Design Mechanism	Implementation Mechanism
Persistence	RDBMS	Oracle
		Ingres
	OODBMS	ObjectStore
Communication	Object request broker	Orbix
		VisiBroker
	Message queue	MSMQ
		MQSeries

An *analysis* mechanism represents an implementation-independent solution. Table 2 shows two analysis mechanisms: persistence and communication. For more examples of analysis mechanisms, see the Analysis Mechanism Summary in Appendix A.

A *design* mechanism is a refinement of an analysis mechanism. It assumes some details of the implementation environment but is not tied to a specific implementation. In our example, the communication analysis mechanism may be realized as a design mechanism such as an object request broker or a message queue.

Finally, an *implementation* mechanism is a refinement of a design mechanism, and specifies the exact implementation of the mechanism. In our example, an object request broker may be implemented using either Orbix or VisiBroker.

Figure 1 summarizes the relationship between requirements and mechanisms, showing refinements of the FURPS requirements, design requirements and implementation requirements as well as architectural mechanisms at different levels of refinement.

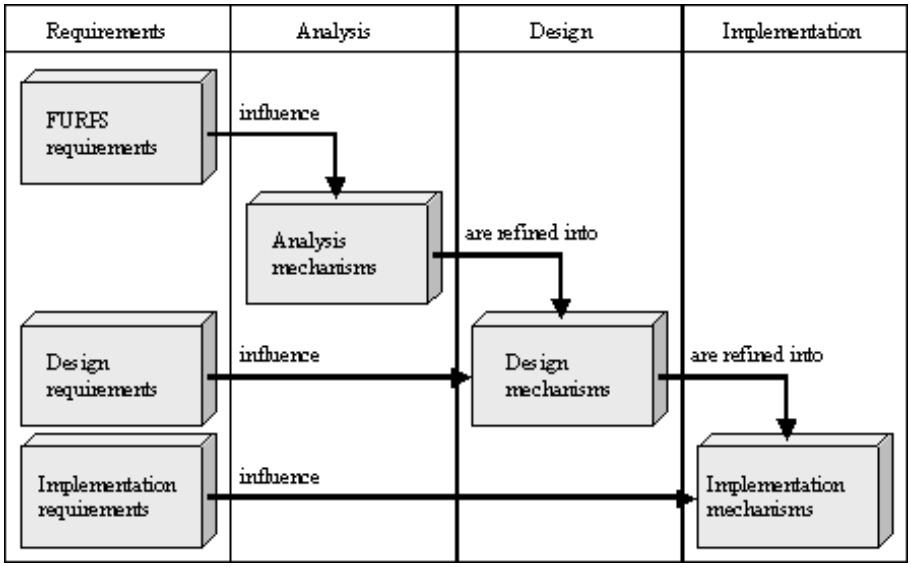

Figure 1: Relationship Between Requirements and Mechanisms

The Challenge of Gathering Architectural Requirements

Before we discuss a systematic approach to gathering architectural requirements, it is worth noting why such an approach is needed. In essence, this is because many of the FURPS+ requirements we mentioned earlier are relevant in a system-wide context and help drive the design of the foundations (i.e., the architecture) of the system we're building. In fact, on some projects, architectural requirements can be significantly more important than their domain-specific counterparts. If you were designing a life-support machine, for example, then availability ("up time") would be pretty high on your list.

So why is it that architectural requirements are often overlooked? Because they are difficult to gather -- and that's why being "systematic" about how these requirements are gathered can make a real difference.

Gathering architectural requirements means venturing into uncharted territory (in contrast to gathering more domain-specific requirements) for a number of reasons:

- In most (although not all) systems, from an end-user perspective, domain-specific requirements are more visible than their architectural counterparts. Consequently, emphasis is placed on gathering these domain-specific requirements because they are perceived as the most valuable.
- Stakeholders are not usually familiar with the majority of architectural requirements. Although they are comfortable with specifying domain-specific features such as "Order Processing" and "Stock Control," they are less familiar with "Reliability" and "Usability," and think of them as technical issues that lie outside their area of concern.
- Systematic techniques for gathering architectural requirements are generally less well known than techniques for gathering domain-specific requirements.

Using a systematic approach can help overcome these difficulties, as we shall see below.

An Approach for Gathering Architectural Requirements

The approach to gathering architectural requirements we will explore is simple:

1. Maintain a complete list of architectural requirements (regardless of whether all items are relevant to a particular project). See Appendix B: Architectural Requirements for a sample list.
2. For each architectural requirement, formulate one or more questions that can help in the specification process. Make sure *all the system's stakeholders* can understand these questions.
3. Assist stakeholders by showing them the potential impact of

answering a question one way or the other.

