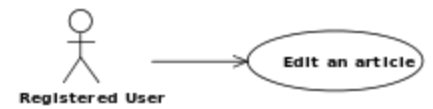


Use case

In software and systems engineering, a **use case** is a list of actions or event steps typically defining the interactions between a role (known in the Unified Modeling Language (UML) as an *actor*) and a system to achieve a goal. The actor can be a human or other external system. In 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 the Systems Modeling Language(SysML) or as contractual statements.



A very simple use case diagram of a Wiki system.

Use case analysis is an important and valuable requirement analysis technique that has been widely used in modern software engineering since its formal introduction by Ivar Jacobson in 1992. Use case-driven development is a key characteristic of many process models and frameworks such as ICONIX, the Unified Process (UP), the IBM Rational Unified Process (RUP), and the Oracle Unified Method (OUM). With its inherent iterative, incremental, and evolutionary nature, use case also fits well for agile development.

Contents

History

Templates

- Cockburn style
 - Design scopes
 - Goal levels
 - Fully dressed
 - Casual
- Fowler style

Actors

Business use case

Visual modeling

Examples

Advantages

Limitations

Misconceptions

Tools

See also

References

Further reading

External links

History

In 1986, Ivar Jacobson first formulated textual, structural, and visual modeling techniques for specifying use cases. In 1992 his co-authored book *Object-Oriented Software Engineering - A Use Case Driven Approach*^[1]helped to popularize the technique for capturing functional requirements, especially in software

development. Originally he had used the terms *usage scenarios* and *usage case* – the latter a direct translation of his Swedish term *användningsfall* – but found that neither of these terms sounded natural in English, and eventually he settled on *use case*.^[2]

Since then, other experts have also contributed a great deal to the technique, notably Alistair Cockburn, Larry Constantine, Dean Leffingwell, Kurt Bittner and Gunnar Overgaard.

In 2011, Jacobson published an update to his work, called *Use Case 2.0*,^[3] with the intention of incorporating many of his practical experiences of applying use cases since the original inception of the concept.^[4]

Templates



There are many ways to write a use case in text, from *use case brief*, *casual*, *outline*, to *fully dressed* etc., and with varied templates. Writing use cases in templates devised by various vendors or experts is a common industry practice to get high-quality functional system requirements.

Cockburn style

The template defined by Alistair Cockburn in his popular book *Writing Effective Use Cases* has been one of the most widely used writing styles of use cases.

Design scopes



Cockburn suggests annotating each use case with a symbol to show the "Design Scope", which may be black-box (internal detail is hidden) or white-box (internal detail is shown). Five symbols are available:^[5]

Scope	Icon	
Organization (black-box)	Filled House	
Organization (white-box)	Unfilled House	
System (black-box)	Filled Box	
System (white-box)	Unfilled Box	
Component	Screw or Bolt	

Other authors sometimes call use cases at Organization level "Business use cases".^[6]

Goal levels

Cockburn suggests annotating each use case with a symbol to show the "Goal Level";^[7] the preferred level is "User-goal" (or colloquially "sea level"^{[8]:101}).

Goal Level	Icon	Symbol	
Very High Summary	Cloud	++	
Summary	Flying Kite	+	
User Goal	Waves at Sea	!	
Subfunction	Fish	-	
Too Low	Seabed Clam-Shell	--	



Sometimes in text writing, a use-case name followed by an alternative text symbol (!, +, -, etc.) is a more concise and convenient way to denote levels, e.g. *place an order!*, *login-*.

Fully dressed

Cockburn describes a more detailed structure for a use case, but permits it to be simplified when less detail is needed. His fully dressed use case template lists the following fields:^[9]

- Title: "an active-verb goal phrase that names the goal of the primary actor"^[10]
- Primary Actor
- Goal in Context
- Scope
- Level
- Stakeholders and Interests
- Precondition
- Minimal Guarantees
- Success Guarantees
- Trigger
- Main Success Scenario
- Extensions
- Technology & Data Variations List

In addition, Cockburn suggests using two devices to indicate the nature of each use case: icons for design scope and goal level.

