IBM

Writing good requirements is a lot like writing good code

Jim Heumann, Requirements management evangelist, IBM, Software Group

Summary: from The Rational Edge: By employing many of the same principles and concepts they already use to write code, developers can effectively serve as requirements engineers. This article reviews those principles and explains how to apply them to create good requirements.

Date: 14 Jul 2004
Level: Introductory

Many software development teams do not have requirements engineers; developers elicit, write, and manage all requirements. This makes sense in terms of resource efficiency: Developers can collect and write requirements during their down time, before serious coding begins. However, the drawback is that programmers are not usually trained in the techniques and tools for writing good requirements. As a result they often struggle, work inefficiently, and sometimes produce substandard requirement specifications.

To write good code, developers must know many things: basic concepts such as controls structures and calling conventions; at least one programming language, including its syntax and structure; fundamentals about the operating system; and how to use technologies such as compilers, debuggers, and IDEs. The good news is that they can leverage all of this knowledge to write good requirements. By employing many of the same principles and concepts they already use to write code, developers can effectively serve as requirements engineers.

Let's look at some of the programming concepts developers can use.

Follow a structure

All programming languages have a structure. Structure refers to how various parts of a program are defined and how they relate to each other. Java programs are structured with classes, COBOL programs have various "divisions," and C programs have a main program with subroutines.

Just as programs have a specific structure, so do requirements. Imagine if you were to put all the code you wrote for a C program into the main program -- it would become unreadable and impossible to maintain. Similarly, if your requirements specification is merely a giant list in no particular order, you will not be able to use it. A group of requirements always has structure, whether you realize it or not. The optimum way to structure in requirements is to organize them by different types, which often correspond to different levels.

To understand the distinctions among different types, let's look at four example requirements for an insurance claims processing application:

1. We must be able to reduce our backlog of claims.
2. The system must be able to automatically check claim forms for eligibility issues.
3. The system shall determine whether a claimant is already a registered user, based on his/her social security number.
4. The system shall support the simultaneous processing of up to 100 claims.

Your intuition may tell you that there is something different about each of these requirements. The first is very high level; it expresses a business need without even mentioning a system. The second expresses what the system should do, but still at a pretty high level; it's still too broad to translate directly into code. The third is a lower-level requirement; it *does* provide enough detail about what the software must do to enable you to write code. And the fourth requirement, though very detailed, does not tell you what the system must do; instead, it specifies how fast the system must be. These requirements are very typical of those you will get as you talk to users and other stakeholders. Perhaps you can see why putting them all into one big, uncategorized list would lead to confusion.

You can make requirements much more usable by putting them into categories, or types, such as:

- Business needs
- Features
- Functional software requirements
- Non-functional software requirements

These are the types suggested in IBM® Rational Unified Process,® or RUP.® They are by no means the only possible types, but they represent one useful approach. Early in your project, you should decide on what types to use. Then, as you collect information from stakeholders, decide which of the requirement types they are describing, and write the requirement.

Note that you can specify functional software requirements in one of two formats: declarative and use case. The third requirement above is stated in declarative form; it is quite granular and uses a "shall" statement. The other form is a use case. It also specifies what the system should do, at a level low enough to write code from. However, it provides more context about how the user and the system will interact to perform something of value. (See below for more detail on use cases.) Before you begin collecting requirements for a project, you should decide which type of functional requirement you want to use and then be consistent.

Use practices that ensure quality

You know that it is possible to write good code and bad code. There are many ways to write bad code. One is to use very abstract function and variable names such as `routineX48`, `PerformDataFunction`, `DoIt`, `HandleStuff`, and `do_args_method`. These names do not provide any information about what the methods or procedures do, forcing the reader to dig into the code to find out. Another bad practice is to use single-

letter variable names such as i, j, and k. You cannot easily search for these with a simple text editor, and the functions are unclear.

Of course, you can write bad requirements in many ways, too. Probably the worst offense is ambiguity. If two people can interpret a requirement in different ways, the requirement is ambiguous. For example, here is a requirement from a real requirements specification:

> The application must be extremely stable with numerous users logged in simultaneously. Speed must not be sacrificed.

