Use Cases are more than System Operations

Rogardt Heldal

Chalmers University of Technology
Gothenburg, Sweden
heldal@cs.chalmers.se

Abstract. Correctly written use cases can be an important artifact for describing how a software system should behave. Use cases should be informal enough to permit anyone in a software project to understand them, in particular the customer (often lacking a formal background). One consequence of adopting use cases to, for example, MDA (Model Driven Architecture) can be an increasing level of formalism, which can severely limit understanding of use cases. Also, too few guidelines for how to write use cases make them both hard to write and understand. Finding the right level of formalism is the topic of this paper. We suggest a new way of writing the action steps of use cases by introducing *action blocks*. The introduction of action blocks makes use cases more formal, but still understandable. In addition, action blocks supports the creation of contracts for system operations.
In this paper we also argue that treating system operations as use cases is a misuse of use cases—system operations and use cases should be separate artifacts. One should be able to obtain several system operations from a use case, otherwise there is no dialog (process) between actors and use cases. We believe that having a clear distinction between use cases and contracts will improve the quality of both.

1 Introduction

In all projects there is a need for informal descriptions of the system. In the Unified Modeling Language (UML) [13], use cases have taken this role. They describe how to use the system in such a way that everyone involved in the project can understand them, in particular the customers without technical training.

To automatically transform use cases into other types of diagrams would require formal treatment of use cases. For use cases to become too formal might not be beneficial. In general, the success of formal methods has been limited in the industry; a lot of informal specifications are still written. Furthermore, all projects need some kind of informal description of the system behavior and use cases are a perfect way to capture this information.

In this paper, we focus on how to write the action steps of a use case—the steps of the main success scenario and the steps of an alternative flow. In our point of view, the action steps are the most important part of a use case. These steps have to be written correctly, otherwise the use case is worthless. This paper is about obtaining higher-quality action steps. We present a way of

making use cases more formal, but still readable by a customer. We group action steps into what we call *action blocks*. The action blocks are not explicitly shown in use cases, they are only a way of structuring a use case. We believe that this grouping will increase the quality of use cases, because the lack of structure today makes them both hard to construct and read.

Furthermore, using the structure of action blocks, system operations, i.e. the calls into the system, can be extracted with ease. We believe that it is crucial to be able to obtain system operations from use cases. The system operations make the implicit communications between actors and the system explicit. Furthermore, the use of action blocks supports writing contracts for the system operations (adding pre- and post-conditions). In contrast to use cases, the semantics of contracts are better understood. More precise, even formal specifications can be obtained for the system using contracts.

We also discuss the problems caused by dressing system operations up as use cases. Use cases are not system operations themselves, but they implicitly contain system operations. Having identified the system operations, contracts can be made. We believe that a clear distinction between use cases and contracts will simplify the writing of both of them.

When reading the literature, one often gets the false impression that use cases can be used in any project and that they can take any form. We also discuss the issue of event driven programs where the use of use cases is limited, but system operations can still be applied. It is important to show that use cases are not perfect in all situations.

In next section we discuss action blocks in detail. The discussion of action blocks is followed by a section on how to obtain contracts from action blocks. Thereafter, we discuss the distinction between use cases and contracts, followed by limitations of use cases in event driven programs. Finally, we will present related work, conclusion and future work.

2 Action Blocks

In this section we first define what an action block is. Thereafter, we give a pattern supporting writing a use case using the structure of action blocks. Then we show a use case obtained according to our pattern. But first we look at what Jacobson considers to be an action step. In his book[9] he writes:

> "Each use case constitutes a complete course of events initiated by an actor and it specifies the interaction that takes place between an actor and the system. A use case is thus a special sequence of related transactions performed by an actor and the system in a dialogue."

To understand transactions in the database sense is too strong, because if a transaction succeeds then changes are made to the system (committed), otherwise the system is reverted to the original state (rollback). The paper of Sendall and Strohmeier[14] relaxes transactions to system operations.

