

In Praise of Use Cases

Ian O'Neill

School of Electronics, Electrical Engineering and Computer Science,
Queen's University Belfast, BT7 1NN, N. Ireland
i.oneill@qub.ac.uk

Abstract. In this paper I argue that student software developers should be concerned not just with tersely expressed requirements ('user stories'), but with more involved 'use cases'. By encouraging developers to consider system behavior in some detail, the use-case approach lays the foundations for a coherent analysis of the object-based components that realize those behaviors. Though not objects in themselves, use cases encourage an object-oriented mindset. I consider some of the pitfalls that students must avoid when they are working with use cases, and propose an automated tutorial to help tackle some recurring difficulties.

Keywords: use cases, UML, educational software.

1 Introduction

In this paper, I examine some common problems associated with use cases in the classroom, while reminding educators of the advantages of including use cases in the software engineering (SE) process. I draw on my work teaching SE to groups of 200+ second-year students on undergraduate computing courses that last three to five years. For seasoned educators, some of my comments may simply confirm their own experiences. Others will find reminders of difficulties to avoid, and may be encouraged to adopt online tutorials, like the one outlined here, as a means of reinforcing key points. Sometimes the challenge of use cases is in the UML notation. Sometimes it involves the very meaning of the term 'use case'. I begin with the latter.

2 Why use cases (still)? – And how they differ from user stories

Booch, Rumbaugh and Jacobson offer a definition that every student developer on a use case driven project should learn by heart: "A *use case* is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor." [1]. A common problem among student developers is the tendency to reduce the use case to a single button click or even an on-screen feature: the resulting use case diagram can resemble a decision diagram, where each use case is a step in a sequence, rather than a sequence in its own right. One likely source of confusion is the similarity (not least in the name!) between *use cases* on the one hand and, on the other, the *user stories* favored by agile methodologies, 'Agile' has gained a

prominence that it did not have when Jacobson’s influential text on use case driven software engineering [2] first appeared 25 years ago, so that now undergraduate students may learn about user stories before use cases. Tersely expressed user stories conceal different degrees of complexity. At one end of the scale are *epics* (e.g. “As a user, I can back up my entire hard drive[, so that...]” [3]) that must be decomposed into smaller, more workable user stories before coding begins. At the other extreme, the user story can indeed represent a ‘single-click’ operation (“As an estimator, I want to see the item we’re estimating, so that I know what I’m giving an estimate for.” [4]). By contrast, much of the value of use cases, as part of an object-oriented (OO) development process, is that they split a development problem into substantial ‘chunks’ of behavior, that encourage the developer to work in an ‘object-friendly’ way. Analyzing the written description of the flows in a use case helps the developer identify suites of collaborating objects that realize the behavior the customer needs. Jacobson has recognized the need to take agile’s user stories into account [5] [6]: in his *Use-Case 2.0* approach, use cases are divided into *use case slices* that correspond to one or more *stories* (similar to the *user stories* of Scrum, etc.). Use cases and user stories are not incompatible: student developers just have to understand, and use, the differences to their advantage.

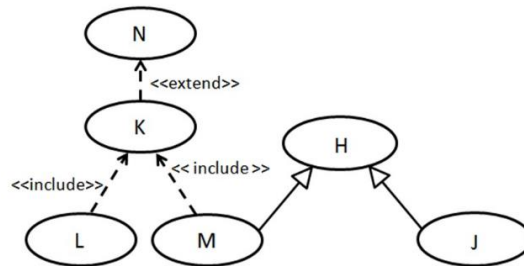


Fig. 1. A simple UML use case diagram

3 Getting use case diagrams right

The UML offers the student some flexibility in the way he or she describes a flow of events within a use case [7] (e.g. bulleted descriptions of normal and alternative flows, pre- and post-conditions, identification of extension points), but it specifically defines a simple diagrammatic notation for representing relationships between the use cases. The intention is that colleagues and clients should be able to interpret the diagram at a glance. However, the three relationships must be used carefully (definitions of <<extend>> and <<include>> especially have been a recurring subject of debate – e.g. [8]):

