

6.9. What do the Sections Mean?

Preface Elements

Scope

The scope bounds the system (or systems) under design. Typically, a use case describes use of one software (or hardware plus software) system; in this case it is known as a **system use case**. At a broader scope, use cases can also describe how a business is used by its customers and partners. Such an enterprise-level process description is called a **business use case** and is a good example of the wide applicability of use cases, but they aren't covered in this introductory book.

Level

In Cockburn's system, use cases are classified as at the user-goal level or the subfunction level, among others. A **user-goal level** use case is the common kind that describe the scenarios to fulfill the goals of a primary actor to get work done; it roughly corresponds to an **elementary business process** (EBP) in business process engineering. A **subfunction-level** use case describes substeps required to support a user goal, and is usually created to factor out duplicate substeps shared by several regular use cases (to avoid duplicating common text); an example is the subfunction use case *Pay by Credit*, which could be shared by many regular use cases.

[EBP](#) p. [88](#)

see the use case ["include" relationship](#) for more on subfunction use cases p. [494](#)

Primary Actor

The principal actor that calls upon system services to fulfill a goal.

Stakeholders and Interests ListImportant!

This list is more important and practical than may appear at first glance. It suggests and bounds what the system must do. To quote:

The [system] operates a contract between stakeholders, with the use cases detailing the behavioral parts of that contract...The use case, as the contract for behavior, captures *all and only* the behaviors related to satisfying the stakeholders' interests [[Cockburn01](#)].

This answers the question: What should be in the use case? The answer is: That which satisfies all the stakeholders' interests. In addition, by starting with the stakeholders and their interests before writing the remainder of the use case, we have a method to remind us what the more detailed responsibilities of the system should be. For example, would I have identified a responsibility for salesperson commission handling if I had not first listed the salesperson stakeholder and their interests? Hopefully eventually, but perhaps I would have missed it during the first analysis session. The stakeholder interest viewpoint provides a thorough and methodical procedure for discovering and recording all the required behaviors.

Stakeholders and Interests:

- Cashier: Wants accurate, fast entry and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- ...

Preconditions and Success Guarantees (Postconditions)

First, don't bother with a precondition or success guarantee unless you are stating something non-obvious and noteworthy, to help the reader gain insight. Don't add useless noise to requirements documents.

Preconditions state what *must always* be true before a scenario is begun in the use case. Preconditions are *not* tested within the use case; rather, they are conditions that are assumed to be true. Typically, a precondition implies a scenario of another use case, such as logging in, that has successfully completed. Note that there are conditions that must be true, but are not worth writing, such as "the system has power." Preconditions communicate noteworthy assumptions that the writer thinks readers should be alerted to.

Success guarantees (or **postconditions**) state what must be true on successful completion of the use case either the main success scenario or some alternate path. The guarantee should meet the needs of all stakeholders.

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Main Success Scenario and Steps (or Basic Flow)

This has also been called the "happy path" scenario, or the more prosaic "Basic Flow" or "Typical Flow." It describes a typical success path that satisfies the interests of the stakeholders. Note that it often does *not* include any conditions or branching. Although not wrong or illegal, it is arguably more comprehensible and extendible to be very consistent and defer all conditional handling to the Extensions section.

Guideline

Defer all conditional and branching statements to the Extensions section.

The scenario records the steps, of which there are three kinds:

1. An interaction between actors.[\[3\]](#)

[3] Note that the system under discussion itself should be considered an actor when it plays an actor role collaborating with other systems.

2. A validation (usually by the system).
3. A state change by the system (for example, recording or modifying something).

Step one of a use case does not always fall into this classification, but indicates the trigger event that starts the scenario.

It is a common idiom to always capitalize the actors' names for ease of identification. Observe also the idiom that is used to indicate repetition.

Main Success Scenario:

1. Customer arrives at a POS checkout with items to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. ...

Cashier repeats steps 3-4 until indicates done.

5. ...

Extensions (or Alternate Flows)

Extensions are important and normally comprise the majority of the text. They indicate all the other scenarios or branches, both success and failure. Observe in the fully dressed example that the Extensions section was considerably longer and more complex than the Main Success Scenario section; this is common.

In thorough use case writing, the combination of the happy path and extension scenarios should satisfy "nearly" all the interests of the stakeholders. This point is qualified, because some interests may best be captured as non-functional requirements expressed in the Supplementary

Specification rather than the use cases. For example, the customer's interest for a visible display of descriptions and prices is a usability requirement.

