# Essential Use Cases and Responsibility in Object-Oriented Development

Robert Biddle, James Noble, Ewan Tempero
School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
(robert,kjx,ewan)@mcs.vuw.ac.nz

## ABSTRACT

Essential use cases are abstract, lightweight, technology-free dialogues of user intention and system responsibility, that effectively capture requirements for user interface design. Employing essential use cases in typical object-oriented development processes requires designers to translate them into conventional use cases, costing time, imposing rework, and delaying work on the object-oriented development until the user interface design is complete. We describe how essential use cases can drive object-oriented development directly, without any intervening translation, allowing user interface development and object-oriented development to proceed in parallel. Working with essential use cases yields some unexpected further benefits: analysts can take advantage of recurring patterns in essential use cases, and the crucial common vocabulary of responsibilities lets designers trace directly from the essential use cases to the objects in their design.

## 1. INTRODUCTION

Use cases are the accepted best practice for capturing requirements for object-oriented software development, and they are widely supported in modeling languages and in development processes. Constantine and Lockwood's *Usage-Centered Design* [9] introduced *essential use cases* — use cases written specially to be abstract, lightweight, and technology-free — to support user interface design. Following a conventional process, essential use cases would be used to produce the user interface design, then, once that design was complete, essential use cases could be translated into more conventional use cases — much more concrete and detailed descriptions of a system and its interface design. Unfortunately, translating essential use cases to a more conventional form requires effort, costs time, and delays work on the object-oriented development until the user interface design is complete.

In this paper, we explore the application of essential use cases directly to object-oriented software development. We speculated that essential use cases would work just as well as conventional use cases as a starting point for object-oriented design; that technology independence could better support requirements gathering, because there would be less need to specify details that are only relevant to the design; and that the brevity of essential use cases would better support communication between developers and stakeholders.

We began using essential use cases as our prime requirements gathering tool. We now suggest that essential use cases are suitable for object-oriented software development in general, and indeed have significant advantages over conventional use cases.

We have four general results to report. The first is what we speculated: that essential use cases can drive object-oriented design directly, without first writing more concrete conventional use cases. The other results were unanticipated. One is that essential use cases provide practical, operational guidance on how to move to an object-oriented design from the requirements. Another is that responsibilities provides a common vocabulary that supports seamless traceability forwards and backwards between essential use cases and objects. The last is that, because essential use cases show only the bare essentials of the use case, we are able to identify patterns of use cases, which can be used to make the requirements gathering more efficient.

This paper is organized as follows. We begin with an introduction to essential use cases, discussing their philosophy as developed by Constantine and Lockwood and comparing them to conventional use cases. Then, in section 3, we describe how to write essential use cases, and use role-play to verify that they are correct and consistent. Section 4 shows how essential use cases can be used to design object-oriented systems, reflecting on the role of responsibilities for both essential use cases and object-oriented design, and including a short example. Section 5 presents patterns for writing essential use cases. Section 6 then discusses some practical issues such as using essential use cases for systems without human users, development processes, and business process design. We then discuss related work in section 7, and finally present our conclusions.

## 2. BACKGROUND

### 2.1 Use Cases

Jacobson defines a use case in his 1992 book as "a behaviorally related sequence of transactions in a dialogue with

the system"[17]. A more recent definition for the Rational Unified Process shows little real change, saying a use case is "a description of a set or sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor"[16].

The general idea of a use case is to represent intended sequences of interaction between a system (even if not yet implemented) and the world outside that system. This idea is very powerful, for several reasons.

In the early stages of development, use cases help to focus on interactions as a way of eliciting desirable system behavior, and so help capture requirements and determine specifications. This technique is effective because interactions can be described in forms very easy for people to recall or imagine, such as narratives or dialogues. This is especially useful when involving a wide range of people in requirements gathering and analysis, such as end-users, background stakeholders, and others with no direct experience or role in actual system development.

In the later stages of development, use cases help again because of the focus on interactions. The interactions can now be regarded as the embodiment of specifications that the system must meet. In design and implementation, a structure must be determined and created that will meet these specifications. In review and testing, use cases can be used to drive system behavior for examination. Their guiding role in design, implementation, and review also assists in providing traceability.

Use cases also lead to a useful partitioning of requirements. This happens naturally, because use cases are based on sequences of interaction, and desirable interactions typically follow a structure of coherent progression, on a limited scale, toward a goal or sub-goal. This partitioning then allows organization by grouping, filtering, prioritizing, and so on, and is helpful in overall management throughout development.

## 2.2 Essential Use Cases

Essential use cases are part of Usage-Centered Design, as developed by Larry Constantine and Lucy Lockwood [9]. Constantine and Lockwood support use cases, and agree with many claims about their advantages. They also see limitations: "In particular, conventional use cases typically contain too many built-in assumptions, often hidden or implicit, about the form of the user interface that is yet to be designed." This is problematic for UI design both because it forces design decisions to be made very early, and because it then embeds these decisions in specifications, making them difficult to modify or adapt at a later time.

Essential use cases were designed to overcome these problems. The term "essential" refers to essential models that "are intended to capture the essence of problems through technology-free, idealized, and abstract descriptions". Constantine and Lockwood define an essential use case as follows:

> An essential use case is a structured narrative, expressed in the language of the application domain and of users, comprising a simplified, generalized, abstract, technology-free and implementation independent description of one task or interaction that is complete, meaningful, and well-defined from the point of view of users in some role or roles in relation to a system and that em-

gettingCash

| User Action | System Response |
|---|---|
| insert card | |
| | read magnetic stripe |
| | request PIN |
| enter PIN | |
| | verify PIN |
| | display transaction menu |
| press key | |
| | display account menu |
| press key | |
| | prompt for amount |
| enter amount | |
| | display amount |
| press key | |
| | return card |
| take card | |
| | dispense cash |
| take cash | |

**Figure 1: A conventional use case for getting cash from an automatic teller system. (From Constantine and Lockwood.)**

gettingCash

| User Intention | System Responsibility |
|---|---|
| identify self | |
| | verify identity |
| | offer choices |
| choose | |
| | dispense cash |
| take cash | |

**Figure 2: An essential use case for getting cash from an automatic teller system. (From Constantine and Lockwood.)**

> bodies the purpose or intentions underlying the interaction.