Cockburn's approach has influenced other authors; for example, Alexander and Beus-Dukic generalize Cockburn's "Fully dressed use case" template from software to systems of all kinds, with the following fields differing from Cockburn:^[11]

- Variation scenarios "(maybe branching off from and maybe returning to the main scenario)"
- Exceptions "i.e. exception events and their exception-handling scenarios"

Casual

Cockburn recognizes that projects may not always need detailed "fully dressed" use cases. He describes a Casual use case with the fields:^[9]

- Title (goal)
- Primary Actor
- Scope
- Level
- (Story): the body of the use case is simply a paragraph or two of text, informally describing what happens.

Fowler style

Martin Fowler states "There is no standard way to write the content of a use case, and different formats work well in different cases."^{[8]:100} He describes "a common style to use" as follows:^{[8]:101}

- Title: "goal the use case is trying to satisfy"^{[8]:101}
- Main Success Scenario: numbered list of steps^{[8]:101}
 - Step: "a simple statement of the interaction between the actor and a system"^{[8]:101}
- Extensions: separately numbered lists, one per Extension^{[8]:101}
 - Extension: "a condition that results in different interactions from .. the main success scenario". An extension from main step 3 is numbered 3a, etc.^{[8]:101}

The Fowler style can also be viewed as a simplified variant of the Cockburn template.

Actors

A use case defines the interactions between external actors and the system under consideration to accomplish a goal. Actors must be able to make decisions, but need not be human: "An actor might be a person, a company or organization, a computer program, or a computer system—hardware, software, or both."^[12] Actors are always stakeholders, but not all stakeholders are actors, since they "never interact directly with the system, even though they have the right to care how the system behaves."^[12] For example, "the owners of the system, the company's board of directors, and regulatory bodies such as the Internal Revenue Service and the Department of Insurance" could all be stakeholders but are unlikely to be actors.^[12]

Similarly, a person using a system may be represented as different actors because of playing different roles. For example, user "Joe" could be playing the role of a Customer when using an Automated Teller Machine to withdraw cash from his own account, or playing the role of a Bank Teller when using the system to restock the cash drawer on behalf of the bank.

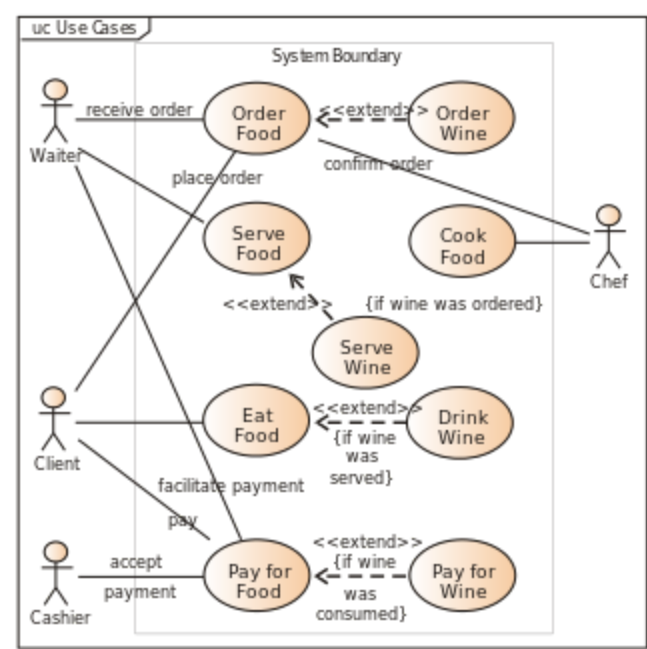
Actors are often working on behalf of someone else. Cockburn writes that "These days I write 'sales rep for the customer' or 'clerk for the marketing department' to capture that the user of the system is acting for someone else." This tells the project that the "user interface and security clearances" should be designed for the sales rep and clerk, but that the customer and marketing department are the roles concerned about the results.^[13]

A stakeholder may play both an active and an inactive role: for example, a Consumer is both a "mass-market purchaser" (not interacting with the system) and a User (an actor, actively interacting with the purchased product).^[14] In turn, a User is both a "normal operator" (an actor using the system for its intended purpose) and a "functional beneficiary" (a stakeholder who benefits from the use of the system).^[14] For example, when user "Joe" withdraws cash from his account, he is operating the Automated Teller Machine and obtaining a result on his own behalf.

