Writing Effective Use Cases

Dan Rawsthorne, Net Objectives, drdan@netobjectives.com

(chapter of forthcoming book: "An Integrated Theory of Effective Software Development" due out in late 2004)

In order to succeed, we must capture and validate the user's requirements for the system we are building. We must do this in ways that allow us to capture and summarize the "whole picture" and drill down into parts even while allowing for incremental development and the inevitable changes. The technique we use is use cases with multiple precision levels.

As described in the introduction to the book, to develop software effectively we need to capture and validate the needs and wants of the user. Unfortunately, this important step is often neglected, or it is done poorly, or it is done well initially but then neglected as new information emerges. The result is the same: Garbage.

Some claim that managing requirements is the single most atrociously executed area in software development today[1]. And for many, the entire approach towards requirements gathering and documentation has to be rethought. In our opinion, development teams make three fundamental mistakes:

- They do not communicate clearly what the system should do
- They don't validate that what they're doing is actually what needs to be done
- They forget that things are going to change constantly

[1] For example, the CMM lists "Requirements Management" as its first Key Process Area.

So let's do it differently.

Begin by throwing out the notion of a perfect requirements document, completed in entirety, before the code writing takes place. It isn't going to happen. In fact, one of the most common ways to delay your project and undermine its chances for success is to spending too much time up front on trying to get the requirements documented nailed down perfectly What we suggest is that you work on pieces of the project, one at a time, making sure that you get each piece as accurate as you know how, and increasing your accuracy and precision as you go. It is more important to know that what you are doing today is right, than it is to know what you are going to do tomorrow.

You will still write requirements, but your requirements document will be comprised of use cases written incrementally, as they are needed, so that they can evolve as your understanding of the requirements evolve, and still remain readable with traceability to the resulting code.

What is a Use Case?

A Use Case[2] describes a conversation between an actor and a system to accomplish a goal. Use cases can be black box or white box, the difference being that a black box use case only shows the conversation with the user interface of system. White box use cases show what happens inside the system, as well. Some people say that "black box" is the only appropriate method of documenting a use case, but we think that this is pedantic. Good use cases provide an appropriate amount of visibility so that we can understand our problem as we unfold and validate.

There are many valid formats for use cases that range from formal to casual depending on the personality of your organization. In this book we use a simple, but complete, format for our use cases. This format is very similar to Cockburn's "fully dressed" or formal use case, and is appropriate for incremental development.

As we increase the precision of our use cases (the functionality of our system), we answer the following questions in order:

Precision Level 0: What's the Scope?
Precision Level 1: Who Wants What?
Precision Level 2: How Do They Get It?
Precision Level 3: What Else Could Happen?
Precision Level 4: How Do We Deal With That?

We refer to the information gathered at precision level 0 and 1 as the "header information". The main success scenario is documented at precision level 2. And the extensions are handled in precision levels 3 and 4. All these terms will be defined and descried soon.

[2] Use Case concepts were independently developed by lots of people throughout the 1980s. However, the term was coined by Ivar Jacobson in his book, Object-Oriented Software Engineering: A Use Case Driven Approach, published in 1992. The best book on Use Cases currently in print is Alistair Cockburn's "Writing Effective Use Cases," which was a recipient of the 2002 Jolt Award. Kudos to Jacobsen and Cockburn.

Precision Level 0: " What's the Scope?"

The goals of this precision level are to:

- Capture the scope and goal(s) of the system;
- Determine what use cases are necessary to accomplish these goals;
- Capture a single goal for each use case and label it with an intention revealing name (verb phrase);
- Determine the actor, level, and type for each case; and
- Validate all of the above information with the users or customers of the system.

Of course, all of this information is most successfully gathered and documented in an iterative, incremental, manner. In actual practice you only need to have one important use case to work on in order to proceed with development

System Scope

The first thing we need to know is what the system is supposed to do. We often refer to this as the "System Scope" or "System Vision". There are a number of ways to document this vision, but all of them should answer the following questions:

- What problem is being solved?
- How will we solve it?
- Why is this an appropriate solution?

For our Home Builder System the following are two versions of a vision statement.

In Table Form:

For	Humongous Builder's Supply (HBS)
Who	Requires that all customer sales be completely managed by HBS sales staff
The HBS3*	Is a web-based, customer-facing application
That	Will enable customers to construct sales orders, schedule deliveries, and provide payment information
Unlike	The "off the shelf" packages we have reviewed
Our Product	Will provide for seamless growth and not constrain how we interact with the back-end systems and other 3rd party tools we already have

* Humongous Builder's Supply Sales System

In paragraph form:

Humongous Builder Supply wants to develop the Humongous Builder Supply Sales System (HBS3). This system will be a web-based, customer-facing system that will allow builders to construct their own sales orders, schedule their own deliveries, and update their credit information. This system is necessary because HBS sales staff is spending a large amount of time transcribing builder's notes and conversations into paper sales orders which must then be handled by the lumber yard and the credit department. We feel that by building this system ourselves we can create better interfaces to our existing systems, provide a forward-looking migration path, and more accurately reflect the needs of both our customers and sales staff.