1. Generalization, represented by an open-headed arrow with a solid-line tail. In this relationship the more general use case is at the arrowhead, while the more specialized use case is at the tail (M and J are specializations of H in Figure 1: where H is allowed, M or J can be used). Using the correct arrow, and positioning the correct use cases at the head and tail, are important details for the student to get right.
2. <<include>>. According to Booch, Rumbaugh and Jacobson [9], “an include relationship between use cases means that the base use case explicitly incorporates the

behavior of another use case at a location specified in the base. The included use case never stands alone, but is only instantiated as part of some larger base that includes it.” This still leaves a degree of ambiguity, for it is not made explicit whether the ‘location specified in the base’ must be in the main flow of the base use case or if it may be in an alternative flow. Often trainers give their students less room for maneuver in the use of <<include>>. For example, in its blog, Requirements Inc. [10] maintains “A includes B [...] – A cannot produce success outcome without running B”. See also Dennis, Wixom and Tegarden’s [11] association of <<include>> with a use case’s normal flow. In explaining the <<include>> relationship to my students, I have favored the more prescriptive approach, so that from the include relationship between L and K shown in Figure 1, it is understood that Use Case L, if it executes normally, always includes Use Case K in its main (or normal) flow. This makes it easier to distinguish <<include>> from <<extend>> which follows.

3. Unlike <<include>>, then, <<extend>> represents optional or conditional behaviour: “The base use case may stand alone, but under certain conditions its behavior may be extended by the behavior of another use case.” [12]. (Some students will struggle initially to appreciate that this is not the same ‘extend’ as the reserved word used to signal *inheritance* in OO languages like Java.) Like <<include>>, the <<extend>> relationship needs to be read in the direction of the dashed stick-arrow, where the use case that provides the ‘extending’ or conditional behaviour is at the tail and the use case that is ‘being extended’ is at the head of the arrow. Thus, in Figure 1, the K-N relationship reads: ‘Use Case K extends Use Case N’. Very often in their diagrams, students place the <<extend>> arrow the wrong way round, confused by the fact that if ‘K extends N’, the functionality of K will be accessed from an ‘extension point’ in a flow of events in N (i.e., put simply, N ‘invokes’ K).

4 Educational software (and some Related Work) to the rescue

For the last two years, in the lead-in to their end-of-year exam, I have provided students with a short, adaptive, on-line ‘use case tutorial’. Both exam and tutorial are written in Questionmark, a widely-used commercial software package for implementing and scheduling multiple-choice/multiple-response assessments [13]. Other multiple-choice/multiple response software is available, sometimes as freeware (e.g. [14]); EduSymp attendees will be familiar with e-learning issues more generally (e.g. [15]). However, this use case tutorial specifically exploits the often neglected, but powerful, conditional ‘jump blocks’ in Questionmark, which can be used to match the system’s follow-up questions to a specific combination of student responses. Drawing on my research into object-based natural language dialogue systems [16], I have used these jump blocks extensively to give the student-system interaction a novel, ‘conversational’ twist. The interaction resembles a branching, frame-based natural language dialogue [17], where a ‘frame’ or ‘set’ of responses determines the next step in the conversation. In a separate article (to appear in *ACM Inroads*) I detail the dialogue management strategy. In this particular case, the student comments on a use case diagram (the example

in Figure 1) and the system responds. Figure 2 shows the system's reaction to an incomplete and inaccurate set of student responses – given this response combination, the system will give feedback and then simply prompt the student to try again. If the student's response combinations indicate that he or she is having difficulty in a particular area (terminology, or interpretation of the symbols), the system will quiz the student on the problem area alone, providing feedback on each new set of responses, and moving on only when the responses are sufficiently accurate. The tutorial concludes when the student gives a good, broad-ranging set of responses. Tenacious but helpful in its comments, the system evokes the small-scale, person-to-person tutorials of earlier years.

It uses the United Markup Language.
 L and M always include the behaviour of K.
 Each oval is called a user case.
 It's in the Unified Modeling Language.
 Each oval represents a use case.
 It's an example of a ULM diagram.
 In certain circumstances N involves the behaviour of K.
 It's written in UML.
 Sometimes - for example, depending on a selection made by the user - K makes use of the behaviour of N.
 H is a combination of the behaviour of both M and J.
 K always makes use of the behaviour of L and M.
 M and J are specialised versions of H.

You didn't spot all the correct examples of how the language and its symbols are described. Check your answers carefully.
 You didn't spot all the details concerning the directions of the arrows and their significance. Check those details again!
 Each oval is called a 'use case' (NOT a 'user case').
 Good, you knew that if K extends N, then, under certain circumstances, N must incorporate the behaviour of K.
 In this diagram **L and M always make use of the behaviour of K** - NOT the other way round! Again, always remember to read the <<include>> relationship in the direction of the arrow.

