

# Dissecting business from software requirements

developerWorks.

Level: Introductory

Jochen Krebs, Senior IT Specialist, IBM Rational software

15 Aug 2005

from The Rational Edge: Business requirements, software requirements, business rules, non-functional requirements, constraints, and use cases are commonly used terms in requirements engineering. Although each of these requirements concerns is different, they are often found mixed together in a more complex requirement statement. This article offers techniques for dissecting complex requirement statements so that business and software requirements become more distinct.

*Does this requirements elicitation scenario look familiar to you?*

*He: "I want the system to be fast."*

*You: "Sure."*

*He: "And easy to use."*

*You: "Understandable."*

*He: "All documents should be then located in one central repository, like our archive in the basement."*

*You: "Agreed."*

*He: "We require every employee to sign a card during check-in and check-out of the document. But we also must comply to the latest guidelines XYZ."*

*You: "OK" (taking notes)*



*Although the effort to capture requirements is more important than the manner in which you document them, it is the job of a business analyst or requirements engineer to document, organize, and structure the requirements of the project. During this elicitation period, which is usually performed in an iterative fashion, new requirements appear and existing requirements are converted into the desired style and structure. It is not uncommon that previously captured requirements are found unnecessary and disappear during requirements negotiation with stakeholders while new requirements emerge in the process.*

*In the brief scenario above, how do you know which requirements are related to business needs, and which are more related to software functionality? The question is important, because separating requirements from one another results in granular and individual requirement statements, critical for assigning priorities (schedule), risks, and costs to the requirements themselves. Revisiting this task in later phases of the project is a time-consuming activity, but also a possible source for ambiguity and errors. Especially during testing, unseparated requirements add an extra burden to your project because precise validation and verification of the requirements are more cumbersome.*

*Once separated, dissected requirement types can save your project time and money. Your requirements management plan is the right tool to manifest your rules about how you expect to document and treat requirements in your project.*

*This article provides advice in three areas related to requirements engineering:*

- *How to clarify the vocabulary of requirement types*
- *How to discuss interdependencies among requirement types with stakeholders*
- *Examples and guidelines for documenting granular requirements*

*A business or software system can be documented by writing single requirement statements, and these statements can be greatly expanded by applying the principles of use-case design. Before we discuss use cases, let's consider some of the principles of traditional requirements analysis regarding business requirements, software requirements, and business rules.*

## Business requirements vs. software requirements

Business requirements are statements about the business, whereas software requirements (a.k.a. functional requirements) specify what "the system shall ... (do)." Software requirements are usually documented for existing or future software systems and should be in-sync with business requirements. It sounds obvious: a planned software system should help the business get its job done more effectively.

Let's step back and look at some scenarios, which show some differences between business requirements and software requirements.

- **Automating the business:** Computers can do most things faster, and in many cases, they are more cost-effective and reliable than humans. Not to mention that they like to work around the clock and take jobs in hazardous environments. Therefore, we try to delegate as many routine jobs and duties as possible to a computer and its system. For example, many of us use a time management system to record our work hours. After capturing the time, we step through an automated approval process. Even though the entire process is electronic, we still use the term "timesheet" or "worksheet." Accessing this document through a Website is much faster than finding the actual paper timesheet in the departments of the organization. In this example, the business requirements (documenting and approving timesheets) have been transformed into software requirements and a paper process was replaced by an electronically supported process.
- **Manual business requirements:** Evaluating indemnity risk for a life insurance company might be subjective based on the experience of the underwriter. Not all business requirements are digitized and manual processes might co-exist alongside computer technology. In this example, the computer system would capture the result of the risk assessment instead of performing the actual risk assessment, and the underwriter will keep performing the business requirement (evaluating risk), whereas the system documents his decision.
- **Improving the business:** Not only can computer systems replace activities, information technology provides organizations new ways of conducting businesses. For example, the way cashiers entered the price of products into the terminal and logistics systems changed drastically with the invention of bar codes and scanners. Totaling prices at the end of a sale transaction, which is still a valid business requirement for a Mom-and-Pop store, can now be executed by software requirements associated with scanner and bar-code technology.

