Capturing business requirements using use cases
Seven principles that can make a difference

Thomas Behrens

from The Rational Edge: This article outlines seven practical principles for capturing business requirements, based on experiences with a business requirements engineering project for Simpay, a payment system that operates via mobile phone.

Suppose you have some experience in the Requirements Engineering discipline, and you are suddenly confronted with a significant business requirements project involving multiple companies and spanning different business domains. Questions begin spinning in your mind: Will use cases work on this project? How will I determine the right level of use case granularity? How should I structure the use case model? Will I have to tailor the standard IBM® Rational Unified Process®, or RUP®, deliverables? This article provides an overview of how Alpheus, an international IT consultancy, addressed these questions in organizing a requirements engineering project for Simpay, an interoperable mobile payment system. It distills what we learned on the project into seven practical principles that illustrate how you can succeed with your own business requirements project.

The discussion assumes a good understanding of requirements engineering using use cases and a basic understanding of RUP.

Simpay is a new initiative to develop an interoperable payments scheme that unites the currently segmented mobile payments market. Figure 1 shows an overview of the business context. Simpay occupies a central position in this context, using open interfaces to integrate mobile merchant acquirers (who represent multiple retailers and/or content providers) and mobile operators (who represent and authenticate the end customer) for online financial transactions. Simpay provides
services for payment authorization, and for clearing and settling funds that flow between the mobile operators and mobile merchant acquirers.\(^3\)

Figure 1: Simpay business context overview

The business requirements process was embedded into a larger process to transition the payment solution (i.e., the Simpay product) into the market. The product represented a new business, built from scratch, using both manual and automated processes. Because the budget was extremely well controlled, decisions about implementing certain automated processes were deferred until the business had been modeled.

The overall project and business characteristics can be summarized as:

- Multi-company (Orange, Telefa Mes, T-Mobile, and Vodafone)
- Significant in size (supporting a customer base of 280 million)
- Multi-national with virtual teams
- Having a potential impact on reputation
- Covering multiple specialist domains (e.g., mobile telecommunications, financial services)
- Regulation-intensive, with various legal constraints

These characteristics put a lot of emphasis and visibility on creating a commonly agreed-upon set of business requirements, with buy-in from all stakeholders.

We introduced a process that was aligned with RUP\(^4\) and spanned the Simpay project lifecycle, from business idea to product deployment. We mapped activities and deliverables to the four phases of RUP (Inception, Elaboration, Construction, and Transition) and their respective milestones. RUP activities and deliverables were originally tailored to the software development
process, whereas this project had a much broader scope (e.g., one deliverable" was the creation of a new company: Simpay Ltd.). Therefore, the activities and deliverables sometimes deviated significantly from RUP. Moreover, this process, in contrast to RUP best practices, was not iterative. However, many individual deliverables were produced on an iterative/incremental basis to allow for early review, validation, and quality assurance, as well as occasionally starting dependent activities earlier.

As we wanted to capture requirements on a business level, without specifying whether a certain deliverable would be manual or automated, we used the RUP Business Modeling discipline as our reference discipline, enriched with some elements from the Requirements Engineering discipline. For the structure of our customized deliverables, see the sidebar in this article.

Practical principles

Below we will discuss the practical principles we distilled from our experience. They will help you to:

- Find the right boundaries for your business requirements (Principle 1: Get the scope right)
- Structure your use case model appropriately (Principle 2: Challenge your use case goals and Principle 3: Use requirement attributes to determine the best use case model)
- Elaborate your business requirements further (Principle 4: Divide et impera: Decompose by business worker)
- Describe your business use cases appropriately (Principle 5: Use case descriptions: State what and not how)
- Connect your business use cases, avoid redundancies, and validate your requirements (Principle 6: Produce a domain model and Principle 7: Use entity lifecycles)

Principle 1: Get the scope right

The 101" of requirements management is to get the scope right. Where are the boundaries? Who is in and who is out? The higher the level of abstraction, the more important this is. A small change in scope may have significant impact on the stakeholders involved, the work to be carried out, and ultimately the deadlines against which the project is executed.