Use Case List

After capturing the goals of the system, the main task is to determine the use cases that will accomplish those goals. Before beginning work on any single use case, check to make sure the goal is within scope of the system you are designing. It might not be. It is hard to tell what goals your interviewees are going to come up with. The things we need to know about the use case at this time are:

- Who or what is the *actor* for this use case?
- What is the actor's *goal* for the use case?
- (optionally) What are the *level* and *type* of the use case?

Actors and Goals

The Actor is a person, role, or thing that calls on the system to deliver one of its services, and the use case documents the conversation the actor has with the system in order to get this service. We say that the Actor has the *goal* of achieving the service, and we name the use case for this goal. We use a verb phrase for the name, such as "Create a Sales Order" or "Manage Customer Invoice" that describes the service being provided.

Levels and Types

For each goal the actor has there are two attributes, the goal's level and its type. These are described below.

Goals/Use Cases Levels:

- <u>System</u> (default) – a single well-defined goal that can be achieved in a single session with the system, like "Withdraw Cash" or "View Inventory"
- <u>Internal</u> – a well-defined goal that is not "complete", like "Login", or "Clean Up"
- <u>Context</u> – a well-defined goal that involves our system, but also uses other systems, as well

If the user addresses a use case that is either at the context level or the internal level, you should have a conversation to try to identify some goals that are at the system level.

Types of Goals/Use Cases:

- <u>System</u> (default) – describes a single, concrete, goal the system accomplishes, like "Create a Sales Order".
- <u>Summary</u> – a well-defined goal that has other goals "hidden" within it. Summary use cases help us collect and organize other use cases into a coherent story. Typical use cases of this type start with "Use…" or "Manage…". For example, the Summary use case "Manage Customer Accounts" has 'internal' use cases like "Create Customer Account", "Modify Customer Account", "Delete Customer Account", and so on. This is particularly useful when discussing data-centric CRUD (Create, Retrieve, Update, Delete) use cases.
- <u>Abuse</u> – goals that we wish our system not to enable. By treating these goals/use cases as "first class citizens" of the use case model it allows us to think about them with more clarity. The Actors for abuse cases are often referred to as "bad actors" in order to further differentiate them. Examples of abuse cases are "Steal Customer Information" or "Get Unearned Discount". Who is most interested in these types of use cases? Security? Who else? These use cases are tougher to think about because they are formed with

negative language and we need to train ourselves to think of these in terms of not succeeding.

- Template – some goals are achieved the same way even though the goals are actually different. Examples are "Generate XXX Report" where the XXX can be replaced with a particular report's name; or "Create Sales Order Item Line" where Sales Order Item can be one of Lumber Item, Plumbing Item, and so on.

Here is the preliminary table of use cases for the HBS3. Remember that we just want to get "most" of them in the sense that we have scoped the system well enough to get going.

In the HBS3 system we found use cases in the following Functional Categories:

- Sales Orders Use Cases
- Delivery Use Cases
- Credit Use Cases
- Others

Note that the style I have used to document the use cases uses "Summary" use cases in each area or category. This is a useful technique, as it allows us to have a "parent" use case that hides the ones we haven't found yet. This means that we can continue to have the conversations we need to have as we go, and flesh out the functional areas as we learn more.

Sales Order Use Cases			
Actor	*Goal (Use Case)*	*Level & Types*	*Use Case Summary*
Customer	Manage Sales Order	System Summary	Includes all the use cases below. Can be thought of as a Category
Customer	Create Sales Order	System	Customer creates a Sales Order, saves it in the System
Customer	Modify Sales Order	System	Customer modifies the Sales Order as needed, saves changes in the System
Customer	Cancel Sales Order	System	Customer cancels the Sales Order, possibly paying a penalty
Sales	Pay Up Front	Internal	Sales enters information that Sales Order has been paid for
Delivery Use Cases			
Actor	*Goal (Use Case)*	*Level & Types*	*Use Case Summary*
Warehouse	Manage Delivery	System Summary	Includes all the Delivery use cases. Can be thought of as a Category
Warehouse	Schedule Delivery	System	Warehouse schedules actual delivery in system
Warehouse	Modify Delivery Date or Location	System	Warehouse alters delivery based on discussions with Customer
Warehouse	Arrange for Drop Shipment	System	Warehouse modifies delivery to support drop shipment by 3rd party
Warehouse	View Pending Deliveries	System	Allows the Warehouse to view the current and future status of deliveries
Warehouse	View Inventory	Internal	Allows the Warehouse to view the current and future status of inventory in order to predict deliveries
Credit Use Cases			

Actor	*Goal (Use Case)*	*Level & Types*	*Use Case Summary*
Credit Dept	Manage Customer Account	System Summary	Perform all the management tasks for Customer accounts, like Create, Delete, Update, View, etc.
Credit Dept	Arrange for Credit	System	Allows a Customer to pay for a Sales Order on Credit; part of Manage Customer Account
Credit Dept	Process Credit Payment	System	Process a payment on a Customer's credit account; part of Manage Customer Account
Credit Dept	View Customer Credit History	System	Allows the Credit Department to publish a Customer's credit history; probably a part of Manage Customer Account
Other Use Cases			
Actor	*Goal (Use Case)*	*Level & Types*	*Use Case Summary*
Customer	Build a House	Context	Includes Customer buying building supplies from HBS – includes buying, delivery, return.
User	Login to System	Internal	Provides appropriate permissions to use functions of the system. Note that the User includes "everybody"
Hacker	Steal Customer Information	System Summary, Abuse	Hacker uses customer-facing feature to steal privacy act info, credit card numbers, etc.