In the scenarios above, software requirements are clearly distinguishable from business requirements, both of which are subject to process improvement opportunities and automation.

### A further distinction: Business rules

Business rules depict and enforce the policy of an organization. They are usually flexible because of the nature of organizational change and often consist of very granular steps, like a decision table associated with a business requirement. Subsequently, business rules are often subject to frequent change. Some business rules are implemented in electronic form, some aren't. Some business rules are also needed to support legal and regulatory requirements, such as the ones from FCC, FDA, FAA,<sup>1</sup> and so on.

By the way, I use the term *business requirements* for standard practices within the particular industry the organization operates in; for example, "Insurance applications must be signed by the applicant." I use the term *business rules* to refer to the specific policies of a particular organization; for example, "For new applicants older than 65 years of age, three underwriters are required to approve the application."

These examples make it clear that *business rules* are more dynamic and more focused on one particular organization, whereas *business requirements* are more stable and consistent according to the industry. Therefore, eliciting good business rules requires the knowledge of experienced employees, who are familiar with the practice of their organization and have domain expertise. If the domain (in this case, the hotel industry) is well understood among requirements engineers, the primary focus of their work is software requirements and business rules. Therefore, I use the term business rules in illustrating the relationship between software and business in the following examples.

The following examples should demonstrate why business rules and other requirements should be viewed separately.

Assuming we develop an ATM system, we look at two different software requirements, SR1 and SR2:

*SR1: The system shall withdraw a given amount of money from the account, but only to a maximum of \$1,000 a day.*

*SR2: The system shall withdraw a given amount of money from the account, limited to a maximum of three withdrawals a day.*

In the examples above, we face the challenge that two requirements are combined into one. In SR1, "the system shall withdraw a given amount of money from the account" is a software requirement, but the \$1,000 maximum complicates the statement. In SR2, the same software requirement is again complicated by another rule. What would we do if our business needs to add a new policy? For example:

*SR3: The system shall withdraw a given amount of money from the account, but only if the resulting balance is not negative.*

Testing these requirements as stated could not return a precise "pass" or "fail" because each complex statement would require a decision table. And as policies change, not only will the amount of requirement statements increase significantly, the complexity with each new policy would increase too, making it more challenging to verify the requirements during testing.

By contrast, the following example demonstrates how business rules can be separated from requirement statements.

*SR1: The system shall withdraw a given amount from the account.*

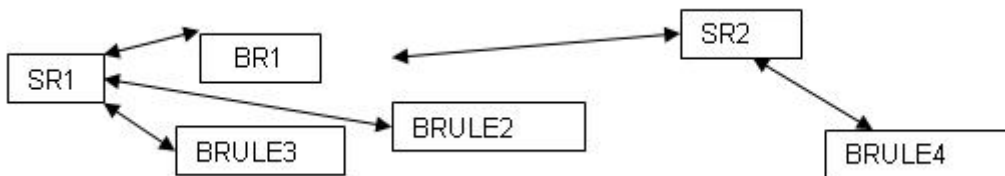
*BRULE1: Limit is \$1,000 a day (24-hour window).*

*BRULE2: No more than three withdrawals a day (24-hour window).*

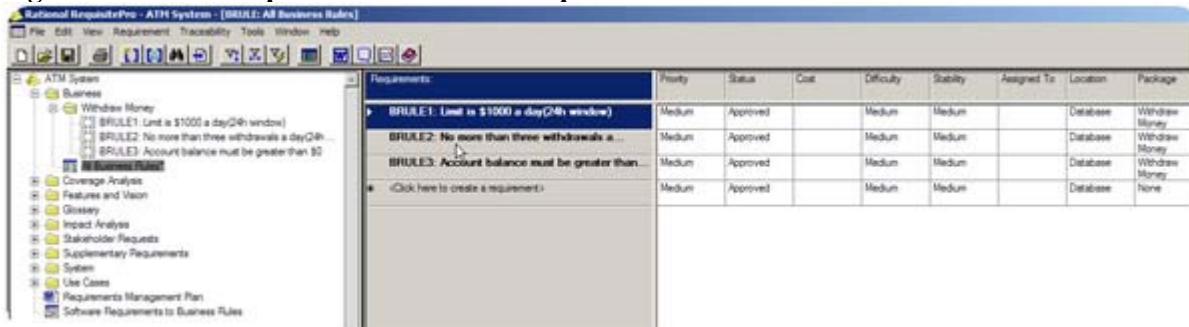
*BRULE3: Account balance must be greater than \$0.*

