Use case

From Wikipedia, the free encyclopedia

A **use case** in software engineering and systems engineering is a description of a system’s behavior as it responds to a request that originates from outside of that system. In other words, a use case describes "who" can do "what" with the system in question. The use case technique is used to capture a system's behavioral requirements by detailing scenario-driven threads through the functional requirements.

Contents

Overview

Use cases describe the system from the user's point of view.

Within systems engineering, use cases are used at a higher level than within software engineering, often representing missions or stakeholder goals. The detailed requirements may then be captured in SysML requirement diagrams or similar mechanisms.

History

In 1986, Ivar Jacobson, later an important contributor to both the Unified Modeling Language (UML) and the Rational Unified Process (RUP), first formulated the visual modeling technique for specifying use cases. Originally he used the terms *usage scenarios* and *usage case*, but found that neither of these terms sounded natural in English, and eventually he settled on the term *use case*.[1] Since Jacobson originated use case modeling many others have contributed to improving this technique, including Kurt Bittner, Ian Spence, Alistair Cockburn, Gunnar Overgaard, Karin Palmquist and Geri Schneider.

During the 1990s use cases became one of the most common practices for capturing functional requirements. This is especially the case within the object-oriented community where they originated, but their applicability is not restricted to object-oriented systems, because use cases are not object-oriented in nature.

Use case topics

Use case focus

"Each use case focuses on describing how to achieve a goal or a task. For most software projects, this means that multiple, perhaps dozens of use cases are needed to define the scope of the new system. The degree of formality of a particular software project and the stage of the project will influence the level of detail required in each use case."

Use cases should not be confused with the features of the system under consideration. A use case may be related to one or more features, and a feature may be related to one or more use cases.

A use case defines the interactions between external actors and the system under consideration to accomplish a goal. An actor specifies a role played by a person or thing when interacting with the system.[2] The same person using the system may be represented as different actors because they are playing different roles. For example, "Joe" could be playing the role of a Customer when using an Automated Teller Machine to withdraw cash, or playing the role of a Bank Teller when using the system to restock the cash drawer.

Use cases treat the system as a black box, and the interactions with the system, including system responses, are perceived as from outside the system. This is a deliberate policy, because it forces the author to focus on what the system must do, not how it is to be done, and avoids the trap of making assumptions about how the functionality will be accomplished.

Use cases may be described at the abstract level (business use case, sometimes called essential use case), or at the system level (system use case). The differences between these is the scope.

- A **business use case** is described in technology-free terminology which treats system as a black box and describes the business process that is used by its business actors (people or systems external to the business) to achieve their goals (e.g., manual payment processing, expense report approval, manage corporate real estate). The business use case will describe a process that provides value to the business actor, and it describes *what* the process does. Business Process Mapping is another method for this level of business description.
- A **system use case** is normally described at the system functionality level (for example, create voucher) and specifies the function or the service that the system provides for the user. The system use case will describe *what* the actor achieves interacting with the system. For this reason it is recommended that system use case specification begin with a verb (e.g., *create* voucher, *select* payments, *exclude* payment, *cancel* voucher). Generally, the actor could be a human user or another system interacting with the system being defined.

A use case should:

- Describe what the system shall do for the actor to achieve a particular goal.
- Include no implementation-specific language.
- Be at the appropriate level of detail.
- Not include detail regarding user interfaces and screens. This is done in user-interface design.

Degree of detail

Alistair Cockburn, in *Writing Effective Use Cases*, identified three levels of detail in writing use cases:[3]

- Brief use case

  consists of a few sentences summarizing the use case. It can be easily inserted in a spreadsheet cell, and allows the other columns in the spreadsheet to record priority, duration, a method of estimating duration, technical complexity, release number, and so on.

- Casual use case

  consists of a few paragraphs of text, summarizing the use case.

- Fully dressed use case

  is a formal document based on a detailed template with fields for various sections; and it is the most common understanding of the meaning of a use case. Fully dressed use cases are discussed in detail in the next section on use case templates.

Appropriate detail

Some software development processes do not require anything more than a simple use case to define requirements. However, some other development processes require detailed use cases to define requirements. The larger and more complex the project, the more likely that it will be necessary to use detailed use cases.

The level of detail in a use case often differs according to the progress of the project. The initial use cases may be brief, but as the development process unfolds the use cases become ever more detailed. This reflects the different requirements of the use case. Initially they need only be brief, because they are used to summarize the business requirement from the point of view of users. However, later in the process, software developers need far more specific and detailed guidance.