This table contains essentially the same information than the ubiquitous Use Case Diagram many of us are used to developing. If you must have a use case diagram, it would look something like the following one.

Sidebar

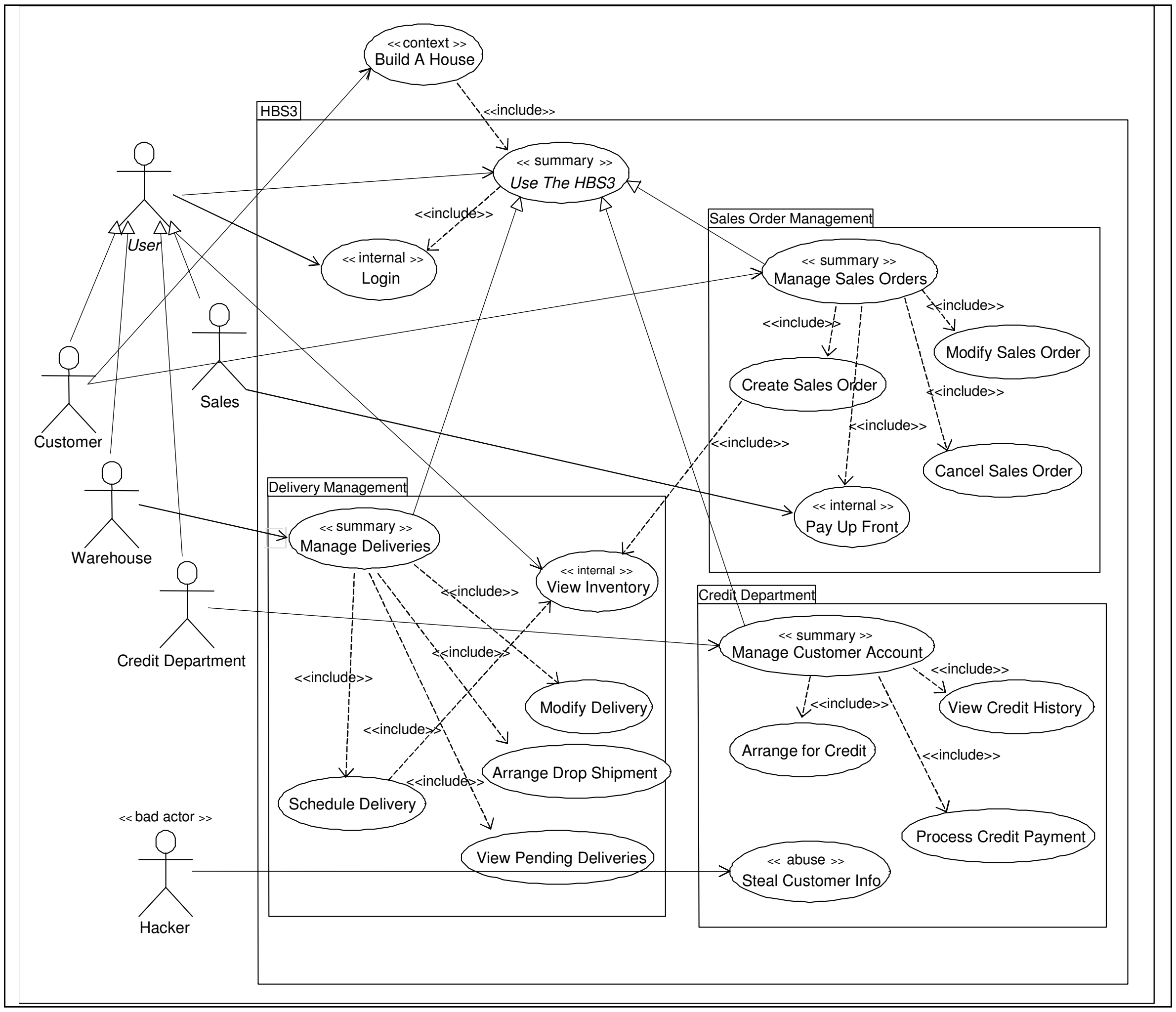
Note that the arguments we can have about this picture are almost infinite. But the arguments about the table aren't. This is one of the reasons why I don't really like these pictures – they cause more problems than they're worth.