Cockburn advises to look for actors among the stakeholders of a system, the primary and supporting (secondary) actors of a use case, the system under design (SuD) itself, and finally among the "internal actors", namely the components of the system under design.^[12]

Business use case

In the same way that a use case describes a series of events and interactions between a user (or other type of Actor) and a system, in order to produce a result of value (goal), a business use case describes the more general interaction between a business system and the users/actors of that system to produce business results of value. The primary difference is that the system considered in a business use case model may contain people in addition to technological systems. These "people in the system" are called business workers. In the example of a restaurant, a decision must be made whether to treat each person as an actor (thus outside the system) or a business worker (inside the system). If a waiter is considered an actor, as shown in the example below, then the restaurant system does not include the waiter, and the model exposes the interaction between the waiter and the restaurant. An alternative would be to consider the waiter as a part of the restaurant system (a business worker), while considering the client to be outside the system (an actor).^[15]



A business Use case diagram depicts a model of several *business use cases* (goals) which represents the interactions between a restaurant (the business system) and its primary stakeholders (*business actors* and *business workers*).

Visual modeling

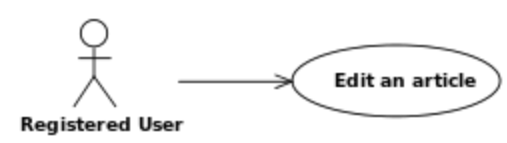
Use cases are not only texts, but also diagrams, if needed. In the Unified Modeling Language, the relationships between use cases and actors are represented in use case diagrams originally based upon Ivar Jacobson's Objectory notation. SysML uses the same notation at a system block level.

In addition, other behavioral UML diagrams such as activity diagrams, sequence diagrams, communication diagrams and state machine diagrams can also be used to visualize use cases accordingly. Specifically, a System Sequence Diagram (SSD) is a sequence diagram often used to show the interactions between the external actors and the system under design (SuD), usually for visualizing a particular scenario of a use case.

Use case analysis usually starts by drawing use case diagrams. For agile development, a requirement model of many UML diagrams depicting use cases plus some textual descriptions, notes or *use case briefs* would be very lightweight and just enough for small or easy project use. As good complements to use case texts, the visual diagram representations of use cases are also effective facilitating tools for the better understanding, communication and design of complex system behavioral requirements.

Examples

Below is a sample use case written with a slightly-modified version of the Cockburn-style template. Note that there are no buttons, controls, forms, or any other UI elements and operations in the basic use case description, where only user goals, subgoals or intentions are expressed in every step of the basic flow or extensions. This practice makes the requirement specification clearer, and maximizes the flexibility of the design and implementations.



Use Case: Edit an article

Primary Actor: Member (*Registered User*)

Scope: a Wiki system

Level: ! (*User goal or sea level*)

Brief: (*equivalent to a user story or an epic*)

The member edits any part (the entire article or just a section) of an article he/she is reading. Preview and changes comparison are allowed during the editing.

Stakeholders

...

Postconditions

Minimal Guarantees:

Success Guarantees:

- The article is saved and an updated view is shown.
- An edit record for the article is created by the system, so watchers of the article can be informed of the update later.

Preconditions:

The article with editing enabled is presented to the member.

Triggers:

The member invokes an edit request (for the full article or just one section) on the article.

Basic flow:

1. The system provides a new editor area/box filled with all the article's relevant content with an informative edit summary for the member to edit. If the member just wants to edit a section of the article, only the original

- content of the section is shown, with the section title automatically filled out in the edit summary.
2. The member modifies the article's content till satisfied.
 3. The member fills out the edit summary, tells the system if he/she wants to watch this article, and submits the edit.
 4. The system saves the article, logs the edit event and finishes any necessary post processing.
 5. The system presents the updated view of the article to the member.

Extensions:

2-3.

- a. Show preview:
 1. The member selects *Show preview* which submits the modified content.
 2. The system reruns step 1 with addition of the rendered updated content for preview, and informs the member that his/her edits have not been saved yet, then continues.
- b. Show changes:
 1. The member selects *Show changes* which submits the modified content.
 2. The system reruns step 1 with addition of showing the results of comparing the differences between the current edits by the member and the most recent saved version of the article, then continues.
- c. Cancel the edit:
 1. The member selects *Cancel*.
 2. The system discards any change the member has made, then goes to step 5.

