



Scenarios: use cases, user stories

In the System book — and occasionally in the Goals and Environment books — you must describe behaviors (1.6.1). One way is to present typical usage scenarios; for example, in an insurance system, how the system will process a claim, as seen from the viewpoint of a claims agent.

Requirements engineering methods use three kinds of scenario:

- A **use case** is a complete run through the system, as in this example.
- A **user story** is of a much smaller scope: one unit of interaction, for example logging into the system. User stories play a particularly important role in the definition of requirements in agile methods such as Scrum.
- A **use case slice** is a combination of the previous two notions.

Section 7.1 describes use cases, 7.2 presents user stories. Section 7.4 discusses the relevance and limitations of these techniques for requirements analysis, in preparation of a comparison with object-oriented requirements in chapter 8.

7.1 USE CASES

Several use case formalisms exist. The most widely publicized ones are graphical, using figures to represent actors of the process. It is convenient to use a tabular and textual notation due to Cockburn (see bibliographical notes). The following example, describing the handling of a claim by an insurance company, is from the same source. (The “*Comment*” parts are not by Cockburn but added here as explanations.)

Use case: process insurance loss claim	
Name	Process_loss_claim <i>Comment: this use case is about processing a “claim”, that is to say a request for payment following a loss event for a customer.</i>
Scope	Insurance company operations
Level	Business summary
Primary actor	Claims adjuster <i>Comment: a “claims adjuster” is a representative of the insurance company in charge of assessing insurance claims.</i>
Context of use	Claims adjuster handles claim.

Preconditions	A loss has occurred. <i>Comment: necessary condition for the use case to proceed.</i>
Trigger	A claim is reported to the insurance company. <i>Comment: event causing the use case to proceed.</i>
Main success scenario	1. A reporting party who is aware of the event registers a loss to the insurance company. 2. A clerk receives and assigns the claim to a claims agent. 3. The assigned claims adjuster: • 3.1 Conducts an investigation. • 3.2 Evaluates damages. • 3.3 Sets reserves. <i>Comment: a “reserve” in the insurance business is an accounting provision made upon receipt of a claim (particularly a large one), based on an estimate of the possible cost to the company, prior to the final assessment which may take days, weeks or more. Provisioning such an estimate in the meantime avoids a disruptive financial surprise at the end of the process in case it does lead to the determination that the company must pay up.</i> • 3.4 Negotiates the claim. • 3.5 Resolves the claim and closes it.
Success guarantee	The claim is resolved and closed.
Extensions	A. Submitted data is incomplete: • A.1 Insurance company requests missing information. • A.2 Claimant supplies missing information. - A.2.1 Claimant does not supply information by deadline: - A.2.1.1 Adjuster closes claim. B. Claimant does not own a valid policy: • B.1 Insurance company declines claim, notifies claimant, updates claim, closes claim.
Stakeholders and interests	<i>Comment: this section describes the various actors in the process — people and organizations.</i> • Insurance company divisions which sell insurance company policies. • Insurance company customers who have purchased policies. • Department of Insurance who sets market conduct. • Claimants who have loss as a result of act of an insured. • Insurance company Claims Division. • Future customers.

The core component (highlighted) is the **main success scenario**, giving the sequence of steps of an instance of the use case. Such scenarios have strong similarities to algorithms or programs:

- Their steps can be decomposed into sub-steps: here step 3 itself consists of a sequence of sub-steps, as does a subroutine (also called subprogram, method, function) in programming.
- Although the example does not illustrate it, a scenario can include conditional steps (two sub-scenarios, of which one will be chosen based on the outcome of a certain condition).

Unlike an algorithm or program, however, a use case (and more generally a scenario, such as a user story as reviewed next) is not meant to be carried out by a computer: instead, it describes the interaction between a human actor and the system.

The **level** entry characterizes the level of abstraction. A use case can describe a process at many levels, from the highest (a bird’s eye view of an overall business process, meant to be complemented by further use cases for the details) down to the detailed descriptions of the system’s actual operation.