I have drawn this picture this way not because I like it, but because it allows me use the most types of arrowheads and make it look as cool as possible. In that regard, I suspect I'm not much different than you are. It can be argued that this diagram actually has more information than the table does, and that's true. For example, this diagram clearly shows that we consider the Customer to be a kind of User while we don't regard to Hacker as a kind of user.

Diagrams like these are pretty cool, from a purely technical perspective. However, I try to avoid them for a number of reasons:

- they are hard to draw yet they hold no more *real* information than the table does
- we get so carried away with making them look pretty that we forget to validate information that is in them – so they're often wrong
- drawing these is a form of passive aggressive behavior. Once our users see a diagram like this they are afraid to change it - and the purpose of this exercise is to get validation, not acquiescence

Of course, you can draw diagrams like this just to show off your UML skills. I expect that some of you are looking at this diagram and critiquing it to see if has been done right. I say "go right ahead." But just remember that whether or not this picture looks pretty has nothing to do with how well we will write our code - the validation is just as good if it comes from the table.



Precision Level 0 Validation

Once you have a collection of use cases, named by their goals expressed as intention revealing names, with identified actors, level type, and so on, you should check it out. This is the most important thing to do before moving forward.

Knowing who it is that will validate your work is a key to your success. There are several options for who will validate your work: A primary customer, an alpha user, your boss, the product manager, or the owner of your business. Determine who will make the decision. Perhaps more than one person will share the responsibility.

Tell the validator(s) which use cases you are planning to work on. Tell them that the next release is going to deliver code that addresses these use cases. Remind them that these are the most important goals you have identified. Then ask them if they agree. Then come back to them later and see if they are still in agreement.

Do not expect them to stay in agreement. It isn't going to happen.

The thing to remember here is that your validators are not always going to play by the rules. They will change their minds. They will learn new things. There will be misunderstandings. There will be mistakes. Prepare yourself ahead of time for this sort of interaction, and check back with them as often as possible.

The sooner you find out that the user has switched direction, the sooner you'll be able to get resynchronized. The effort you can divert from working on something unimportant to something your validator now feels he really needs is money in the bank.

Precision Level 1: "Who Wants What?"

Assume our project chooses to work on "Create Sales Order" as our first use Case. So far we have already validated the following based on the PL0 analysis:

Name: Create Sales Order
Summary: Customer creates and saves a Sales Order.
System: HBS3
Actor: Customer

We now need the rest of the "Header" material. So far we know what the main goal of the use case is – it's the name of the use case. However, this is other stuff we need to know before we start designing the conversation that will achieve the goals:

- Who are the stakeholders, and what do they want and need?
- What are the preconditions for this use case?
- What are the postconditions (what we actually achieve) for this use case?

Stakeholders and Interests

Stakeholders are entities (someone or something) that have an interest in the behavior of a use case. Everything that interacts with the system during the execution of this use case is a stakeholder, but many of the more interesting stakeholders don't interact with the system at all. Examples of stakeholders are the owner of the system, the company's board of directors, and regulatory bodies such as the Internal Revenue Service, Security, IT Infrastructure (the group who supports your system) and the Department of Insurance.

A use case can be thought of as a contract between the system, the actor, and the stakeholders. Therefore, we need to know what the stakeholders want from the system – what their interests are. Listing the interests of each of the stakeholders both helps us catch all the requirements, and helps involve people (the stakeholders or those that represent them) in the requirements process. We do not need this information to be exhaustive. Since a requirement is merely a "promise for a conversation"[3], they need only remind us of the appropriate conversation to have when we're actually building our system, whether it is a conversation when designing, testing, assembly.

For our use case, "Create Sales Order", we get the following Stakeholders and Interests.

Name: Create Sales Order
…

[3] Paraphrasing Ron Jeffries: " a Story is a promise for a conversation"

Stakeholders and Interests:
Customer: Wants an accurate sales order; wants proposed delivery dates to be accurate; wants costing to be accurate.
HBS Owners: Wants accurate sales order, pricing, warehouse and inventory information.
Warehouse: Wants items listed as "in stock" to actually be there; wants accurate SKUs for items
Credit Department: Want sales order to be available in order to arrange payment.

Separate Document: In some processes (such as RUP) the stakeholders and their interests are listed in a separate document. This is laudable, but we need to know which of their interests pertain to *this* use case, not just what they want in general. Therefore, we like to have their interests explicitly stated for each use case. Of course, we may document global stakeholder needs and interests in a central place, where it is assumed to part of all use cases and is readily accessible to all developers.

Preconditions and Trigger

Not only do we need to know what the stakeholders want, we also need to know what state we're in when we get into this use case. There are two things we need to know:

- Preconditions - things that the system will guarantee to be true before the use case starts. They don't need to be checked as part of the use case, because they are already true. The existence of a precondition implies that there is another use case that exists to put the universe into that state, and that use case has already been done.
- Trigger – the action that caused us to enter this use case. Often, this action is not unique, but it helps our users if they see something here.

For our Create Sales Order use case we have the following Precondition ad Trigger.