4. Capture the responses from your stakeholders to each of the questions.
5. Assist the architect by ensuring that the stakeholders -- in addition to answering these questions -- assign a priority or weighting to each architectural requirement. This weighting will allow the architect to make tradeoffs between requirements.

It is worth noting that this approach is possible because, at a high level, the set of architectural requirements that must be considered is finite. You can also apply this approach to requirements gathering within particular problem domains that also have finite, well-defined, sets of considerations. For a financial system, for example, there would be an imperative to pose certain finance-related questions.

The Architectural Requirements Questionnaire

This approach is best represented in the form of a simple table provided to stakeholders as an Architectural Requirements Questionnaire. Table 3 shows a portion of such a questionnaire and includes example answers. For a complete Architectural Requirements Questionnaire template, see Appendix C.

Table 3: Portion of an Architectural Requirements Questionnaire

Requirement	Questions	Impact	Answers	Priority
Licensing	Will the system, or parts of the system, be licensed? Are there any constraints on the mechanism used to provide licensing capability?	The greater the sophistication of the licensing mechanism, the longer the time to market, and the greater the long-term maintenance cost.	The stock control module will be marketed as a separate component of the system and will require licensing. The FlexLM tool is used throughout our organization to provide licensing capability.	Medium
Availability	Are there any requirements regarding system "up time"? This may be specified in terms of Mean Time Between Failures (MTBF).	The higher the availability, the longer the time to market.	Availability is a key product feature. The product must have an MTBF of 60 days.	High

Platform support	What platforms must the system support?	Development for a single platform shortens the time to market. It also allows closer integration with platform features. Development for multiple platforms lengthens the time to market. Close integration with platform features is lessened, increasing the maintenance of the system.	The product will be released on the following UNIX platforms: Sun Solaris IBM AIX HPUX	High

Note that this questionnaire is used in the Elicit Stakeholder Requests activity in the Rational Unified Process (RUP). Once completed, the questionnaire is then used in the definition of some of the most important artifacts used in the development process, including the Use-Case Model and Supplementary Specification (which together provide a formal specification of system requirements).

Rational RequisitePro® can be of great help in relation to the questionnaire:

- It allows you to create multiple "views" of the questionnaire. This is particularly valuable if you are interviewing different stakeholders because you can filter by role. If you are interviewing a marketing person, for example, you might want his or her responses to a certain subset of questions but not to complex technical issues. Rational RequisitePro allows you to assign a "role" attribute to each question, which makes information gathering more efficient.
- It gives you traceability between architectural stakeholder requests and both use-case requirements and supplementary requirements. You can specify this traceability through links, and RequisitePro provides traceabilty matrices to help you visualize these links.

Avoiding Common Pitfalls

When gathering any requirements -- not just architectural requirements -- there are a number of potential pitfalls. This section discusses these pitfalls and provides suggestions for avoiding them.

The "Shopping Cart" Mentality. The technique described in this article has been used on more than twenty projects to date, and every single one of these projects has suffered from stakeholders' false impression that specifying requirements is like filling up a shopping cart. In the absence of any other criteria, a requirements gathering effort can amount to a futile exchange along the following lines:

Analyst:	Does the product need to be localized?
Stakeholder:	That sounds good. We should plan to address foreign markets.
Analyst:	And what about security?
Stakeholder:	Oh yes, the product should be secure.
Analyst:	Tell me about your reliability expectations.
Stakeholder:	Definitely twenty-four by seven -- no down time. That's what we need to beat our competitors.

And so on. If you could offer your stakeholders a solution that solved the worldwide shortage of dilithium crystals[3], they'd want it. But every desired requirement comes at a price; stakeholders can't just put everything they want into a shopping cart at no cost. Be careful not to fall into the trap of presenting them with the equivalent of a shopping list from which they pick and choose.

The trick is to ensure that your stakeholders understand the cost of their purchases, which is why an impact statement is associated with each question in the questionnaire.

The "This Is Too Technical for Me" Attitude. Some stakeholders may dismiss the Architectural Requirements Questionnaire as a technical document whose content lies outside their area of concern. Often, the reason for this perception is that the questions don't discuss their familiar domain concepts, so they treat the document as less important than techniques (such as use-case modeling) for capturing more visible, domain-specific requirements.

Once again, it's important to ensure that your stakeholders understand the value of taking time to answer questions your questionnaire poses. It's often easiest to demonstrate this value by giving examples of problems that arise in the absence of a questionnaire!

