Softw Syst Model (2010) 9:141–160
DOI 10.1007/s10270-009-0112-9

REGULAR PAPER

Improving the quality of use case models using antipatterns

Mohamed El-Attar · James Miller

Received: 28 May 2008 / Revised: 30 December 2008 / Accepted: 13 January 2009 / Published online: 15 February 2009

Abstract Use case (UC) modeling is a popular requirements modeling technique. While these models are simple to create and read; this simplicity is often misconceived, leading practitioners to believe that creating high quality models is straightforward. Therefore, many low quality models that are inconsistent, incorrect, contain premature restrictive design decision and contain ambiguous information are produced. To combat this problem of creating low quality UC models, this paper presents a new technique that utilizes antipatterns as a mechanism for remedying quality problems in UC models. The technique, supported by the tool ARBIUM, provides a framework for developers to define antipatterns. The feasibility of the approach is demonstrated by applying it to a real-world system. The results indicate that applying the technique improves the overall quality and clarity of UC models.

Keywords Use cases · Antipatterns · UML · Use case modeling qualityattributes · OCL

1 Introduction

The unified modeling language (UML) [11,35] has been widely accepted as the de-facto language for producing software blueprints [9,24,25,28,37,40]. Use case (UC) modeling, part of the UML framework, is a popular method to capture a system's functional requirements [1,2,5,16,22,25,32,44]. UC models are comprised of unstructured text and diagrams that adhere to a small notational set. This simplicity allows all stakeholders, including non-technical stakeholders, to fully understand the models allowing everyone to share a common understanding of the system being modeled.

The notation and guidelines for creating UML artifacts, including UC models, are clearly defined in [35]. However, mechanisms to construct semantically correct and verifiable diagrams are not discussed [35]. Therefore, UC models are vulnerable to inappropriate techniques and decisions. UC modeling mainly occurs in the early stages of development and hence influences the construction of other UML artifacts. Defects in a UC model are likely to propagate through to later development phases where the cost of repairing defects escalates [10,20,48].

A UC model must accurately represent an analytical view of a system's functional requirements. This is the principal research problem addressed in this paper. The proposed approach aims to tackle this issue by searching for the existence of antipatterns in UC models. Antipatterns represent debatable diagrammatic and textual structures. Their detection prompts a "review" of the debatable structures to either undertake corrective actions or to verify their correctness. The effectiveness of our proposed technique is dependent on the original state of a UC model. Applying our proposed technique will help bring a given UC model into a form that more accurately represents its system's functional requirements, ideally yielding a flawless UC model.

The remainder of this paper is structured as follows: Sect. 2 provides a brief background on UC modeling. Section 3 outlines the proposed approach and presents a taxonomy of 26 UC modeling antipatterns. In Sect. 4, we introduce ARBIUM and explain how it provides automated support for the proposed approach. A real world case study is presented in

Communicated by Prof. Robert France.

M. El-Attar (✉) · J. Miller
STEAM Laboratory, Electrical and Computer Engineering Department, University of Alberta, Edmonton, AB, Canada
e-mail: melattar@ece.ualberta.ca

J. Miller
e-mail: jm@ece.ualberta.ca

Sect. 5 to demonstrate the effectiveness of our approach. Section 6 concludes and discusses future work.

2 Background

A UC model consists of three main components: a diagram, textual descriptions and a glossary. The diagram provides a visual summary of the services offered by a system and its interaction with its environment (actors). The textual descriptions explain the interactions between a system and its actors. The glossary is used to remove namespace ambiguities.

2.1 Related work on UC model quality improvement

Many researchers and practitioners have devised techniques to improve the quality of UC models. The following is a brief summary of their approaches:

2.1.1 Computer-supported verification of UC models—state of the art

Berenbach [7] describes a set of software-supported (Design-Advisor) heuristics to create large verifiable analysis models. This approach can be highly restrictive as many organizations only use a subset of UML; moreover, many organizations have procedures that utilize in-house design heuristics. These restrictions are resolved by ARBIUM, the tool presented in this paper. ARBIUM provides support for analysts to define and verify their own heuristics in addition to being equipped with a set of predefined rules that are applicable to any UC model. The antipatterns defined in this paper encompass all of the heuristics presented in [7] that pertain to UC modeling. It is believed that this approach, Berenbach [7], presents the current state of the art in computer-supported verification of UC models; and hence, this approach will be compared against our approach in the case study presented in Sect. 5. The heuristics in [7] will be presented in Sect. 5 and the antipatterns that embody these heuristics will be identified.

2.1.2 Other approaches

The antipatterns developed in this paper are based on widely accepted guidelines and practices, such as those presented in [1,4,6,8,9,13–19,23,26,27,29–33,38,40–42,44]. Hence, the work presented in this paper should be regarded as building upon foundations laid by others. However, most of these pre-existing guidelines are informal and are provided at a very abstract level. In this section, we will briefly outline other related work which tackles the identified problem.

UC modeling inspection technique developed in [3], based upon recommendations provided in [5,32,44], is focused on textually-oriented domain-dependent defects in UC models. In order to effectively apply these guidelines and inspection techniques, a great deal of UC modeling expertise is required and therefore these techniques will not be evaluated in Sect. 5. Linguistic techniques [18,32] and tools [32,39,43] do not perform any verification upon the semantics of the UCs and their relationships. However, UC modeling semantics are carefully considered when developing antipatterns and applying our technique. UC refactorings [12,38,39,42,49] were developed to address simple defects in UC models. The refactorings are based on simple heuristics which can be found in a small subset of our antipatterns. Ryndina et al. [43] developed a computer-supported approach to verify UC models. However, the approach does not support the basic UC modeling syntax defined in [35]; specifically (a) all types of relationships amongst UCs, (b) the generalization relationship between actors and (c) multiple actor associations with a single UC. ARBIUM is designed to support these basic UC modeling notations.

It should be noted that it is not necessary to apply the antipatterns technique exclusively. In fact, we recommend that other approaches should be used in addition to using antipatterns. The resulting UC models will be of higher quality in comparison to using any approach exclusively.

2.2 Quality attributes of UC models

In order to develop high quality UC models, it is required to identify the quality attributes that should exist in UC models. Table 1 provides a summary of the most important quality attributes as stated by [2,5,6,9,14–16,22,27,43]. Our technique aims to improve the overall quality of a UC model by improving these attributes.

3 UC modeling antipatterns

This paper focuses on deficiencies that require human cognition to verify. Therefore, the approach can be characterized as "risk-based", meaning that a "poor" UC modeling structure does not necessarily indicate that a defect certainly exists; rather it indicates that the structure in question may lead to potential defects. In this section, we describe a new technique to find these situations in UC models. The final judgment, with regard to correctness, can only be taken by a domain expert. The proposed quality improvement technique is based on identifying modeling practices that are likely to lead to harmful consequences. While it is impossible to formally analyze the unstructured Natural Language (NL) found in textual descriptions, UC diagrams can be formally analyzed due to their adherence to a rigorous syntax [35]. Therefore, an informal review process will be required to analyze

Table 1 Quality attributes of a UC model

Quality Attribute	Definition	Implications of absence
Correctness	The UC diagram along with textual descriptions must accurately represent the requirements.	May lead to a faulty system.
Consistency	All components of the UC model must conform to the same concepts and represent a consistent view of the requirements.	May lead to a faulty system.
Analytical	The model should describe a system's requirements with respect to: what it needs to achieve; instead of how it achieves it. The model should not contain any interface details, implementation or design decisions.	Restricts the designers' creativity and prevents them from devising an optimal solution. It becomes unclear what services the system actually offers.
Understandability	The model must be unambiguous and readable. All stakeholders must be able to understand it and commonly agree upon the presented functional requirements.	May lead to a faulty system. The creation of test cases that don't test the actual requirements. The project may experience delays and cost overruns.

textual descriptions, while inappropriate design decisions in UC diagrams can be formally detected.