Essential use cases are documented in a format representing a dialogue between the user and the system. This resembles a two-column format used by Wirfs-Brock [27]. In Wirfs-Brock's format, the column labels refer to the *action* and the *response*. In contrast, the essential use case format labels the columns *user intention* and *system responsibility*.

These new labels indicate how essential use cases support abstraction by allowing the interaction to be documented without describing the details of the user interface. Note that the abstraction does not really relate to the use case as a whole, but more to the steps of the use case. In this way an essential use case does specify a sequence of interaction, but a sequence with abstract steps.

Constantine and Lockwood give the examples shown in figures 1 and 2. The dialogue in figure 1 is for a conventional use case, described in terms of actions and responses. The dialogue in figure 2 is for an essential use case, described in terms of intentions and responsibilities. The steps of the essential use case are more abstract, and permit a variety of concrete implementations. It is still easy to follow the dialogue, however, and the essential use case is shorter.

Jacobson created use cases to support object-oriented software design; Constantine and Lockwood introduced essen-
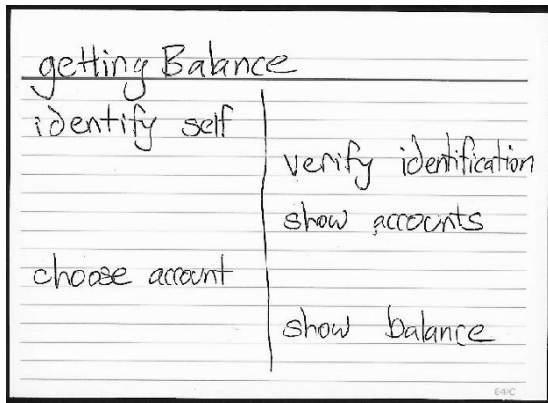
**Figure 3: An Essential Use Case Card**

tial use cases, and the larger framework of essential modeling, for user interface design and development. Our observation is that there is actually nothing about essential use cases that rules out their use for object-oriented software development, which means their advantages may also apply in that area.

## 3. ESSENTIAL USE CASES AND REQUIREMENTS

We began our exploration of essential use cases in object-oriented software development for practical reasons. We wanted to improve use case understanding and elaboration as a team activity. We were familiar with the CRC (class-responsibility-collaborator) technique for design [4], and decided to develop a similar technique for use case analysis. Essential use cases brought many characteristics beneficial in our new technique.

Like CRC, our technique involves using index cards and role-play. Teams work together to determine candidate use cases, allocate one card per use case, and write the name of the use case at the top. Each card is then divided by a vertical line, with the left hand side for the user and the right hand side for the system. (See figure 3.) Teams then work in pairs exploring dialogue, with one person playing the user, and another playing the system, together writing the dialogue for the card. Pairs then role-play in front of the team who review the use case.

Essential use cases are a dialogue between a user and the system. This facilitates direct use of role-play, as the players can regard the use case as a script, and one team member plays the part of the user, and another plays the part of the system. The dialogue also helps because the the interaction is very visible. Wirfs-Brock points out that use cases can easily be seen as a "conversations", and this familiar form of interaction assists in the modeling process [28]. This has the important effect that the use case dialogue and role-play really help determine the boundary of the system, making it clear what is done outside and what is done inside the system.

We have now used this approach working with a number of development teams, both in industry and at our university, and have gained experience about the effects of essential use cases on aspects of the development process. Essential use cases are abstract, and this brings many benefits. The dialogue is brief, and so able to fit on a card, and the ab-

straction can also quicken the analysis process. In discussion and exploration, many specific ideas for interaction will arise. However, the benefit of abstraction is that no particular concrete interaction sequence need be determined at this early point. This allows useful exploration to help determine the essential use case, while avoiding the need to make time-consuming implementation decisions.

The abstract nature of an essential use case means that a direct role-play does not yield "realistic" dialogue. However, role-play in early stages of development needs to be lightweight, and realism is not critical. If concrete examples of abstract elements of the dialogue are needed to clarify issues, it can be useful to explore a concrete scenario, referred to as an "enactment" of the essential use case, representing a possible implementation.

### 3.1 User Intention

In essential use cases, the dialogue specifies not simply the user *actions*, but the user *intentions*. The effect of this quickly became clear in exploring the dialogue through role-play.

An important aspect of role-play is that people identify with the role they are playing, and tend to think from that point of view. The emphasis on intention seems to intensify this effect, because the role player must examine their motivation more deeply. The need to identify user intention requires understanding about the kind of person the user is, and consideration of the situation they are in. With this encouragement, people playing the user role tend to make efforts to determine their context, and then really focus on expressing intent.

Driven by this, the use case role-play becomes more significant for the other team members or a wider audience. The strong concern of the user role-player with user intention makes the role-play evocative. More importantly, reviewers have more to consider than just the users actions or words: the issue of intent invites deeper consideration and closer scrutiny. This makes it easier to evaluate coherence, and determine whether all the critical elements of the dialogue have been identified.

Concentrating on the issues beyond the use case is an important advantage. Use cases focus developer attention on how a system interacts with the world, and in particular on how the system is used. In most system development this usage is not precisely pre-determined, and working out the use cases requires understanding and creative effort. Essential use cases do focus on usage, but by requiring the identification of user intention, they also ensure that usage can be determined on the basis of understanding the users. In this way, the term "user intention" acts as a heuristic to guide the specification of the use cases.

In some software development processes, such as the Rational Unified Process, the user interface prototype is developed early [16]. This allows the user interface to act as an input to later system design. This is beneficial because the development of the user interface will involve work to understand the users and determine what they need. System development based on essential use cases will also accomplish this, because essential use cases require understanding of the user intentions. In this way, essential use cases explicitly build a concern for the user into the process. Moreover, this is done in a lightweight way, without the need to generate actual user interface prototypes, so allowing more rapid

development.

Essential use cases were created with user interfaces in mind, and the term "user" refers to a human user. When we began using essential use cases, we accepted this because our use cases were driven by human users. We later changed from the term "user" to the term "actor", following Wirfs-Brock's terminology, and also UML.

## 3.2 System Responsibility

In essential use cases, the dialogue specifies not just system *response*, but system *responsibility*. Some of the effects of this arise early, and in a similar but more complex way to those involving user intention. Other effects become clear only later in design.