Name: Create Sales Order
…
Precondition: Customer Logged in to HBS3
Trigger: Customer selects "New Sales Order" from main menu

Wish-Away Preconditions: Many people like to list preconditions that need to be checked by the use case or the system and can't possibly be guarantee, like:

- The Customer has a sufficient Credit Line (how do we know until we know how much he wants to buy?)
- The Sales Items are in stock in sufficient quantity (how do we know until we know what the customer wants?)

These are not legitimate preconditions. We'd like to be able to assume all the conditions that make our use case work as preconditions, but then we "wished away" the complexity in our use case. This leaves us with a problem that is neither properly understood or analyzed. Use cases can fail. That's one of the things they do. The failures are captured when you ask the question "What Else Could Happen" in precision level 3. These failure conditions are addressed as extensions.

Post conditions

After defining the stakeholders, their interests, and any existing preconditions, the next thing to do is figure out what the use case is *actually* going to do. In other words, it is now time to balance the various needs of the stakeholders and "make the call".

There are two kinds of postconditions:

- Success – what will be true if the use case succeeds. . We would like for all stakeholders to be satisfied with the results, but that often is just not possible – their interests conflict, they're too expensive, whatever… The way to find the success Post condition is to decide which of the stakeholders' interests you will provide for, and then provide for them.
- Minimum – what will be true if the use case fails. The way to find the minimum post conditions is to imagine the worst possible thing that can happen in the middle of the use case, and then describe what the system can actually do in that situation. Most commonly, at a minimum your system will log how far it gets. This is important enough to not be overlooked, yet very often it is overlooked anyway.
- Other - defined by other states, like "Network Down", "Database Down", and so on, that are dependent on their own set of business rules. These cases are often found when doing further analysis, design, testing, or investigation in the field. We then go back and document them as the discussion may be useful for other use cases not yet analyzed…

For our use case, this is what we come up with.

Name: Create Sales Order
…
Postcondition:
Success: Sales Order completed and saved. Sales Item information is accurate, including quantities in stock and proposed delivery dates. Pricing is accurate and complete.
Minimal: Transaction logged to completion or point of failure

Precision Level 1 Validation

Following the listing and writing in use case development at precision level one, once again go to your validator, and in this case your stakeholders, and check the work you've done.

Tell them once again the use case you're working on. Repeat the idea that you are working on this because it is most important. Review with them the list of stakeholders and the minimum success post conditions. Go over the preconditions.

What the use case looks like now.

Name: Create Sales Order
Summary: Customer creates and saves a Sales Order.
System: HBS3
Actor: Customer
Stakeholders and Interests:
Customer: Wants an accurate sales order; wants proposed delivery dates to be accurate; wants costing to be accurate.
HBS Owners: Wants accurate sales order, pricing, warehouse and inventory information.

Warehouse: Wants items listed as "in stock" to actually be there; wants accurate SKUs for items
Credit Department: Want sales order to be available in order to arrange payment.
Precondition: Customer Logged in to HBS3
Trigger: Customer selects "New Sales Order" from main menu
Postcondition:
Success: Sales Order completed and saved. Sales Item information is accurate, including quantities in stock and proposed delivery dates. Pricing is accurate and complete.
Minimal: Transaction logged to completion or point of failure

You can only hope that the use case you have is the right use case to be working on, and that it is done right. It might be good to surround the validation with some kind of ceremony. At a minimum, keep records of what you validate, and the decisions you make. After all, validation of the use case at this precision level determines what your team will be working on. From that standpoint it is very important.

Domain Validation

At some point you need to step back and review your goals and decide whether they are achievable within your domain. The Analyst creating the use cases from the goals must understand the domain before continuing further, in order to avoid writing stupid use cases. This is a "Duh" statement, but you'd be surprised how often the analyst goes into further use case analysis without proper domain knowledge.

Domain knowledge falls into two categories:

- Problem Domain Knowledge – understanding how the user talks about his problem. This knowledge can be increased through a glossary. Future validation will be with users, so you must speak his language in order to make use case "validatable."
- Solution Domain Knowledge - what the solution "looks like". Is it a website, a desktop application, an artificial intelligence system, etc? This is based on talking with the Architect in order to understand the basics of the solution. This is important so that the solution given at PL2 makes sense to the architect and developers. Future validation will be with architect, so the architect must understand what is "validatable" and produce a technical vision.

Precision Level 2: "How Do They Get It?"

Whereas the first two precision levels resulted from conversations with the customers, users, and stakeholders, this precision level also involves the developers and the testers. Of course you will be continuing to talk with the user, and the user is going to be asked to check this level as well.

The goals of precision level 2 are to:

- Complete the main success scenario for the use case, documenting the steps for success as clearly as possible.
- Validate with the team (users, customers, architect)

Main Success Scenario

At this time your goal is to document the main success scenario of the use case. The main success scenario is a documented conversation between the Actor and the System that accomplishes the Success Postconditions. It is often seen as the "sunny day" scenario, or what you wish would happen when the use case is executed.