To effectively apply this approach, a repository of (anti) patterns which articulate poor UC modeling habits and decisions is required; our initial repository is described in Sect. 3.5. An advantage to this approach is that it can be applied in the early phases of the development cycle where UC models are often incomplete.

3.1 Advantages of using antipatterns: what can antipatterns do?

Learning from previous experiences and mistakes is the main concept behind using antipatterns. An antipattern explains why a given structure may cause deficiencies in a UC model. An antipattern will also provide a detection mechanism to guide modelers to areas in the UC model where an antipattern may exist, be it in the UC diagram, the descriptions, or both. Most importantly, an antipattern will explain why such a debatable structure seemed appropriate in the first place. Finally, an antipattern provides suggestions upon improving the current structure to avoid potential consequences. Basically, an antipattern provides key information to guide modelers from a fallacious solution to a superior solution [47]. Table 2 shows the antipattern template used in this paper. The purpose of each field is described briefly in Table 2 and in more detail in Sect. 3.2.

3.2 Matching antipatterns with UC models

As mentioned earlier, poor modeling decisions may exist in the UC diagram, the descriptions, or both. The "Detection" section in an antipattern contains detailed guidelines to match the antipattern. For poor modeling decisions that exist in UC diagrams, an antipattern will outline a set of diagrammatic elements that represent a debatable structure. Detecting a match for such antipatterns can be achieved by juxtaposing the antipattern's stated unsound diagrammatic structure with the actual UC diagram. As for poor decisions that exist in textual descriptions, the "Detection" section will guide analysts to particular field(s) of a UC template where an antipattern match can be detected. If an antipattern is matched; the analysts are then required to verify the correctness of the UC model.

Upon reviewing an antipattern match, corrective measures may be required. If corrective measures were undertaken, this may consequently eliminate previously detected antipattern matches that have not been reviewed. Alternatively, undertaking corrective measures may cause new antipatterns to surface. Therefore, the antipattern matching process must be performed iteratively until all antipattern matches have been addressed.

3.3 Using OCL to describe unsound diagrammatic structures

Unsound structures described in NL are inherently ambiguous. Ambiguity can be eliminated by describing unsound diagrammatic structures referred to by antipatterns using OCL constraints [46]. During the matching process, if the constraints were not satisfied, then an antipattern match is detected. Wherever possible, antipatterns will be augmented with OCL statements to automate or semi-automate their detection.

Traditionally, OCL statements are used to describe constraints in class diagrams or object models. In order to describe diagrammatic UC structures using OCL, the UC diagram must be transformed to an object model. This is possible since every instance of a UC diagram conforms to the metamodel provided by OMG [35]. Each element in a UC diagram maps onto one or more metaclasses. However, it is clearly impractical to expect analysts or domain experts to study hundreds of pages of documentation explaining thousands of metaclasses, most of which are not exclusive to UC diagrams, in order to construct their OCL statements. To increase the accessibility of our approach, a simplified

Table 2 An antipattern template

Antipattern name: The title of the antipattern.
Description: A description of the faulty decisions or techniques.
Rationale: A list of the deceptive or seductive reasons as to why the fallacious solution seemed to be appropriate.
Consequences: A list of the harmful consequences that could be sustained from applying the fallacious solution.
Detection:
Where—A guide to the areas where the antipattern can exist.
How—Instructions that are used to positively identify a match for the antipattern.
Improvement: A list of actions that can be performed to convert a fallacious solution into a superior solution or avoid the fallacious solution.

metamodel was created (see Fig. 1), which contains only four classes and a limited number of associations linking these classes together. All these metaclasses are exclusive to UC diagrams. The simplified metamodel does not need to support the entire notational set of UC diagrams. The smaller metamodel will encourage the adoption of the metamodel by analysts and minimize the learning curve while supporting the notational subset most commonly used, and which encompasses most UC diagrams. The metamodel can easily be extended to support any additional notation required.

The metaclasses shown in Fig. 1 represent actors, UCs, the association relationship, the *generalization* relationship (both between actors and UCs), abstraction, the *include* relationship, the *extend* relationship and extension points. The following is a brief description of the metamodel elements:

- Instances of the UseCase and Actor classes are assigned names using the name attribute, and a Boolean abstract attribute that indicates whether they are *abstract* or *concrete*. The extensionLocation attribute of the ExtendsAt association class is used to state the extension point to which an *extend* relationship link is referring.
- For the ExtendsAt association class, the extensionUC role indicates that the *extension* UC *extends* the *base* UC. The *extended* extension point is referenced by the extensionLocation attribute. The ExtendsAt relationship is required since the extension point referred to is a property of the *extend* relationship. The *base* UC is specified using the base role.
- An *extend* relationship that does not refer to an extension point is indicated using the Extends association. For the Extends association, the extension role indicates the *extension* UC. Meanwhile, the *base* UC in turn is specified using the base role.
- An *include* relationship is specified using the Includes association. For the Includes association, the inclusion role indicates the *inclusion* UC being included by a *base* UC which is in turn indicated by the base role.
- The *generalization* relationship between UCs is supported by the Specializes_use_case association. The Specializes_use_case association has one UseCase object assigned the parent role, while another UseCase object is assigned the child role.
- The *generalization* relationship between actors is supported by the Specializes_actor association. The Specializes_actor association has one Actor object assigned the parent role, while another Actor object is assigned the child role.
- The Associated_With association represents an association relationship between an actor and a UC. The actor end of the association relationship is assigned the actorEnd role, while the UC end of the relationship is assigned the useCaseEnd role.
- Directed associations are represented with the DirectedAssociation association class, the directedActorEnd role indicates the actor involved in the association, while the directedUCEnd role indicates the UC involved in the association. The String attribute directTowards can be set to either "UC" or "Actor" to indicate where the association link is directed towards.

Automated support is available to examine diagrammatic constructs using OCL and the above metamodel. Unfortunately, examination of textual descriptions remains a manual process. The tool supported antipatterns shown in Sect. 3.5 are augmented with OCL statements whenever possible to automate or semi-automate their detection.

3.4 Domain independent versus domain dependent antipatterns

Antipatterns can either be domain-independent (DI) or domain dependent (DD). DI antipatterns make no assumptions about the underlying domain and hence are applicable to any UC model. Researchers can derive DI antipatterns by understanding the semantics of the UC modeling notation and the purpose behind each component of a UC model. DD antipatterns represent additional, specialized antipatterns which seek to encode an organization's specific objectives for a specific project or domain. Analysts should collaborate with domain experts to develop DD antipatterns. Using OCL