In role-play, the user or actor role often allows identification with a known kind of person, thereby allowing some inspiration about intention. The system role involves identification with an unknown entity, giving less opportunity for motivation. It is known that the system should correctly interact with the user. However, this provides little leverage in discovering motivation, allowing identification, or determining desirable interaction.

To gain insight about any element of dialogue, one needs to consider the purpose beyond it. For the user, essential use cases employ the term "intention" to denote the purpose. This reflects that nature of the user as external but understandable, and intention is something we are able to estimate.

For the system, we must instead describe something internal to the system which will guide its design. This is what *responsibility* is all about: it is an expression of *what* needs to be done, without unnecessary detail of *how* it will be done. This a more subtle motivation than "intention", but when understood it does assist determining the use case, and it also assists role-play. The motivation for the user is the intention to accomplish goals; the motivation for the system is the responsibility to fulfill obligations.

Essential use cases harness abstraction to ensure the user interface is not designed too early, and can be designed to be independent of any particular user interface technology. In the user role, the focus on intention supports abstraction by avoiding the need to decide details of how the intention is expressed. In the system role, the focus on responsibility supports abstraction by avoiding the need to decide details of how the responsibility will be implemented.

Essential use cases do have some user interface heritage that must be addressed in the context of more general system development. In examples of essential use cases employed to develop user interfaces, the system responsibilities typically concern presenting some information to the user. In this way, the system plays its part in the dialogue. However, in more general system development, the system will have responsibilities that go deeper. For example, the banking system essential use case gettingCash shown in figure 2 clearly relates to the user interface, but says little about responsibilities relating to the accounts and money in the banking system.

Of course, ordinary use cases may also have this same weakness. If the use case only documents the dialogue between the user and the system, important context may not be obvious. For example, the ordinary use case in figure 1 also says little about the system role relating to accounts and money.

getting Cash

| *User Intention* | *System Responsibility* |
|---|---|
| identify self | |
| | verify identity |
| | *log transaction start* |
| | offer choices |
| choose | |
| | dispense cash |
| | *adjust balances* |
| take cash | |
| | *log transaction finish* |

**Figure 4: An essential use case for getting cash, augmented to show system responsibilities not directing involved with user communication.**

With essential use cases, the focus on responsibility, rather than response, addresses this limitation. It seems reasonable to extend essential use case practice, and identify significant responsibilities that are not directly concerned with communicating with the user. For example, consider figure 4, where the essential use case for gettingCash has been augmented to show system responsibilities beyond communicating with the user. These do not strictly follow the dialogue form, although they resemble dramatic "asides" that precede communication. Most importantly, they add to the completeness and coherence of the scene, and aid system development.

## 4. ESSENTIAL USE CASES AND DESIGN

In object-oriented design, the term "responsibility" already holds a special role. In particular, responsibility is the pivotal concept in CRC cards, and in Responsibility-Driven Design.

In the CRC technique [4], responsibilities are associated with objects, and identify problems to be solved. Objects may send messages to other objects in the course of satisfying responsibilities, and these other objects are designated as collaborators. The arrangement and delegation of responsibilities and collaborators is then iteratively adjusted through many versions until a satisfactory structure emerges. In this way, responsibility guides the articulation of a system by partitioning classes to distribute responsibility. While CRC may have been originally cast as a pedagogical tool, it is now seen as useful in the context of practical system development [24, 5].

In responsibility-driven design [25, 26], the idea of responsibility is used more thoroughly and on a larger scale. Responsibilities are associated with objects, and represent knowledge an object maintains, or actions an object can perform. Responsibilities thus emphasize abstract behavior while being silent about possible implementation structure. However, objects may fulfill responsibilities by collaborating with other objects through message sending, and responsibilities may also be factored to higher level abstract classes. Together, such principles lead to designs where responsibilities are apportioned at high levels of abstraction, without any mention of object implementation.

The basic idea of "responsibility" seems to be the same in both CRC and responsibility-driven design, and the ways in which it is used seem consistent. Both techniques suggest that every object should have a coherent and well-

understood set of responsibilities, and involve the concept of distributing responsibility sensibly as a guide to making design decisions. In essence, both use responsibility as a design heuristic. A basic principle of object-oriented design is that objects involve behavior and information that work together. Responsibility is good heuristic for determining this, because the word "responsibility" suggests both a duty to do something, and the resources with which to do it. Responsibility also allows delegation, allowing large responsibilities to be managed by delegating smaller responsibilities to others. Responsibility involves both abstraction and encapsulation, as Wirfs-Brock et al.[25] explain:

> The responsibility-driven approach emphasizes the encapsulation of both the structure and behavior of objects. By focusing on the contractual responsibilities of a class, the designer is able to postpone implementation considerations until the implementation phase.

In essential use cases, the idea of a responsibility is to identify what the system must do to support the use case, without making commitments as to how it will actually be done. This resembles object encapsulation, where the internals of an object cannot be directly accessed from outside, and has similar benefits.

This role of responsibility in use cases is entirely consistent with the role of responsibility in design. Both describe behavior without describing implementation. This commonality presents valuable opportunities to link the way we work when determining requirements and the way we work when determining design.

## 4.1 Determining the System Boundary

In our experience, a major issue in determining requirements is distinguishing what the system should do from what it should not do. It often difficult to make decisions about this boundary, but it is also often difficult to communicate about this issue with all the people involved, analysts and stakeholders. We have found that it is very helpful to apply an approach familiar in design. We present the system as a "black-box", with an explicit boundary, describing the behavior of the system by essential use case responsibilities.

We can regard the system as a single *system object*, with a set of responsibilities like any object, and an implementation not yet under consideration. Jacobson [17] proposes a similar idea, but takes it in a different direction. With essential use cases, we can use the responsibilities to help determine the boundary of the system. If the system is like a single object, then the use cases are like methods of this object. They allow access to the system behavior, and no other access is possible. The interaction in a use case resembles method parameters and return values, but managed in a sequential way.