The words *extremely* and *numerous* are open to broad interpretation -- so this requirement is ambiguous. In fact, to achieve clarity, you should really express it as three highly specific requirements:

1. Mean time between system failures must be no greater than once per week.
2. The system shall support 1,000 simultaneous users, all doing queries against the database at the same time, without crashing or losing data.
3. The average response time of the system shall be less than one second with up to 1,000 simultaneous users.

Quality requirements have many more attributes; see the IEEE guidelines for more information.[1]

Elaborate with comments

Well-written programs include comments that add information to the code to explain what it is doing or why it was written a certain way. A good comment does not explain *how* the code *does* something -- which should be obvious from the code itself -- instead, it provides knowledge that helps users, maintainers, and reviewers to understand what the code does and ensure quality. Similarly, requirements have *attributes* -- information that makes the requirements more understandable or usable. As you elicit the requirements you should also discover attribute information. For example, one important attribute is *origin*: Where did the requirement come from? Keeping track of your sources will save significant time if you need to go back for more information. Another attribute is *user priority*. If a user gives you fifty requirements, he should also let you know how important each one is relative to the others. Then later in the project lifecycle, when time is getting short and you realize you cannot meet every requirement, at least you will know which ones are most important.

Just as there are no rules that tell you exactly what comments to write in your code, there is no universal list of the "right" attributes. Origin and priority are almost always useful, but you should define others that are suited to your project. As you gather requirements, try to anticipate what information the team might need when you start to design the system and write code.

Know the language

Obviously, developers must know the language they use for coding, whether it is Java, COBOL, C++, C, Visual Basic, Fortran or one of many others. To write good code, you must understand the language's nuances. Although basic programming concepts are the same in each language, they take different approaches to specific operations. For example, the Java loop structure uses "for"; Fortran's is "DO". In C you call a subroutine by referencing its name, with parameters; in Fortran you use a CALL statement.

To write requirements well you must also know the language. Most requirements are written in a natural language (French, English, etc.). Natural languages are very powerful but also very complex; developers not trained in composition sometimes have difficulty communicating complex ideas in writing. We don't have space for a full-blown writing lesson here, but some guidelines can help.

First, use complete sentences for declarative requirements (i.e., those expressed as *shall* statements or with a similar structure); check for a subject and verb in each sentence.

Second, use simple sentences. Statements consisting of only one independent clause that conveys a single idea are easier to understand and easier to verify or test. If your requirement seems too complex for simple sentences, try breaking it down into smaller requirements that you can more easily define. Compound and complex sentences may introduce dependencies (branching); in other words, they may describe variables that depend on certain actions. The result is often an unnecessarily complicated requirement that makes testing difficult.

> **Simple sentence**: The system shall be able to display the elapsed time for the car to make one circuit around the track.
> **Compound sentence**: The system shall be able to display the elapsed time for the car to make one circuit around the track, and the time format shall be hh:mm:ss. (This is two requirements; one is a functional requirement specifying what the system should do, and the other is a user interface requirement specifying the time format.)
> **Complex sentence**: The system shall be able to display the elapsed time for the car to make one circuit around the track within 5 seconds of the lap completion. (This also is two requirements; a functional requirement, and a performance requirement.)

To write adequate tests based on the compound or complex example, you would have to separate the two requirements within each one. Why not make it easy and do that to begin with? Here is one way to translate the complex sentence above into simple sentences:

> The system shall be able to display the elapsed time for the car to make one circuit around the track.
> The format for elapsed time display shall be hh:mm:ss.
> The elapsed time display shall appear within 5 seconds of the end of a lap.

Notice that, in addition to being more testable, these requirements are easier to read.

Here's one more tip for writing good requirements: Use a consistent document format. You already have a format or template for writing code. Use one for writing requirements, too. Consistency is the key; each specification document should use the same headings, fonts, indentions, and so forth. Templates can help. In effect, they act as checklists; developers writing requirements don't have to start from scratch or reinvent the wheel to write a specification that looks good. If you want example templates, RUP has many.

Follow guidelines

Most development teams use coding guidelines such as the following:

- Place module specifications and implementations in separate files (C++).
- Indent your code within the scope of a code block (Java).
- Place the high-activity data elements at the beginning of each group of the WORKING STORAGE

SECTION variables (COBOL).