A use case always contains more than one scenario, but there is only one main success scenario. This scenario, as written, has only one outcome – and it must achieve the success postconditions. If there is another possible outcome (which there always is) that outcome is documented in an extension to the use case, but not in the main success scenario. This feature helps us maintain our traceability into the code.

Writing Guidelines

Use Cases are textual things meant to be understood by actual human beings. It is not good to make them so cryptic that they are unclear. At the same time, they are also relatively formalized, and terse. The terseness helps us review (validate easily) and communicate when we get to the point of writing code and writing tests.

We like to say that 8th-grade English is appropriate tone for the steps in a scenario. If you can forget that hollow content-free writing style you learned in college, you'll be better off.

When writing the Main Success Scenario of a use case, there are some simple rules:

1. **Keep It Short**. Use cases must be understandable. If they contain too much "stuff" they get impossible to understand. We recommend they have from 3 (enough for something real to happen) to 9 (still possible to understand) steps. If you find yourself writing a use case that contains 10 or more steps, consider finding sub-goals and extracting them as their own use cases, referencing the 'new' use cases from within the one you have.

2. **Keep It Simple**. Use simple, declarative, sentences. For example:

   Subject => verb => direct object => prepositional phrase.

   "The system deducts the amount from the account balance."

3. **Keep it Clear**. This is a conversation. Make sure you know *who* is saying or doing what, and that you understand *what* they are saying or doing. The combination of keeping it short and keeping it simple goes a long way to making sure that it is clear, but make sure that there is as little ambiguity as possible. Each step should be clearly understood to be made up of:
   - Interactions – passing info between the participants
   - Validations – making sure something is ok before doing something else
   - Internal changes – updating data, doing calculations, etc.

4. **Keep it Positive**. This is a 'success scenario' that is showing how the success postconditions are being satisfied. There are no "ifs" – either expressed or implied. We'll take care of all that "but… what else…" stuff when we get to Precision Level 3. Rather than saying "if xxx is true" we say "validate that xxx is true" – we'll take care of the "xxx isn't true" case later.

5. **Use These Tricks**.

Looping. Sometimes you just gotta loop. Use the idiom

"Do steps X-Y until Z" for this, as

"Do steps 2-4 until the Customer selects 'exit'"

Show the other Players as Necessary. When writing "white-box" use cases we often need to show the participation of other systems, either external or internal to the one we are dealing with. So we write steps like:

"Pass the login and password to the *Login Subsystem* for validation" or

"The *Login Subsystem* validates the login and password" or

"Get the number on hand from the *Inventory System*" or

"The *GPS Subsystem* calculates and returns the current location"

Timing Issues. Sometimes a step has to happen within a given time frame or the system is not usable. We just insert or attach this to the step like

"The system calculates the price and displays it within 5 seconds" or

"The system displays the list of employees. (Within 5 seconds)

Precision Level 2 Validation

For our use case, this is what we wind up with, after much conversation, arguing, writing and rewriting – this isn't easy, you know… see the note in the MSS that shows the conditions under which the MSS works – this is what you do instead of wish-away preconditions.

Name: Create Sales Order
Summary: Customer creates and saves a Sales Order.
System: HBS3
Primary Actor: Customer

Stakeholders and Interests:
Customer: Wants an accurate sales order; wants proposed delivery dates to be accurate; wants costing to be accurate.
HBS Owners: Wants accurate sales order, pricing, warehouse and inventory information.
Warehouse: Wants items listed as "in stock" to actually be there; wants accurate SKUs for items
Credit Department: Want sales order to be available in order to arrange payment.
Precondition: Customer Logged in to HBS3
Trigger: Customer selects "New Sales Order" from main menu
Postcondition:
Success: Sales Order completed and saved. Sales Item information is accurate, including quantities in stock and proposed delivery dates. Pricing is accurate and complete.
Minimal: Transaction logged to completion or point of failure

Main Success Scenario
Note: the MSS works if: the customer knows everything (SKU, quantity, color,

etc.) for all the items he wants to buy; everything he wants is in stock; he has delivery details ready; and knows the payment information. All other cases will be handled by extensions.

1. HBS3 displays a form for entering sales order information
2. Steps 3-5 are repeated until the customer selects "done"
3. The customer enters sales item details including merchandise SKU, quantity, color, style, etc, as appropriate for the individual sales item.
4. The system validates the sales item details (including verifying with the Inventory System that there is enough in stock), and presents them on the screen. (within 5 seconds)
5. The system calculates a running total of the total price "so far" (not including delivery and taxes) and presents it on the screen
6. The customer enters the customer, delivery location and date details.
7. The system validates the location and date, calculates the delivery charge and taxes and presents the completed sales order to the customer.
8. The customer okays the sales order and the system saves it.

Once you have a good scenario written, short and clear, it is time to validate again. The customer, the developers, and once again the users are all participants in precision level 2 validation. There are four types of validation:

1. **Does the MSS actually accomplish the success post conditions?** Does the "truth of the use case" match the post conditions? Is it useful? Done by a second pair of technical eyes, either developer or tester – a QA check.