The Rational Unified Process invites developers to write a brief use case description in the use case diagram, with a casual description as comments and a detailed description of the flow of events in a textual analysis. All those can usually be input into the use case tool (e.g., a UML Tool, SysML Tool), or can be written separately in a text editor.

Use case notation

In Unified Modeling Language, the relationships between all (or a set of) the use cases and actors are represented in a use case diagram or diagrams, originally based upon Ivar Jacobson's Objectory notation. SysML, a UML profile, uses the same notation at the system block level.

Use cases and the development process

The specific way use cases are used within the development process will depend on which development methodology is being used. In certain development methodologies, a brief use case survey is all that is required. In other development methodologies, use cases evolve in complexity and change in character as the development process proceeds. In some methodologies, they may begin as brief business use cases, evolve into more detailed system use cases, and then eventually develop into highly detailed and exhaustive test cases.

Use case templates

There is no standard template for documenting detailed use cases. A number of competing schemes exist, and individuals are encouraged to use templates that work for them or the project they are on. Standardization within each project is more important than the detail of a specific template. There is, however, considerable agreement about the core sections; beneath differing terminologies and orderings there is an underlying similarity between most use cases. Different templates often have additional sections, e.g., assumptions, exceptions, recommendations, technical requirements. There may also be industry specific sections.

Use case name

A use case name provides a unique identifier for the use case. It should be written in verb-noun format (e.g., *Borrow Books*, *Withdraw Cash*), should describe an achievable goal (e.g., *Register User* is better than *Registering User*) and should be sufficient for the end user to understand what the use case is about.

Goal-driven use case analysis will name use cases according to the actor's goals, thus ensuring use cases are strongly user centric. Two to three words is the optimum. If more than four words are proposed for a name, there is usually a shorter and more specific name that could be used.

Version

Often a version section is needed to inform the reader of the stage a use case has reached. The initial use case developed for business analysis and scoping may well be very different from the evolved version of that use case when the software is being developed. Older versions of the use case may still be in current documents, because they may be valuable to different user groups.

Goal

Without a goal a use case is useless. There is no need for a use case when there is no need for any actor to achieve a goal. A goal briefly describes what the user intends to achieve with this use case.

Summary

A summary section is used to capture the essence of a use case before the main body is complete. It provides a quick overview, which is intended to save the reader from having to read the full contents of a use case to understand what the use case is about. Ideally, a summary is just a few sentences or a paragraph in length and includes the goal and principal actor.

Actors

An actor is someone or something outside the system that either acts on the system – a primary actor – or is acted on by the system – a secondary actor. An actor may be a person, a device, another system or sub-system, or time. Actors represent the different roles that something outside has in its relationship with the system whose functional requirements are being specified. An individual in the real world can be represented by several actors if they have several different roles and goals in regards to a system. These interact with system and do some action on that.

Stakeholders

A stakeholder is an individual or department that is affected by the outcome of the use case.[4] Individuals are usually agents of the organization or department for which the use case is being created. A stakeholder might be called on to provide input, feedback, or authorization for the use case.[5] The stakeholder section of the use case can include a brief description of which of these functions the stakeholder is assigned to fulfill.

Preconditions

A *preconditions* section defines all the conditions that must be true (i.e., describes the state of the system) for the *trigger* (see below) to meaningfully cause the initiation of the use case. That is, if the system is not in the state described in the preconditions, the behavior of the use case is indeterminate. Note that the preconditions are *not* the same thing as the "trigger" (see below): the mere fact that the preconditions are met does NOT initiate the use case.

However, it is theoretically possible *both* that a use case should be initiated whenever condition X is met *and* that condition X is the only aspect of the system that defines whether the use case can meaningfully start. If this is really true, then condition X is *both* the precondition and the trigger, and would appear in both sections. But this is *rare*, and the analyst should check carefully that they have not overlooked some preconditions which are part of the trigger. If the analyst has erred, the module based on this use case will be triggered when the system is in a state the developer has not planned for, and the module may fail or behave unpredictably.

Triggers

A 'triggers' section describes the event that causes the use case to be initiated. This event can be external, temporal or internal. If the trigger is not a simple true "event" (e.g., the customer presses a button), but instead "when a set of conditions are met", there will need to be a triggering process that continually (or periodically) runs to test whether the "trigger conditions" are met: the "triggering event" is a signal from the trigger process that the conditions are now met.

There is varying practice over how to describe what to do when the trigger occurs but the preconditions are not met.