We have found use case diagrams useful in reinforcing this idea. We use a form of the use case diagram that, as in UML [23], shows actors (as stick figures) and their involvement with use cases (as ellipses). We also explicitly show the system boundary, depicted as a box surrounding the use cases, with the lines between the actors and the use cases crossing the boundary, as shown in figure 5. This clearly separates the actor's intention from the system's responsibility. This convention of showing the system boundary in use case diagrams was used by Jacobsen [17] but does not typically
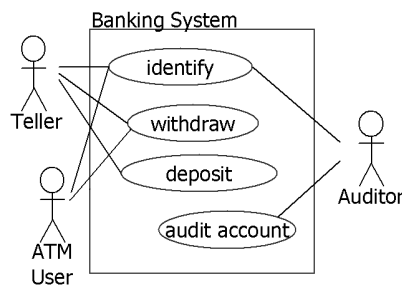


**Figure 5: A use case diagram, explicitly showing the boundary around the *system object*.**

feature in use of UML or in the Rational Unified Process [16]. We have found it worthwhile, consistent with the idea of the system as an object, and helpful in resolving issues about determining the boundary of system responsibilities.

## 4.2 Use Case Responsibilities and Design

For any particular system, the responsibilities in the essential use cases must be strongly related to the responsibilities of the objects internal to the system. Essential use case responsibilities must reflect the behavior of the overall system, and the object responsibilities must together reflect the same behavior.

This focus on responsibility in both essential use cases and in design suggests a way in which to strongly link system requirements and system design. The responsibilities from the essential use cases can be used as a starting-point for system design. This provides positive operational guidance when beginning design, and later leads to explicit traceability from the design back to the use cases.

To begin design we can start with a set of essential use cases, and the responsibilities they describe for the system object. We can then consolidate these where possible by using consistent language. Design work can then begin, which will determine a set of collaborating objects that will together meet these same responsibilities.

A strict approach to design might begin just with the system object and work from there by identifying related responsibilities and creating classes with those responsibilities. This approach could then be continued carefully, distributing responsibilities and eventually determining a design. This approach is essentially the same as refactoring, primarily discussed as a technique for improving the design of existing code [11]. Some of the refactoring techniques can easily be applied just to designs: for example **Extract Class** is a common technique applied in the early stages of the design process.

We do not advocate such a strict approach. One reason is that it does not harness any domain model. This will likely lead to difficulty in creating a correspondence between the domain model and the design, and thus fail to deliver the advantages of understandability and maintainability that are associated with that correspondence. Another reason is that a complete system may have many use cases and responsibilities, making a strict decomposition very difficult. Finally, a strict approach would make it difficult to allow consideration of design structures that arise from elsewhere.

In CRC or responsibility-driven design, design begins with finding a set of key candidate objects and classes, on the basis of a model of the application domain. Initial responsibilities are then assigned to these objects and classes, typically informed by knowledge of the domain and by design heuristics. This yields an initial design which can be explored and improved iteratively by with a small set of focal use cases.

Essential use cases do not require change in either CRC or responsibility-driven design. However, the responsibilities from essential use cases can play a helpful role. In both CRC and responsibility-driven design, there is a significant element of rapid exploratory design consideration. At significant points in the design process, the ability to check object responsibilities with use case responsibilities presents a valuable way of checking to see whether a design still meets the requirements.

One such significant point is at the beginning of the design. When assigning initial responsibilities, consideration can be given to the responsibilities required by the use cases. Alternatively, the initial design responsibilities might still be created from domain knowledge. These can then be compared with those from the essential use cases, and can give us valuable early feedback, and allow us to avoid future difficulties that may otherwise result. This approach provides better guidance for designers at a critical point in system development.

Design is rarely undertaken in a void, and there are typically many existing design assets that can be reused as part of any new design. For example, there may well be legacy components, component libraries, frameworks, or design patterns. Even where these are themselves already implemented, harnessing them may well affect the system design. As with the alternatives that arise in exploration, the responsibilities from essential use cases provide a valuable way to check how the resulting design matches the requirements.

Unlike alternatives from CRC or responsibility-driven design, however, other design structures may not come with responsibilities already identified. The comparison is then more arduous, and will involve careful examination of components and other structures. Even where the assets are actually implemented already, it is not the implementation that must be examined, but more the behavior: in fact the responsibilities. We believe such care and examination is valuable, and ultimately unavoidable to facilitate successful reuse.

When a discrepancy between use case and design responsibilities is detected, there are several avenues of resolution. The design may have gaps, either unintended or simply a reflection of temporary priorities in design activity. In either case, the design should be improved. On the other hand, there are sometimes important advantages to designs even if they fail to meet some requirements. For example, the design may be based on valuable existing artifacts that fall slightly short of requirements. In such cases, it may be reasonable to revisit to the use cases, and explore whether they should be changed in order to allow use of the design assets.

In all these situations, the ultimate aim is the same: consistency between requirements and design. By using responsibilities in requirements and in design, our approach also leads to a significant improvement in traceability.

## 4.3 Example

In this section, we present a small example to illustrate how our approach works. This design is for a small part of a library system. The domain model consists of two classes of objects: books and borrowers. Figure 6 shows the sequence of steps. At the top, we have identified the two focal use cases: borrowing and returning a book.

Consolidating the system responsibilities of just these two essential use cases results in a system object, which we show as a CRC card (marked "Iteration 1"). It is important to be aware that not all responsibilities of the system object will be explicit at this point. For example, in order that the system verify that the book may be borrowed, the system has the implicit responsibility to record what books are borrowed. Implicit responsibilities will typically be identified when they are distributed to the collaborating objects.

While the system object must clearly be able to enact the two use cases, if we had more use cases the resulting object would be large and unwieldy to implement. Considering the domain model, we could construct a design using three classes, a singleton Library System class, a Book class (one instance per book) and a Borrower class (one instance per borrower). We now need to decide how the responsibilities are distributed between the classes.

Iteration 2 shows the CRC cards for this design. There are several points we should note. For example, the implicit responsibility of recording whether or not a book has been borrowed has now been made explicit, and delegated to the Book class. Furthermore, the responsibility of updating whether or not the book has been borrowed has also been moved to the Book class. Note however that the Library System still has the responsibility of initiating the check and record update, which together may be regarded as the responsibility to issue the book.

The Library System has also acquired another responsibility that was not apparent in the original system object, that of "knowing all books", that is, being able to locate a Book instance given its identity (such as call mark). This responsibility has become important because we have created the Book class, and so the Library System must now become responsible for managing the Book instances. The situation with Borrower is similar.

Further consideration of the design could see the "knowing all books" responsibility moved into a separate singleton Catalogue class, perhaps implemented with a standard Collection object (Iteration 3, with the Book and Borrower classes unchanged).