2.1 System Operations

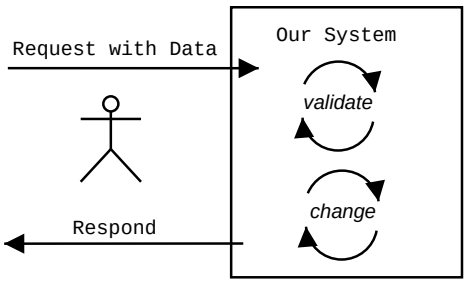

Fig. 1. A transaction has four parts

Cockburn interprets in his book [3] what Jacobson [9] means by a transaction in the following four points (see Fig. 1)[1]:

I. The primary actor sends request and data to the system.
II. The system validates the request and the data.
III. The system alters its internal state.
IV. The system responds to the actor with the result.

If we take the view that the actor sees the steps I to IV as an atomic action, all the things which happen inside the system are grouped together in a transaction. This is fine as long as the transaction succeeds, but what about failing transactions? The normal interpretation of a transaction is that if it fails the system should go back to the original state. Cockburn does not elaborate on this topic.

In this paper, we investigate the four steps described above in much more detail. We call the four steps an *action block* schema. We do not consider an *action block* to be a transaction by default. Whether one can view an action block obtained from the action block schema as a transaction depends entirely on how the action block is specified, as we will see in this section.

2.2 Request with Data

First, we look at steps I and IV. These two steps together constitute the signature of a system operation. System operations are the operations which deal with the events coming from outside of the system. From a customer's point of view these are the important operations: they define the functionality of the system. In the case when step IV is left out, there will be no return value. Describing a use

[1] Cockburn used Arabic numerals. We use Roman numerals instead to avoid confusing these numbers with Arabic numbers used in the main success scenario

case by using I and IV from the action block schema gives a use case of the form "action—response", for example[2]:

- Administrator registers a person
- System provides a student number

It is recommended to keep the sentences of action steps simple, for example subject-verb-object [3]. This is a good idea, but is sometimes too limited; for example, one might want to pass more than one piece of data to the system. Furthermore, if a domain model exists it is recommended that the data passed into the system and returned from the system should either be a concept from the domain model or a value of a primitive data type.

It can be useful to annotate verbs that are used as operation name, for example:

- Administrator registers a person

Here, we have underlined the name of operations. As we will see later, the operation annotation makes the relationship between contracts and use cases more explicit. In a tool, this annotation could instead be hyperlinked to contracts. It could be that, for readability, the name of the system operation and the name used in the use case are different. In that case one could include the system name in parentheses, for example:

- user gives PIN (authenticate)

2.3 Validation

To only consider the interaction between actors and the system as we described in the previous subsection makes use cases less useful than if we also consider the responsibility of the system. For example, it is important to validate that the request and data given to the system permit the user action to take place. The step II of the action block schema deals with this issue:

II. The system validates the request and the data.

Let us consider a fragment of the use case "Withdraw Money" for an ATM machine:

- user requests withdrawal of an amount of money
- system checks that the account balance is high enough
- system dispenses cash

It is at the validation step that something can go wrong, for example the user might not have enough money on his/her account. This will require an alternative action:

- account does not contain enough mone
  - system informs that there was not enough money on account

[2] Since our action steps are fragment of whole scenarios we will only use - in front of each action step. In the example shown later of a complete use case, we will use numbers to show the sequence which is the standard way

2.4 System Changes

A use case can describe what internal states are changed and what other actors are called. This is taken care of by step III of the action block schema:

- The system alters its internal state and calls other actors

In this step, we have included that the system can call other actors as well. In the ATM example above one can specify that the system should decide about the amount taken out from the ATM in the main success scenario:

- user requests withdrawal of an amount of money
- system checks that the account balance is high enough
- system subtracts from account the amount taken out from the ATM
- system gives back card and dispenses cash

and in the alternative case one can specify that no money is withdrawn:

- account does not contain enough money
  - system does not change the account
  - system informs that there was not enough money on account

One can ask if one really wants to write down things which do not happen? This might be considered as an over-specification of a use case, but on the other hand it is more precise. In the above example it is quite clear that the account should not change even if we do not specify it. It is important that the balance of the account stays the same as before the interaction. In this particular case we require withdrawal to behave like a transaction.

