A Pragmatic Approach to Use Cases

Jürgen Börstler, Thomas Boman, Katarina Sigerud

UMINF-97.24
ISSN-0348-0542

UMEÅ UNIVERSITY
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN
A Pragmatic Approach to Use Cases

Jürgen Börstler, Thomas Boman, Katarina Sigerud
Department of Computing Science
Umeå University
S–901 87 Umeå, Sweden
e-mail: jubo@cs.umu.se
URL: http://www.cs.umu.se/~jubo/

UMINF-97.24
ISSN-0348-0542

Abstract
Currently the notion of use cases and (task) scripts get much attention in the OOA community. These notions are well-suited to develop an external view of a system, i.e. a description of a system from the user’s point of view. Use cases and (task) scripts serve as useful tools to support requirements capturing, object model derivation, traceability, and test case development. Furthermore they are natural entities for user documentation.

In this paper we describe a pragmatic approach to use cases and how this approach is embodied in the prototype tool RECORD (REquirements COllection, Reuse, and Documentation). RECORD provides a form-based user interface for stakeholders to input their requirements. These forms are analyzed to generate initial object models, which can be used to query a repository for matching object models. In addition we produce a set of hypertext documents documenting the use cases, their interrelationships, as well as their relationships to the generated and/or reused object models. Goal of the whole process is to produce a hypertext based requirements model, supporting the traceability between external and internal views of the system.

Most of the system is still under development and our current prototype covers only parts of the system.
1 Introduction

The notion of use cases was developed to build an external view of a system. Each use case describes a “... behaviorally related sequence of transactions in a dialogue between the user and the system” ([Jac + 92]). From the user’s point of view this is a natural way to describe the (functional) requirements for a system. Most approaches to object-oriented analysis (OOA) and design propose a similar concept, the scenario. Scenarios define possible system executions as time ordered sequences of interactions between objects. The usage of scenarios then allows for a smooth transition from analysis to design. The gap between requirements and design seems to be solved. But there is a subtle difference between use cases and scenarios. Uses cases are described in terms of the user’s terminology, whereas scenarios are described in terms of the systems’ terminology, i.e. objects, messages, etc. So there still is a gap when using traditional object-oriented approaches. To bridge this gap use cases get much attention in current approaches to OOA ([BoRu 95], [JaCh 95], [Reg + 95], [Rum 94]).

There are also object-oriented approaches focusing on a user-oriented view of the requirements. In [RuGo 92] and [Gra 95] for example (task) scripts are used to record the use of the (proposed) system. Use cases and (task) scripts are well-suited to describe a system from the users’ point of view. They serve as useful tools to support requirements capturing, object model derivation, traceability, and test case development. Furthermore they are natural entities for user documentation.

Goal of the RECORD (REquirements COollection, Reuse, and DOcumentation) project is to provide a pragmatic approach to user-centered requirements modeling, object model derivation and reuse, and the generation of traceable documentation.

In the next chapter we give an overview over the approach taken in RECORD. In Chapter 3 we describe how our current prototype realizes these ideas. In Chapter 4 we shortly compare RECORD to related approaches. A conclusion and description of future work concludes the paper.

2 Overview over the RECORD Approach

In RECORD stakeholders play an active role to ensure that the right requirements are got. The RECORD system will then support the process of getting these requirements right, deriving an initial design, and finally documenting them in an appropriate notation.

One important feature of our approach is its user-orientedness. Stakeholders are integrated into the requirements engineering process. This is supported by providing

- an user-oriented requirements model, where stakeholders can input their requirements by themselves, and
- the generation of a hypertext based documentation as a basis for requirements traceability and validation.
Requirements engineering in RECORD proceeds in phases. In the requirements collection phase the stakeholders use forms to input their requirements. In the reuse phase an initial design is derived with the help of previously defined object models stored in a repository. Goal of the documentation phase is then to integrate the requirements model and the initial design into a hypertext based documentation to support traceability. The stakeholders are the central users of a system implementing the RECORD approach. The phases are summarized in figure 1.

![Diagram of RECORD phases]

Figure 1: Overview over the RECORD phases.