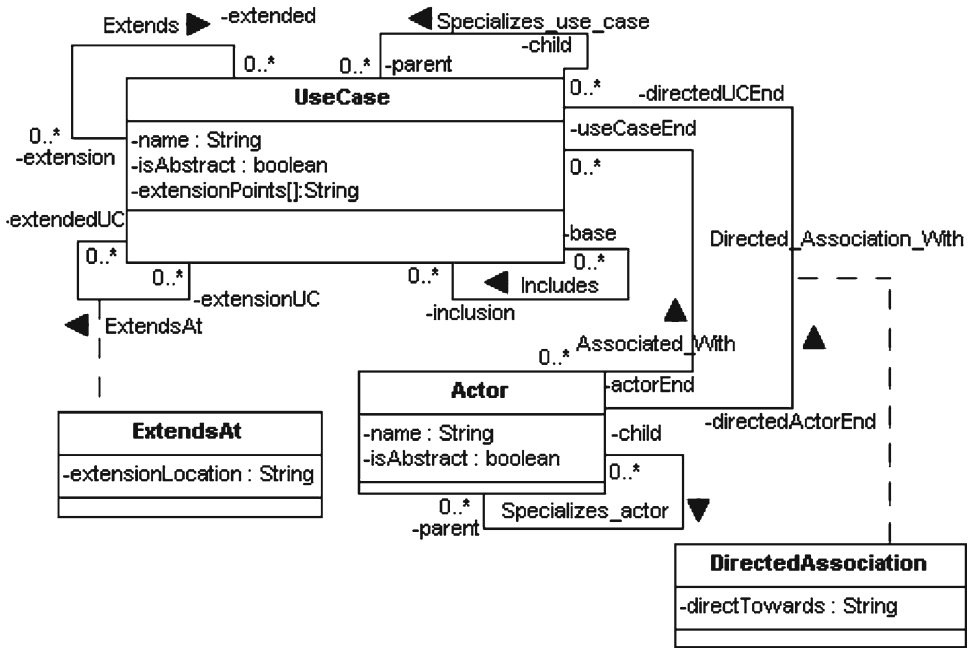


Fig. 1 The simplified version of the UC metamodal used in ARBIUM[1]

and the simplified metamodel, analysts can quickly define DD antipatterns.

Antipatterns can be further subdivided—with respect to their suitability of being machine readable (see Table 3). The principal advantage of antipatterns that are machine readable is that they can be (semi-)automatically matched. Diagrammatic structures are described using OCL as a set of constraints, which is used by ARBIUM to perform the matching process. The process of matching textual patterns cannot be automated as unstructured natural language cannot be formally analyzed. Therefore, a review process is required to detect matches for textual patterns described by antipatterns. The availability of tool support to match an antipattern is dependent on the information provided in its "Detection" section.

Type (1): This paper mainly focuses on this type of antipattern. A large number of Type (1) antipatterns are presented in detail in Sect. 3.5. The detection process of these antipatterns can be fully automated as they only require analysis of UC diagrams.

Type (2): For this type of antipattern, ARBIUM can be used to detect the diagrammatic structure described by an antipattern; subsequently a review process is required to analyze the corresponding textual descriptions.

Type (3): For this type of antipattern, the review process needs to be conducted manually since these antipatterns require the examination of textual descriptions. No automation support can be provided to detect this type of antipattern. A number of Type (3) antipatterns are described in Sect. 3.5.

Type (4): These DD antipatterns are machine-readable. Analysts use the simplified metamodel to compose OCL statements that describe the debatable construct. Analysts will need to collaborate with domain experts to develop these antipatterns.

Type (5): Similar to Type (4) antipatterns, analysts will need to collaborate with domain experts to develop this type of antipatterns. Unlike Type (4) antipatterns, Type (5) antipatterns require a review or inspection of textual descriptions in addition to the UC diagrams.

Type (6): DD antipatterns are again developed through a collaborative effort between analysts and domain experts. A manual review process needs to be performed in order to match this type of antipatterns as they require examination of textual descriptions.

3.5 Examples of UC modeling antipatterns

In this section we present a number of UC modeling antipatterns. Due to space limitations, an abridged version of a subset (8 out of 26) of the current repository of antipatterns is presented. Also due to space limitations, antipatterns "a3.", "a4." and "a5." are clustered as they address the same issue and hence contain a number of similar components. A detailed description of all antipatterns, including elongated versions of the antipatterns presented below, augmented with

[1] Some elements of the metamodel are present to more effectively benefit from the features provided by USE (UMLbased Specification Language), which is used in addition to ARBIUM to automate the detection process (Sect. 4)

Table 3 Types of antipatterns

Situation	Full automation support available	Semi-automation support available	No automation support available
Domain-independent Antipatterns	(1)	(2)	(3)
Domain Dependent Antipatterns	(4)	(5)	(6)

Table 4 Antipatterns presented in this Section

Antipattern Name	Comment	Automation Support
a1. Accessing a *generalized concrete* UC [9,35,37]	Abridged	Type (1)
a2. UCs containing common and exceptional functionality [1,7–9,11,12,35,37]	Abridged	Type (1)
a3. Functional decomposition of UCs using the *include* relationship [9,16,35]	Abridged and clustered	Type (1)
a4. Functional decomposition of UCs: using the *extend* relationship [35,37]		Type (1)
a5. Functional decomposition of UCs: using pre and postconditions [9,14,15]		Type (3)
a6. Accessing an *extension* UC [35,37]	Abridged	Type (1)
a7. Multiple actors associated with one UC [5,11,14,15,30,35]	Abridged	Type (1)
a8. A description of an actor that is not depicted in the UC diagram [1,14,15,30,41]	Abridged	Type (2)

examples, is presented in [45]. The antipatterns presented in this Section are listed in Table 4. In Table 4, each antipattern name is followed by literature references of which the antipattern is based upon.

- **Antipattern name**

a1. Accessing a *generalized concrete* UC—**Automation Support: Type (1)**

- **Description**

A family of UCs that represent a framework of services offered by a system can be defined using the *generalization* relationship. Modelers can define a hierarchy between the UCs using the *generalization* relationship. To access this framework of services, an actor is associated with *generalized* UCs to indirectly access the services offered by this hierarchy of UCs.

- **Rationale**

An association can be created between an actor and a *generalized* UC for two reasons:

(1) The *generalized* UC contains behavior that individually is useful to the actor.
(2) The operational mechanisms of the *generalization* relationship in UC diagrams are similar to that of class diagrams. Therefore, modelers may utilize the concept of polymorphism in their UC model. Hence, when an actor initiates a *generalized* UC, the service request can be delegated to one of its *specializing* UCs.

- **Consequences**

Often *generalized* UCs contain fragments of general behavior that are completed by their *specializing* UCs. Therefore, *generalized* UCs are often incomplete. If a *generalized* UC is *concrete*, it can be standalone as a complete UC that can be exclusively initiated. However, if an actor makes an exclusive initiation request to such a *generalized* UC, incomplete behavior will be executed.

- **Detection**

Where—Search for any *generalized* UCs in the "UC Diagram". **How**—If the *generalized* UC is *concrete*[2] and associated with an actor.

OCL Description:

```
  context UseCase
  inv AccessingGeneralizedUseCaseBy-
Actor:
  not ((not (self.isAbstract)) and self.
actorEnd->size > 0 and self.child->size
>0)
```

[2] *Concrete* UCs are labeled using a regular font. Meanwhile, *abstract* UCs are labeled using an *italic* font.

- **Improvement**

(1) Set the *generalized* UC to be *abstract*. Unlike *concrete* UCs, *abstract* UCs cannot be initiated.
(2) Remove the association relationship between the actor and the *generalized* UC and replace it with explicit associations between the actor and the *specializing* UCs. Associations with the *specializing* UCs will force a service request to be performed through one of the *specializing* UCs. Hence, the *generalized* UC will not be exclusively initiated.

- **Antipattern name**

a2. UCs containing common and exceptional functionality—**Automation support: Type (1)**

- **Description**

The reuse of a UC is optimized by making it both an *extension* and an *inclusion* UC.

- **Rationale**

Object-oriented modeling and design strongly promotes the concept of reuse. Modelers are keen to reuse much of the functionality contained in preexisting UCs. Reusing UCs prevents the cluttering of the UC model with many redundant UCs. However, when applying the concept of reuse, the *include* and *extend* relationships can be misused, leading to the creation of UCs containing both common and exception-handling behavior.

- **Consequences**

The shared UC currently contains common and exceptional behavior required by the two *base* UCs. Therefore, when either *base* UCs initiate the shared UC, additional undesired functionality is performed.