The bank system (*BankSystem*), cash dispenser (*CashDispenser*), and card (*Card*) can be viewed as actors for the ATM machine and be part of the use case description. Here is an example of how we can rewrite the use case fragment above:

- user requests withdrawal of an amount of money
- system checks that the account balance is high enough
- system subtracts from account the amount taken out from the ATM
- system calls returnCard from Card and calls giveOutCash from CashDispenser

We have specified that the *CashDispener* and *Card* are called. One might regard this as too implementation related for a use case. In general one will not include this level of details.

2.5 Pattern for writing action steps

Now, we combine the information from the previous subsections to give a pattern for how to obtain use case steps. For each action block one should consider

- What request and data is sent to the system.

Furthermore, one should consider the following three questions:

- Does the system need to validate the request and the data?
- Does the system need to alter its internal state and call other actors?
- Does the system need to respond to the actor?

Each of these questions can lead to several action steps.

The alternative flows does not follow the action block structure. For each alternative flow one should consider:

- The condition for taking the alternative flow.

These conditions most often represent the case when the condition of the validation of request and data in the main success scenarios goes wrong. There should be an alternative flow condition which true for any possible validation failure. Furthermore one should consider for the alternative (failure) situation:

- Does the system need to alter its internal state and call other actors?
- Does the system need to respond to the actor?

Thereafter the alternative flow can continue with action blocks just as in the main success scenario. There is one more question to consider in the case of alternative flow:

- Is it possible to go back to the main success scenario?

There are several conventions for how to go back to the main success scenario, two that are quite known can be found in Cockburn's and Larman's books [3, 11].

We believe our pattern will improve the quality of action steps. It does restrict the way of writing use cases, so whether this pattern is suitable in all situations has to be further investigated. In the rest of this section we will show the result of applying our pattern for obtaining action blocks.

The example in Fig. 2 shows a use case for *Withdraw Money* written using the action block schema. We include the part of the use case template which should be common for all use cases: name, main success scenario, extensions, and postcondition. Looking at the first extension, we have written 2-8a to mean that the action steps form 2 to 8 of the main scenario shall be substituted by the action steps described under the condition of the extension when it is true. If there had been another condition for replacing steps 2-8 we would have written 2-8b and etc. In the second extension we cannot use this numbering schema, since we return to an earlier action step 3.

If we look at the main success scenario, we can see that it contains 3 action blocks. We indicate the start of each action block by a hyphen "-". The first action block only contains step I and II of an action block schema. After step 2 of the main success scenario, we could have included "system requests PIN" to show that the handling of the card is over. This is a matter of taste, we prefer not to include unnecessary steps — steps which are obvious. The form of action block 2 is the same as action block 1. Only action block 3 contains all steps of

Use case: Withdraw Money
Main Success Scenario:

1. - user identifies himself by a card
2. system reads the bank ID and account number from card and validates them
3. - user authenticates by PIN
4. system validates that PIN is correct
5. - user requests withdrawal of an amount of money
6. system checks that the account balance is high enough
7. system subtracts the requested amount of money from account balance
8. system returns card and dispenses cash

Extensions:
2-8a. Wrong type of card or destroyed card:
1. System gives back the card

4a. Wrong pin less than 3 times:
1. System updates number of tries
2. start from action step 3

4-8a. Wrong pin 3 times:
1. System keeps the card

6-8a. Not enough money on account:
1. account does not contain enough money
2. system does not change the account
3. system returns card

Postcondition:

- If the customer entered the PIN stored on the Card, and the customer's balance was greater than or equal to the requested amount, then the customer got the requested amount and the amount was deducted from the balance.
- If the customer entered the wrong PIN three times, the card was retained.
- If the customer requested too much money, the card was returned to the customer.