2.1 Requirements Collection using Use Cases

From cognitive science and human-computer interaction we know that people appear to understand the world in terms of specific situations ([Car+ 94]). Use cases resemble these ideas. They describe in the terms of the user, how the interaction between the system and a user is accomplished to fulfil a specific task. How the system works internally is only relevant insofar as the systems reactions are visible to the user. Use cases are a valuable tool to acquire requirements from the users of a system, since they correspond very closely to the users’ view of the functional requirements of the system.
The use case mode consists of actors and use cases ([Jac+ 92]). The actors represent everything that interacts with the system (via use cases). The use cases describe how the system is used by the actors. Each use case describes a specific case of system usage. Take for example an access control system (see figure 2), where room access is controlled by a system using card readers and key pads. Typical use cases for such a system would be Open Door, New Code, and Register New Card. Typical actors would be user and system administrator. All actors in our system may open the door and change the code of their own card, but only operators are allowed to register new cards.

A use case can be seen as a black box which contains a description of all potential situations caused by a specific user stimulus. This black-box view makes the use case model very powerful, since it abstracts away all details not related to user stimuli. The model is therefore easy to understand by the stakeholders of the system.

![Access Control System Diagram](image)

Figure 2: Black-box view of the use case model of an access control system.

In RECORD the stakeholders themselves can input requirements. They are therefore provided with a form-based user interface to support structured input of requirements in the form of an use case model. Different kinds of forms are used to collect primary functional requirements and to successively build a unified terminology.

### 2.2 Derivation of an Initial Design

In this step the requirements model is semi-automatically transformed into an initial design model. This is done by an analysis of the use cases to extract objects, relationships between objects, operations, and states or attributes. This step takes advantage of the structure of our use case forms and the information on the vocabulary used in the use case descriptions (i.e. the importance of words and their word classes). The extracted information is then used to create initial object model.

This kind of textual analysis is often criticized since it is not a rigorous approach, oversimplifies the problem, and does not scale up. But there is evidence that textual analysis works quite well for (a) small problem statements, (b) if applied semi-auto-
matically ([SHE 89]), or (c) together with more advanced natural language processing tools ([GrRo 93]). Another big advantage of this approach is its ability to handle problem statements described in natural language. Since use case descriptions are structured and use quite simple sentences only (see example in figure 6), such an approach is used in RECORD to analyze use cases. Furthermore this approach is easy to implement and leads to object models, which can easily be related to the requirements.

This correspondence between use cases and object models is an important step to support full traceability. Stakeholders will now be able to follow the development process as far as traceability is provided throughout the following phases of the software development process ([BôJa 92]).

The components of the initial design will then be matched against a repository of previously defined object models to find reusable components. This step will be supported by means of feature-oriented classification as described in [Bôr 95]. This reuse phase is described in more detail in a forthcoming paper.

2.3 Documentation Generation

Outcome of the documentation phase are an easily accessible requirements document, an initial design document, and the traceability information linking both documents. We therefore produce a set of interlinked documents.

For each use case a HTML version is produced in a straightforward way. HTML documents have the advantage that we can introduce links to definitions in the dictionary, related use cases, and corresponding object models. They are therefore a natural tool for the implementation of traceability. Furthermore are HTML documents easily accessible on all computer configurations. This makes it very easy for the stakeholders to validate their requirements. In our example a stakeholder can easily check if the reaction to the exception Card invalid in our example use case Register New Card (see figure 6) is what she had in mind by following the link associated with this exception. Each use case is also linked to all corresponding object models (and vice versa). These links are the basis for traceability. If a dictionary entry, a use case, or an object model is modified it is very easy to identify all documents, which are potentially affected by this change. The simplicity and ease of use of such a document makes it a useful tool during the whole development process.

3 Prototype Implementation

Our very first prototype for the collection phase was implemented using HTML forms, so it could be used together with a WWW browser. All modifications to the use case model had to be inserted, submitted, and reloaded to see the effects of a change. This turned out to be very awkward for the users. The current prototype is implemented using Borland Delphi for Windows and has a traditional graphical user interface. The prototype supports the following tasks:
Figure 3: Summary form.