Business rules like the ones above are easier to manage and test. For example, a test engineer could verify SR1 individually, before the test for the individual policies are executed.

Dissecting business rules from software requirements is a good technique for depicting, analyzing, and validating requirements. However, dissected requirements, which become granular statements, need to be linked to the software requirements to establish the context. Figures 1, 2, and 3 show how that is accomplished.

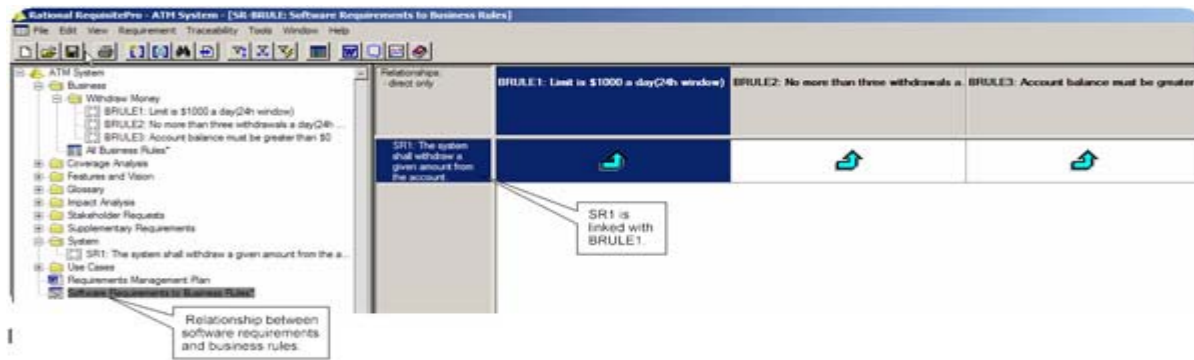


**Figure 1: Relationships between software requirements and business rules**



**Figure 2: IBM Rational RequisitePro allows you to capture business rules during the requirements analysis process.**

[Click to enlarge](#)



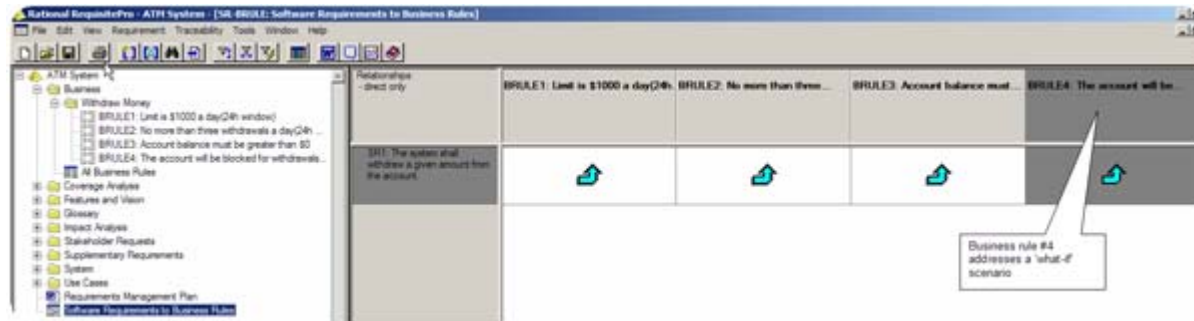
**Figure 3: Tracing software requirements to business rules using IBM Rational RequisitePro**

[Click to enlarge](#)

Communicating business rules is only half the story. The other piece is the "policy," which should be enforced once the original business rule is violated (what-if scenario). For example, how should the system react if somebody attempts for a fourth time to withdraw money within a 24-hour window? How a business reacts if a business rule is violated is basically determined by another set of business rules. If the business wants to get that business rule incorporated into the software, it must be stated. For example:

*BR4: The account will be blocked for withdrawals after more than three withdrawals in a 24-hour window.*

Figure 4 illustrates how this what-if scenario is captured in RequisitePro.