Fig. 2. Use Case for Withdrawn Money

an action block schema. Action block 3 was explained in detail in Sect. 2. It is important to point out that an action blocks can contain more than 4 action steps as well. For example there might be several validation steps in one action block.

We have not included all possible extensions, for example power failure or communication failure. We have focused on the condition when step II of the action block schema goes wrong.

Even though the post-conditions are not an issue discussed in this paper we have included post-conditions for completeness of the use case description,

since post-conditions are an important part of any use case template. For further details of post-conditions we refer to [6].

Use cases experts would of course come up with something similar to the use case above. For beginners, we believe that applying our action block pattern should improve the quality of use cases. In the next sections we will see further benefits of using the pattern.

3 System Sequence Diagram

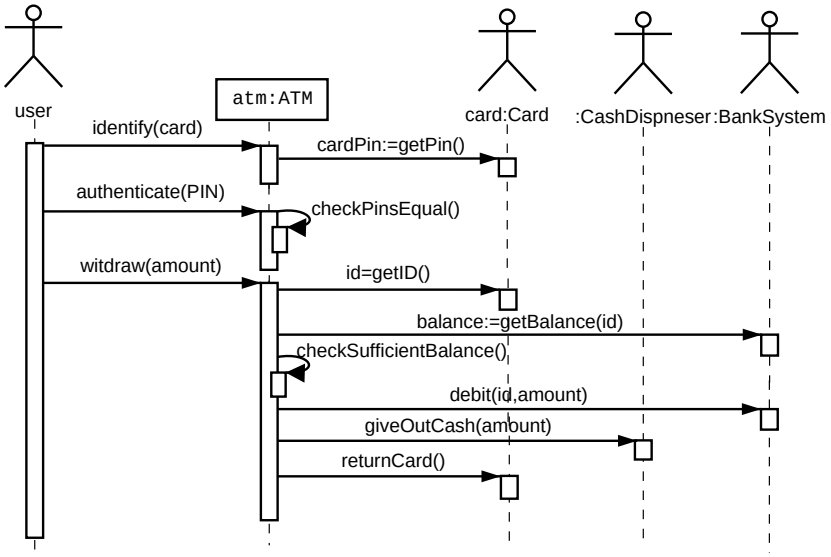

Fig. 3. Sequence diagram for the normal flow of the "withdraw" use case

The purpose of this section is to obtain a better understanding of the relationship between system operations and use cases. Here we will only look at the main success scenario, the 3 action blocks of the example in Fig. 2. The normal flow of control is illustrated in Fig. 3. As we can see from the use case *Withdraw Money*, the whole sequence of events is initiated by the user presenting the card to the ATM, which is modeled by the call `identify(card)` to the system object `atm`. To be able to check the PIN entered by the user, the card is queried for the correct PIN, `cardPin`.

After authentication, the customer is asked for the desired amount, which is passed to the ATM in the call `withdraw(amount)`. As we can see from the use case the amount needs to be compared to the customer's bank account balance. For this purpose, an identification `id` of the bank account is fetched from the

card, and used to query `balance` from the bank's central system. After checking that the balance is sufficient, the `CashDispenser` unit is instructed to deliver the required amount to the user. Finally, the card is returned to the user.[3]

In Fig. 3 we have only shown the main success flow of the use case Withdraw Money, but one can write a sequence diagrams for each alternative flow, a generic system sequence diagram, or a state chart diagram as we did in [6] showing all the possible ways through the use case. The strength of a system sequence diagrams is that they make the implicit communication among actors and the system explicit. In the next section we will see how the action blocks of use cases can be used to support the creation of contracts for system operations.