- **Detection**

Where—Search for any *inclusion* UCs in the UC diagram.
How—If the *inclusion* UC is *extending* other UCs.
OCL description:

```
Context UseCase
inv UsingIncludeAndExtendToImplement-
AbstractUC:
not ((self.isAbstract) and (self.
inclusion->size > 0 or
self.extension->size > 0 or self.
extensionUC->size > 0))
```

- **Improvement**

(1) Check if the shared UC contains functionality suitable for only one of the *base* UCs. This can be achieved by examining the contents of the shared UC.
   - If the shared UC contains functionality suitable only for the *base* UC that *includes* it, then the *extend* relationship should be removed. A new *extension* UC should be created to handle the exceptional situation generated by the other *base* UC.
   - If the shared UC contains functionality suitable only for the *base* UC that it *extends*, then the *include* relationship should be removed. A new UC should be created and *included* by the other *base* UC.
(2) In the case that the shared UC does indeed contain both common and exception handling behavior. The shared UC should be split into two separate UCs. Each of the newly created UCs should only contain functionality appropriate to its corresponding *base* UC.

In all cases, the UCs should be renamed to accurately describe their purpose.

- **Antipattern names**

a3. Functional decomposition: Using the *include* relationship—**Automation support: Type (1)**
a4. Functional decomposition: Using the *extend* relationship—**Automation Support: Type (1)**
a5. Functional decomposition: Using pre and postconditions—**Automation support: Type (3)**

- **Description**

a3. Functional decomposition commonly occurs due to the misuse of the *include* relationship. *Inclusion* UCs are set to describe tasks that are required to perform parts of a complete service offered by their *base* UC. The tasks described by the *inclusion* UCs represent functions in a program, or menu options.
a4. Another form of functional decomposition is the improper use of the *extend* relationship. Naturally, this additional behavior is very specific to the respective base UC. If an *extension* UC contains general behavior that would be useful to more than one *base* UC, this would be a strong indication that the *extension* UC has degraded into a function.
a5. Modelers misuse the pre and postconditions in UCs to explicitly declare a virtual call sequence between the UCs. It can be deduced that the virtual sequence is likely to be the result of UCs degrading into functions. There are two methods that can be used to apply this concept: (a) by stating

as a postcondition in UC "N" that UC "N+1" must start, and stating as a precondition in UC "N+1" that UC "N" must successfully end; (b) implicitly, by restating the postconditions of UC "N" as the preconditions UC "N+1" and vice versa. Even though this method may be valid, it is likely to lead to the over-specification of the conditions in one of the UCs.

- **Rationale**

Dissecting UCs into functions yields a set of "smaller" UCs that are easier to understand and code. Consequently, in later development phases, the "smaller" UCs will be easier to test and maintain. Functional decomposition can be used to embody design decisions that analysts would like to enforce throughout the development of a system. In the case of functional decomposition using the *extend* relationship, there might be times where the *extension* UC is used to provide general functionality that is specialized by the UCs it *extends*.

- **Consequences**

The "smaller" UCs offer no value to the system's users if initiated individually. Being able to discern the actual service offered by these numerous function-oriented "smaller" UCs is a very difficult task. One can at best guess what service these UCs will offer when performed together. Therefore, the "smaller" UCs created as a result of functional decomposition obscure the real purpose of the system. For complex systems, it is likely that this "guess" will be incorrect. Functional decomposition of UCs may lead to more complex descriptions of the interactions between the actors. Functional decomposition embodies premature design decisions which severely limits the creativity of designers and forces them to abide to these decisions. In the case of functional decomposition using the *extend* relationship, it is often the case that the *extension* UC does not properly handle the exceptional situations caused by the *base* UCs.

- **Detection**

a3. Where—Search for an *inclusion* UC inside the UC diagram. **How**—If the *inclusion* UC is *included* by a single *base* UC. It is important to note that the *inclusion* UC must not be associated with any actors or UCs.

OCL description:

```
context UseCase
inv NotJustOneInclude:
not (self.base->size = 1)
```

a4. Where—Search for an *extension* UC in the UC diagram. **How**—If the *extension* UC *extends* more than one UC. **How -** Examine the behavior described by the *extension* UC to check that it is not too generic.

OCL description:

```
context UseCase
inv ExtendingMoreThanOneUseCase:
not ( (self.extended->size) + (self.
extendedUC->size) > 1)
```

a5. Where—In the preconditions and postconditions of each *base* UC description. **How**—(a) If it is stated as a postcondition for a UC is that another UC needs to be initiated. (b) If it is stated as a precondition of a UC that another UC needs to be successfully completed. (c) If it is found that the precondition of a UC and a postcondition of another UC state the same requirements.

- **Improvement**

The behavior described in the functionally decomposed UCs must be combined into UCs that individually offer a complete service. For the case when *extension* UCs are used to specialize general behavior described by their *base* UCs, a *generalization* relationship should be used instead of the *extend* relationship.

- **Antipattern name**

a6. Accessing an *extension* UC—**Automation support: Type (1)**

- **Description**

Similar to *base* UCs, *extension* UCs can be initiated by actors. Therefore, modelers may associate an *extension* UC with any actor.

- **Rationale**

Extension UCs differ from regular *base* UCs in that they contain behavior that is of exceptional or optional nature. Modelers may require an actor to access behavior contained in an *extension* UC for a number of reasons:

(1) If the *extension* UC contains optional behavior relative to the *base* UC that may be exclusively useful to an actor. Therefore, an explicit association is created between the actor and the *extension* UC to allow the actor to execute this optional behavior without initiating the *base* UC first.
(2) When an *extension* UC is handling an exceptional situation, it may be desired to notify a particular actor that such an exceptional situation has occurred. This can be achieved by creating an association between the actor and the *extension* UC.

(3) The operation of the *extension* UC may require certain information to be provided by an actor. An association between the actor and the *extension* UC is created to allow the actor to convey this information.

- **Consequences**

(1) This situation is appropriate because the *extension* UC contains behavior that is complete and optional. Moreover, it is desirable to execute this optional functionality without initiating the *extended base* UC.
(2) The actor is supposed to communicate with an *extension* UC only in the case of an exceptional situation arising. However, since the navigability of the association between the actor and the *extension* UC is not specified, the actor may initiate the *extension* UC. This is an undesired effect as it allows an actor to initiate an *extension* UC regardless of the situation. It should only be initiated when an exceptional situation has occurred.
(3) Same as (2).

- **Detection**

Where—Search for any *extension* UCs in the UC diagram. **How**—If the *extension* UC detected is directly associated with any actor in the model.
OCL description:

```
context UseCase
inv AccessingExtensionUseCaseByActor:
not((self.extended -> size > 0 or
self.extendedUC->size>0) and self.
actorEnd-> size > 0 )
```

- **Improvement**

(1) The association between the actor and the *extension* UC is required in this situation. Therefore, no corrective actions are required.
(2) The modelers need to explicitly state that the association between the actor and the *extension* UC is a one-way communication link. Unfortunately, UML lacks the required notation to depict this type of association between actors and UCs. To work around this limitation, a UML "note" can be connected to the association link between the two entities to explicitly state that it is a directed association. Moreover, the navigation direction of the association link can be specified to ensure that the interaction between the two entities is started by the UC.
(3) The *base* UC should be used to convey information to the *extension* UC. Therefore, the association between the actor and the *extension* UC should be removed. The *extension* UC should retrieve the required information from the *base* UC that it *extends*. The *base* UC should have the required information since the actor would have provided it when initially performing the *base* UC. If the actor only communicates with the *extension* UC after it has been initiated, then the navigation direction of the association link should be set to point towards the actor.