- Creation and modification of use cases.
- Manual creation and modification of the unified terminology.
- Creation and modification of relationships between use cases (see section 2.1).
- Generation of hypertext documentation in HTML.
- Generation of a simple text-based object model.

All steps in the current prototype are implemented straightforward. The forms used are text-based and have a common look and feel. The reuse of existing object models is currently not supported. All object models are created from scratch by means of textual analysis of the use case forms. This step takes advantage of information on the vocabulary used in the use case descriptions (i.e. importance of words and word classes). In our current prototype this information must be added manually. The
resulting object model describes all classes identified by a name, a short textual
description required operations and attributes, and relationships to other classes. The
prototype is described in more detail in [BoSi 96].

Figure 3 shows the main window of the current prototype. From here all the
tasks listed above can be initiated. In this form all available actors and use cases are
listed. Detailed information on the listed actors and use cases is available on selection.

3.1 Requirements Collection

Requirements can be manipulated via the actor, use case, or dictionary forms. In an
actor form (see figure 4) only the Name: and Description: fields are editable. The field

![Image of an actor form]

Figure 4: Example of an actor form.

Related Use Cases: contains all use cases which refer to the current actor in their func-
tional descriptions. This field is updated by the system, when actors are inserted or
changed in existing newly created use cases. The Description: field may contain arbi-
trary text, which is used later on for documentation purposes.

The use case form is the most interesting and complex part of the prototype. We
support abstract and concrete use cases. Abstract use cases allow us to share common
parts between different (abstract and concrete) use cases. Abstract use cases cannot
stand on their own, they only make sense as a part of an concrete use case. Actors can
only communicate with the system via concrete use cases (see [Jac+ 92]).
In our example system there may be several use cases where an operator mode has to be entered first, for example to register a new card. *Supervisor Mode* is an example of such an abstract use case (see figure 5). The *Description*: field contains the sequence of interactions between the *system administrator* and the system. In the field *Related use cases*: the system automatically lists all use cases that share this abstract use case. Deviations from the usual sequence of interactions between an actor and the system exception all use case forms provide an *Exceptions*: part. See figures 6 and 7 for more information on exceptions.

Figure 6 shows an example of a concrete use case sharing the abstract use case *Supervisor Mode*. The description of this use case is already textually analyzed. Classified words in its *Description*: field appear in curly brackets. Classification can be done by means of the buttons beneath the *Description*: field. We currently support the word categories *Word*, *Actor*, *Abstract*, and *Concrete*.

*Word* is reserved for words which are neither an actor nor an abstract or concrete use case. They can be further classified as being a proper noun (for example cardread-
Figure 6: An example of a concrete use case.

...er), a verb phrase (for example enters), or an adverb phrase. Actor denotes actors. Abstract and Concrete refer to shared use cases. These can be abstract use cases as described above or extended (concrete) use cases. An extended use case is a concrete use case, which is extended with additional steps in its description and/or additional pre-/postconditions or exceptions.

Pre- and postconditions are currently arbitrary text describing the state before and after the application of an use case. The Summary: field contains a short textual description to aid stakeholders in getting a quick overview over the system. The Contact address: field will later be used to automate change notification and may contain arbitrary text to handle messages.

Exceptions are described in a use case like form (see figure 7). The form shows where the exception can be raised (From: field) and where the raising use case has to be resumed after completion (To: field). To support independence and reuse of exception forms this information will be moved into the forms raising the exception. Since exceptions may affect the post conditions of use cases the form provides a Post Conditions:
field to allow for the description of differing use cases. The default for this field is *Same*, i.e. the original post condition is not affected. Since exceptions may raise new exceptions this form contains an *Exceptions* field too.

### 3.2 Derivation of an Initial Design

The generation of an initial object model is preceded by an analysis of all use cases. This analysis identifies objects, operations, etc. by means of word classes and relationships between use cases and actors. In the current prototype this classification is partly done manually and partly by a simple textual analysis of the use case actions to identify subject, predicate and object of a sentence. Classified words appear in curly brackets in the *Description:* field of the use case form (see figure 6).
The cardreader can read the type of smartcards that the access control system is designed for.

The cardreader is connected to the door lock and the door sensor. It is also connected to the main computer.