**Figure 4: Tracing requirements to business rules in RequisitePro with what-if scenario**

[Click to enlarge](#)

Because business rules are dynamic and subject to change with regulatory and organizational changes (new products, services, bundles, mergers, spin-offs, etc.), they can be implemented in an organization-wide "rule engine," which offers a great opportunity to software engineers. This rule engine can act as the police of the organization and monitor changes to the existing policy in the organization. For example, as illustrated in Figure 5, the bank decides to increase the limit for withdrawal. Through the association with a software requirement, a suspect is generated which gives a software engineer a clear indication as to which software requirements are affected.

Object-oriented software design, in which business rules are assigned to objects, can enable this idea very nicely. Figure 5 shows how this appears in RequisitePro.



**Figure 5: Tracing changes in business rules to software requirements**

[Click to enlarge](#)

## Non-functional requirements

Beginning with the phrase, "The system shall be ...," and followed by adjectives such as "fast," "easy to use," "intuitive to learn," etc., non-functional requirements (NFRs) are often found mixed in with other requirement types. Traditionally, non-functional requirements apply to the whole system -- such as "The system shall be secured with 128-bit encryption" or "The system should provide a two-second response time," and so on. However, that is not always the case.

Many users experience the human-computer interface as "the system." The graphical user interface is therefore a popular item among non-technical stakeholders' wish-lists. For example:

*It should be easy to see the history of an insurance policy.*

There are two problems within that statement which make this a complex requirement:

- Functionality (history) is mixed with non-functionality (ease of use)
- "Easy" is a subjective concept that's not elaborated.

Often, NFR statements are soft and not clearly defined, which makes them hard to measure and test against. In this particular example, I would question the word "easy" to find out if this requirement should be valid for the entire system or just in combination with that functionality, which is in this example the history of this policy.

Usability requirements in particular represent the style and taste of individuals; not all requirements attract all your stakeholders. Filter carefully which ones apply to the majority of users and your sponsors. Sometimes, look-and-feel standards can replace long debates about colors, fonts, and navigation. What do you think about the following requirement statement? Is it used in controlling a nuclear power plant or a game of solitaire?

*The system shall be reliable.*

It is important to separate functionality and non-functionality of a system, but also to precisely scope the non-functionality of a system as in the example below.

*SR1: The system shall capture a history of all changes made to a policy.*

*NFR1: The history of a policy shall be easy to retrieve. Attributes (in mouse-clicks):*

*Easy - 0*

*OK - 1*

*Not Easy - 2 or more*

## Constraints

When NFR statements contain the word "must," that usually indicates one or more constraints that apply to the entire system; for example, complying with guidelines, laws, or industry-wide standards. Constraints should be treated as separate from other requirement types in the same way as any other form of non-functional requirement. For example:

C1: *The system must comply to governance act "XYZ."*

*The system must be fault tolerant.*

*(Attribute: System-crashes each calendar week)*

C2: *Very good: 0-1*

*Acceptable: 2-5*

*No way: > 5*

---

## Use-case design

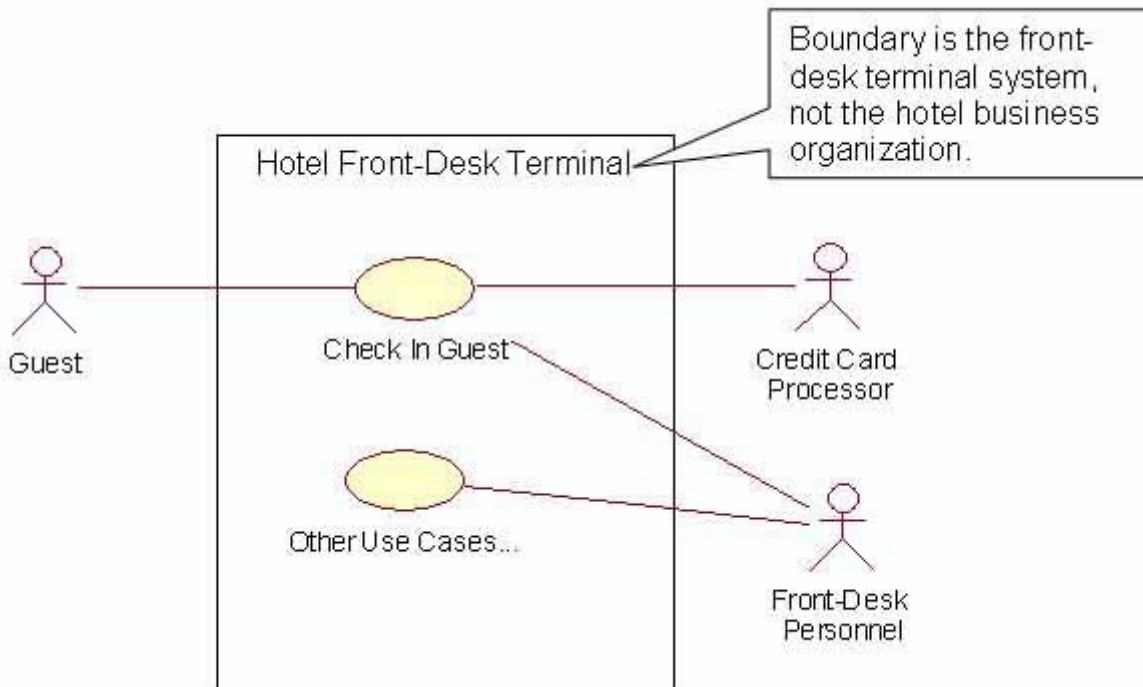
Many books have been written about use cases, discussing style, depth, and detail for writing them. My purpose is not to discuss these subjects again, but to clarify how business use cases and system uses cases can be documented separately.

Let's consider a practical example. Assume we are building a front-desk terminal for a hotel. This hotel primarily operates via manual processes. Keys, calendars, erasers, and pens are utilities at the front desk. Before we start planning the future front-desk application, let's look at the current, existing use case, also called the *as-is scenario*. The procedure for the business use case "Check In Guest" could look like this:

Use case: "Check In Guest"

1. The use case begins when the guest enters the lobby.
2. The front-desk personnel asks the guest about room preferences.
3. The guest tells the front-desk personnel the preferences.
4. The front desk checks the schedule and availability and offers a room to the guest.
5. The guest decides to take the room.
6. The front-desk personnel asks the guest for a driver's license and form of payment.
7. The guest provides the driver's license and credit card.
8. The front-desk personnel captures the guest information in the calendar, and approves the credit card using the credit card terminal.
9. The credit card terminal approves the card.
10. The front-desk personnel gets the key.
11. The front-desk personnel returns the identification, credit card, and room key to the guest.
12. The guest walks away with the information about the room and the key.

After business scenarios (as-is scenarios) are documented for the hotel front desk, the stakeholders decide which steps will be automated using a front-desk terminal system and develop a *to-be scenario* describing the process after the system is deployed. First, a use-case diagram sketch (as shown in Figure 6) is needed for the system to be built that outlines the system boundary and the use cases to be developed against. Second, more granular system use cases describe the interaction between the actors and the system. Once specified, those use cases contain the software requirements.



**Figure 6: Partial use-case diagram created in Rational Rose**

This diagram provides important information about the scope of the planned system. For example, the system does not build a credit card processor (external existing system) but will have an interface to it. Also, we see what kind of things the front-desk personnel can accomplish with the future system (checking in guests, etc.).