4a. Timeout:

...

Advantages

Since the inception of the agile movement, the user story technique from Extreme Programming has been so popular that many think it is the only and best solution for agile requirements of all projects. Alistair Cockburn lists five reasons why he still writes use cases in agile development.^[16]

1. The list of goal names provides the *shortest* summary of what the system will offer (even than user stories). It also provides a project planning skeleton, to be used to build initial priorities, estimates, team allocation and timing.
2. The main success scenario of each use case provides everyone involved with an agreement as to what the system will basically do and what it will not do. It provides the context for each specific line item requirement (e.g. fine-grained user stories), a context that is very hard to get anywhere else.
3. The extension conditions of each use case provide a framework for investigating all the little, niggling things that somehow take up 80% of the development time and budget. It provides a look ahead mechanism, so the stakeholders can spot issues that are likely to take a long time to get answers for. These issues can and should then be put ahead of the schedule, so that the answers can be ready when the development team gets around to working on them.
4. The use case extension scenario fragments provide answers to the many detailed, often tricky and ignored business questions: "What are we supposed to do in this case?" It is a thinking/documentation framework that matches the if...then...else statement that helps the programmers think through issues. Except it is done at investigation time, not programming time.
5. The full use case set shows that the investigators have thought through every user's needs, every goal they have with respect to the system, and every business variant involved.

In summary, specifying system requirements in use cases has these apparent benefits comparing with traditional or other approaches:

User focused

Use cases constitute a powerful, user-centric tool for the software requirements specification process.^[17] Use case modeling typically starts from identifying key stakeholder roles (*actors*) interacting with the system, and their goals or objectives the system must fulfill (an outside perspective). These user goals then become the ideal candidates for the names or titles of the use cases which represent the desired functional features or services provided by the system. This user-centered approach ensure that what has the real business value and the user really want is developed, not those trivial functions speculated from a developer or system (inside) perspective.

Use case authoring has been an important and valuable analysis tool in the domain of User-Centered Design (UCD) for years.

Better communication

Use cases are often written in natural languages with structured templates. This narrative textual form (legible requirement stories), understandable by almost everyone, complemented by visual UML diagrams foster better and deeper communications among all stakeholders, including customers, end-users, developers, testers and managers. Better communications result in quality requirements and thus quality systems delivered.

Quality requirements by structured exploration

One of the most powerful things about use cases resides in the formats of the use case templates, especially the main success scenario (basic flow) and the extension scenario fragments (extensions, exceptional and/or alternative flows). Analyzing a use case step by step from preconditions to postconditions, exploring and investigating every action step of the use case flows, from basic to extensions, to identify those tricky, normally hidden and ignored, seemingly trivial but realistically often costly requirements (as Cockburn mentioned above), is a structured and beneficial way to get clear, stable and quality requirements systematically.

Minimizing and optimizing the action steps of a use case to achieve the user goal also contribute to a better interaction design and user experience of the system.

Facilitate testing and user documentation

With content based upon an action or event flow structure, a model of well-written use cases also serves as an excellent groundwork and valuable guidelines for the design of test cases and user manuals of the system or product, which is an effort-worthy investment up-front. There is obvious connections between the flow paths of a use case and its test cases. Deriving functional test cases from a use case through its scenarios (running instances of a use case) is straightforward.^[18]

Limitations

Limitations of use cases include:

- Use cases are not well suited to 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.
- As there are no fully standard definitions of use cases, each project must form its own interpretation.
- Some use case relationships, such as *extends*, are ambiguous in interpretation and can be difficult for stakeholders to understand as pointed out by Cockburn (Problem #6)^[19]
- 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. In software engineering, this difficulty is resolved by applying requirements traceability, for example with a traceability

matrix. Another approach to associate UI elements with use cases, is to attach a UI design to each step in the use case. This is called a use case storyboard.