4 System Operation Contracts

Having written the use case in the style of action blocks, the contracts can easily be obtained—system operations together with pre- and post-conditions. Let us look at the third action block of the use case *Withdraw Money* of Fig. 2:

- user requests withdrawal of an amount of money
- system checks that the account balance is high enough
- system subtracts from account the required amount
- system returns card and dispenses cash

This can be turned into the system operation contract:

Operation: *withdraw(amount:int)*
Cross References: *Use Cases: Withdraw*
Precondition:
Postcondition: **if** *account contains enough cash*
then *the amount of the account is decreased by the amount taken from the account AND the card has been returned AND cash has been dispensed*
else ??

The case when there is not enough money on the account can be found in the extension of the use case:

- system does not change the account
- system returns card

So, the complete contract will be:

[3] It can be questioned where the call of `returnCard()` should be directed. We chose `card`, because the controller obviously communicates directly only with the card reading device and not with the card itself. If `card` stands for the card reader, than it is the right goal for the `returnCard()` call.

Operation: *withdraw(amount:int)*
Cross References: *Use Cases: Withdraw*
Precondition:
Postcondition: **if** *account contains enough cash*
then *the amount of the account is decreased by the amount taken from the account AND the card has been returned AND cash has been dispensed*
else *the account has not been changed AND card has been returned*

To have "the account has enough money" as a precondition would be too weak, since then we will only guarantee that the operation will behave correctly in the case where there is enough money on the account, but give no guarantee of behavior in the case when there is not enough money. Given the system operation sequence diagram in Fig. 3 and an appropriate class diagram, the contract can be written in a formal way using OCL [13]:

```
Context ATMController::withdraw(amount:long) post:
  if ( amount <= bank.getBalance(card.getID()) ) then
        cashDispenser^giveOutCash(amount)
    and   bank.getBalance(card.getID())
        = bank.getBalance@pre(card.getID()) - amount
    and card^returnCard()
  else
        not cashDispenser^giveOutCash(?)
    and   bank.getBalance(card.getID())
        = bank.getBalance@pre(card.getID())
    and card^returnCard()
```

Our language technology group has a tool for translating OCL to natural language [7, 2]. At the moment it can only handle version 1.5 of OCL. The OCL expression above uses the *hasSent* operator which is part of OCL 2.0. When the translator works for 2.0 we will have the benefit of automatically producing natural language text from formal contracts written in OCL. One benefit will be that the informal contract can be understood by people not trained in formal methods. So, both people trained in formal method and those who are not can find inconsistencies between the contract and the action blocks.

In the above OCL example, only one extension is used, but more extensions can be used. The conditions needs to cover all possible flow.

If there are no conditions one will only have internal state changes, return of values or both. In the ideal case when the ATM always gives out money the use case would not have to contain any conditions involving the account and the postcondition of the contract would be:

Postcondition: *the amount of the account is decreased by the amount taken from the account AND*
the card has been returned and cash dispensed

The transformation from action blocks to contracts has to be done by hand. One experiment can be to try making a formal language for action blocks, using the domain model, in such a way that contracts can be automatically produced from use cases. The problem might be that this language would be too formal, for example, not benefiting the customer.

5 Distinguishing Use Cases and Contracts

The situation where the actor calls the system once and obtains an answer can be described as a contract (one action block). A use case having only one action block is a contract written as a use case. We believe that treating everything as a use case severely harms their usefulness. A contract can be written differently than a use case. For example, in a contract it is natural to include explicit information about input and output data.

One could include input and output data as a part of the use case template as in the work of Jürjens et al. [10]. We consider this to be wrong, because if a use case has several action blocks it is hard to know which action block input and output is related to. Even worse, the same data might be input to several action blocks.

We believe that mixing contracts and use cases styles make neither a good use case nor a good contract. A contract can be written when one has one action block and a use case can be written when one has several action blocks. In addition, a use case is required to be complete [3]. So, just having more than one action block does not in itself make them a use case.

Jürjens et al. [10] added the extra information about input and output data to use cases to specify the critical data of the system. We argue that it is the action block which should be modified in this case since action blocks handle input and output data. So, each action block could be modified to specify the critical data.

In next two subsections we will discuss whether log-on is a use case or not.

