Agile Modeling

System Use Cases: An Agile Introduction

A use case is a sequence of actions that provide a measurable value to an actor. Another way to look at it is a use case describes a way in which a real-world actor interacts with the system. In a system use case you include high-level implementation decisions. System use cases can be written in both an informal manner and a formal manner. Techniques for identifying use cases are discussed as well as how to remain agile when writing use cases.

Informal System Use Cases

Let's start by considering the types of use cases that you'll write as part of your initial requirements modeling efforts during "the Inception phase" of your projects. These use cases will either be essential use cases or "informal" system use cases, a detailed example of which is presented in Figure I-1. As you can see the steps are written in very brief, bullet/point-form style. They contain just enough information to get the idea across and no more. Note that this version takes technology issues into account, for example the text "Student inputs her name and address" implies some sort of information system. The reference to the system also implies the same.

Figure I-1. *Enroll in seminar* as an informal system use case (automated solution).

Name: Enroll in Seminar

Identifier: UC 17

Basic Course of Action:

- Student inputs her name and student number
- System verifies the student is eligible to enroll in seminars. If not eligible then the student is informed and use case ends.
- System displays list of available seminars.
- Student chooses a seminar or decides not to enroll at all.
- System validates the student is eligible to enroll in the chosen seminar. If not eligible, the student is asked to choose another.
- System validates the seminar fits into the student's schedule.
- System calculates and displays fees
- Student verifies the cost and either indicates she wants to enroll or not.
- System enrolls the student in the seminar and bills them for it.
- The system prints enrollment receipt.

Figure I-2 presents an alternate version, this time as a manual process involving a registrar (a person) instead of an automated system. Choosing a manual process over a software-based one is still a technical architecture decision, in this case a low-tech architectural decision. The differences between the two versions illuminates how system use cases are not analysis and arguably even design artifacts, not requirements artifacts.

Figure I-2. Enroll in Seminar as an informal use case (manual solution).

Name: Enroll in Seminar

Identifier: UC 17

Basic Course of Action:

- Student provides her name and student number on the enrollment form.
- Registrar verifies the student is eligible to enroll in seminars. If not eligible then the student is informed and use case ends.
- Registrar asks the student which seminar they'd like to enroll in. If they don't know, the registrar provides the student with course catalog if required.
- Student chooses a seminar or decides not to enroll at all.
- Registrar checks the student record to see if student has previously passed prerequisite courses. If not eligible the student is asked to choose another.
- Registrar validates the seminar fits into the student's schedule.
- Registrar calculates fees
- Student verifies the cost and either indicates she wants to enroll or not.
- Registrar enrolls the student in the seminar and bills them for it.
- The Registrar writes a payment receipt.

Figure I-3 provides yet another alternate form, in this case as a very high-level use case written on an index card. Very Agile teams start with this level of detail, captured during their initial high-level requirements modeling efforts.

Figure I-3. *Enroll in seminar* as a very high-level use case.

Enroll in Seminar

- Student chooses a seminar to enroll in
- System checks that the student can enroll in the seminar
- System calculates fees
- Student pays fees and is enrolled

Formal System Use Cases

Figure 1 presents a formalized version of Figure I-1. This version is much more detailed than the corresponding use case, and is typical of the type of use cases that people will write in documentation-intense environments. Frankly, use cases like this are overkill for many projects yet many project teams are required to write them in this manner (or something similar) because senior management is convinced that they require this level of documentation. My advice is to keep your models as simple as possible and only document them this thoroughly if it adds actual value.

A formalized system use case refers to specific user interface components-such as screens, HTML pages, or reports-something you wouldn't do in an essential/business use case. During analysis, you make decisions regarding what will be built, information reflected in your use cases, and, arguably, even how it will be built (effectively design). Because your use cases refer to user interface components, and because your user interface is worked on during design, inevitably design issues will creep into your use cases. For example, a design decision is whether your user interface is implemented using browser-based technology, such as HTML pages or graphical user interface (GUI) technology such as Windows. Because your user interface will work differently depending on the implementation technology, the logic of your system use cases, which reflect the flow of your user interface, will also be affected.

By referring to other artifacts, instead of embedding the information in your use cases, you reduce the chance that you're writing Use Cases of Mass Destruction.

Figure 1. Enroll in Seminar as a formal system use case.

Name: Enroll in Seminar

Identifier: UC 17

Description:

Enroll an existing student in a seminar for which she is eligible.

Preconditions:

The Student is registered at the University.

Postconditions:

The Student will be enrolled in the course she wants if she is eligible and room is available.

Basic Course of Action:

1. The use case begins when a student wants to enroll in a seminar.

2. The student inputs her name and student number into the system via *UI23 Security Login Screen*.

3. The system verifies the student is eligible to enroll in seminars at the university according to business rule *BR129 Determine Eligibility to Enroll*. [Alt Course A]

4. The system displays *UI32 Seminar Selection Screen*, which indicates the list of available seminars.

5. The student indicates the seminar in which she wants to enroll. [Alt Course B: The Student Decides Not to Enroll]

6. The system validates the student is eligible to enroll in the seminar according to the business rule *BR130 Determine Student Eligibility to Enroll in a Seminar*. [Alt Course C]

7. The system validates the seminar fits into the existing schedule of the student according to the business rule *BR143 Validate Student Seminar Schedule*.