A **precondition** is a limiting condition governing the use case’s applicability. A **trigger** is an event that sets off the use case. Note the difference: the precondition is a logical condition which should hold if the use case is to be executed, but does not by itself cause such an execution. Only the trigger does. In the example, the precondition is that a loss has occurred; the trigger is that the loss has been reported. Beyond their application to use cases, preconditions are a general requirements concept, also known as an “assumption” and discussed in 1.7.

The **success guarantee** characterizes the state resulting from a successful execution of the use case, per the “main success scenario”. In the terminology of mathematical software verification, from which the term “precondition” is borrowed, it would be called a “postcondition”. Like preconditions, postconditions (success guarantees) are a general concept of requirements, which we encountered in 1.7. We may also relate success guarantees to a testing concept, “oracle”, which denotes the condition that a successful test must ensure.

Extensions describe departures from the main success scenario. They serve two purposes:

- An extension can specify an alternate path for cases in which the main success scenario hits a condition that prevents it from proceeding — such as, in the example, a claimant who does not have an applicable insurance policy.
- Extensions also support *reuse* of elements common to several use cases. It is good practice to divide a set of related use cases into a base use case (covering common elements) and extensions. For example, in a further development of the claim-processing use case, variants for accident, loss and robbery might have both common elements and specific details. Rather than writing three separate use cases with significant repetition, you can write one base claim-processing use case, then extensions covering the specifics of each variant.

The use case lists, towards the beginning, the **main actor** responsible for carrying out instances of the scenario. It concludes with a list of **stakeholders**: other groups of people who may be affected. We saw how important it is for requirements in general to include consideration of all possibly affected stakeholders (“Categories of stakeholders”, 1.10.1, page 13).

7.2 USER STORIES

A use case is a complete path taken by an actor through the system. A user story is also a usage scenario for actors interacting with the system, but much more limited in scope and complexity.

User stories play an important role for requirements in *agile methods*, a set of development methods which promote incremental program construction. Two of the best-known agile methods are XP (Extreme Programming), which introduced user stories, and Scrum, today the most widely used. In agile methods, the basic development iteration involves a developer picking the next item from a list of functions to be implemented, implementing it, and moving on to the next one. For such a scheme to work, each such function has to be small; a use case in the above style is not adequate because it describes a collection of related functions rather than a unit of functionality. For this reason, agile methods generally rely on a finer-grain unit: the user story.

The standard format for a user story consists of three elements:

- A role, denoting the main actor of a use case (remember “**Special case: role**”, page 11).
- A desired function — part of the system’s behavior (“**Behavior**”, page 8).
- A business purpose (a goal in the general requirements terminology: “**Goal**”, page 6).

For example:

As a claims agent, I want prompt claim processing, so as to achieve customer satisfaction.

A standard set of quality criteria for user stories goes by the acronym INVEST:

The INVEST criteria for user stories	
Criterion	Explanation
I ndependent	Each user story should be describable on its own.
N egotiable	A user story is not a cast-in-stone specification imposed by a business analyst, but a proposal intended for discussion with stakeholders and, particularly, with the programmer who implements it and may find reasons to suggest changes.
V aluable	The functionality captured by a user story should bring the organization a demonstrable business benefit.
E stimable	The user story should be described precisely enough to allow at least a rough estimate of the effort required to implement it.
S mall	Unlike use cases, user stories capture small units of functionality, typically implementable in a few person-days or at most weeks.
T estable	Every user story should lend itself to the design of one or more tests assessing whether a candidate implementation meets it. In keeping with other principles of agile development, particularly XP, the agile advice is to write these tests immediately after writing the user story, before starting the implementation.

7.3 EPICS AND USE CASE SLICES

User stories and use cases lie at opposite ends of the size and complexity range for scenarios:

- A user story describes a small and focused scenario, which a programmer can typically implement in a few days.
- A use case covers an entire path of interaction with the system, all the way to obtaining a meaningful result. It may involve a considerable amount of system functionality.

Compare, in a system supporting an e-commerce platform, “*validate seller login*” and “*enable seller to enter a new range of products*”.