- Use cases can be over-emphasized. Bertrand Meyer discusses issues such as driving system design too literally from use cases, and using use cases to the exclusion of other potentially valuable requirements analysis techniques.^[20]
- Use cases are a starting point for test design,^[21] but since each test needs its own success criteria, use cases may need to be modified to provide separate post-conditions for each path.^[22]
- Though use cases include goals and contexts, whether these goals and motivations behind the goals (stakeholders concerns and their assessments including non-interaction) conflict or negatively/positively affect other system goals are subject of goal oriented requirement modelling techniques (such as BMM, I*, KAOS and ArchiMate ARMOR).

Misconceptions

Common misunderstandings about use cases are:

User stories are agile; use cases are not.

Agile and Scrum are neutral on requirement techniques. As the Scrum Primer^[23] states,

Product Backlog items are articulated in any way that is clear and sustainable. Contrary to popular misunderstanding, the Product Backlog does not contain "user stories"; it simply contains items. Those items can be expressed as user stories, use cases, or any other requirements approach that the group finds useful. But whatever the approach, most items should focus on delivering value to customers.

Use cases are mainly diagrams.

Craig Larman stresses that "use cases are not diagrams, they are text".^[24]

Use cases have too much UI-related content.

As some put it,

Use cases will often contain a level of detail (i.e. naming of labels and buttons) which make it not well suited for capturing the requirements for a new system from scratch.

Novice misunderstandings. Each step of a well-written use case should present *actor* goals or intentions (the essence of functional requirements), and normally it should not contain any user interface details, e.g. naming of labels and buttons, UI operations etc., which is a *bad* practice and will unnecessarily complicate the use case writing and limit its implementation.

As for capturing requirements for a new system from scratch, *use case diagrams* plus *use case briefs* are often used as handy and valuable tools, at least as lightweight as user stories.

Writing use cases for large systems is tedious and a waste of time.

As some put it,

The format of the use case makes it difficult to describe a large system (e.g. CRM system) in less than several hundred pages. It is time consuming and you will find yourself spending time doing an unnecessary amount of rework.

Spending much time in writing tedious use cases which add no or little value and result in a lot of rework is a *bad smell* indicating that the writers are not well skilled and have little knowledge of how to write quality use cases both efficiently and effectively. Use cases should be authored in an iterative, incremental and evolutionary (*agile*) way. Applying use case templates does not mean that all the fields of a use case template should be used and filled out comprehensively from up-front or during a special dedicated stage, i.e. the requirement phase in the traditional *waterfall* development model.

In fact, the use case formats formulated by those popular template styles, e.g. the RUP's and the Cockburn's (also adopted by the OUM method) etc., have been proved in practice as valuable and helpful tools for capturing, analyzing and documenting complex requirements of large systems. The quality of a good use case documentation (*model*) should not be judged largely or only by its size. It is possible as well that a quality and comprehensive use case model of a large system may finally evolve into hundreds of pages mainly because of the inherent complexity of the *problem* in hand, not because of the poor writing skills of its authors.

Tools

Text editors and/or word processors with template support are often used to write use cases. For large and complex system requirements, dedicated use case tools are helpful.

Some of the well-known use case tools include:

- CaseComplete
- Enterprise Architect
- MagicDraw
- Rational Software's RequisitePro - one of the early, well-known use case and requirement management tools in the 1990s.
- Wiki software - good tools for teams to author and manage use cases collaboratively.

Most UML tools support both the text writing and visual modeling of use cases.

See also

- Abuse case
- Business case
- Event partitioning
- Feature
- List of UML tools
- Misuse case
- Requirement
- Requirements elicitation
- Scenario
- Storyboard
- Test Case
- Use Case Points
- Entity-control-boundary