Methods

- `enters_card` used by `system_administrator` trace `Register_new_card`
- `validates_card` used by `system` trace `Register_new_card`
- `enters_card` used by `employee` trace `New_code`
- `enters_card` used by `employee` trace `Open_door`
- `enters_card` used by `system_administrator` trace `Unregister_card`
- `validates_card` used by `system` trace `Unregister_card`

Attributes

- card
- system

Uses

Has

- card

Figure 8: Documentation of the generated object model.

Currently we proceed as as described in [SHE 89], i.e. in a typical sentence describing a use case action the subject can be interpreted as the object sending a message (the predicate) and the object of the sentence can be interpreted as its receiver. Consider actions 2, 3 and 4 of our example use case from figure 6. We get the objects `system administrator`, `cardreader`, `system`, and `database` from the subject and object of the actions. From the predicates of the actions we get the verb phrases `enters_card`, `validates_card`, and `registers_card`, which are transformed into methods in the object class. See figure 8 for an example of the class description for `cardreader`. For better information we added the source of this information to the `Methods` section of the class documentation. All direct objects (i.e. the nouns in the compound method names) are added to the `Attributes` section. If there exists a class description for the direct object,
it is also added to the Has section. The System entry in cardread Attributes section derives from another use case.

Although use case analysis is manual, the object model is created without user interaction. This makes it necessary to polish the object model, since misspellings and varying formulations result in the repetition of (mainly) methods and attributes. Currently we produce simple text files to store the object model. These are then translated to HTML documents and interlinked with other information (see figure 8 for an example). This makes it very easy to detect peculiarities in the generated information.

### 3.3 Documentation Generation

In the documentation phase we produce an interlinked set of HTML documents, which make the requirements model as well as the initial design easily accessible.

Figure 9 shows the index page to the generated information. TREAD is the acronym for the current prototype of our Tool for Requirements Elicitation and Documentation (see [BoSi 96]). We generate HTML documents for all actors and use cases, the dictionary and the object model. The object model is structured according to the classes identified during use case analysis. In the current prototype we generate one HTML document per class (see figure 8 for an example).

### 4 Related Approaches

There are very few approaches that are user-oriented and support all phases of the requirements engineering process. Notable exceptions are AIR ([Mai+ 94]) and CARE/ALECSI ([GrRo 93]).

AIR (Advisor for Intelligent Reuse) supports guided and unguided user input. Input can be formal (ERDs) or informal (hypertext). The AIR system uses a knowledge base of predefined abstractions. These abstractions are empirically verified domain models of restricted application domains (patterns). Guided input is connected to an analogical reasoning engine to find matching knowledge. Critiquing of requirements is also supported.

Informal input is done by fill-in-the-blanks forms, where attributes are queried to discriminate between domain models. How development is supported beyond requirements acquisition is unclear.

In CARE (Computer Aided Requirements Engineering) natural language sentences are analyzed according to Fillmore’s “case for case” theory. The knowledge extracted is used to build a real-world model in terms of actors, events, etc. This real-world model is then refined into a semantic network. User inputs are interpreted as facts. Transformation and refinement knowledge is stored as rules in a knowledge base. The semantic network is finally transformed into an object-oriented conceptual schema.
Actors

- Visitor
- Employee
- Security guard
- System administrator
- Door sensor

UseCases

- Open door
- Register new card
- Door alarm
- New code
- Unregister card
- Supervisor mode

Object Model

Figure 9: Summary of the generated HTML documentation.

5 Conclusion and Future Work

We have presented a pragmatic approach to user-oriented requirements engineering using use cases. We have shown that use cases can serve as a basic tool for requirements collection, analysis, reuse, documentation, and traceability. Since they are easy to understand by stakeholders they are a natural choice for user-oriented approaches to software development. The prototype described, even in its current simple form, can be a valuable tool for requirements capturing, analysis, and traceability.

We have also built a separate prototype supporting the classification and matching phase, but is not integrated with the RECORD prototype. For the next prototype
we are planning to integrate the two prototypes and to automate parts of the dictionary construction and and object model generation by means of natural language processing tools. We are furthermore searching for a suitable tool to support graphical object modeling, which will also be integrated into the prototype.

References