A need also exists for scenarios at levels of granularity in-between these extremes. To address it, one may start from either end: gather user stories into larger units; or decompose use cases into smaller units. These two ideas lead to *epics* and *use case slices*.

The first approach starts from the user story side, towards larger granularity. An epic is a collection of user stories, all directed towards a common goal (in the general requirements sense of this term, the “G” in PEGS). A typical epic for an e-commerce platform could be “*decrease the percentage of fraudulent sellers by 20%*”, involving many user stories.

The second approach starts from the use case side, towards finer granularity. “Use Case 2.0”, the revision (introduced around 2006) of the original use case methodology from the 1990s, includes the notion of use case slice. In the words of the concept’s originators (see the bibliographical notes), a use case slice consists of “*one or more stories selected from a use case to form a work item that is of clear value to the customer*”. The use case slice “*acts as a placeholder for all the work required to complete the implementation of the selected stories*”. Consider (in the e-commerce example) a sophisticated use case describing the process of registering a new seller on the e-commerce site. It may be interesting to extract from it a use case slice describing all the paths through the use case that have relevance to issues of security and fraud-prevention.

The term *scenario* covers use cases, user stories, epics, use case slices and any other technique that specifies system behavior by describing paths of interaction with the system. The discussion of scenarios, in the following section and in the next chapter, will mostly take use cases and user stories as examples but applies just as well to all other variants.

7.4 THE BENEFITS OF SCENARIOS FOR REQUIREMENTS

Scenarios — in particular use cases and user stories — describe units of behavior. They can help requirements in two different ways:

- As an **elicitation tool**. Chapter 6 emphasized how important it is to use a diversity of techniques to ensure that stakeholders express their needs. Scenarios, which express snapshots of system behavior in a form that speaks directly to future users, can be easier to obtain from them than more abstract specifications.

- As a **verification tool**. Both use cases and user stories describe scenarios of possible use that have been identified as essential: whatever else the system does, it must support these scenarios. Put another way, they are *system-level tests*. What a test case is to a single program element, a use case or user story is a test of the entire system or a subsystem.

To cover these two roles, the Standard Plan of requirements reserves not just one but two chapters to scenarios:

- **G.5** in the Project book, for the principal high-level scenarios, of interest to all stakeholders and directly connected to the Goals.
- **S.4** in the System book, for detailed scenarios (use cases or, depending on the project's choice, user stories) exercising specific functions of the system.

References to these scenarios can also occur in the verification plan (**S.6**).

7.5 THE LIMITATIONS OF SCENARIOS FOR REQUIREMENTS

Use cases and user stories cannot by themselves define requirements. Several reasons make them insufficient for that purpose.

Note first that scenarios only apply — out of the four PEGS — to the System and to the Goals. They bring nothing to the specification of the Project and Environment.

For the description of system properties (behaviors), the most important limitation is that a scenario only describes one case (or a set of related cases if we take advantage of conditional branching and extensions). Typically, it describes the most desirable case (“*main success scenario*” as illustrated in 7.1) and some variants (“*extensions*”). Those cases, however, are the easy part.

If software development could focus on the simple, straightforward cases, it would be a cakewalk. Much of the difficulty of software and particularly of software requirements comes from the myriad of variants that any significant system must be prepared to handle. Even in a basic case such as insurance-claim processing in 7.1, any attempt to specify things seriously leads to an explosion of cases, which no scenario-based technique can handle. Consider this part of the “extension” (extracted from the complete text on page 130):

Extensions	B. Claimant does not own a valid policy: • B.1 Insurance company declines claim, notifies claimant, updates claim, closes claim.
------------	--

There will inevitably be cases of the company mistakenly declaring that the claimant does not own a policy and then correcting the mistake (for example, the renewal was in the mail). There has to be a scheme for that case, such as reopening the claim. As another example, the “main success scenario” includes the following:

Main success scenario	[Extract only, from the full scenario on page 130] 3. The assigned claims adjuster: • 3.1 Conducts an investigation. • 3.2 Evaluates damages. • 3.3 Sets reserves. • 3.4 Negotiates the claim. • 3.5 Resolves the claim and closes it.
-----------------------	---

It will inevitably occur in practice that some customer, during the “negotiates the claim” step, contests the result of the earlier “evaluates damages” step as too low, or the adverse party’s insurer contests it as too high. Then step 3.2 needs to be reopened after 3.4, causing what in programming would be called a loop.