5.1 Why Log-on is not a Use Case

We consider a debatable use case: log-on, describing how to log-on to a system by using login name and password. Some consider this to be a use case and some do not. One can find it described as a use case in the books of Cockburn and Larman [3, 11].

The reason why the use case log-on is debated in the first place is about whether log-on return a value for an actor. In general, nobody just wants to log-on, but to use the system to obtain a goal, for example, pay a bill. There are use case experts that consider log-on such an important issue that it should be a use case. We believe that log-on is not only important but crucial, but that does

not make it into a use case by itself. If log-on procedure takes as input a user name and a password and decides whether user log-on is successful or not, it should in our point of view be a system operation $logOn(user, password)$ which can be given a contract.

The purpose of log-on and to authenticate by card and PIN as in the use case Withdraw Money in Fig. 2 is similar, in the sense that the actor wants to be able to use the system to obtain a goal. The interaction needed for obtaining the authority to use the ATM is slightly more complex than with log-on, since the use case needs two action blocks to describe the interaction between actor and the system. The fact that there are more than one action block does not make it a use case in itself. Use cases also has to fulfill a goal for the actor. So, a use case should contain more than one action block and fulfill a goal.

5.2 Incomplete Use Cases

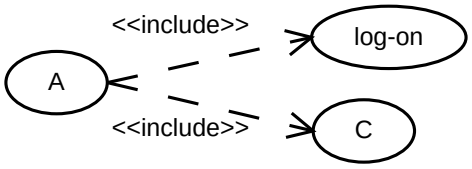

Fig. 4. Using Incomplete Use Cases

So far, in the paper we have only talked about complete use cases, i.e., use cases which return a result to the primary actor. The use of *include* use cases and *extend* use cases make things problematic, since it is generally accepted that they may not be complete use cases, i.e., return no results to the primary actor[4].

In Fig 4 we give an example of include relations. The use case A is complete, but *log-on* and C are incomplete. How to deal with this situation of incomplete use cases requires further research. One suggestion might be to permit the above use case diagram, but have the stereotype:≪complete≫, ≪incomplete≫, and ≪operation≫ on the use cases. Use cases with stereotype operation will be incomplete use cases which can be specified as contracts rather than as use cases. A similar problem exists for interaction overview diagram, where some of the interaction occurrences might represent only a single action block such as login.

6 Limitation of Use cases

In the book "Object-oriented software construction", Meyer [12] claims that use cases are not good because they decide the sequence of events too early in the

[4] This issue still debated among methodologists.

development process. For example, it is not important whether one first finds a house or first obtains a loan. The important thing is that one cannot buy a house before one has obtained both a loan and found a house. Cockburn does not consider this to be a problem. One can only write into the use case that "the following steps can happen in any order". One can also use activity diagrams to show parallel activities.

The problem is not when only two or three things can happen in any order, but rather when ten or twenty or more things can happen in any order. This might be the case for event driven programs. Some years ago we ran a project for creating an editor for an architect to draw houses in. In the project we used use cases for describing the workflow. For some small part of the system, we used use cases quite successfully, but when it came to the process of drawing the house, use cases did not really help. We did not want to put any restrictions on the order of steps involved in drawing a house.

For many systems use cases work fine, but for heavily event driven systems the use of use cases is less useful. Writing contracts can be more fruitful for event driven program, since in this case a contract relates directly to a functional requirement from the customer. However, one can still use scenarios to show uses of the system. Bear in mind that there can be infinitely many scenarios.

The fact is that the use case literature always gives the impression that it is always possible to use use cases successfully. This section has shown one case, even driven program, where use cases are difficult to use.

7 Related Work

Use cases were invented by Jacobson [9]. They are part of the UML [13] and supported by UML tools, and also widely used in the industry. There are a number of books dedicated only to use cases and countless papers discussing possible interpretations of use cases. Some of the papers which have influenced our understanding of use cases are [4, 5, 8] and the books [3, 1].