Let's look back at Figure 1, this time thinking of it as a marketing view. This view shows the purchase (Shopping Interaction) and payment interactions; it also shows Simpay as the payment mediator. Compare it now to Figure 2. In addition to using a different notation (UML), Figure 2 presents two different contexts. The top one is the context of the merchant owning the shopping functionality represented by the Purchase Product use case. The bottom one is the context of Simpay owning the payment functionality represented by the Request Payment use case.

If you look carefully, you'll see that this figure divides the term Simpay from the marketing view into two parts: Simpay the product, and Simpay Ltd., the company. This distinction was necessary when we found out that the Simpay product encompassed not only requirements for Simpay Ltd. (i.e., the central entity), but also requirements for the mobile operator and the mobile merchant.
acquirer entities. Together, these entities realize the product functionality; therefore, all three are within the project’s scope.

Is this splitting hairs? No! It helped to clearly establish the project boundaries. This meant we could avoid unnecessary discussions from the early days on. In hindsight these distinctions may look obvious, but for a new business, establishing boundaries for the parties involved requires effort. However, this is effort worth investing.

**Figure 2: Merchant context and Simpay context**

**Principle 2: Challenge your use case goals**

Once you have confirmed your scope, you can start identifying use cases and actors. A common problem is that the use case model explodes in size very quickly, especially when you are working on the abstraction level of business requirements. In very short order, you will find you have a lot to say. Literally, stay on top of your use case model to avoid the 700 use case syndrome." If your use case model is exploding, challenge the granularity of your use cases. For Simpay we ended up with about twenty primary use cases on the highest abstraction level. Remember that a use case delivers something of value to an actor; it fulfills a goal for that actor. Ensure that all use case goals are on the same level, and that all goals are at the level of abstraction for business modeling.

Here is a concrete example from the Simpay context. A payment can either be fulfilled immediately -- so that payment authorization and payment capture happen at the same time -- or it can be deferred, so that the authorization happens before the capture. Figure 3 shows a possible use case diagram. However, note that the goals represented by the use cases are not on the same level. The merchant’s goal is to get paid. He must obey the rules of conduct and split the payment transaction into a separate authorization followed by the capture, which may not be to his liking. Also, you are forced to express some relationship between Request Payment Authorisation and
Request Payment Capture. A simpler solution is to combine those two use cases into a single one named Request Deferred Payment. You end up with one fewer use case, and you're a step closer to having an easily comprehensible use case model.

What if there is a long time gap between the authorization and the capture? You don't care! Time gaps are not an indicator that you should break use cases apart; business use cases, especially, can be long running. The essential criterion for deciding whether to break use cases apart is whether their goals are different.

Another danger leading to proliferation of use cases is what I call the water lily syndrome.” It begins when you find a variation for a use case; in the Simpay example it may be the channel through which the payment can happen (e.g., SMS, WAP). Immediately you might be tempted to multiply the use cases in our example (Request Immediate Payment, Request Deferred Payment) by three, to make sure that each case is handled through each channel (Request Immediate Payment using WAP, Request Immediate Payment using SMS, etc.). There may be more dimensions as well. Soon, you will cover your use case diagram with use cases like water lilies cover a pond. Instead, challenge the goal. If you do some analysis, you'll see that the goals are really the same for all of these use cases. Focus on the essential use case, knowing that there are better ways to represent these variations later (e.g., in the special requirements section of the use case description).

Also, when you identify supporting goals, such as maintenance of important business entities, factor them out into separate supporting use case packages with their own diagrams.

![Use case diagram showing different goal levels]

**Figure 3: Different goal levels**

**Principle 3: Use requirement attributes to determine the best use case model**

Requirement attributes contain information about requirements. Priority” indicates how important a requirement is to the business user. Iteration” indicates to which iteration the requirement is
assigned. Stability” indicates how much the requirement is exposed to change. There are more attributes, but let’s focus on those three. Make sure you capture these attributes for your use cases diligently. Why? Because they can make your life easier. In particular, you will often have multiple choices about how to structure your use case model; when you do, align your use case model with the above attributes. This creates the least restructuring work and focuses your effort.