You should use guidelines for writing requirements, too. For example, if you decide to specify software requirements with use cases, then your guidelines should tell you how to write the flows of events. Use-case flows of events explain how the system and a user (actor) interact to do something significant. Your guidelines should describe what goes in the *main* flow (the success scenario) and what goes in *alternate* flows (exception scenarios), as well as how to structure these flows. They should also suggest lengths for both flows and individual steps within them. If you decide to use traditional, declarative requirements, then the guidelines should explain how to write them. Fortunately, many of these guidelines already exist in RUP and other respected sources, so you need not write them yourself.[2]

Understand the operating environment

To develop good code, you must know the machine on which your system will run and how to use its operating system. If it is Windows, you must know MFC and .Net. If it is Linux, you must know UNIX system calls.

To be good at writing requirements you must understand not the operating system but the operator. You must also understand not the user interface but the user. Java developers think about *classpaths*; requirements writers think about getting people on the right path to a class (or workshop).

Eliciting requirements is a people-centric task. You don't make up requirements; you gather them from other people. This may be challenging for introverted developers, but if they apply their existing skills in the right way, they can be successful.

Often, users do not know what they want; or if they do, they don't know how to describe it. Developers have skills that can help: They often have to decipher arcane and cryptic error messages from the compiler. For example, a Java developer writing an applet might encounter this message:

```
load: com.mindprod.mypackage.MyApplet.class can't be instantiated.
java.lang.InstantiationException: com/mindprod/mypackage/MyApplet
```

What does it mean? If the developer is not sure, she will investigate by looking in her code, the compiler documentation, and maybe even via a search engine such as Google.® Eventually she will figure it out: Her code is missing the default constructor for the applet she is writing.

If you were collecting requirements for a weather forecasting system and a stakeholder told you that the system should be able to "...display wind speed and direction at various heights in the atmosphere over a 200 square mile area, using standard arrows with little tails," you would need to dig deeper. You might ask to see a report from a similar system, consult a book on meteorology, or ask another stakeholder to describe the request more accurately. You would keep investigating until you had enough detail about the desired functionality. Then you would restate the requirements to make them clear and unambiguous, providing enough detail to support a design.

Another tip for eliciting requirements is to try not to ask leading questions. Although you may have ideas about what users should want, if you let those slip out, you may not get a true picture of what *they* really want. Instead, ask open-ended questions such as, "What separate data displays would you like to see?" and avoid questions such as, "Do you want to see a combined air pressure and temperature chart?"

Follow established principles

Among the basic principles for designing and writing good programs are information hiding, coupling, and cohesion. Each one has its counterpart for writing good requirements.

Information hiding

This refers to the principle that the user/caller of a piece of code should not be able to access, or even know about, the data's internal details. All access and modifications to the data should be through function calls. That way, you can change the internal data structures without affecting the calling programs.

This is a good principle for requirements too, especially if you are expressing them in use cases. As we noted above, use cases have flows of events. Poorly written use cases often have flows of events that are packed full of data definitions. Consider this basic flow of events for a use case called **Manage Purchase Request**:

Basic flow of events:

1. The system displays all pending purchase requests.
2. Each pending request will include the following information about the request (limit by `char`):
    - Approval ID (internal only)
    - PO #
    - Reference ID
    - Distributor Account Abbreviated Name
    - Dealer Account Name (first 10)
    - Dealer Account Number
    - Reason Codes
    - Amount Requested
    - Request Date
    - Assigned to (Internal)
    - Comments Indicator
3. The Approval Admin can do one of the following 1) approve 2) reject 3) cancel or 4) assign the request. He/she chooses 1) approve.
4. ...and so forth until all steps are complete.

Of the fifteen lines shown, eleven are dedicated to telling what data goes with a pending request. This is important information, but it obscures what is happening in the use case. A better solution is to hide the data somewhere else. The steps would then look like this:

Basic flow of events:

1. The system displays all *pending purchase requests*.
2. The Approval Admin can do one of the following 1) approve 2) reject 3) cancel or 4) assign the request. He/she chooses 1) approve.
3. ... and so forth until all steps are complete.

Pending purchase requests is italicized to indicate that the data is defined elsewhere (usually in the special requirements section of the use case or perhaps in a glossary). This makes the flow of events, which represents the true functional requirements, much easier to read and understand.