Even though Cockburn takes up transactions in his book [3] he does not elaborating very much on the idea. In his view there are several goal levels of use cases. He has blue use cases, which indicate user goal use cases. This is the type of use cases considered in our work. He has even use cases on higher and lower goal level, for example use case on the lowest goal level can be viewed as sub-functions. We believe it can be confusing with all these goal levels of use cases, particularly below the blue use cases. In our framework, we do not have these low goal level use cases, but instead contracts. We believe this separation is important, since it makes it clear what a use case is.

The most related and inspirational work for this paper is the work of Sendall and Strohmeier [14]. Our work can be seen as an extension of their work in the sense that we go more into detail about how to write the steps of a use case (giving a pattern) and have more details about how to obtain contracts from use cases. We also include a discussion around the misuse of use cases.

8 Conclusion and Future Work

Having a clear separation of use cases and contracts will improve the quality of system specifications. In particular one should avoid to dress up system operation contracts as use cases. Having identified the system operations we know how to write contract for them in a clear and precise way, even formal. In this paper we have made this distinction clear. Both system operation contracts and use cases are necessary, either used separately or together. We have shown misuse of use cases and discussed the limitations of use cases in the case of event driven programs.

We have defined *action blocks* and shown that they can give support for obtaining use cases of high quality. Furthermore, we have shown that action blocks support the writing of contracts. We have also given a pattern that supports the writing of action blocks in use cases.

We will look into tool support for producing contracts from an action block. By putting enough restrictions on the language for producing action blocks this translation could be made automatic or partially automatic. Our pattern currently does not in itself restrict the language used in each action step, because we believe this could make the pattern less usable. There is a severe danger in making the whole process too complex by requiring a more formal language for action blocks. This has to be looked into.

Acknowledgements. We thank Karol Ostrovsky, Kristofer Johannisson, Fredrik Lindblad, and anonymous reviewer for reading the draft version of this paper. In particular we want to thank Philipp Rümmer and Marcin Zalewski for both careful reading of the final version of this paper and several discussions.

References

1. F. Armour and G. Miller. *Advanced Use Case Modeling.* Addison-Wesley, 2001.
2. David Burke and Kristofer Johannisson. Translating formal software specifications to natural language—a grammar-based approach. In P. Blache, E. Stabler, J. Busquets, and R. Moot, editors, *Logical Aspects of Computational Linguistics*, number 3492 in LNAI. Springer, 2005. To appear.
3. Alistair Cockburn. *Writing Effective Use Cases.* Addison Wesley, 2001.
4. L. L. Constantine and L. A. D. Lockwood. *Structure and Style in Use Cases for User Interface Design*, chapter 7, pages 245–279. Object Technology. Addison-Wesley, 2001.
5. Gonzalo Génova, Juan Llorens, and Víctor Quintana. Digging into use case relationships. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 115–127. Springer, 2002.
6. Martin Giese and Rogardt Heldal. From informal to formal specifications in UML. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc. of UML2004, Lisbon*, volume 3273 of *LNCS*, pages 197–211. Springer, 2004.

7. Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in LNCS, 2002.
8. Sadahiro Isoda. A critique of UML's definition of the use-case class. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 280–294. Springer, 2003.
9. I. Jacobson, M., Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.
10. Guido Wimmel Jan Jürjens, Gerhard Popp. Use case oriented development of security-critical systems. *Information Security Bulletin 2*, pages 55–60, 2003.
11. C. Larman. *Applying UML and Patterns, second edition*. Addison-Wesley, 2002.
12. B. Meyer. *Object-Oriented,Software Construction*. Prentice Hall PTR, 1997.
13. OMG. *Unified Modeling Language Specification version 1.5 (2.0)*, 2005. `http://www.omg.org/technology/documents/formal/uml.htm`.
14. Shane Sendall and Alfred Strohmeier. From use cases to system operation specifications. In Stuart Kent and Andy Evans, editors, *UML'2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference*, volume 1939 of *LNCS*. Springer, 2000.