One final point to make with this example is that we have only listed responsibilities as identified by the use cases. In reality, when we develop the domain model we will usually identify other responsibilities that the classes are likely to have. This information would also feed into the process described above. For example, recording whether or not a book is borrowed is a responsibility very likely to be identified from the domain analysis.

As the figure shows, tracing the responsibilities from the requirements through to the different designs is not only straightforward, but naturally falls out of the design decision making process. This is useful for communicating with stakeholders, and in particular when carrying out reviews.

**ESSENTIAL USE CASES**

borrowingBook

| User Intention | System Responsibility |
|---|---|
| identify self | verify borrower id |
| identify book | verify that book may be borrowed |
| | record book as issued to borrower |

returningBook

| User Intention | System Responsibility |
|---|---|
| identify self | verify borrower id |
| identify book | record book as returned by borrower |

**CRC CARDS**
**ITERATION 1**

**System Object**

verify borrower id
verify that book may be borrowed
record book as issued to borrower
record book as returned by borrower

**ITERATION 2**

**Library System**

verify borrower id
issue book to borrower
know all borrowers
know all books

Borrower
Book

**Borrower**

**Book**

verify that I may be borrowed
issue myself to borrower
return myself

**ITERATION 3**

**Library System**

verify borrower id
issue book to borrower
know all borrowers

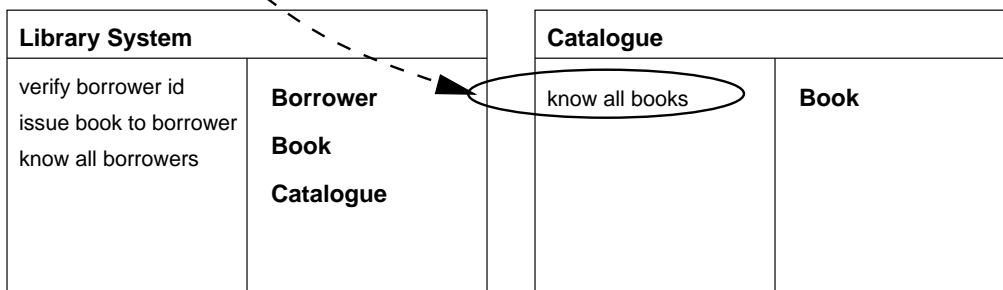Borrower
Book
Catalogue

**Catalogue**

know all books

Book
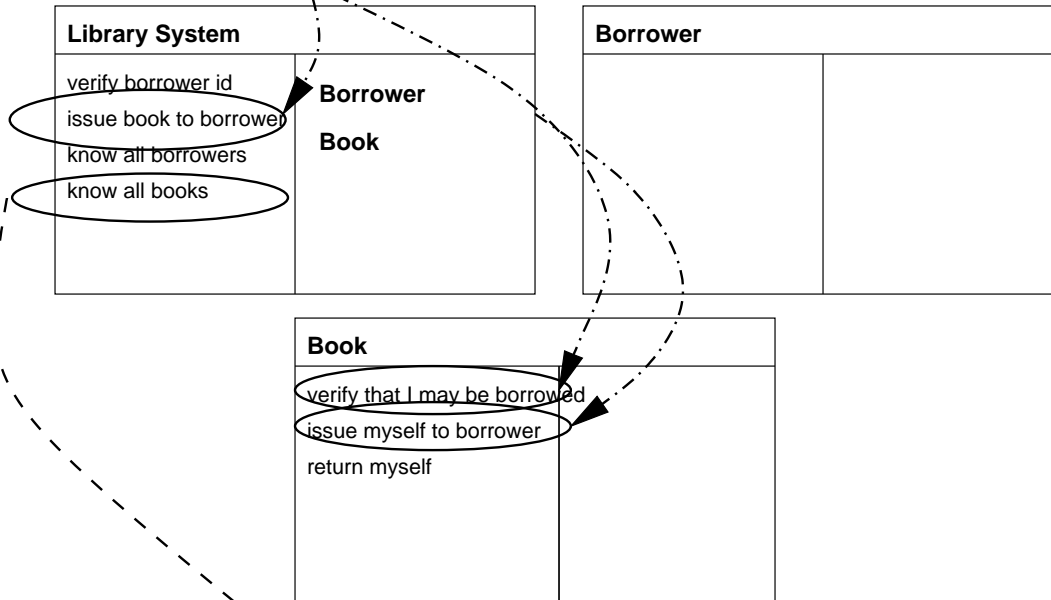
Figure 6: Tracing responsibilities from essential use cases to design.

# 5. PATTERNS OF USE CASES

While writing and designing software using essential use cases, we noticed that particular sequences of user intentions and system responsibilities reoccurred, both within single systems, and, more importantly across systems as different as network help systems, supermarket stocktaking, and micro-controller programming environments. Importantly we did not write the use case models for the majority of these systems ourselves.

As we investigated these recurring sequences in use case bodies, we realized that they represented solutions to particular problems in use case modeling: the sequences were common because they provided solutions to common problems in modeling the interfaces of systems to the actors surrounding them, that is, to the world outside. Following other patterns work [10, 12], we then identified (or "mined" in patterns terminology) a series of patterns for essential use cases [6].

To move from descriptions of repeating use case sequences to patterns, we also analyzed the *forces* acting in the use case, that is, the important considerations impacting on the use case [1]. For essential use case patterns, the forces capture characteristics of the interaction between actor and system: issues of initiative, information flow, and usability, such as preferring selection (from menus) over memory (for obscure command names), and promoting safety by requiring confirmation of irreversible actions. Each pattern gives positive and negative consequences of in terms of some of these forces.

These essential use case patterns are philosophically closer to Analysis Patterns [10] than they are to Design Patterns [12] or other user interface patterns [13, 7]. Like Analysis Patterns, these patterns describe analysis artifacts, rather that object-oriented designs or user interface designs for completed systems. Essential use case patterns describe patterns in essential use cases — that is, characteristic dialogues of user intention and system responsibility — while Analysis Patterns describe patterns in business domain models.

## 5.1 Some example use case patterns

We have identified at least 18 use case patterns overall, and space does not permit us to include them all in this paper [6]. Figure 5.1 presents a précis of one of the first patterns we identified, the **Alarm Use Case Pattern**. The problem this pattern addresses is to write an essential use case to model an interaction where the system needs to notify an actor about an important event, such as a change in its internal state or a potential violation of an internal invariant or business rule. The full pattern form includes rather more discussion, more variants, and a list of known uses: all of which have been elided in the figure.