Now let's look at a more granular description of this use case, in Figure 7, which more carefully considers system boundaries and helps reveal opportunities to separate needs ever further.

Actor	System
1. This use case begins when the guest wants to check-in a room	
2. The front desk personnel ask the guest about the preferences.	
3. The guest requests a queen-size bed room for one night.	
4. The front desk personnel checks for availability	5. The system offers the available rooms and the room information.
6. The front desk personnel communicates the options to the guest	
7. The guest decides to take the room	
8. The front desk personnel enters the guest information and credit card info.	9. The system communicates the credit card information to the credit card processor.
	10. The credit card processor approves the credit card.
	11. The system blocks the room for the duration of the stay and associates the guest with the room.
12. The front desk personnel picks the key for the room.	
13. The front desk personnel hands the credit card, identification and room key back to the guest.	
14. The use case ends when the guest walks away from the front-desk.	

Activities outside the system boundary

Tasks the system has to fulfill inside the system boundary.

Not decided yet whether the card will be swiped or keyed in, but the system will expect this information.

**Figure 7: Use-case description (to-be scenario): "Check In Guest" using a two-column style**

Even though use cases provide more scope around the actual system interaction, the system column provides a clear picture of the system responsibility indicated in the right column. Let's look closer at Step 11 from the example above: "The system blocks the room for the duration of the stay and associates the guest with the room."

As in our earlier examples, where the requirement statements were divided between software requirements and related business rules, we can separate the steps in the use-case description and mark them as individual requirements toward the system. The new flow could look like this:

11. The system marks the room as occupied for the duration of a stay.
12. The system associates the guest with the room.

One use-case step dissected into two steps like this makes both requirements independent. They could be separately verified and validated, and they will be easier to track for completion. The interaction between actors and the system demonstrated in the original "Check In Guest" description can now be used as part of a kiosk application (self-check-in), thus automating the business.

The check-in procedure described in Figure 7 follows a standard process within the hotel industry. Similar to the discussion about software requirements and their association to business rules, business rules could be linked to a particular use-case step. For example, in Step 5 in Figure 7, we never clarified what the word *available* exactly means. This particular hotel might have a business rule that defines *availability* as meaning *only for the same price range, continuous period of days (no relocation), or any kind of rooms during the requested period*. The chosen business rules indicating *availability* can now be associated with a step in the use-case flow and will give software engineers a precise answer to determine room availability.

### Further discoveries

Discovering the automation and business process improvement opportunities becomes more obvious when we look at purely

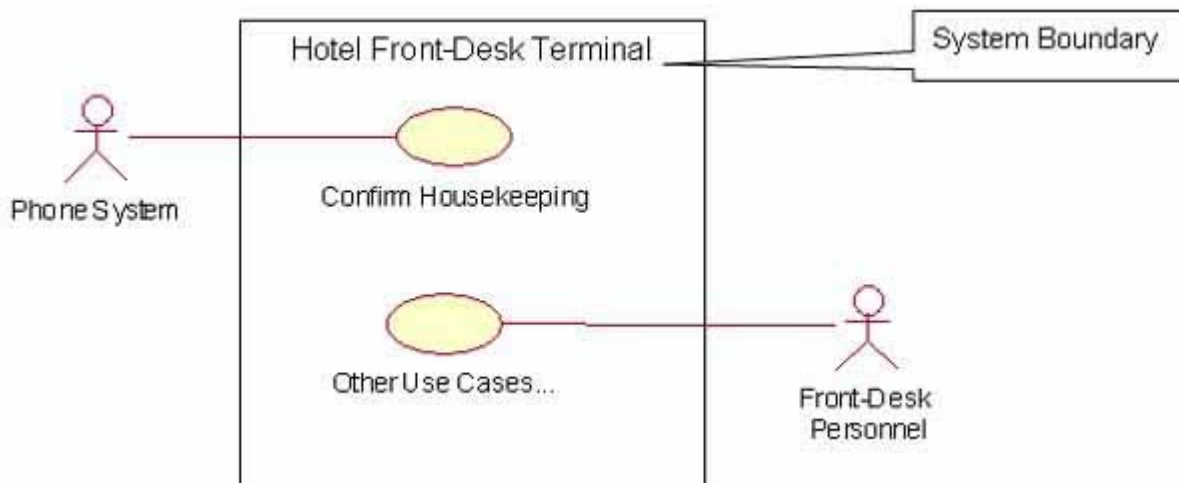
manual processes, such as housekeeping. Prior to new room rentals (see Step 4 in Figure 7 above), clean room availability needs to be communicated back to the front desk. There are various ways for doing this reporting. For example, the cleaning personnel could either 1) carry a sheet, which they submit at the end of their shift to the front desk, or 2) they could call the front desk from the room after cleaning is performed. The sheet has the disadvantage that it is slow -- the front desk would not have access to real-time information. The second solution would place more workload on the front desk, which should be involved with customer facing activities.