- **Antipattern name**

a7. Multiple actors associated with one UC—**Automation support: Type (1)**

- **Description**

A UC is associated with more than one actor.

- **Rationale**

(1) The associated actors play a similar role when performing the shared UC. In other words, the actors communicate with the shared UC in a similar fashion.
(2) The instances of the system's users are depicted instead of the role played by the users.
(3) The functionality performed by the shared UC is too generic.
(4) Execution of the shared UC requires communication with multiple actors.

- **Consequences**

(1) Actors should have unique roles when interacting with a shared UC. This leads designers to create different implementations of a UC when it is interacting with different actors, even though the implementation should be the same.
(2) This situation violates the true semantics of an actor. This will lead to similar consequences to those described in (1). Moreover, the model will need to be changed frequently as instances of a type of the system's users are frequently added and removed.
(3) The actual functionality developed will only cater to one of the actors, or perhaps none.
(4) This situation is appropriate as actors have different roles when the shared UC is performed.

- **Detection**

Where—Search for any UCs associated with actors in the UC diagram. **How**—If the UC is associated with multiple actors.
OCL Description:

```
context UseCase
inv MultipleActorsAssociatedWithUC:
not (self.actorEnd->size > 1)
```

- **Improvement**

(1) This situation can be fixed by extracting the overlapping roles between the associated actors and creating a new actor that represents these roles.
(2) An actor representing the true role of the depicted users should be created to replace all user instances.
(3) The shared UC should be split into separate UCs which accurately represents the behavior of the system when interacting with each actor.

- **Antipattern name**

a8. A description of an actor that is not depicted in the UC diagram—**Automation support: Type (2)**

- **Description**

The UC model contains a description of an actor; however, the actor is not depicted in the UC diagram.

- **Rationale**

(1) Even though the behavior of the actor might be known, it is not clear at the time how the actor will interact with the system.
(2) The actor represents a device used by the system to perform its UCs, without it being directly associated with the UCs. For example, the system might require a timer to control or initiate the activation of certain UCs.
(3) The actor described is associated with many UCs. Therefore, depicting the actor and its association with multiple UCs will clutter the UC diagram.

- **Consequences**

(1) It is acceptable to describe an actor before deciding how it interacts with the system. However, it is essential that the actor's association with the system be eventually defined; otherwise the actor's involvement with the system will be ambiguous.
(2) Timers and other input/output devices do not constitute actors. This issue is discussed in the description of the "Representing devices as actors" antipattern "a19." (see [45]).
(3) This situation can be the result of having too many UCs. The "Too many UCs" antipattern "a25." (see [45]) describes the consequences of this situation. If an actor remains missing from the UC diagram, the actor's involvement with the system can be misinterpreted.

- **Detection**

Where—For every actor described. **How**—(a) If the actor is not depicted in a UC diagram.

- **Improvement**

(1) The actor must be depicted and associated with the appropriate UCs in the UC diagram.
(2) The behavior of these devices should be included in the behavior of their associated UCs. If it is necessary to describe the behavior of a device, such a description should be available in the supplementary requirements document.
(3) First, the improvements stated by the "Too many UCs" antipattern (**a25.**) should be undertaken. If any actor remains associated with too many UCs, then reorganizing the layout of the UC diagram can be beneficiary. In any event, the actor and its associations with the system must be depicted.

The remaining 18 antipatterns are presented in [45]

4 Tool support using ARBIUM

Examining the structure of a UC diagram is a process that can be fully automated. For complex systems, a UC model may contain hundreds of UCs [7]; in addition, these UCs are not depicted in any chronological order. Moreover, various types of relationships are depicted linking those UCs. Inherently, these relationships are not depicted in any chronological order either. Such systems also usually contain a large number of actors that are associated with UCs using association relationship links. Ultimately the UC diagram can be viewed as a large mesh of UCs, actors and relationship links. Attempting to detect a match for a given diagrammatic structure described by an antipattern can be very challenging, cumbersome and error prone. ARBIUM (**A**utomated **R**isk-**B**ased **I**nspector of **UC** **M**odels) provides automation support for detecting diagrammatic structures. The presented technique does not target deficiencies that can be detected via static analysis, such as syntax errors. ARBIUM is geared towards detecting potential deficiencies that require human validation.

Unsound structures described in antipatterns are entered into ARBIUM as OCL statements. The OCL statements adhere to the simplified metamodel presented earlier (Fig. 1). In addition to being able to describe and search for custom made antipatterns, ARBIUM is provided with a set of predefined antipatterns, which analysts may utilize to improve their models. The predefined antipatterns are of the DI variety so that they can be applied to any UC model regardless of its domain.

The matching process is aided by the tool USE (**U**ML-based **S**pecification **E**nvironment). USE is a tool that checks the integrity of information systems against constraints described in OCL [21]. ARBIUM generates two input files for USE: a specifications file and a script file. The specifications file describes the class structure of the metamodel, and contains the set of antipatterns specified by the analyst. The script file loads an object representation of the actual UC diagram, based on the simplified metamodel. After completing the matching process, USE presents any antipattern matches for analysts to review. An overview of how ARBIUM, incorporating USE, can be used to search for antipatterns is shown below (see Fig. 2). A more detailed discussion of ARBIUM is presented in [45].

5 Evaluation

In this section we present a real world case study to demonstrate the application of our proposed technique and to examine its feasibility. In addition, we compare the results of using ARBIUM to drive the inspection process to the results of using DesignAdvisor [7].

5.1 Definition and motivation

The main research question posed by this case study is whether the detection of antipatterns and analysis of the resulting matches can improve the overall quality of UC models. This is achieved on two fronts: (a) by restructuring the UC diagrams to adhere to the notational syntax rules and semantics set by OMG [35]; and (b) by changing UC descriptions to comply with recommended guidelines and widely accepted practices (Sect. 2). Therefore, the effectiveness of using our proposed approach will be assessed by comparing the resulting UC model with the original UC model, with respect to the aspects mentioned in (a) and (b).

5.2 Case study formulation

The proposed approach was applied to the MAPSTEDI (**M**ountains **a**nd **P**lains **S**patio-**Te**mporal **D**atabase **I**nformatics) [34] UC model. ARBIUM was utilized to perform the matching process. The MAPSTEDI system is being developed to allow the University of Colorado Museum (UCM), Denver Museum of Nature and Science (DMNS), and Denver Botanic Gardens (DBG) to merge their separate collections into one distributed biodiversity database. The merged collections will include over 285,000 biological specimens. The system will also be used as a research toolkit by geocoders to analyze biodiversity data in the southern and central Rocky Mountains and the northern plains both spatially and temporally. The MAPSTEDI system will be developed over three phases. Upon completion of the project, MAPSTEDI will be able to "georeference" the museum collection databases. Users' search results will be provided by the MAPSTEDI website in GIS-linked spatio-temporal coverage.

The UC model of the MAPSTEDI system contains several UC packages that are used to model different subsystems of the target system. The UC model is accompanied with UC descriptions. The descriptions play an essential role in examining the validity of the UC diagrams and the model as a whole. The MAPSTEDI UC model contains five UC packages which represent different aspects of the system's functionality. Each UC package contains one UC diagram:

- **Database access** (Fig. 3): The purpose of this UC package is to state who may access the database and how. Users of the system can search and download collections data. Users may also visualize biodiversity analysis. Only research users are permitted to access sensitive data.
- **Database queries** (Fig. 4): This UC package provides a hierarchal outline of the query functionalities performed by the system. The subsystem queries local and distributed databases for collections data. There are two distributed databases, the DMNS and DIGIR databases.
- **Database integrator** (Fig. 5): This UC package shows how the collections data from separate databases (local and remote) are integrated after being updated.
- **Database edits** (Fig. 6): This UC package outlines the operational mechanisms for editing and updating the databases. The geocoder edits the collections data and the databases are updated accordingly.
- **Administrative process** (Fig. 7): This UC package shows the administrative functionalities and responsibilities. The subsystem backups and restores collections data and application code. The subsystem also installs any new updates.