One can add “extensions” covering such situations, or write new use cases to address them. Scaling up such a strategy is, however, hopeless: for any non-trivial system there are so many combinations that it is impossible to describe them all, or even a small subset. Requirements must provide a description of the system (and other PEGS components) at a more abstract level.

The difference between scenarios and requirements is a generalization of the difference between a test and a specification. We could attempt to specify an integer function by stating that it yields 0 for the argument 0, 1 for 1, 4 for 2, 9 for 3 and 16 for 4. That would not be a specification, however, just a set of tests. It says nothing about the function’s result for 5, 6, or one million. (You may try, for fun, to use a curve-fitting program, easy to find on the Web, to see which functions — besides “square” — match these tests.) A true specification is: “ $f(n) = n^2$ for every integer n .” It captures all possible cases, not just a few out of a myriad or infinity.

The influence of scenario-based specifications can be seen in many Web applications which work well for users whose needs exactly fit one of the planned schemes but let you down as soon as your need is in any way special. A serious requirements effort cannot stop at identifying the most common cases; it must define general behaviors.

7.6 THE ROLE OF USE CASES AND USER STORIES IN REQUIREMENTS

The consequences of the preceding discussion are clear. Scenarios are not suitable by themselves as requirements. Combined with other techniques, however, they can play an important part in the requirements effort, in two ways:

- As a **requirements elicitation tool**. Identifying user stories and use cases helps identify key components and behaviors of the system.
- As a **requirements verification tool**. While scenarios can only cover a very small subset of the possible paths through the system — a drop in the ocean — they can cover some of the *most important* paths. In this role they address an important form of completeness: the system’s ability to handle the scenarios of most importance to key stakeholders. The notion of scenario completeness (11.3) will address this concern.

Useful requirements demand more abstract descriptions of behavior. Modern object-oriented requirements analysis, studied in the next chapter, provides a better approach.

The following principle summarizes this assessment:

Scenario Principle

Take advantage of use cases, user stories, use case slices, tests and other forms of scenarios as tools for elicitation and verification of requirements.

Do not use them as a substitute for actual requirements specifications.

7-E EXERCISES

7-E.1 Scenarios for graphical manipulation

Numerous interactive systems make it possible to produce figures, typically as part of a text, and perform various manipulations of them (move, rotate, group, ...). Typical examples in the Microsoft world are Word and PowerPoint. Using any such systems with which you are familiar as a user, and assuming that you were instead in the designer's seat, write the fundamental use cases and user stories for the figure-related capability. Define any epics or use case slices that may be deduced from them.

7-E.2 User stories from a use case

From the insurance-related use case of this chapter (first presented in 7.1), derive user stories.

BIBLIOGRAPHICAL NOTES AND FURTHER READING

The book that originally popularized use cases was [Jacobson 1992], which covers a broader spectrum of software engineering techniques and remains an important reference. A shorter but effective exposition of use case techniques is [Cockburn 2001]. The example in section 7.1 is based (with permission) on its use cases 19 and 21 (themselves credited by Cockburn to the Fireman's Fund insurance company).

[Glinz 2000], a classic paper on the suitability of UML for requirements, contains an in-depth critical analysis of the limitations of use cases, which also provides important insights (for anyone working with use cases anyway) on how to use them effectively.

A discussion of requirements, [Meyer 2012a], takes seriously the idea of using a curve-fitting tool to find functions matching the first few values of the square function (page 135), leading to a few unexpected solutions.

On user stories, a good reference is a book about requirements in agile methods: [Leffingwell 2011]. On agile methods in general see [Meyer 2014]. The INVEST set of criteria for user stories (7.2) appears in [Wake 2003-2017], from which the explanations on page 132 are taken in part.

[Jacobson-Spence-Kerr 2016] is the reference on use case slices and more generally on Use Case 2.0.