- One way is to handle the "error" within the use case (as an exception). Strictly, this is illogical, because the "preconditions" are now not true preconditions at all (because the behavior of the use case is determined even when the preconditions are not met).
- Another way is to put all the preconditions in the trigger (so that the use case does not run if the preconditions are not met) and create a different use case to handle the problem. Note that if this is the local standard, then the use case template theoretically does not need a preconditions section!

Basic course of events

At a minimum, each use case should convey a *primary scenario*, or typical course of events, also called "basic flow", "normal flow," "happy flow" and "Happy path". The main basic course of events is often conveyed as a set of usually numbered steps. For example:

```
1. The system prompts the user to log on,
2. The user enters his name and password,
3. The system verifies the logon information,
4. The system logs user on to system.
```

Alternative paths or Extensions

Use cases may contain secondary paths or alternative scenarios, which are variations on the main theme. Each tested rule may lead to an alternative path and when there are many rules the permutation of paths increases rapidly, which can lead to very complex documents. Sometimes it is better to use conditional logic or activity diagrams to describe use case with many rules and conditions.

Exceptions, or what happens when things go wrong at the system level, may also be described, not using the alternative paths section but in a section of their own. Alternative paths make use of the numbering of the basic course of events to show at which point they differ from the basic scenario, and, if appropriate, where they rejoin. The intention is to avoid repeating information unnecessarily. The description of an exception should indicate how the system will respond to, or (if possible) recover from, the error condition.

An example of an alternative path would be: "The system recognizes a cookie on the user's machine", and "Go to step 4 (Main path)". An example of an exception path would be: "The system does not recognize a user's logon information", and "Go to step 1 (Main path)".

(NOTE) Many use case designers prefer to put the complete series of steps in an Alternate Flow rather than referring back (rejoin) to steps in the Primary or Happy path Flow. The reason for this preference is that Test Engineers may be receiving segments of a use case in order to design test cases. As such, they may not have the full picture of the sequences. Another reason of making the Alternate Flow stand on its own is that of reuse. In the area of error control and reporting, an Alternate Path may be identical in all aspects except for the error message. Being able to reuse the Alternate Flow can save significant design time. And finally, it is much easier to read and follow an Alternate Path when the steps are present rather than having to jump between the Primary Path and the Alternate Path. All this being said, although it is a preference to make the use case as readable as possible, there are few set rules on how to define the paths.

According to Anthony J H Simons and Ian Graham[*citation needed*] (who openly admits he got it wrong - using 2000 use cases at Swiss Bank), alternative paths were not originally part of use cases. Instead, each use case represented a single user's interaction with the system. In other words, each use case represented one possible path through the system. Multiple use cases would be needed before designs based on them could be made. In this sense, use cases are for exploration, not documentation. An Activity diagram can give an overview of the basic path and alternative path.

Postconditions

The *post-conditions* section describes what the change in state of the system will be after the use case completes. Post-conditions are guaranteed to be true when the use case ends.

Business rules

Business rules are written (or unwritten) rules or policies that determine how an organization conducts its business with regard to a use case. Business rules are a special kind of requirement. Business rules may be specific to a use case or apply across all the use cases, or across the entire business. Use cases should clearly reference business rules that are applicable and where they are implemented.

Business Rules should be encoded in-line with the Use Case logic and execution may lead to different post conditions. E.g. Rule2. that a cash withdraw will lead to an update of the account and a transaction log leads to a post condition on successful withdrawal - but only if Rule1 which says there must be sufficient funds tests as true.

Notes

Experience has shown that however well-designed a use case template is, the analyst will have some important information that does not fit under a specific heading. Therefore all good templates include a section (e.g. "Notes to Developers") that allows less-structured information to be recorded.

Author and date

This section should list when a version of the use case was created and who documented it. It should also list and date any versions of the use case from an earlier stage in the development which are still current documents. The author is traditionally listed at the bottom, because it is not considered to be essential information; use cases are intended to be collaborative endeavors and they should be jointly owned.

Limitations of use cases

Use cases have limitations:

- Use case flows are not well suited to easily capturing non-interaction based requirements of a system (such as algorithm or mathematical requirements) or non-functional requirements (such as platform, performance, timing, or safety-critical aspects). These are better specified declaratively elsewhere.
- Use case templates do not automatically ensure clarity. Clarity depends on the skill of the writer(s).
- There is a learning curve involved in interpreting use cases correctly, for both end users and developers. As there are no fully standard definitions of use cases, each group must gradually evolve its own interpretation. Some of the relations, such as *extends*, are ambiguous in interpretation and can be difficult for stakeholders to understand.
- Proponents of Extreme Programming often consider use cases too needlessly document-centric, preferring to use the simpler approach of a user story.
- Use case developers often find it difficult to determine the level of user interface (UI) dependency to incorporate in a use case. While use case theory suggests that UI not be reflected in use cases, it can be awkward to abstract out this aspect of design, as it makes the use cases difficult to visualize. Within software engineering, this difficulty is resolved by applying requirements traceability through the use of a traceability matrix.
- Use cases can be over-emphasized. In *Object Oriented Software Construction (2nd edition)*, Bertrand Meyer discusses issues such as driving the system design too literally from use cases and using use cases to the exclusion of other potentially valuable requirements analysis techniques.
- Use cases have received some interest as a starting point for test design.[6] Some use case literature, however, states that use case pre- and postconditions should apply to all scenarios of a use case (i.e., to all possible paths through a use case) which is limiting from a test design standpoint. If the postconditions of a use case are so general as to be valid for all possible use case scenarios, they are likely not to be useful as a basis for specifying expected behavior in test design. For example, the outputs and final state of a failed attempt to withdraw cash from an ATM are not the same as a successful withdrawal: if the postconditions reflect this, they too will differ; if the postconditions don't reflect this, then they can't be used to specify the expected behavior of tests. An alternative perspective on use case pre- and postconditions more suitable for test design based on model-based specification is discussed in [7].

See also

- Business case
- List of UML tools
- User story
- Use case diagram
- Misuse case

References

1. ^ Alistair Cockburn, "Use cases, ten years later"
2. ^ http://www.omg.org/docs/formal/07-02-03.pdf §16.3.1
3. ^ A. Cockburn (2001). *Writing Effective Use Cases*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN 0-201-70225-8.
4. ^ Use case modeling By Kurt Bittner, Ian Spence
5. ^ Liddle, Stephen. Stake holders for use case specifications.[1] 2009
6. ^ Frank Armour and Granville Miller (2000). *Advanced Use Case Modeling: Software Systems*. Addison-Wesley Professional. ISBN 0201615924.
7. ^ Richard Denney (2005). *Succeeding with Use Cases: Working Smart to Deliver Quality*. Addison-Wesley Professional. ISBN 0321316436.

Further reading

- Alexander I., Maiden N. Scenarios, Stories, Use Cases. Wiley 2004. ISBN 0470861940.
- Aurum A. Cox, K. and Jeffery. An experiment in inspecting the quality of usecase descriptions. Journal of Research and Practice in Information Technology, 36(4):211–229, 2004.
- Phalp K. Cox, K. and M. Shepperd. Comparing use-case writing guidelines. roc. Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'01), pages 101–112, 2001.
- Colom J.M. Ezpeleta, J. and J. Martinez. Comparing use-case writing guidelines. A Petri net based deadlock prevention policy for flexible manufacturing systems, 11(2):173–184, 1995.
- E. B. Fernandez and J. C. Hawkins. Determining role rights from use cases. In RBAC '97: Proceedings of the second ACM workshop on Role-based access control, pages 121–125, New York, NY, USA, 1997. ACM Press.
- R. Hurlbut. A survey of approaches for describing and formalizing use-cases. Technical Report 97– 03, Department of Computer Science, Illinois Institute of Technology, USA., 1997.
- Woo Jin Lee, Sung Deok Cha, and Yong Rae Kwon. Integration and analysis of use cases using modular petri nets in requirements engineering. IEEE Trans. Softw. Eng., 24(12):1115–1130, 1998.
- F. Torner, M. Ivarsson, F. Pettersson, and P. Ohman. Defects in automotive use cases. In ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering, pages 115–123, New York, NY, USA, 2006. ACM Presss.

External links

- Use case point calculator - Free online tool to estimate software development effort using use case points methodology
- EZEstimate - A Free tool to estimate software development effort using use case points methodology
- Understanding Use Case Modeling Basic introduction to use case modelling
- Precise Use Cases
- Estimating With Use Case Points This article describes the process to measure the size of an application modeled with UML, using use case points.
- Use Case Template This PDF file from Bredemeyer contains a useful template for Use Cases.
- Use Case Tutorials An introduction to use cases and use case-driven development.
- Use Cases (Usability.gov)
- Basic Use Case Template by Alistair Cockburn
- Getting Started With Use Cases
- "An Academic Survey on the Role of Use Cases in the UML"
- Practical Use Case Example
- User Stories vs. Use Cases Agile developer blog post

Retrieved from "http://en.wikipedia.org/wiki/Use_case"
Categories: Software project management | Software requirements | Unified Modeling Language | SysML