Practically, consider the following guidelines to determine your choices:

1. If any of these guidelines breaks Principle 2 and leads to use case proliferation, then don’t apply the guideline!
2. Avoid merging two use cases if they have different priorities.
3. Avoid merging two use cases if they are assigned to different iterations.
4. Avoid merging two use cases if they have a different stability classification.
5. Don’t spend too much time getting the model right for unstable use cases; this functionality may change anyway. For now, focus on the stable use cases to get the model right.

Remember: How you restructure the use case model will affect the efforts of others. Business users will have to find their way through the new model, and traceability across various deliverables will have to be updated. Constructing the model correctly from the outset will save other people’s time as well.

In the example above, we decided against merging deferred and immediate payment functionality into a single use case (e.g., Request Payment) as it quickly turned out that the initial Simpay deployment could and would include either deferred or immediate payment; whichever payment mechanism was chosen, the other would be scoped for a later release. This variation in use case iteration attributes motivated the split into Request Immediate Payment and Request Deferred Payment.

**Principle 4: Divide et impera - Decompose by business worker**

Now you have a well-structured and understandable business use case model. Where do you go from here? An important thing not to do is to functionally decompose your use case model; that is, do not break your use cases into smaller parts. Such parts would become isolated, defying one of the most important advantages of a use case approach: presenting requirements in the context of an actor's goals. The answer instead is to recursively re-apply a use case-based approach to the innards of your current context.

Let’s look back at Figure 3. The bottom Simpay product context shows the Simpay product realized by Simpay Ltd., the mobile operator, and the mobile merchant acquirer. The RUP term for these entities is *business worker*. These business workers collaborate to create the Simpay product functionality. One, two, or all of the business workers are involved in each Simpay product use case. With this information, you can create a context for each of the business workers (divide) from the Simpay product context. Each context is created by (1) partitioning the use cases over the business workers and (2) deriving new actors from existing business workers. This process is illustrated in Figure 4, which shows an example outcome for the mobile operator. A new mobile
operator context with two partitioned use cases and a new actor (Simpay Ltd.) has been derived from the higher abstraction level. Now you can separately control (impera) the new contexts. 

Is this different from functional decomposition? Yes! Instead of creating isolated parts in the Simpay product context that are not motivated by an actor’s goal, you have created smaller worlds, with new actors representing refined goals. Take a look at the resulting mobile operator context. Simpay Ltd. initiates the Request Payment use case and represents the goal to get paid for the mobile operator. Simpay Ltd. does this obviously on behalf of the mobile merchant acquirer, who, in turn, acts on behalf of the merchant. However, the world of the mobile operator is self-contained; it requires no knowledge about what the Simpay Ltd. actor does. In contrast, a functional decomposition approach would break a use case into many parts: Obtain Merchant Details; Process Foreign Exchange; Authorize Payment; and Capture Payment. Which approach is better suited to managing the project scope?

Figure 4: Business worker partitioning. Click to enlarge.

Principle 5: Use case descriptions: State WHAT and not HOW

We were lucky; we had both interested business representatives who quickly felt comfortable with the use case approach and a capable technology team whose goal was to translate the
requirements to be automated into a technical specification. However, as it turned out, the business representatives were motivated to state as much as possible, and the technology team was nervous about having unnecessary constraints imposed upon them. Use cases represent requirements by describing what the product does to achieve the goal for an actor. The distinction between what and how the product will behave is sometimes unclear to people, and particularly in globally distributed teams, misinterpretations are common. Here are four style guidelines that helped us avoid these mistakes:

1. Certain natural language expressions have a technical connotation (e.g., loop"). If you use these expressions, make sure they cannot be misinterpreted as a technical design suggestion.
2. Business use cases describe business flows; they determine who is the source of the information and who is the target. They do not imply a technical implementation of the flow. The do not explicitly define whether information is to be pushed or pulled, whether the information exchange is batch or online, or whether caching should be used. Such technical decisions relate mainly to non-functional requirements. Be sure people understand that, in business use cases, words such as contacts, requests, or pushes do not imply technical constraints for the underlying technical communication infrastructure.
3. Strive for a simple communication model between the involved actors and the product. Try to follow a synchronous request/response communication model. This implies that a response to a request is either received or not received (e.g., in case of a time out) when the description of the use case continues. It is not possible to receive a response when the use case has moved on in its description. The required expressiveness is sufficient, and it simplifies the description. Emphasize that it does not imply that a technical implementation follows the same model.
4. When you number use case steps, people often think you are ordering how the functionality should be achieved. However, the sequence of use case steps is often mandatory only in
relation to the next interaction with an actor. Follow the practice of explicitly stating when a specific sequence within such a group of steps is mandatory.