The "All Requirements Are Equal" Fallacy. Another common pitfall stems from giving all requirements the same priority. Without exception, projects that fall into this trap classify all requirements as high priority.

Architectural requirements must be prioritized to indicate to the architect -- or anyone else -- which are the most important requirements for the finished system. No design tradeoffs can be made if all requirements are assigned the same priority. If you start to get bogged down when prioritizing requirements, try considering them two at a time. The most important requirements should naturally "bubble up" to the top of the list.

The "Park It in the Lot" Problem. In some organizations, stakeholders dutifully collect requirements because an analyst told them to and prioritize them (again, because an analyst told them to). And then, for some reason, these requirements are then placed "on the shelf," never to be used again. It's critical to ensure that your stakeholders understand the

value of capturing architectural requirements and that these requirements are applied throughout the development of the system.

The "Requirements That Can't Be Measured" Syndrome. This can be a pitfall for both domain-specific and architectural requirements. It's important to ensure that all stated requirements are both unambiguous and measurable. "The system will have a state-of-the-art interface," for example, does not meet these criteria, because the definition of state-of-the-art is highly subjective.

The "Not Enough Time" Complaint. Specifying architectural requirements is a complex task that can't be done quickly. If stakeholders balk at scheduling adequate time for the activity or grow impatient during the process, remind them how important these requirements are in building a foundation for the system, and don't allow them to short-circuit your efforts.

The "Lack of Ownership" Problem. It is critical for the system analyst and software architect to work collaboratively to create the Architectural Requirements Questionnaire, and for the analyst to fully understand the content. After all, the analyst will be gathering the requirements. If he or she conveys the impression that the questionnaire is too technical to comprehend and cannot provide clarification when necessary, that will only reinforce stakeholders' misperceptions that the questionnaire is of little use.

The "All Stakeholders Are Alike" Misconception. Many different stakeholders have input during the requirements gathering process, and you need to address the right questions to the right people. For example, product management is probably the group to ask about specifying the ability to license elements of the system, but requirements for system usability are best specified by end users. Before you talk with stakeholders, take the time to identify which group is responsible for answering which questions.

The "Too General" Tendency. The Architectural Requirements Questionnaire should be treated as an elicitation technique similar to interviewing, brainstorming, and so on. The only difference between this technique and others is that the primary focus is on capturing architectural requirements for the system. Architectural requirements are often specified at a very general level. For example, a requirement might state that the response time of any transaction is less than three seconds. It's likely, however, that although response times for specific transactions must conform to this particular requirement, some transactions will take significantly longer. If you don't make requirements as specific as possible, then elements of your system are likely to be over-engineered, to comply with architectural requirements that may not apply. Once again, the RUP is helpful here. It provides three possible locations for architectural requirements, which reflect three different levels of specificity, as shown in Table 4.

Table 4: Levels of Specificity and RUP Locations for Architectural Requirements

Relates to:	Example	Location in a RUP Artifact
A particular event flow within a use case (most specific)	If, in the basic flow, the plane undercarriage fails to engage, then an alarm will be sent to the central monitoring station in less than one second.	"Flow of events" section in a use-case specification.
An entire use case	Any order that is processed must scale to support more than 10,000 items.	"Special requirements" section in a use-case specification.
The entire system (most general)	Due to the nature of our target markets, the system must be deployed in English, French, Chinese, and Arabic.	Supplementary specification.

Where to Go from Here

As we've seen, capturing architecturally significant requirements is difficult and fraught with pitfalls. But help is available. Use a tried and true classification scheme for architectural requirements and an elicitation technique to ensure that these requirements are not overlooked. Use the appendices in this article, which contain a Sample Architectural Requirements Questionnaire, and summaries of architectural analysis mechanisms, and architectural requirements.

You can also take advantage of the best practices and workflow capabilities built into the Rational Unified Process and Rational RequisitePro. These tools are based on the experiences of many software professionals just like yourself who have been down the same challenging path, so using them will get you off to a great start and sustain you on the journey, too.

Acknowledgments

I'd like to thank a number of my colleagues at Rational for reviewing the content of this article and providing valuable feedback, including Dave Brown, Maria Ericsson, Kelli Houston, Volker Kopetzky, and Wojtek Kozaczynski.

Appendices

Appendix A: Analysis Mechanisms

Appendix B: Architectural Requirements

Notes

[1] Much of the advice in this article can be applied to requirements gathering in general.

[2] As presented in Robert Grady, *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, 1992.

[3] As every Trekkie knows, dilithium can work in conjunction with deuterium to power both warp drives and weaponry. Unfortunately, it is native only to planets heretofore unvisited by Earthlings.

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