References

1. Jacobson Ivar, Christerson Magnus, Jonsson Patrik, Övergaard Gunnar, *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992.
2. "Alistair Cockburn, "Use cases, ten years later"". Alistair.cockburn.us. 2002. Retrieved 17 April 2013.
3. Jacobson, Ivar; Spence, Ian; Bittner, Kurt (December 2011). "Use Case 2.0: The Guide to Succeeding with Use Cases". Ivar Jacobson International. Retrieved 5 May 2014.
4. "Business Analysis Conference Europe 2011 - 26-28 September 2011, London, UK". Irmuk.co.uk. Retrieved 17 April 2013.
5. Cockburn, 2001. Inside front cover. Icons "Design Scope".
6. Suzanne Robertson. *Scenarios in Requirements Discovery*. Chapter 3 in Alexander and Maiden, 2004. Pages 39-59.
7. Cockburn, 2001. Inside front cover. Icons "Goal Level".
8. Fowler, 2004.
9. Cockburn, 2001. Page 120.
10. Cockburn, 2001. Inside rear cover. Field "Use Case Title".
11. Alexander and Beus-Dukic, 2009. Page 121
12. Cockburn, 2001. Page 53.
13. Cockburn, 2001. Page 55.
14. Alexander and Beus-Dukic, 2009. Page 39.
15. Eriksson, Hans-Erik (2000). *Business Modeling with UML*. New York: Wiley Computer Publishing. pp. 52. ISBN 0-471-29551-5.
16. Cockburn, Alistair (9 January 2008). "Why I still use use cases". *alistair.cockburn.us*.
17. Karl Wiegers (March 1997). "Listening to the Customer's Voice". *Process Impact*. Software Development.
18. Peter Zielczynski (May 2006). "Traceability from Use Cases to Test Cases". IBM developerWorks.
19. "Alistair.Cockburn.us - Structuring use cases with goals". *alistair.cockburn.us*. Retrieved 16 March 2018.
20. Meyer, 2000. (page needed)
21. Armour and Miller, 2000. (page needed)
22. Denney, 2005. (page needed)
23. Pete Deemer; Gabrielle Benefield; Craig Larman; Bas Vodde (17 December 2012). "The Scrum Primer: A Lightweight Guide to the Theory and Practice of Scrum (Version 2.0)". InfoQ.
24. Larman, Craig. *Applying UML and patterns*. Prentice Hall. pp. 63–64. ISBN 0-13-148906-2.

Further reading

- Alexander, Ian, and Beus-Dukic, Ljerka. *Discovering Requirements: How to Specify Products and Services*. Wiley, 2009.
- Alexander, Ian, and Maiden, Neil. *Scenarios, Stories, Use Cases*. Wiley 2004.
- Armour, Frank, and Granville Miller. *Advanced Use Case Modeling: Software Systems*. Addison-Wesley, 2000.
- Kurt Bittner, Ian Spence, *Use Case Modeling*, Addison-Wesley Professional, Aug 20, 2002.
- Cockburn, Alistair. *Writing Effective Use Cases*. Addison-Wesley, 2001.
- Larry Constantine, Lucy Lockwood, *Software for Use: A Practical Guide to the Essential Models and Methods of Usage-Centered Design*, Addison-Wesley, 1999.
- Denney, Richard. *Succeeding with Use Cases: Working Smart to Deliver Quality*. Addison-Wesley, 2005.
- Fowler, Martin. *UML Distilled (Third Edition)*. Addison-Wesley, 2004.
- Jacobson Ivar, Christerson M., Jonsson P., Övergaard G., *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992.
- Jacobson Ivar, Spence I., Bittner K. *Use Case 2.0: The Guide to Succeeding with Use Cases*, IJI SA, 2011.

- Dean Leffingwell, Don Widrig, *Managing Software Requirements: A Use Case Approach*, Addison-Wesley Professional. Dec 7, 2012.
- Kulak, Daryl, and Eamonn Guiney. *Use cases: requirements in context*. Addison-Wesley, 2012.
- Meyer, Bertrand. *Object Oriented Software Construction. (2nd edition)*. Prentice Hall, 2000.
- Schneider, Geri and Winters, Jason P. *Applying Use Cases 2nd Edition: A Practical Guide*. Addison-Wesley, 2001.
- Wazlawick, Raul S. *Object-Oriented Analysis and Design for Information Systems: Modeling with UML, OCL, and IFML*. Morgan Kaufmann, 2014.

External links

- [Alistair Cockburn's use case column](#)
- [Use Cases \(Usability.gov\)](#)
- [Basic Use Case Template by Alistair Cockburn](#)
- [Application of use cases for stakeholder analysis "Project Icarus: Stakeholder Scenarios for an Interstellar Exploration Program", JBIS, 64, 224-233](#)
- ["An Academic Survey on the Role of Use Cases in the UML"](#)
- [Search use case in IBM developerWorks](#)