It is important to communicate these guidelines. For Simpay, we captured them in the Product Overview document (see sidebar). The best place in RUP to do this is the Use Case Modeling Guidelines document. Even more important, if you are working with a customized process, have a trial. Pick a use case and go a mile deep" with the whole team, so that everybody feels comfortable with how you will produce the deliverables and manage expectations.

**Principle 6: Produce a domain model**

In RUP a domain model (see Figure 5) is defined as the place to capture the most important types of objects in the context of the domain. The domain model is a subset of the Business Analysis model that contains the business use case realizations describing how the business workers (see Principle 4) and business entities achieve the use case’s functionality. The domain model is useful, even if you do not produce business use case realizations. It interconnects your use cases by providing a common vocabulary of terms with precise meanings for your use case descriptions. The domain model ensures that the same term is used to refer to the same business entity. It is also the right place to capture requirements of the following types:

- Relationships between business entities and their multiplicities (e.g., a mobile user uses one or many mobile phones)
- Constraints (e.g., a payment cannot exceed a certain value) and derivation rules (e.g., the computation of interest), especially when these are relevant to more than one use case

Structure your domain model document to create placeholders for this information (see Figure 5). In my experience, business users feel quite comfortable with this slightly increased formality, and it gives you an improved consistency and completeness check:

- Are all business entities and their attributes defined?
- Have you captured all relationships?
- Are all business entities referred to in at least one use case?
- Are all business entities referred to in use cases in the domain model?

In addition to following common sense and the preferences of your audience when creating the domain model, you can use the following guidelines:

1. Use diagrams to support the descriptions, especially to show how business entities are related to each other.
2. Prefer redundancy to generalizations; otherwise, you may demand too much from your business users.
3. Prefer attributes for simple types to relationships; otherwise, you will clutter your diagrams.
4. Omit the types of attributes when they are obvious to avoid creating distraction.
5. Omit identifiers (such as Customer ID) if they are not relevant for the business.
6. Use role names only when they help to clarify the context. Otherwise, imply the business entity name as the role name to avoid cluttering your diagrams.
7. Create an easy navigation experience between related items. The domain model will be used frequently to find information. For Simpay, we used IBM Rational Rose to capture the information for the domain model and IBM Rational SoDa to produce an easily navigable document.

How does a domain model relate to a glossary? A glossary defines important terms used by the project. Based on this definition, a domain model is a subset of the terms in the glossary. But avoid redundancy: Define an item in the glossary or in the domain model, but not in both. Often, terms get transferred from the glossary into the domain model as you continue to learn more about your business domain. However, your glossary is still a useful deliverable for those terms that do not represent business entities.

Figure 5: Domain model diagram and related text. Click to enlarge.

**Principle 7: Use entity lifecycles**

Entity lifecycles are under-used in business specifications, but a good solution is available: UML statechart diagrams. Why should you use these, and how do they relate to use cases? Business entities (e.g., a mobile user) generally have lifecycles that span use cases. A use case transitions (often different) business entities between states. The statechart diagram gives a unified view of where and how your important business entities are manipulated. Figure 6 shows an example for a mobile user business entity. The transitions represent use cases\(^{15}\) that manipulate the mobile user, transitioning it between the different states.

As an analyst, you can also use statechart diagrams to validate your requirement set for consistency and completeness. How does my business entity come into existence? Have I missed an important state? Have I described all behavior in my use cases to support all transitions? Have I missed special conditions that affect the transitions?