Currently the MAPSTEDI UC model suffers from a number of issues (listed below) that decrease its quality. These issues are determined after examining the UC diagrams and the corresponding UC and actor descriptions:

1. The public and research users are shown to have different roles when accessing certain functionalities offered by

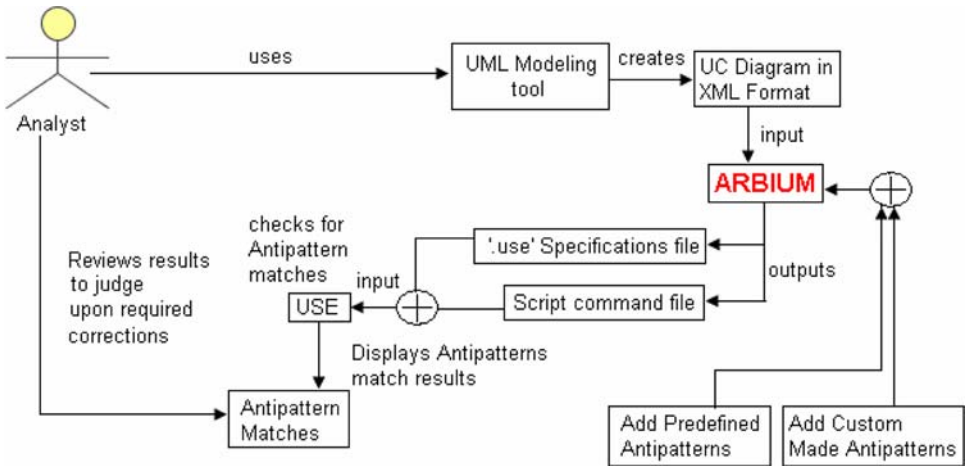


Fig. 2 An overview of how ARBIUM and USE can automate the detection process

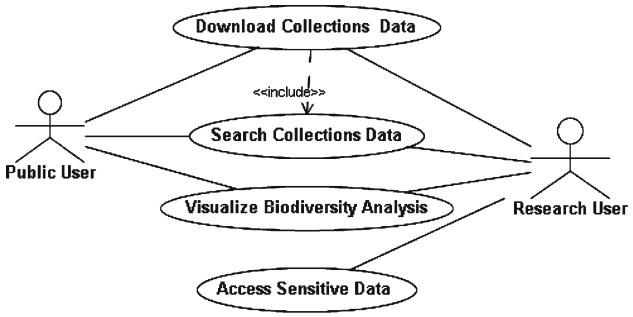


Fig. 3 The UC diagram of the "Database Access" subsystem

the system, however they perform the same role. Moreover, the UC diagram indicates that both public and research users need to be involved with the system in order to perform certain functionalities which is incorrect.

2. A dependency is created between UCs Download Collections Data and Search Collections Data through improper use of pre and postconditions. (Please refer to antipattern **(a5.)** for details regarding the implications of this issue).
3. UCs in the Query Databases UC diagram are shown to *extend* each other, meaning that some UCs introduce optional or exceptional behavior to the functionality described in other UCs which is incorrect. The UCs have a hierarchical relation with respect to the query services that they offer, which is not shown.
4. The UC model presents a number of functionally decomposed UCs, such as the Edit Collections Data, Upload DGB and UCM Data and Run QC Tests UCs. This is detrimental to the analytical quality of the UC model. Further implications of functional decomposition are presented in antipatterns **(a3.), (a4.)** and (**a5.**).
5. The database edits UC diagram shows an incorrect type of dependency between the Geocode Specimen UC and the Update Collections Data UC.
6. The UC model contains a superfluous actor: Data Editor.
7. The administrative process UC diagram shows three UCs that are too generic to allow either of the administrator actors to perform their intended duties.
8. The UC Model describes two actors that are system functionality not actors.
9. UC Query Remote Database is indirectly accessed by an actor while it exclusively does not describe complete and meaningful functionality.

Many of these issues may have severe consequences downstream in the development process. It is crucial to remove these issues from the UC model. In the following subsections, our proposed technique will be applied to the UC model in order to assess its ability to resolve these issues. All UC diagrams will be juxtaposed with the entire set of antipatterns. While performing the matching process, it is important to consider overlapping entities. That is, UCs or actors that exist in more than one diagram. Considering overlapping entities help reveal antipattern matches that may exist over multiple UC diagrams.

5.3 Analysis and interpretation of the results

The resulting antipattern matches shown in Table 5 require human inspection to verify the correctness of the UC model. A total of 11 antipattern matches were detected across all of the UC packages. An analysis of the antipattern matches of the first iteration is shown in Table 6. All antipatterns detected in the first iteration, with the exception of antipattern matches 1.1.2 and 1.6.1, are of Type (1). Therefore, they were detected automatically by ARBIUM. Antipattern match 1.1.2 (Type (3)) was detected by manually applying the anti-pattern template to the descriptions of the Download Collections Data and Search Collections Data UCs of the Database Access UC diagram. Meanwhile, antipattern match 1.6.1 (Type (2)) was detected by manually applying the anti-pattern template

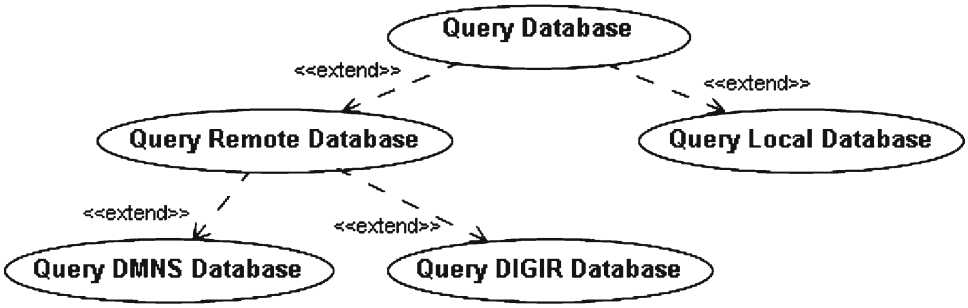


Fig. 4 The UC diagram of the "Database Queries" subsystem

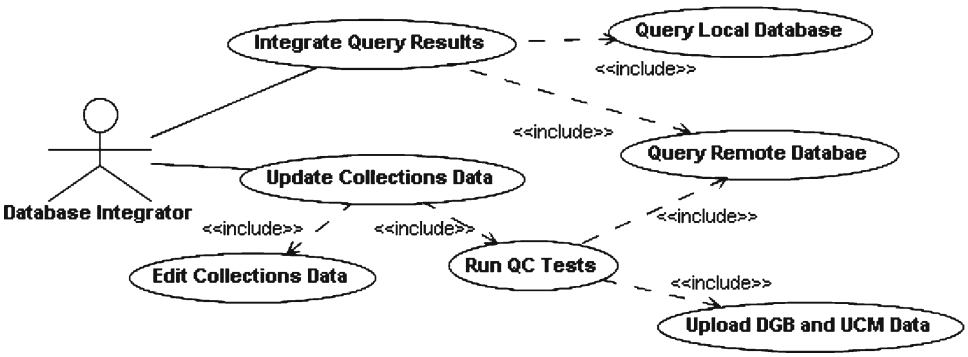


Fig. 5 The UC diagram of the "Database Integrator" subsystem

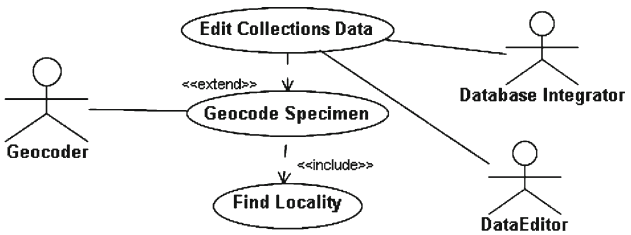


Fig. 6 The UC diagram of the "Database Edits" subsystem

to the actor descriptions, while ARBIUM searched for these actors in the UC diagrams.