8. The system calculates the fees for the seminar based on the fee published in the course catalog, applicable student fees, and applicable taxes. Apply business rules *BR 180 Calculate Student Fees* and *BR45 Calculate Taxes for Seminar*.

9. The system displays the fees via *UI33 Display Seminar Fees Screen*.

10. The system asks the student if she still wants to enroll in the seminar.

11. The student indicates she wants to enroll in the seminar.

12. The system enrolls the student in the seminar.

13. The system informs the student the enrollment was successful via *UI88 Seminar Enrollment Summary Screen*.

14. The system bills the student for the seminar, according to business rule *BR100 Bill Student for Seminar*.

15. The system asks the student if she wants a printed statement of the enrollment.

16. The student indicates she wants a printed statement.

17. The system prints the enrollment statement *UI89 Enrollment Summary Report*.

18. The use case ends when the student takes the printed statement.

Alternate Course A: The Student is Not Eligible to Enroll in Seminars.

A.3. The registrar determines the student is not eligible to enroll in seminars.

A.4. The registrar informs the student he is not eligible to enroll.

A.5. The use case ends.

Alternate Course B: The Student Decides Not to Enroll in an Available Seminar

B.5. The student views the list of seminars and does not see one in which he wants to enroll.

B.6. The use case ends.

Alternate Course C: The Student Does Not Have the Prerequisites

C.6. The registrar determines the student is not eligible to enroll in the seminar he chose.

C.7. The registrar informs the student he does not have the prerequisites.

C.8. The registrar informs the student of the prerequisites he needs.

C.9. The use case continues at Step 4 in the basic course of action.

Figure 2. *Enroll in University* as a formal system use case.

Name: Enroll in University

Identifier: UC 19

Description:

Enroll someone in the university.

Preconditions:

- The Registrar is logged into the system.
- The Applicant has already undergone initial checks to verify that they are eligible to enroll.

Postconditions:

- The Applicant will be enrolled in the university as a student if they are eligible.

Basic Course of Action:

1. An applicant wants to enroll in the university.

2. The applicant hands a filled out copy of form *UI13 University Application Form* to the registrar. [Alternate Course A: Forms Not Filled Out]

3. The registrar visually inspects the forms.

4. The registrar determines that the forms have been filled out properly. [Alternate Course B: Forms Improperly Filled Out].

5. The registrar clicks on the *Create Student* icon.

6. The system displays *UI89 Create Student Screen*.

7. The registrar inputs the name, address, and phone number of the applicant. [Extension Point: *UC34 Perform Security Check*. Applicable to Step 17]

8. The system determines that the applicant does not already exist within the system according to *BR37 Potential Match Criteria for New Students*. [Alternate Course F: Students Appears to Exist Within The System].

9. The system determines that the applicant is on the eligible applicants list. [Alternate Course G: Person is Not Eligible to Enroll]

10. The system adds the applicant to its records. The applicant is now considered to be a student.

11. The registrar helps the student to enroll in seminars via the use case *UC 17 Enroll in Seminar*.

12. The system calculates the required initial payment in accordance to *BR16 Calculate Enrollment Fees*.

13. The system displays *UI15 Fee Summary Screen*.

14. The registrar asks the student to pay the initial payment in accordance to BR19 Fee Payment Options.

15. The student pays the initial fee. [Alternate Course D: The Student Can't Pay At This Time]

16. The system prints a receipt.

17. The registrar hands the student the receipt.

18. The use case ends.

Alternate Course A: Forms Not Filled Out

A.2. The Applicant asks for a set of forms.

A.3. The Applicant fills out the forms as appropriate.

A.4. The use case continues at step 2 in the basic course of action.

Alternate Course B: and so on.

Figure 2 presents a formalized system use case (also called a traditional or concrete use case) for enrolling in the university. Interesting points about it:

- The system use case has many implementation details embedded within it. For example, it references "the term system" indicating a decision has been made to automate many of the mundane aspects of enrollment. The writer of system use cases is analyzing and describing requirements imposed by the problem, intermingled with implicit decisions about what the user interface is going to be like.
- The system use case makes references to screen and reports, for example, *UI23 Security Login Screen* and *UI89 Enrollment Summary Report*. Once again this reflects implementation details, someone has decided the system will be implemented as screens, as opposed to HTML pages perhaps, and printed reports.

- The use case references business rule definitions-such as *BR129 Determine Eligibility to Enroll* - because business rules reflect essential characteristics of your domain that your system must implement. For very simple systems without very many complex business rules I'll often keep it simple and document the rule within the use case. Different situations call for different approaches, hence the importance of AM's *Local Adaptation* principle.