Here is a third solution, which delegates the update of the room information away from the front desk personnel to the cleaning personnel, in Step C of a new "Confirm Housekeeping" business use case.

Use case: "Confirm Housekeeping"

1. The cleaning personnel calls the phone system from the room.
2. The cleaning personnel enters a code to signal that the room is clean.
3. The phone system communicates the update to the front-desk information system.

This example demonstrates how the front-desk personnel become liberated from this active messaging activity and how the room information becomes up-to-date. The business requirement that the cleaning personnel are required to inform the front desk when the room becomes available still exists, but a new software requirement has been discovered in the process of writing down and separating the business concerns from the software requirements. With the proposed solution, the hotel phone system will need to interface with the front-desk information system. The following use-case diagram illustrates the difference between boundaries and actors; we realize that the actor "Cleaning Personnel" does not directly interact with the actual front-desk information system, but through the phone system. This external system will have an interface to our Hotel Front Desk Terminal and a new software requirement has been uncovered based on this technique.



**Figure 8: Partial use-case diagram, including "Confirm Housekeeping"**

Use cases are very popular among stakeholders, because they tell a story about the business and system they are part of. The clear separation of business versus system use cases and the distinction of inside versus outside the box allows engineers to take a specific view on relevant processes, which then determine areas for improvement and innovation. By looking carefully at the activities of actors in business use cases and identifying which activities can be replaced by information technology, new software requirements are likely to appear. A positive side-effect is that with the smooth integration into the business world, IT systems are more likely to meet and exceed customer expectations.

## Conclusions

The introduced technique, dissecting requirements, is one way to focus on granular requirements and their interdependencies to each other. This technique allows business analysts a critical check of existing requirements scope, and a way to brainstorm

requirement gaps.

Additionally, I found the technique useful to separate functionality documented in use cases, and non-functionality documented in the supplementary requirements specification (SRS).

---

## Further reading

Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2000

Suzanne Robertson and James Robertson, *Mastering the Requirements Process*, Addison-Wesley, 2000

Donald C. Gause and Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset Publishing House, 1989

Geri Schneider and Jason P. Winters, *Applying Use Cases*, Addison-Wesley, 2001

---

## Notes

<sup>1</sup> United States federal agencies: respectively, the Federal Communications Commission, the Food and Drug Administration, and the Federal Aviation Administration.

---

---

## About the author



Jochen (Joe) Krebs (<http://www.jochenkrebs.com>) is a senior IT specialist for the Rational Software Brand within the IBM Software Group. He is responsible for successful enablement of Rational products and services for clients in the financial sector. Prior to joining IBM Rational, he worked as an instructor and senior consultant with a focus on project management, requirements management, software engineering processes, and object-oriented technologies using Smalltalk and Java. He holds his MSc in computing for commerce and industry at the Open University. He can be reached at [mail@jochenkrebs.com](mailto:mail@jochenkrebs.com)

---

## Rate this content

---

Please take a moment to complete this form to help us better serve you.

Did the information help you to achieve your goal?  Yes  No  Don't know

Please provide us with comments to help