The database edits, database queries and database integrator UC diagrams were merged since they contain a number of overlapping entities. The merged UC diagram ("Merged UC Diagram") is presented in Fig. 9.

As mentioned earlier, the proposed technique must be applied iteratively as corrections and changes applied upon reviewing an antipattern match might cause new antipatterns to surface. The matching process is repeated for a second iteration. Table 7 shows the antipattern matches detected during the second iteration, and Table 8 shows the corresponding analysis. All antipatterns matched are of Type (1) and hence were detected by ARBIUM. The antipattern matches were detected in the Merged UC diagram shown in Fig. 9.

5.4 Discussion of results and validation

In this section we assess whether the application of our technique resolved the issues that existed in the original MAPSTEDI UC model (see end of Sect. 5.2). Table 9 provides a summary of the issues resolved by applying our technique:

5.5 Comparison of alternative approaches

In order to fully evaluate the effectiveness of using antipatterns, it should be compared to alternative approaches by applying them to the MAPSTEDI UC model. As mentioned earlier in Sect. 2, only the approach presented by Berenbach in [7] can be compared to our approach since it does not require significant human cognition to apply. The MAPSTEDI UC model was examined to determine if it violates any of the heuristics presented in [7]; these violations will then be "resolved" in the model. Table 10 presents the heuristics from [7], and presents the number of violations found in the MAPSTEDI model. Each heuristic is stated is followed by the antipatterns that embody it; the heuristics from [7] believe that the model is defect free!

6 Conclusions, future work and suggestions

UC modeling is a very powerful requirements modeling tool, providing great flexibility for requirements engineers to capture the behavioral essence of the target system. In a UC driven development process, UC models are used to create other UML artifacts leading to the eventual implementation of the target system. Thus poorly constructed UC models may yield many inconsistencies between subsequent UML

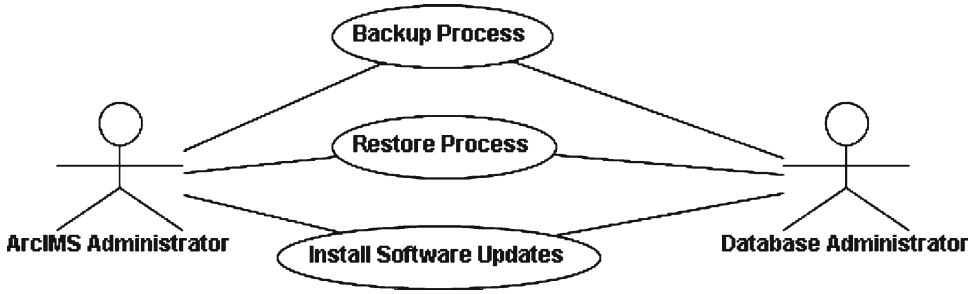


Fig. 7 The UC diagram of the "Administrative Process" subsystem

Table 5 First iteration matches

Match No.	UC diagram	Antipattern matched	Elements involved
1.1.1	Database Access	**a6.** Multiple actors associated with one UC	Actors: Public User and Research User UCs: Download Collections Data, Search Collections Data and Visualize Biodiversity Analysis
1.1.2		**a8.** Functional decomposition: Using pre and postconditions	UCs: Download Collections Data, Search Collections Data
1.2.1	Database Queries	**a3.** Functional decomposition: Using the *extend* relationship	UCs: All five UCs illustrated in the corresponding UC diagram.
3.1	Database Integrator	**a5.** Functional decomposition: Using the *include* relationship	UCs: Edit Collections Data and Update Collections Data.
1.3.2		**a5.** Functional decomposition: Using the *include* relationship	UCs: Upload DGB and UCM Data, Run QC Tests and Update Collections Data.
1.4.1	Database Edits	**a5.** Functional decomposition: Using the *include* relationship	UCs: Geocode Specimen and Find Locality
1.4.2		**a4.** Accessing an *extension* UC	Actors: Data Editor and Database Integrator. UCs: Edit Collections Data and Geocode Specimen.
1.5.1	Administrative Process	**a6.** Multiple actors associated with one UC	Actors: Database Administrator and ArcIMS Administrator UCs: Backup Process
1.5.2		**a6.** Multiple actors associated with one UC	Actors: Database Administrator and ArcIMS Administrator UCs: Restore Process
1.5.3		**a6.** Multiple actors associated with one UC	Actors: Database Administrator and ArcIMS Administrator UCs: Install Software Updates
1.6.1	System Wide	**a7.** A description of an actor that is not depicted in the UC diagram	Actors: Database Upload Process and Database QA/QC Process

artifacts, ultimately leading to many defects in the eventual code. Unfortunately, UC modeling is often misapplied resulting in significant numbers of defects. Hence, it is essential to produce high quality UC models.

In this paper we devise a technique based on antipatterns that helps improve the quality of UC models throughout the development cycle. The application of the technique does not require any artifacts in addition to UC models, this allows the technique to be applied early in the development cycle, where other design artifacts are usually unavailable and the cost of removing defects is minimal. Given the "informality" of UC models, many approaches provide abstract guidelines towards improving UC models. Using antipatterns provides analysts with a more systematic approach to improve UC models, significantly reducing the dependency on skill and experience. A large repository of antipatterns was developed to guide analysts in improving their UC models. The repository contains 26 domain-independent antipatterns that can be applied to any UC model. The majority of the developed antipatterns benefit from (semi-)automation support to increase the accuracy and speed of their detection. In addition to the provided antipatterns, a framework was developed for analysts to create their customized antipatterns based on a simplified UC modeling metamodel, where analysts can create their own antipattern descriptions using OCL. The complexity of the metamodel was intentionally designed to encourage