Coupling and cohesion

For those who write code, *coupling* refers to the principle that individual modules in a program should be as independent as possible. Processing in one module should not depend on knowledge of the internal workings in another module. *Cohesion*, refers to the principle that inside a given module, all the code should work to accomplish a single goal. These principles make a program both easier to understand and easier to maintain.

These principles apply to requirements, too -- particularly to use cases. Use cases should stand alone (i.e., have little or no coupling).[3] Each use case should specify a significant chunk of functionality and show how the system provides value to an actor. The actor focus is important; you can specify what the system does for the actor without worrying about ordering the use cases sequentially.

All the functionality in one use case should be about accomplishing one goal of an actor (i.e., have high cohesion). In a system for a typical automated teller machine (ATM), one use case would be "Withdraw Cash" and another "Transfer Funds." Each use case concentrates on a single goal. If you were to combine these functions into a single use case, it would have low cohesion (and undesirable dependencies).

However, be aware that many use-case beginners go overboard and create too many low-level use cases. I once saw a model for a bank's debt collection system that had 150 use cases with names such as "Modify Data." This project had a team of ten and was scheduled to last about a year. However, the organization was having a lot of trouble moving forward because these use cases were too fragmented. They described low-level functions that didn't specify value to the user; they were hard to understand and hard to use. Each use case was extremely cohesive, but consequently the use cases had a high degree of coupling. Raising the level to more specific activities such as "Collect Debt" yielded appropriate degrees of both cohesion and coupling.

Use automated tools

Developers use software tools to do their jobs. It is not possible to compile code by hand, so they always use compilers. Integrated development environments (IDEs) are growing more comprehensive and more popular as organizations strive for efficiency. UML modeling tools for code generation and reverse engineering are also used widely.

Automated tools are also extremely useful for writing and managing requirements. The first step toward automation is to use a word processor. Using Microsoft Word to record requirements represents a significant step up from keeping them on whiteboards or paper napkins, or in someone's head. A word processor provides spell checking, formatting, and document templates that function as "containers" for requirements. You can keep them all in one place and send them around for review. However, Word documents cannot help you sort and filter large requirements lists or establish traceability.

Spreadsheets such as Microsoft Excel facilitate more sophisticated sorting and filtering, but at the expense of losing the context a document provides. Also, you can use spreadsheets for traceability, but the operations are still manual.

A homegrown tool based on a database has some of the same advantages as a spreadsheet; plus, they are often pretty good for doing filtering and traceability. However, because the tool's functionality is often specific to a given project structure, it is hard to adapt. It may also lack complete and up-to-date documentation.

Tools designed specifically for requirements management (RM tools) are usually a little more complex than Word or Excel but not nearly as complicated as a compiler or an IDE. They also offer significant advantages:

- Almost all RM tools allow you to import existing requirements documents into the tool. If both the tool

and your documents are good, the tool will be able to identify automatically the actual requirements in the document. IBM Rational RequisitePro® provides a dynamic link from the requirements in the document to those stored in the tool (or backend database), so that the requirements are always "live."

- RM tools allow you to easily create requirement types and give them attributes. This allows for sorting and filtering, giving the user a flexible query mechanism to find requirements of interest quite easily. These tools also allow you to sort requirements by attribute values. For example, if you had the attributes "user priority" and "risk" for your feature requirement type, you could create the following query: "Show me all the high-priority, high-risk features." This could help you decide which features to implement in early iterations to ensure that you do not leave out important functionality and that you mitigate risk early in the project.
- Good RM tools provide traceability between requirements; a *really* good one provides traceability to other tools and artifacts, such as designs and tests. Traceability is an important capability that helps you validate and verify your system.
- A requirements management best practice is to track each requirement's history. An RM tool can help with this, too. It tells you not only the requirements' origin, but also why decisions were made and who made them.
- RM tools can also help with *baselining*: taking a "snapshot" of your requirements at a particular point in time that you can compare to future snapshots. A baseline provides a stable set of requirements upon which to work. It also provides a branch point in the project lifecycle that you can reference, should you want to copy your requirements for a new development effort.

So, like compilers and IDEs, RM tools help developers do things they could not easily do manually (or perhaps not at all) and helps them achieve greater efficiency.