2. **Is the MSS something the User can live with?** Does it satisfy the usability criteria? Done by a user or surrogate, perhaps the customer or a tester that understands usability concerns. Note that the analyst has some knowledge of the problem domain in order to make this validation easier – it's in the right language at least.

3. **Is the solution buildable?** Does it satisfy our architectural concept? Can it be unfolded to software requirements we recognize? Done by architect or developer that understands the architecture. This could be part of a peer review. Note that the analyst understands something of the solution domain in order to make this validation easier – at least the use case speaks the same language as the architect.

4. **Is the solution testable?** Part of validation centricity, should be automatic as the developer has been talking to the tester all along, right?

In some cases, for the very important scenarios, the validation of the usability requires us to create a story board version of the scenario with screen shots and sample interactions. This leads us directly to the ever-unfolding story described in the next chapter.

It's Time to Develop: At this step, though it is not the end of your requirements gathering task, it is time for you to develop. You have a validated story board and a document describing the main success scenario of one of your use cases. Go build it. After you build it, release it and let your users test it out. To move from use cases to development, use the Ever Unfolding Story. When developing, make sure to use good coding practices and to test frequently.

There is no need to wait or stockpile undeveloped requirements or use cases. If the use case you worked out wasn't enough to keep your developers busy, ask the user which is the next most important, and start the use case for that. The key thing here is the increment rather than the iteration. Every time we add on a piece we validate the whole thing.

Precision Level 3: "What Else Could Happen?"

So, we have a validated Main Success Scenario. We might even have analyzed, designed, and coded it. Now what? Now it's time to figure out the "rest of the story" – what else could happen…

Specifically, we are looking for extension conditions – things that can happen that prevent us from getting from one step to the next in the Main Success Scenario. Extensions keep appearing as we develop our use case: we have business extensions that we can find before we do any further analysis, we find some more as we unfold, we find still more as we design and develop, and we find a lot of them as we test.

Identifying the Extension Conditions

For now we'll document some of the "obvious" extensions - those things that we just have to handle. This is so that we can do this without having unfolded first. The idea is that the MSS is not the "whole story" – there are some extensions that are necessary for understanding what is actually going on from a business, or user, perspective.

Some of the things we look for are:

1. **Validation Failing** - These conditions are low hanging fruit. If your system is validating information, the validation failure is always a possibility. An example for us would be if there isn't enough of the sales item we want in stock.

2. **Critical Performance Failures** - The performance areas that you monitor will vary from system to system. Response times are a typical performance metric. If your performance falls outside the limits you decided were acceptable, what do you do? For us, what do we do if it takes more than 5 seconds to display what we want?

3. **Alternative Success Paths** - This is a condition that can happen when the user is logically presented with options in the main success scenario. Since there is only one option on the MSS, the other options must be documented as extensions. In our case, the MSS assumes that the order is being delivered. If the customer wants to just take it home right now, this must be an extension condition.

4. **The Actor Behaving Incorrectly** - This can happen at any time. Your actor is rarely going to do just what you intend them to. They walk away from the system, causing it to time out. Or they just hit "Cancel" when you least expect it. Or…

5. **Internal Failures in System** - This sort of thing is also inevitable, but sometimes less easy to foresee. Sometimes we get them early, sometimes we

get them after the system has been deployed. Believe me, they show up… These would be detectable internal failures, like the paper jams in a printer.

6. **Abnormal Failure Modes** - These might be conditions triggered by safeguards not part of the scenario at all. Duplicate transactions occurring, or violated transaction rules, are possible examples. These are seldom documented this early, but they show up eventually.

Documenting the Extension Conditions

So, by asking the question: "What else could happen?" and following the guidance just given, our team has identified and documented a number of extension conditions. There are a few rules about how they are documented:

- The extensions are listed separately from the MSS, so as not to obscure the MSS. This allows for constant revalidation of the MSS every time somebody goes in to look at it.
- The extension condition is identified by the step number that it extends. That is, if the condition prevents you from getting from step 2 to step 3, it extends step 2. Since there can a number of extensions of step 2, they are numbered 2a, 2b, etc. In those weird cases where the extension prevents you from getting started with step 1, you can number like 0a…
- An extension that can happen at any time is identified with an "*" for the step number.
- The extension condition itself is described by the event that caused the extension or by the event that the system detects. Either will do at this time. The idea is that the customers and users can identify it in order to determine what to do about it.
- For each extension condition feel free to enter notes about what you might want to do about. Don't think too hard, though, it's not time. But if you know what to do, or have questions, or whatever, go ahead and put in a note.

So, here it is…

Use Case: Create Sales Order
… Header Info …

Main Success Scenario

Note: the MSS works if: the customer knows everything (SKU, quantity, color, etc.) for all the items he wants to buy; everything he wants is in stock; he has delivery details ready; and knows the payment information. All other cases will be handled by extensions.