The key to this pattern is that the essential use case to model this interaction begins with the system's responsibility to signal the user, rather than beginning with a user's intention to trigger some kind of alarm: in the first example in the figure (to warn of an undersold performance in a theater) this is the *only* step in the use case. While perhaps straightforward in retrospect, our experience teaching and working with essential use cases is that use cases following this pattern are counterintuitive at first sight: it is not obvious that a use case can simply involve the system's responsibility for communicating some information to an action, without any requirement for the actor to initiate or confirm

## Alarm Use Case

*How do you have the system inform the user about something?*

### Forces

- The system needs to draw actor's attention to a change in its internal state.
- The system is about to break a business rule.
- The notification should be asynchronous, that is, actors should not have to trigger the use case.

**Therefore:** *Write a use case that begins with the system taking the responsibility to warn the user.*

### Example

**Warning Start of Performance**

| User Intention | System Responsibility |
|---|---|
| | Signal "performance about to start" |
| | Show name, theater, and times of performance |

### Consequences

- \+ The system takes responsibility for initiating the use case.
- \+ The system can pass information about the alarm to the actor.
- \+ The actor does not have to interrupt their current task immediately to respond to the alarm.
- \- The actor can ignore the alarm.

### Variant

If the alarm is important, you may need to include a **Confirming Step**:

**Warning Theater Performance Undersold**

| User Intention | System Responsibility |
|---|---|
| | Signal "performance undersold" |
| | Show name, theater, time or performance, and percentage of seats sold |
| Confirm warning | |

This variant has the following different consequences to the main pattern:

- \- The actor cannot continue with their current task: they must interrupt it to confirm the alarm.
- \+ The actor cannot ignore the alarm.

**Figure 7: A pattern for alarms**

## Prompting Step

*How should you write a use case when the system knows some information that would help an actor make a decision?*

### Forces

- The system knows information that the actor may not know.
- Even if the actor knows all the information in the system, they may not know which information will be applicable at any given time.
- For human actors, making choices is easier than remembering commands or internal codes.
- For system actors, receiving current information from the system is easier than them having to maintain duplicate copies of that information.

**Therefore:** *Give the system the responsibility of offering that information before the actor makes the decision.*

### Example

**Booking Seats in Performance**

| User Intention | System Responsibility |
| --- | --- |
| | Offer Unbooked Seats |
| Choose Seats | |

### Consequences

- \+ The actor does not have to remember or store information that is known to the system.
- \- The interaction will be less efficient because the system has to present more information to the user.

**Figure 8: A pattern for prompting actors**

that information. Early in our essential use case modeling work we came across several examples that clearly fit this pattern, and generally chose to tolerate them grudgingly. Further experience seeing more examples of this kind of interaction lead us to realize that these interactions could all be modeled using this pattern.

Subsequently, we have seen this pattern appearing in many use cases for a wide variety of systems: to provide warnings that a new program will overwrite the current contents of a programmable logic controller; to advise a stockroom that a supermarket shelf is running low and should be restocked from the warehouse; and to notify users when they have new mail, to give just three examples.

The alarm pattern also illustrates the move from *System Response* in Wirfs-Brock's two-column use cases to *System Responsibility* in Constantine's (and our) essential use cases: a *response* requires some prior user interaction to which it can respond, while a *responsibility* includes both initiating and responding to communication with actors.

The figure also shows a variant of the alarm pattern: where it is important that the actor confirm receipt of the alarm, the use case can be written to capture the actor's intention with an extra, explicit, confirmation step. As the figure explains, this variant pattern resolves the forces in the use case in a slightly different way: the alarm cannot

be ignored, but it will be more intrusive, interrupting the current task by requiring the actor to confirm it explicitly. Microsoft Windows' "hard disk full" can be seen as an implementation of this variant pattern: when a machine's disk is full, the system warns the user of this fact, and expects confirmation that the warning has been received.

Figure 5.1 gives another example of a similar pattern — the **Prompting Step Pattern** — where the actor intends to make a decision, the system can supply that actor with up-to-date information before the decision is made, thus removing the responsibility to remember, store, or track the information maintained by the system. Although more intuitive than the alarm pattern, the prompting step pattern has also been useful to analysts writing essential use cases, to determine whether (or not) to include a prompting step in their models.

## 5.2    Patterns and Responsibilities

As with all types of patterns, we do not claim the designs or analyses behind these patterns are novel — rather, we have identified each of these patterns in at least three separate systems: systems that were completed *before* we began investigating patterns systematically. We hope that none of these patterns will come as any surprise to those experienced with writing any form of use cases. We have, however, found that these simple patterns extremely useful to people have identified an interaction as a potential use case, but do not known how best to actually write the essential use case to represent that interaction.

None of these patterns are specific to essential use cases — there are obvious analogues for conventional forms of use cases. We do not claim that these patterns could not be developed by examining conventional use cases, although no such patterns have yet been identified to our knowledge. We do believe, however, that these patterns would not be anywhere nearly as obvious written with a conventional use cases forms, because the amount of information and the length of the more conventional forms would make analysis significantly harder than with essential use cases.

In terms of identifying patterns, the key advantage of essential use cases is precisely that they focus on the essential, abstract nature of the interaction between an actor and a system, and characterize a system's interface without describing the implementation of that interface in detail. This is clearly useful for user interface design: this paper argues that this is useful in object-oriented design more generally. This is precisely the kind of information that you need to recognize or use patterns: to quote Gamma et. al.: *"A design pattern names, abstracts, and identifies the key aspects of a common design structure . . . . The design pattern identifies . . . the distribution of responsibilities."*. In the same way, an essential use case abstracts and identities the key aspects of an interaction between an actor and a system — the sequence of user intentions and system responsibilities. The presentation of these use case patterns is simplified by having to only discuss the minimal details associated with essential use cases, making the patterns easier to understand, in much the same way that the presentation of a design pattern concentrates primarily on objects, their relationships, and their important responsibilities, rather than the low-level details of the code that will be needed to implement the pattern.

# 6. DISCUSSION

In this section we discuss a number of issues related to essential use cases and responsibilities in object-oriented design: applications to systems without user interfaces (such as embedded or real-time systems); to systems with set, predefined user interfaces; iterative and incremental processes; parallel development of user interfaces and applications; and the object-oriented design of business processes.

