Reuse in Use-Case Models: <>, <>, and Inheritance

One of your goals during analysis is to identify potential opportunities for reuse, a goal you can work toward as you are developing your use-case model. Potential reuse can be model through four generalization relationships supported by UML use-case models:

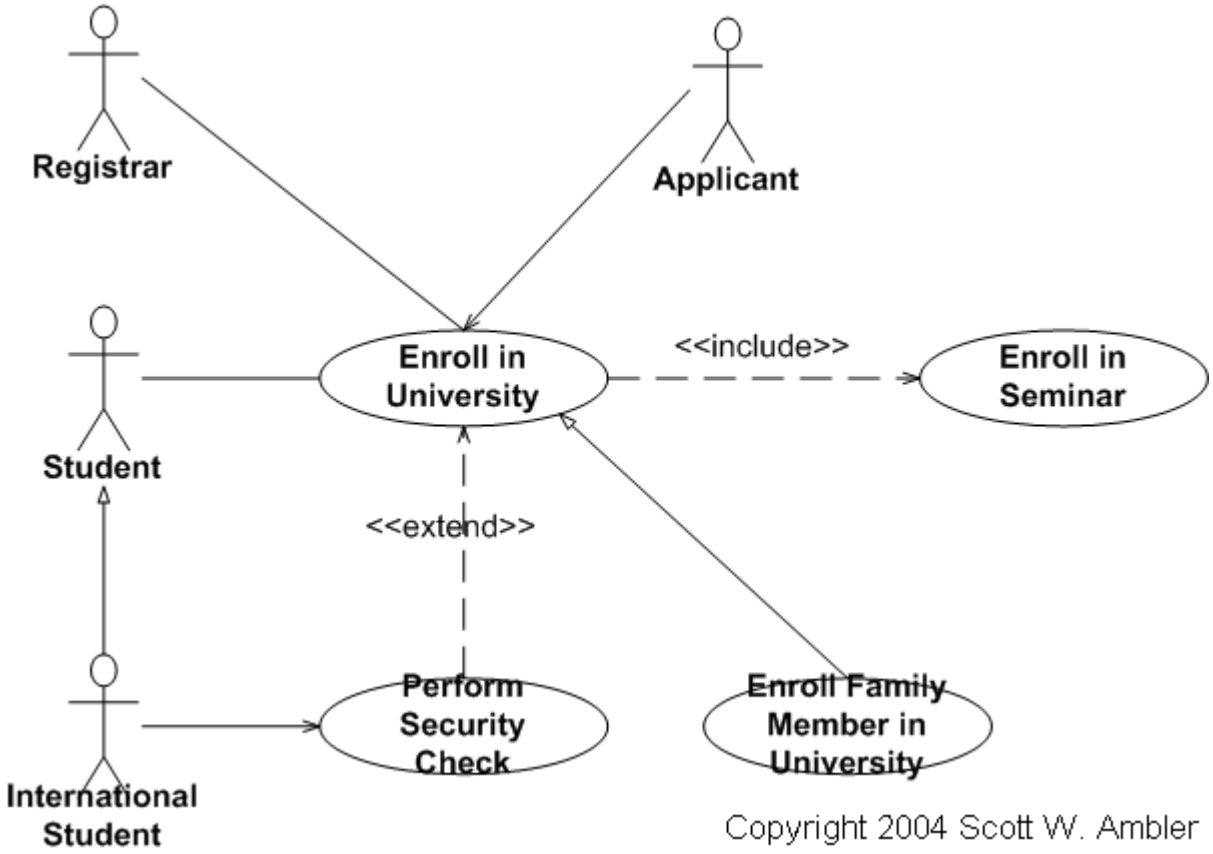
- Extend dependencies between use cases
- Include dependencies between use cases
- Inheritance between use cases
- Inheritance between actors

Extend Dependencies Between Use Cases

An extend dependency, formerly called an extends relationship in UML v1.2 and earlier, is a generalization relationship where an extending use case continues the behavior of a base use case. The extending use case accomplishes this by conceptually inserting additional action sequences into the base use-case sequence. This allows an extending use case to continue the activity sequence of a base use case when the appropriate extension point is reached in the base use case and the extension condition is fulfilled. When the extending use case activity sequence is completed, the base use case continues. In the use case diagram of Figure 1 , you see the use case "Perform" extends the use case "Enroll in University," the notation for doing so is simply a normal use-case dependency with the stereotype of <>. In this case, "Enroll in University" is the base use case and "Perform Security Check" is the extending use case.

An extending use case is, effectively, an alternate course of the base use case. In fact, a good rule of thumb is you should introduce an extending use case whenever the logic for an alternate course of action is at a complexity level similar to that of your basic course of action. I also like to introduce an extending use case whenever I need an alternate course for an alternate course; in this case, the extending use case would encapsulate both alternate courses. Many modelers avoid the use of extend dependencies as this technique has a tendency to make use-case diagrams difficult to understand. My preference is to use extend dependencies sparingly. Note, the extending use case¾in this case "Perform Security Check"would list "UC33 Enroll in University," the base use case, in its "Extends" list.

Figure 1. The opportunities for reuse in use-case models.