1. HBS3 displays a form for entering sales order information
2. Steps 3-5 are repeated until the customer selects "done"
3. The customer enters sales item details including merchandise SKU, quantity, color, style, etc, as appropriate for the individual sales item.
4. The system validates the sales item details (including verifying with the Inventory System that there is enough in stock), and presents them on the screen. (within 5 seconds)
5. The system calculates a running total of the total price "so far" (not including delivery and taxes) and presents it on the screen
6. The customer enters the customer, delivery location and date details.
7. The system validates the location and date, calculates the delivery charge and taxes and presents the completed sales order to the customer.
8. The customer okays the sales order and the system saves it.

Extensions

*a The Customer hits the “Cancel” button
*b The Customer walks away from the system, and it times out
3a. SKU not found in the database
3b. The Customer does not know the SKU of the item
Note: Maybe we build a look-up system that enables other searches?
4a. Details entered by Customer are invalid
Note: Make the Customer reenter them…
4b. Some is in stock, but not enough
Note: Interesting. Give the number in stock, and a projected delivery date for the rest.
4c. It takes more that 5 seconds to display
Note: maybe we have different time limits for different situations. What should the rule be?
5a. Interesting pricing rules exist
Note: Like What? 3 for price of 2? Stuff like that? Whoaa!
6a. Customer wants to just take it with him/her
Note: so the delivery location is “here” and it is automatically valid…
7a. Delivery Location is not valid or unknown
7b. Proposed delivery date is invalid
8a. The Customer does not OK the sales order

Precision Level 3 Validation

The extensions, added to the end of the Main Success Scenario, provide you with another opportunity for feedback. Go back to the customer, the business owner, the user and the developer, and review with them the work you've done. Show them the revised Main Success Scenario and show them the extensions you have identified. See what they think. Let them help you find things you forgot.

Precision Level 4, How Do We Deal With That?

For each extension condition, you must do something. It may be trivial, it may be complicated, it may be the final solution, or it may be a bandaid. What you do is based on the priorities of the extension. If the extension is something that almost never happens, you may not do much of a fix, but if it is something that happens a lot you may put a lot of effort into it.

In any case, you’ve got to think about what to do, because if you don’t the developer writing the code will do the thinking for you. If you’re the user or the customer, ask yourself if this is what you want… or do you want to be involved.

Extension Handling

For each Extension Condition you listed in the previous precision level, you now have several choices. You can:

- Clean Up Mess and Bail Out (CUMABO) – graceful failure
- Do something to fix the problem (Interact with the user for a few extra steps, perhaps), and reenter the MSS
- Fork to a different Use Case in order to try to succeed in a different way

The choices you have to make are rarely obvious, therefore you should get as much guidance as possible. In fact, deciding which Extension Conditions to handle, and how, is one of the biggest problems you will have. First of all, you don't know all of them all up front, so some of the most interesting ones will hit you unexpectedly. Secondly, they might take emergency resources to solve.

So. Let's say that we have decided to work on the following extension:

3a. The Customer does not know the SKU of the item.

Let's say that after much discussion with the users and customer we decide to build a lookup, search, system. To do this, we need to document the steps that will actually happen – a miniature use case… this documentation can take place right there in the extensions section

3a. SKU not found in the database
    …
3b. The Customer does not know the SKU of the item
    1. The Customer selects "Search for Item"
    2. The System pops up a form for the Customer to enter search criteria
    3. The Customer enters search criteria (e.g., brand name, words to search for in description field) and hits "Search"
    4. The System displays a list of Items that match the search
    5. The Customer selects the item he wants and continues
4a. Details entered by Customer are invalid

Note that there a couple of interesting things happening here:

- There are enough steps for this solution to be its own use case (between 3 and 9)
- There are some obvious extensions to the extension steps themselves
  4a. No items match the search
  5a. The Customer didn't find what he was looking for

For these reasons it is reasonable to create a new use case, "Search for Item", and just reference it here, as

3a. SKU not found in the database
    …
3b. The Customer does not know the SKU of the item
    The Customer searches for the item using the Search for Item use case and continues
4a. Details entered by Customer are invalid

This new use case needs to be analyzed, and will then have its own extensions, etc. This analysis of the extension handling use case is an example of the Ever Unfolding Story in action (see the next chapter), and points out the inherent fractal nature of use cases and requirements in general.

Precision Level 4 Validation

Validation of the extension handling is much like validation of the MSS itself. The extension handling has just defined another scenario, or path, through the use case, which either leads to success or failure:

- If it leads to success, we must assure that it satisfies the success postconditions
- If it leads to failure we must assure that it satisfies the minimal postconditions or whatever other postconditions may apply to the extension
- We must validate that the solution is usable – that the user is willing to put up with it
- We must validate that the solution is doable within our architectural framework

Summary

Writing an effective use case is an incremental process. As we move through the precision levels we learn more, document more, and validate more. We ask the following questions:

PL0:	What's the Scope?
PL1:	Who Wants What?
PL2:	How Do They Get It?
PL3:	What Else Could Happen?
PL4:	How Do We Deal With That?

As complete scenarios get developed and validated we can move to further development. In the next chapter we show how the Ever-Unfolding story is used to analyze a scenario to tease out the software elements lurking within.