Restrict lifecycles to the most important business objects, but describe all states in your documentation. Your use case descriptions should refer to these states as well-defined domain

---

**3.12 Payment Transaction**

The Payment Transaction is \(\ldots\)

The following attributes are defined on this business concept:

- Amount
  - The Amount is the \(\ldots\)

The following business entities are associated to a Payment Transaction:

- Payer (**Mobile User**, 1) performs Payment Transaction
  - The Payer is the \(\ldots\)

The following constraints apply:

1. The Amount cannot exceed 5.00€ if the Mobile User is Pre-Paid.

The lifecycle for a Payment Transaction is defined in **Appendix C**.
model terms, removing ambiguity from your specification while still maintaining readability. A use case description (related to the example in Figure 6) would read like this:

& 4. The mobile operator validates that the mobile user is not barred.

\textit{Barred} in this case refers to a state the mobile user is in, and therefore represents a very precise definition of the condition. Depending on your preferences, you can use formatting to emphasize states in the text.

Lifecycles are best placed into the domain model as appendices that are cross-linked through the business entity definition (see Figure 5). You can use IBM Rational Rose to maintain the statechart diagrams and document the states, and you can use IBM Rational SoDa to create easily navigable documentation.

Following this approach you can leverage the benefits of using entity lifecycles without making your requirements look too techie\textsuperscript{16} when you place them in front of the business user.

A look into the future: With Model Driven Architecture\textsuperscript{17}, you will benefit from specifying precise behavior up front, independent of a specific platform. In addition, executable UML will allow you to validate your models up front, before embarking on costly refinement of your requirements.

![Figure 6: Statechart diagram: mobile user -- lifecycle](image)

\textbf{Summary}

Use cases are an excellent way to capture business requirements. They scale well over large projects, as long as you define appropriate abstraction levels to allow for an understandable and traceable elaboration from top to bottom. Just increasing the number of use cases does not work. Business users who validate your requirements can feel quite comfortable with a use case-based
approach. The technical team that elaborates your to-be-automated business use cases into technical requirements is often even more receptive to this approach.

So, if you are assigned to a complex business requirements project, don't despair. By using a use case approach to requirements engineering, supported by a sound development process and refined by the principles presented in this article, you can overcome many challenges and make the project a success. You might even find, as I did, that it's a satisfying experience.

Acknowledgments

I would like to thank the people at Simpay for their commitment to the process described in this article, in particular Tim Jones, Simon Richards, Martin Rugfelt, and Jim Wadsworth. Special thanks go to my colleague Martin Elliffe for his helpful comments and suggestions.

Notes

1 In this article, I use requirements engineering as a short form for business requirements engineering.

2 The examples used in this article are from the wider Simpay context. Some have been simplified to demonstrate essentials only and observe confidentiality as required. Therefore, readers should not interpret these examples as representations of actual Simpay behavior.

3 For more details of the Simpay business model, see www.simpay.com


5 The main obstacle for a non-iterative delivery was the fact that the IT solution and its operation were outsourced to a third party, integrated into the overall process through a set of legal contracts. An iterative process would have added significant complexity to these legal contracts.

6 Though not part of the standard 4+1 view associated with RUP, this view is essential. In the end, you have to sell your product.

7 The two contexts are obviously related. As part of the Purchase Product use case in the merchant context, the merchant triggers the Request Payment use case in the Simpay context.

8 The merchant confirms that the mobile user can pay and has the money reserved for him.

9 The merchant requests the amount to be paid to him.

10 In fact you may even want to consider going one step further and have a single use case Request Payment." We did decide against it. The reason was scope management, but read on.
11 I do not discuss extend and include relationships here. In my experience, it is best to avoid these in your business use case model.

12 In *The Object Advantage* (Addison-Wesley, 1995, pg.173), E.A. Jacobsen introduces an algorithm used to map a business use case model onto a system use case model. A similar algorithm is used here to map a business use case model onto another business use case model in a different context.

13 In certain circumstances you may have a request without a response. This is entirely consistent with the proposed model and should be used if the response has no business implication.

14 Conversely, I strongly encourage you to use a fine-grained numbering style for use case descriptions. It allows for improved consistency and completeness checks as well as improved cross-referencing.

15 If a use case transitions a business entity between multiple states, you can refer the transition to alternative flows or to groups of steps (e.g., using the tag or number notation of your use cases description).

16 Let me emphasize that they may look techie," but they aren't techie," as they document business decisions only.