## 6.1 Systems without User Interfaces

The approach we have outlined, based on essential use cases and object responsibilities, is quite applicable even to those systems without a traditional user interface, such as embedded systems, or software engines that interact only with system actors.

The key point here is that any system has to interact with the *"outside world"*. For any system, it is important to determine the boundaries between the interior and exterior of the system; to determine the principal interactions between the system's actors (human or machine); and to characterize those interactions in a way that facilitates later design. We have found that essential use cases retain most of the same benefits for these kind of systems (or rather, these kind of interfaces to systems) as they do when used to describe user interfaces: by working at the right level of abstraction, they capture the essence of the actors intentions and system responsibilities while eliding the accidental details of syntax and implementation. As we have described, essential use cases are smaller (and thus quicker to write, review, and modify) than longer, more detailed use cases, and facilitate system responsibilities flowing seamlessly from analysis to design: all these benefits apply equally to interactions with system actors as well as with human users.

Real-time systems often impose non-functional constraints (response time, transaction rate, information volume, reliability) upon their use cases. When using more conventional use cases, such constraints can be associated with the use cases narratives, and managed along with them. Managing these constraints is orthogonal to the way use cases are structured, and this information can be handled in much the same way when using essential use cases. Usage-Centered Design [9], for example, has always associated this kind of property with essential use cases, since they are crucial to the design of high-performance user interfaces.

The traceability between the system responsibilities of the essential use cases, the consequent responsibilities of the system object, and the resulting responsibilities of the internal objects making up the system design is also beneficial in ensuring the system does meet such non-functional constraints. For example, the system-level constraints (on use cases) can be propagated forwards with their responsibilities to objects during design, and objects' non-functional performance can be tracked backwards via their responsibilities to the use cases. Either way, the explicit traceability gained by using a common notion of responsibility can be used to verify that such non-functional constraints can be met in a final design.

## 6.2 Systems with Set Interfaces

Although originally developed to design user interfaces, we have found that essential use cases can be used for the object-oriented design of systems even when the interface is fixed in advance — either because the user interface design has been completed, or because the only interfaces are to other system actors via existing protocols. To design systems with these interfaces, we begin by analyzing the interfaces and writing essential use cases to describe their interactions, and then we develop the object-oriented design from these use cases. This allows us to capture most of the benefits we have outlined — in particular, we can design the system focusing on the essential parts of the interactions, especially the system responsibilities — and also gives some confidence that the internal system design will be stable in the face of changes to the details of these interfaces.

## 6.3 Designing Business Processes

Conventional use cases have long been advocated for business process design as well as software design [15]. Such designs typically begin with use cases specially written to describe business process interactions — that is interactions between people or business units rather than people interacting with computer systems.

Essential use cases are technology free: they describe the abstract intentions and responsibilities in a use case, and so any given use case can be implemented in a range of interface technologies. This is practically useful: exactly the same essential use cases can be reused where one system has more than one interface (for example, as a desktop application for a call center, a web site, and an interactive voice-response system). This also means, however, that they can characterize designs using *no* technology — that is, business processes — as well as designs based on intensive computational support.

We have had some experience with applying essential use cases and system responsibilities to business processes design and have been successful: the same advantages of abstraction, dialogues, and common responsibilities accrue to business process design as to software design.

## 6.4 Development Processes

In this paper, we have primarily described the models that we build for software design, and have made only passing references to the process by which those models are constructed. It is important to understand that although information *conceptually* flows through the models from essential use cases to the system object and then into the internal object-oriented design, we are describing a *philosophical* view of the relationships between these design models, and not a *temporal* or *process* view of the way the models should be constructed. In particular, we do not practice or advocate a waterfall approach, beginning with essential use cases for "analysis", then making sure they are "complete" and "correct" before proceeding onto the next phase of "design", and so on.

Rather, we expect that the design process will be incremental and iterative: starting by coming up with a list of candidate use cases, and developing a rough domain model in parallel. Then, some focal use cases could be elaborated, identifying user intentions and system responsibilities and writing them onto use case cards. An initial design could be developed for these use cases, resulting in responsibilities being tracked backwards from objects and the wording of the use cases being refined. Then more use cases could be elaborated, more object modeling attempted, existing work revisited in the light of later developments, and so on, as is accepted good practice.

We have found that using essential use cases and respon-

sibilities throughout the whole design iterative development rather than hindering it. Common levels of abstraction and a common vocabulary between use cases and objects make it easier to work either forwards or backwards during an iteration, tracing use cases to objects or vice versa. This also makes it easier to answer the frequently-asked question about when a design is complete [20]: the design is done when all the use cases and objects use the same vocabulary, when all the system responsibilities have been delegated to internal objects within the design, and when every use case can be executed by the object model.

## 6.5 Parallel Development

Basing a system's internal object-oriented design upon essential use cases has several advantages when a user interface is also being designed using essential use cases, typically via Usage-Centered Design. Duplication of work can be avoided: the essential use cases developed for the user interface design can be reused directly by the software designers, without having to be recast into other, more detailed forms of use case. This allows user interface design and software design to proceed in parallel, both working from the same essential use cases.

This is in contrast to the serial approach taken in many formal processes (RUP in particular [16]) where user interface design is performed early, in the inception phase, and that user interface design is then input into a later software design phase. Followed strictly, the serial approach delays software design until the user interface design is complete: for large, user interface intensive projects this can add several months to the time frame of the whole project.

Parallel design is particularly useful for web development, as the front-end interface can be designed separately from the back-end server software: each could be outsourced to different vendors with suitable expertise. Such parallel development does require some kind of iterative project management to coordinate changes and elaborations to the use cases as each design proceeds.

## 7. RELATED WORK

Improving software development is ongoing research by many people. Here, we concentrate on how these various efforts deal with the requirements-design nexus.

Object-Oriented Software Engineering (OOSE) provides some advice on how to relate objects in an object model to use cases [17]. Specifically Jacobson et al. describe how to develop an analysis model by identifying the Interface (now generally referred to as Boundary), Control, and Entity objects needed to realize the use case. While this is useful advice, it does require some interpretation to follow, particularly in assigning behavior to the resulting classes. Our approach is more operational in this regard.