Manage change

Good development shops manage the changes to their code. Developers write code according to designs and specifications; they do not add features at their own discretion. In addition, the code is under source control; when they change the code, developers specify why they did it. Periodically, they also baseline the code, integrate it, and test it for release.

Requirements need change control, too. Change is inevitable; it is important to plan for it. At the beginning of a project, requirements are usually (and appropriately) in a state of flux. But at some point, before too much code gets written, it is important to draw a line in the sand and create a requirements baseline. After that, requirements changes must be approved, typically by an appointed Change Control Board (CCB). However, some organizations designate just one or two people to review change requests periodically.

Teams that do not have a requirements change control process must field change requests from all quarters and often have difficulty saying "no." If you want to avoid this, along with having to constantly rewrite code to keep pace with the requirements changes, start a CCB or the equivalent. A process for reviewing changes can help ensure that the changes you make will provide business value and that everyone understands their impact. Changes with no business value simply eat up resources with little reward. Similarly, changes that do have business value but would also have great impact on existing requirements, designs, code, and tests may not be worth the effort.

Another way to assess a change's potential value and feasibility is through traceability. It allows you to track (trace) the justification for a requirement and understand all related artifacts. By tracing a software requirements to higher-level business, or user requirements, you can ensure that it has value. If you cannot trace it in this way, the software requirement probably does not have a business justification. In addition, by

tracing from high-level to low-level requirements and on to design, code, and test, you can easily see the impact of a requirements change. A traceability matrix -- or better yet an RM tool -- will clearly show all relevant artifacts and provide the knowledge necessary to decide whether a change request is worthwhile.

Planning

Most successful software development projects have a plan that guides the project, specifying who does what, how things will be done and what the milestones will be. The architect typically creates a document that provides a comprehensive overview of the system's architecture. It also enables communication between the architect and other project team members regarding architecturally significant decisions and guides developers as they implement the system.

Like these plan documents, a requirements management (RM) plan can provide tremendous benefit to a project. For developers who write requirements, the plan describes necessary requirements artifacts, along with requirement types and their respective attributes. It specifies information the developers must collect and mechanisms for controlling requirements changes.

As we saw earlier, requirement types might include business needs, features, and functional and non-functional software requirements. You may also have user requirements and marketing requirements. A plan encourages you to think about and specify requirement types you will need, which in turn helps to ensure consistency and readability for written requirements.

As we also noted earlier, attributes provide supplemental information that helps you understand and use requirements specifications more effectively.

The RM plan also describes the documents you will use. RUP recommends three categories: a vision document, use-case documents, and a supplementary specification document describing requirements that do not warrant use cases.

The RM plan also describes the change management process so that everyone on the project will understand it.

If you are already working on a project that does not have an RM plan, you can write one yourself. It doesn't have to be long: One or two pages might contain all the information you need to promote the creation of consistent, high-quality requirements.

Good developers can write good requirements

The principles and practices developers typically apply to creating code can serve them well when they have to elicit and record requirements. If you are a developer who thinks that you do not have the background or education to write effective requirements, I hope this article has convinced you otherwise. Simply apply the knowledge and the principles you use every day to this new task, and you will succeed.

Notes

[1] *IEEE Recommended Practice for Software Requirements Specifications*, Software Engineering Standards Committee of the IEEE Computer Society. Approved 25 June 1998

[2] In RUP, look under Artifacts -> Requirements Artifact Set -> Use-Case Model -> Use-Case Modeling Guidelines. Also see *IEEE Recommended Practice for Software Requirements Specifications*, Software Engineering Standards Committee of the IEEE Computer Society. Approved 25 June 1998

[3] Unified Modeling Language, or UML, includes the concepts of *extend* and *includes*. You can learn about these advanced topics in IBM Rational Unified Process. Another good reference is *Use Case Modeling* by Kurt Bittner and Ian Spence (Addison-Wesley 2003).

About the author

Jim Heumann has worked in the software industry since 1982, doing analysis, development, design, training, and project management for both large and small organizations. Since joining Rational in 1998, he has focused on helping customers understand and implement software development processes and tools. Currently, as IBM Rational's requirements management evangelist, he specializes in front-end development issues. He holds an M.S. in management information systems from the University of Arizona.