Fig. 2. What can you tell me about Figure 1? The student responds and system corrects.

Student feedback on the automated tutorial has been very encouraging. Comments taken from the surveys (also implemented in Questionmark) that accompany the tutorial include: "I thought that the way that you could repeat the questions again if you didn't pass was good, and I thought that the personalized feedback was great and really helpful"; "It was good, told me what I didn't understand and confirmed what I already knew"; "I thought it explained it quite well, and wasn't too critical on my wrong answers, which I liked a lot". The main requests from the surveys are for longer tutorials, and tutorials on other subjects. Having trialed the short use case tutorial as a revision aid, I hope to introduce it earlier in the academic year, where it will be a ready reminder of some key points as students embark on a substantial use case driven SE project.

5 Concluding remarks

On traditional OOSE pathways, when students write their 'final-year dissertation', they are expected to describe their software solution and justify their approach. A use case

model gives clear structure to the required behavior and so to the development process – provided the student, with the educator’s help, has learnt to avoid common conceptual and notational pitfalls associated with use cases. Automated tutorials can support that learning process. In software engineering, use cases encourage developers and clients to think beyond isolated actions and consider the system in the round, from an early stage of the development process. Use cases may have a similar effect in life more generally! At a talk I attended in London in the 1990s (sadly I do not have a more detailed reference), Jacobson told how use cases had even influenced the layout of his kitchen. So, if *Make toast* is a use case, realize it properly: make sure that your bread bin, toaster, cutlery drawer and fridge with the butter in it are all beside each other!

References

1. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. 2nd edn, p. 228. Addison-Wesley, Upper Saddle River (2005).
2. Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G.: Object-Oriented Software Engineering – A Use Case Driven Approach. ACM Press/Addison-Wesley, Wokingham (1992).
3. Mountain Goat Software, <https://www.mountaingoatsoftware.com/agile/user-stories#section-3>, last accessed 2017/06/27.
4. Mountain Goat Software, <https://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template>, last accessed 2017/06/27.
5. Jacobson, I., Spence, I., Bittner, K.: Use-Case 2.0. <https://www.ivarjacobson.com/publications/white-papers/use-case-ebook>, last accessed 2017/08/07
6. Jacobson, I., Spence, I., Kerr, B.: Use Case 2.0 – The Hub of Software Development. ACM Queue, 4 (1), 94–122 (2016).
7. Booch, G., Rumbaugh, J., Jacobson, I.: *ibid.*, p. 230.
8. Laguna, M.A., Marqués, J.M.: On the Multiplicity Semantics of the Extend Relationship in Use Case Models. In: Cordeiro, J., Shishkov, B., Ranchordas, A., Helfert, M. (eds.) Software and Data Technologies, Third International Conference, ICSoft 2008, pp. 62–75, Springer, Heidelberg (2009).
9. Booch, G., Rumbaugh, J., Jacobson, I.: *ibid.*, p. 233.
10. Requirements Inc Blog, <http://requirementsinc.com/uml-basics-include-and-extend-stereotypes-in-use-cases/>, last accessed 2017/06/27.
11. Dennis, A., Wixom, B.H., Tegarden, D.: System Analysis & Design – An Object-Oriented Approach with UML. 5th edn. Wiley, Hoboken, N.J. (2015).
12. Booch, G., Rumbaugh, J., Jacobson, I.: *ibid.*, p. 234.
13. Questionmark Homepage, <https://www.questionmark.com/>, last accessed 2017/06/27.
14. Moodle, <https://www.moodle.org/>, last accessed 2017/08/08.
15. Stikkolorum, D.R., Demuth, B., Zaytsev, V., Boulanger, F., Gray, J.: The MOOC Hype: Can We Ignore It? Reflections on the Current Use of Massive Open Online Courses in Software Modeling Education. In: MODELS Educators Symposium, EduSymp 2014, pp. 75–86, <http://ceur-ws.org/Vol-1346>, last accessed 2017/08/07.
16. O’Neill, I., Hanna, P., Liu, X., Greer, D., McTear, M.F.: Implementing advanced spoken dialogue management in Java. Science of Computer Programming, 54 (1), 99–124. (2005).
17. McTear, M.F.: Spoken Dialogue Technology: Toward the Conversational User Interface. Springer-Verlag, London (2004).