As others before us have have observed, use cases are a convenient way to organize the requirements to better match an object-oriented design [18]. As we discussed in section 2, Jacobson introduced the use case concept [17] and Constantine and Lockwood developed essential use cases as a modification of this original idea for use in user interface design [9]. There has been much more written about use cases and their use in software development. Cockburn's recent book provides a detailed account of how to write use cases, and includes a good summary of the different styles of use cases [8]. Armour and Miller discuss how use cases can be

discovered and managed as part of the requirements gathering process[2]. However, these works say little on how the use cases are then fed into the analysis and design phases. Discussions that do relate use cases to design generally advocate the development of domain models (for example [22]). The domain model then provides the first step in the development of the design. We believe the development of a domain model is important, as we have discussed above. But, we believe there is still a large step between identifying the requirements and determining how the domain model satisfies those requirements.

The importance of traceability is widely discussed in the literature. Pfleeger identifies both vertical and horizontal forms of traceability [21]. Vertical traceability refers to identifying relationships within models in the development process, whereas horizontal traceability refers to relationships between models. There is practical evidence to show that emphasizing traceability does improve the quality of aspects of the software development process and reduces maintenance costs [19].

Our work addresses the horizontal traceability between requirements and design artifacts. In our approach, the relationships between the requirements and the design correspond directly to the decisions on how to distribute the responsibilities. This means that the traceability is ensured as part of the design process.

Any software development effort follows some kind of overall process. One of the most talked about processes of late is the Rational Unified Process (RUP) [16]. RUP is a use case driven process, meaning use cases impact each phase of the process. Use cases are as important to our approach as they are to RUP. We differ in the choice of form of use case. Our use of essential use cases is not inconsistent with any of the RUP descriptions, and in fact can be regarded as a refinement of the Analysis and Design work-flow. This work-flow is based on OOSE, which we addressed above.

Another software development process is OPEN (more accurately, a framework for software development methods, but we concentrate on the process part of it). Of particular interest is the OPEN Toolbox of Techniques [14], which provides a comprehensive survey of techniques that can be used to accomplish various tasks needed for software development. There are several techniques the can be used to produce a design model from the initial requirements, including: Collaborations Analysis, CRC card modeling, Delegation Analysis, Domain analysis, Generalization and inheritance identification, and Transformations of the object model. Any of these techniques can be used in conjunction with our approach, and again, the key contribution of our approach is more operational guidance and the direct provision of traceability.

Recently there has been a lot of interest in lightweight processes such as Extreme Programming (XP) [3]. XP's rules on design include keeping the design as simple as possible for the required functionality, and not adding functionality before it is needed. XP is a strongly incremental process that takes individual "user stories" (like use cases but written completely concretely) and then incrementally refactoring an extant program to implement those use cases. Some parts of the underlying dynamic of our approach and XP are similar, such as the idea that all design can be seen as refactoring. In XP, design happens when the minimal program is extended to support a new use case, then refactored so that

it is well designed. In our approach, the design begins with a complete system object that is imagined to implement the entire program, and *all* refactorings preserve the responsibilities of this object and simply redistribute them amongst its internal components.

## 8. CONCLUSIONS

Use cases are seen as beneficial in many aspects of system development, and the refinement of *essential* use cases was originally made to address needs in user interface design. We have explored the application of essential use cases in object-oriented system development, and in this paper have reported on our findings.

Essential use cases strike the right level of abstraction to facilitate simple and rapid progress in determining requirements. They support communication well through role-play, and help to determine system boundaries. They are brief, easy to learn, and can be developed quickly because they avoid unnecessary debate about implementation details. They also make it easy to detect use case patterns. The level of abstraction works well for both user interface and system design, and allows the two to proceed in parallel.

Essential use cases identify system responsibilities, and these responsibilities can play an empowering role in linking requirements to object-oriented design. In design, responsibility is often used as a heuristic to apportion abstract behavior among collaborating objects. Using essential use cases to identify requirements, together with a responsibility-driven approach to design, leads both to better operational guidance in design, and explicit traceability between design and requirements.

## 9. REFERENCES

[1] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] Frank Armour and Granville Miller. *Advanced Use Case Modeling: Software Systems, Volume 1*. Addison-Wesley, 2001.

[3] Kent Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.

[4] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 1–6, 1989.

[5] David Bellin and Susan Suchman Simone. *The CRC Card Book*. Addison-Wesley, 1997.

[6] Robert Biddle, James Noble, and Ewan Tempero. Patterns for essential use cases. In *Proceedings of KoalaPLoP 2001*, 2001. To appear.

[7] Marc Bradac and Becky Fletcher. A pattern language for developing form style windows. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design*, volume 3, pages 347–358. Addison-Wesley, 1988.

[8] Alistair Cockburn. *Writing effective use cases*. Addison-Wesley, 2001.

[9] Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design*. Addison-Wesley, 1999.

[10] Martin Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[13] Neil Harrison, Brian Foote, and Hans Rohnert, editors. *Pattern Languages of Program Design*, volume 4, chapter Part 7: Patterns of Human-Computer Interaction, pages 445–593. Addison-Wesley, 2000.

[14] Brian Henderson-Sellers, Anthony Simons, and Houman Younessi. *The OPEN toolbox of techniques*. Addison-Wesley, 1998.

[15] Ivar Jacobson. *The Object Advantage : Business Process Reengineering With Object Technology*. Addison-Wesley, 1995.

[16] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[17] Ivar Jacobson, Mahnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[18] Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 2000.

[19] Mikael Lindvall and Kristian Sandahl. Practical implications of traceability. *Software—Practice and Experience*, 26(10):1161–1180, October 1996.

[20] Pete McBreen. When are use cases done? OOTips web site at `http://ootips.org/use-cases-done.html`, January 1998.

[21] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 1998.

[22] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: A practical approach*. Addison-Wesley, 1999.

[23] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[24] Nancy Wilkinson. *Using CRC Cards - An Informal Approach to OO Development*. Cambridge University Press, 1996.

[25] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In Norman Meyrowitz, editor, *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 71–75, 1989.

[26] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object Oriented Software*. Prentice Hall, 1990.

[27] Rebecca J. Wirfs-Brock. Designing scenarios: Making the case for a use case framework. *The Smalltalk Report*, 3(3), 1993.

[28] Rebecca J. Wirfs-Brock. The art of meaningful conversations. *The Smalltalk Report*, 4(5), 1994.