Just as you indicate the point at which the logic of an alternate course replaces the logic of a portion of the basic course of action for a use case, you need to be able to do the same thing for an extending use case. This is accomplished through the use of an extension point, which is simply a marker in the logic of a base use case indicating where extension is allowed. The red text in Figure 2 presents an example of how an extension point would be indicated in the basic course of action of the "Enroll In University" use case. Notice how the identifier and the name of the use case is indicated. If several use cases extended this one from the same point, then each one would need to be listed. A condition statement, such as "Condition: Enrollee is an international student," could have been indicated immediately following the name of the use case but, in this example, it was fairly obvious what was happening.

Figure 2. The Enroll in University Use Case (narrative style).

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: …..

Include Dependencies Between Use Cases

A second way to indicate potential reuse within use-case models exists in the form of include dependencies. An include dependency, formerly known as a uses relationship in UML v1.2 and earlier, is a generalization relationship denoting the inclusion of the behavior described by another use case. The best way to think of an include dependency is that it is the invocation of a use case by another one. In Figure 1 , you see the use case "Enroll in University" includes the use case "Enroll in Seminar," the notation for doing so is simply a normal use-case dependency with the stereotype of <>. The blue test in Figure 2 presents an example of how you would indicate where the use case is included in the logic of the including use case. Similar to calling a function or invoking an operation within source code, isn't it?

You use include dependencies whenever one use case needs the behavior of another. Introducing a new use case that encapsulates similar logic that occurs in several use cases is quite common. For example, you may discover several use cases need the behavior to search for and then update information about students, indicating the potential need for an "Update Student Record" use case included by the other use cases.

As you would expect, the use case "Enroll in University" should list "UC17 Enroll in Seminar" in its "Includes" list. Why should you bother maintaining an "Includes" and an "Extends" list in your use cases? The answer is simple: your use cases should stand on their own, you shouldn't expect people to have your use-case diagram in front of them. Yes, it would be nice if everyone has access to the use-case diagram because it also contains this information, but the reality is that sometimes you use different tools to document each part of your model. For example, your diagrams could be drawn using a drawing package and your use cases documented in a word processor. Some of your project stakeholders may have access to the word processor you are using, but not the drawing package. The main disadvantage of this approach is you need to maintain these two lists in parallel with the diagram, the danger being they may become unsynchronized.

Inheritance Between Use Cases

Use cases can inherit from other use cases, offering a third opportunity to indicate potential reuse. Figure 1 depicts an example of this, showing that "Enroll Family Member in University" inherits from the "Enroll In University" use case. Inheritance between use cases is not as common as either the use of extend or include dependencies, but it is still possible. The inheriting use case would completely replace one or more of the courses of action of the inherited use case. In this case, the basic course of action is completely rewritten to reflect that new business rules are applied when the family member of a professor is enrolling at the university. Family members are allowed to enroll in the school, regardless of the marks they earned in high school, they don't have to pay any enrollment fees, and they are given top priority for enrollment in the university.

Inheritance between use cases should be applied whenever a single condition, in this case, the student is a family member of a professor, would result in the definition of several alternate courses. Without the option to define an inheriting use case, you need to introduce an alternate course to rework the check of the student's high-school marks, the charging of enrollment feeds, and for prioritization of who is allowed to enroll in the given semester.

The inheriting use case is much simpler than the use case from which it inherits. It should have a name, description, and identifier, and it should also indicate from which use case it inherits in the "Inherits From" section. This includes any section that is replaced, particularly the pre-conditions and post-conditions as well as any courses of action. If something is not replaced, then leave that section blank, assuming it is inherited from the parent use case (you might want to put text, such as "see parent use case," in the section).

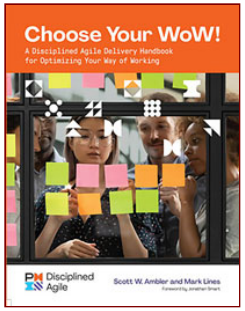
Inheritance Between Actors

The fourth opportunity for indicating potential reuse within use-case models occurs between actors: an actor on a use-case diagram can inherit from another actor. An example of this is shown in Figure 1 , the "International Student" actor inherits from "Student." An international student is a student, the only difference being is he or she is subject to different rules and policies (the international student pays more in tuition and is restricted from taking several seminars because of political reasons). The standard UML notation for inheritance, the open-headed arrow, is used and the advice presented about the appropriate use of inheritance still applies: it should make sense to say the inheriting actor is or is like the inherited actor.

Source

This article is excerpted from Chapter 5 of the book The Object Primer 3/e.

Recommended Reading



This book, Choose Your WoW! A Disciplined Agile Delivery Handbook for Optimizing Your Way of Working, is an indispensable guide for agile coaches and practitioners to identify what techniques - including practices, strategies, and lifecycles - are effective in certain situations and not as effective in others. This advice is based on proven experience from hundreds of organizations facing similar situations to yours. Every team is unique and faces a unique situation, therefore they must choose and evolve a way of working (WoW) that is effective for them. Choose Your WoW! describes how to do this effectively, whether they are just starting with agile/lean or if they're already following Scrum, Kanban, SAFe, LeSS, Nexus, or other methods.

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.