Extension scenarios are branches from the main success scenario, and so can be notated with respect to its steps 1...N. For example, at Step 3 of the main success scenario there may be an invalid item identifier, either because it was incorrectly entered or unknown to the system. An extension is labeled "3a"; it first identifies the condition and then the response. Alternate extensions at Step 3 are labeled "3b" and so forth.

Extensions:

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

An extension has two parts: the condition and the handling.

Guideline: When possible, write the condition as something that can be *detected* by the system or an actor. To contrast:

5a. System detects failure to communicate with external tax calculation system service:

5a. External tax calculation system not working:

The former style is preferred because this is something the system can detect; the latter is an inference.

Extension handling can be summarized in one step, or include a sequence, as in this example, which also illustrates notation to indicate that a condition can arise within a range of steps:

3-6a: Customer asks Cashier to remove an item from the purchase:

1. Cashier enters the item identifier for removal from the sale.
- System displays updated running total.

At the end of extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (such as by halting the system).

Sometimes, a particular extension point is quite complex, as in the "paying by credit" extension. This can be a motivation to express the extension as a separate use case.

This extension example also demonstrates the notation to express failures within extensions.

7b. Paying by credit:

1. Customer enters their credit account information.
- System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.

2a. System detects failure to collaborate with external system:

1. System signals error to Cashier.
- Cashier asks Customer for alternate payment.

If it is desirable to describe an extension condition as possible during any (or at least most) steps, the labels *a, *b, ..., can be used.

*a. At any time, System crashes:

In order to support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered at any step in the scenario.

1. Cashier restarts the System, logs in, and requests recovery of prior state.
- System reconstructs prior state.

Performing Another Use Case Scenario

Sometimes, a use case branches to perform another use case scenario. For example, the story *Find Product Help* (to show product details, such as description, price, a picture or video, and so on) is a distinct use case that is sometimes performed while within *Process Sale* (usually when the item ID can't be found). In Cockburn notation, performing this second use case is shown with underlining, as this example shows:

3a. Invalid item ID (not found in system):

1. System signals error and rejects entry.
2. Cashier responds to the error:

2a. ...

2c. Cashier performs Find Product Help to obtain true item ID and price.

Assuming, as usual, that the use cases are written with a hyperlinking tool, then clicking on this underlined use case name will display its text.

Special Requirements

If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 2 and 6.

Recording these with the use case is classic UP advice, and a reasonable location when *first* writing the use case. However, many practitioners find it useful to ultimately move and consolidate all non-functional requirements in the Supplementary Specification, for content management, comprehension, and readability, because these requirements usually have to be considered as a whole during architectural analysis.

Technology and Data Variations List

Often there are technical variations in *how* something must be done, but not what, and it is noteworthy to record this in the use case. A common example is a technical constraint imposed by a stakeholder regarding input or output technologies. For example, a stakeholder might say, "The POS system must support credit account input using a card reader and the keyboard." Note that these are examples of early design decisions or constraints; in general, it is skillful to avoid premature design decisions, but sometimes they are obvious or unavoidable, especially concerning input/output technologies.

It is also necessary to understand variations in data schemes, such as using UPCs or EANs for item identifiers, encoded in bar code symbology.

Congratulations: Use Cases are Written and Wrong (!)

The NextGen POS team is writing a few use cases in multiple short requirements workshops, in parallel with a series of short timeboxed development iterations that involve production-quality programming and testing. The team is incrementally adding to the use case set, and refining and adapting based on feedback from early programming, tests, and demos. Subject matter experts, cashiers, and developers actively participate in requirements analysis.

That's a good evolutionary analysis process rather than the waterfall but a dose of "requirements realism" is still needed. Written specifications and other models give the *illusion* of correctness, but models lie (unintentionally). Only code and tests reveals the truth of what's really wanted and works.

The use cases, UML diagrams, and so forth won't be perfect guaranteed. They will lack critical information and contain wrong statements. The solution is not the waterfall attitude of trying to record specifications near-perfect and complete at the start although of course we do the best we can in the time available, and should learn and apply great requirements practices. But it will never be enough.

This isn't a call to rush to coding without any analysis or modeling. There is a middle way, between the waterfall and ad hoc programming: iterative and evolutionary development. In this approach the use cases and other models are incrementally refined, verified, and clarified through early programming and testing.

You know you're on the wrong path if the team tries to write in detail all or most of the use cases before beginning the first development iteration or the opposite.

This list is the place to record such variations. It is also useful to record variations in the data that may be captured at a particular step.

Technology and Data Variations List:

3a. Item identifier entered by laser scanner or keyboard.

3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

7a. Credit account information entered by card reader or keyboard.

7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.