- Each use case step reflects one activity and one activity only. Several advantages exist to this approach: the use case becomes easier to test because each statement is easier to understand and to validate; alternate courses are easier to write because it is easier to branch from a statement when it does one thing only.

- Use case steps are written in the active voice. For example, the statement "The registrar informs the student of the fees" is in active voice whereas "The student is informed of the fees by the registrar" is in passive voice. Writing in the active voice leads to succinct sentences.

- I like to end the basic course of action within a use case with a closing statement. This is often something along the lines of "The use case ends" or "The use case ends when . . .", indicating that the logic for the course of action has been completely defined.

- An alternate course of action is an infrequently used path of logic in a use case. Alternate courses are identified whenever there is an alternate way to work, an exception, or an error condition that must be handled. The use case text references several alternate courses, think of them simply as the use case way of doing if/then logic, one of which is described at the bottom of the use case.

Identifying Use Cases

How do you go about identifying potential use cases? Constantine and Lockwood (1999) suggest one way to identify essential use cases, or simply to identify use cases, is to identify potential services by asking your stakeholders the following questions from the point of view of the actors:

- What are users in this role trying to accomplish?
- To fulfill this role, what do users need to be able to do?
- What are the main tasks of users in this role?
- What information do users in this role need to examine, create, or change?
- What do users in this role need to be informed of by the system?
- What do users in this role need to inform the system about?

For example, from the point-of-view of the *Student* actor, you may discover that students:

- Enroll in, attend, drop, fail, and pass seminars.
- Need a list of available seminars.
- Need to determine basic information about a seminar, such as its description and its prerequisites.
- Obtain a copy of their transcript, their course schedules, and the fees due.
- Pay fees, pay late charges, receive reimbursements for dropped and cancelled courses, receive grants, and receive student loans.
- Graduate from school or drop out of it.
- Need to be informed of changes in seminars, including room changes, time changes, and even cancellations.

Remaining Agile

It is very easy for use case modeling to become un-agile. To prevent this from happening you need to focus on creating artifacts that are just barely good enough, they don't need to be perfect. I've seen too many projects go astray because people thought that the requirements had to be worded perfectly. You're not writing the Magna Carta!!!!!!! For example, in Figure 2 there are several imperfections, the alternate courses aren't labeled in order (D appears after F

and G) and the letters C and E aren't used (they were at some point in the past but then were dropped). The use case isn't perfect yet the world hasn't ended. Yes I could invest time to fix these issues but what would the value be? Nothing. Always remember AM's Maximize Stakeholder Investment principle and only do things that add value. Repeat after me: My use cases need to be just good enough. My use cases need to be just good enough. My use cases need to be just good enough. Why does this work? Because in an agile environment you'll quickly move to writing code based on those requirements, you'll discover that you don't fully understand what is required, you'll work closely with your stakeholder to do so, and you'll build something that meets their actual needs. It's software development, not documentation development.

Source

This artifact description is excerpted from Chapter 5 of The Object Primer 3rd Edition: Agile Model Driven Development with UML 2.

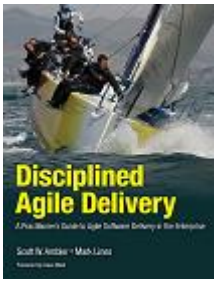
Translations

- Japanese

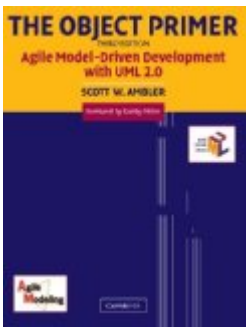
Let Us Help

We actively work with clients around the world to improve their information technology (IT) practices, typically in the role of mentor/coach, team lead, or trainer. A full description of what we do, and how to contact us, can be found at Scott Ambler + Associates.

Recommended Reading



This book, Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise describes the Disciplined Agile Delivery (DAD) process decision framework. The DAD framework is a people-first, learning-oriented hybrid agile approach to IT solution delivery. It has a risk-value delivery lifecycle, is goal-driven, is enterprise aware, and provides the foundation for scaling agile. This book is particularly important for anyone who wants to understand how agile works from end-to-end within an enterprise setting. Data professionals will find it interesting because it shows how agile modeling and agile database techniques fit into the overall solution delivery process. Enterprise professionals will find it interesting beause it explicitly promotes the idea that disciplined agile teams should be enterprise aware and therefore work closely with enterprise teams. Existing agile developers will find it interesting because it shows how to extend Scrum-based and Kanban-based strategies to provide a coherent, end-to-end streamlined delivery process.



The Object Primer 3rd Edition: Agile Model Driven Development with UML 2 is an important reference book for agile modelers, describing how to develop 35 types of agile models including all 13 UML 2 diagrams. Furthermore, this book describes the fundamental programming and testing techniques for successful agile solution delivery. The book also shows how to move from your agile models to source code, how to succeed at implementation techniques such as refactoring and test-driven development(TDD). The Object Primer also includes a chapter overviewing the critical database development techniques (database refactoring, object/relational mapping, legacy analysis, and database access coding) from my award-winning Agile Database Techniquesbook.