Table 6 First iteration analysis

Antipattern Match 1.1.1:
Analysis
Upon analysis of the three UCs which the actors Public User and Research User are associated with, the actors were found to have similar roles when performing the UCs.
Corrective actions
The role that the actors play in correspondence to the three given UCs will be *generalized* into a separate actor (called User). The *generalized* actor is then associated with the UCs, while the Research User remains the only actor associated with UC Access Sensitive Data (Fig. 8).
Antipattern match 1.1.2
Analysis
The precondition of the Download Collections Data UC states that the Search Collections Data UC must be initialized beforehand.
Corrective actions
Each UC offers a complete service individually hence they should remain separate. However, the precondition stated by the Download Collections Data UC should be removed.
Antipattern match 1.2.1
Analysis
The *extend* relationship was used to represent the hierarchy between the query services offered by the system.
Corrective actions
The *extend* relationships should be replaced with *generalization* relationships (Fig. 9).
Antipattern match 1.3.1
Analysis
The Edit Collections Data UC represents subroutine type behavior that is required by the Update Collections Data UC.
Corrective actions
The functionality described in the Edit Collections Data UC should be merged with the description of the Update Collections Data UC and represented as a "Sub-flow[3] Subsequently, the UC Edit Collections Data and it's *include* relationship link with UC Update Collections Data are removed from the diagram (Fig. 9).
Antipattern match 1.3.2
Analysis
Analysis of the involved UCs show that updating the database requires the DGB (Denver Botanic Gardens) and UCM (University of Colorado Museum) data to be uploaded. Meanwhile, the task of uploading any data also requires that the data undergo Quality Control (QC) tests.
Corrective Actions
The Upload DGM and UCM Data and Run QC Tests UCs should be merged into the Update Collections Data UC by modeling each as a separate "Sub-flow" component. Moreover, the description of the "Sub-flow" component responsible for uploading the data should indicate a requirement to execute the other "Sub-flow" that is responsible for running the QC tests. UCs Upload DGM and UCM Data and Run QC are removed from the Database Integrator diagram. Meanwhile, an *include* relationship will be directed from the Update Collections Data UC to the Query Remote Database UC, to replace the *include* relationship that was present between the Run QC Tests and Query Remote Database UCs (Fig. 9).
Antipattern Match 1.4.1
Analysis The Find Locality UC represents subroutine type behavior that is required by the Geocode Specimen UC.
Corrective actions
The functionality described in the Find Locality UC should be merged and represented as a "Sub-flow" component of the Geocode Specimen UC. Hence, UC Find Locality and it's *include* relationship with UC Geocode Specimen are removed from the diagram (Fig. 9).
Antipattern match 1.4.2
Analysis The Edit Collections Data UC was merged into the Update Collections Data UC as a result of antipattern match 1.3.1 in the Data Integrator UC diagram. Therefore, actors Data Editor and Database Integrator are now associated with the UC Update Collections Data. Moreover, UC Update Collections Data now *extends* the Geocode Specimen UC.
Upon analyzing the *extended* UC Geocode Specimen, it is discovered that updating the database represents part of its required functionality.
The data-editing role played by the Geocoder actor is modeled using the Data Editor actor. However, the Geocoder is the only actor that edits this data. Moreover, the model shows that the Geocoder already has indirect access to the Update Collections Data UC, through the Geocode Specimen UC.

[3] A "Sub-flow" is a component of a UC description that describes subroutine-like behavior that is exclusive only to the belonging UC."

Table 6 continued

Antipattern match 1.1.1
Corrective actions
The *extend* relationship between the involved UCs was used to indicate subroutine type behavior. Therefore, this relationship should be replaced with an *include* relationship directed from the Geocode Specimen UC to the Update Collections Data UC. Hence, the Data Integrator actor is no longer directly accessing an *extension* UC (Fig. 9).
Since the Geocoder actor already has indirect access to the Update Collections Data UC, the Data Editor actor is no longer required and should be removed (Fig. 9).
Antipattern matches 1.5.1, 1.5.2 and 1.5.3
Analysis All three antipattern matches resulted from the same issue; the shared UCs are too general to suit either the ArcIMS Administrator or the Database Administrator actor. After reviewing the tasks of both actors, it was determined that the ArcIMS Administrator actor accesses the system to backup and restore the application code and to install code updates. Meanwhile, the Database Administrator actor accesses the system to backup and restore the *collections* data, and to install database updates.
Corrective actions
The three shared UCs should be split down into six UCs in order to properly represent the administrative duties of the actors (Fig. 10).
Antipatterns match 1.6.1
Analysis Two actors Database Upload Process and Database QA/QC Process where described but never depicted in any UC diagram. The descriptions of the actors however simply state functionality that is performed by the system itself, and hence should be part of the UC descriptions.
Corrective actions
No corrective actions are required since the actor tasks were already stated in the UC descriptions. The superfluous actors should be removed from the UC model.

Table 7 Second iteration matches

Match No.	UC Diagram	Diagrammatic-antipattern matched	Elements involved
2.1.1	Merged UC diagram (Fig. 9)	**a1.** Accessing a *generalized concrete* UC	Actors: Database Integrator
			UCs: Query Remote Database and Integrate Query Results.
2.1.2		**a2.** UCs containing common and exceptional functionality	UCs: Query Remote Database, Update Collections Data and Geocode Specimen.
2.1.3		**a4.** Accessing an *extension* UC	Actor: Database Integrator
			UCs: Update Collections Data

Table 8 Second iteration analysis

Antipattern match 2.1.1
Analysis
This antipattern match resulted from replacing the inappropriately used *extend* relationships with *generalization* relationships. The *generalized* UC Query Remote Database is *concrete* and is indirectly accessed by the Database Integrator actor through the Integrate Query Results UC.
Corrective actions
According to the "Accessing a *generalized concrete* UC" antipattern **(a1.)**, this situation may be fixed by setting the *generalized* Query Remote Database UC to be *abstract*. To conserve space, this minor change to the merged UC diagram (Fig. 9) will not be shown.
Antipattern match 2.1.2
Analysis
The shared UC Update Collections Data contains subroutine behavior relative to the Geocode Specimen and Query Remote Database UCs.
Corrective actions
The *include* relationship with the UC and the Update Collections Data UC should remain intact. Meanwhile, the *extend* relationship between the Update Collections Data UC and the Query Remote Database UC should be replaced with an *include* relationship. To conserve space, this minor amendment to the merged UC diagram (Fig. 9) will not be shown.
Antipattern match 2.1.3
Analysis:
This antipattern no longer exists due to the corrective actions undertaken after analyzing antipattern match 2.1.2.

Table 9 Addressing issues in the MAPSTEDI UC model

Issue	Discussion and Validation	Resolved
1	The newly created *generalized* actor User represents the only role that exists while performing the shared UCs. Hence, the *generalization* relationship between the actors Research User and Public User, and their parent actor User, correctly indicates that they have the same role while performing the shared UCs. Having a single *generalized* actor access the previously shared UCs also eliminates the misinterpretation that both the Research User and Public User actors are required to be involved with the system simultaneously in order to perform the UCs. The only unshared UC Access Sensitive Data remains associated only with the Research User actor.	✓
2	Removal of the improper preconditions from the Search Collections Data UC complies with a widely accepted authoring guideline discussed in [9]. Now the Search Collections Data UC is appropriately dependent on the Download Collections Data UC through an *include* relationship only.	✓
3	The *generalization* relationships appropriately represent the hierarchy of services offered by the UCs in the Query Databases UC diagram.	✓
4	The analytical value of the UC model is greatly improved as the functionally decomposed UCs (Edit Collections Data, Upload DGB and UCM Data and Run QC Tests) are removed. Their respective functionalities are appropriately merged into their respective *base* UCs so that the *base* UCs describe complete and useful behavior.	✓
5	The Geocode Specimen UC is now appropriately set to *include* the Update Collections Data UC, representing the correct type of dependency that exists between the UCs.	✓
6	Each actor must have a distinct role. The Data Editor actor represents part of the role already performed by the Geocoder actor. The superfluous Data Editor actor is now removed from the UC model, eliminating redundancy and improving understandability.	✓
7	UCs must contain the correct level of detail in order to provide a complete and meaningful service to an actor. Each of the three overly general UCs shared by the administrator actors are now split into two separate UCs. The newly created UCs contain specific behavior to allow each administrator actor to perform their respective administrative duties.	✓
8	Every actor described in the UC model must be depicted at least once in a UC diagram. An actor is invalid if its description states functionality that is performed by the system itself. Therefore, actors Database Upload Process and Database QA/QC Process are now removed from the UC model.	✓
9	A UC should only be *concrete* if it can offer a complete service to an actor, which is not the case with the Query Remote Database UC. Therefore, the Query Remote Database UC was set to be *abstract* to force one of its implementing UCs to carryout the specific behavior of querying a remote database.	✓

Table 10 Examining the MAPSTEDI UC model for violations of the heuristics presented in [7]

#	Heuristic	Violations detected
1	"Every UC must be defined." **(Covered by a17.)**	0
	Analysis: Every UC was appropriately defined. The template used for each UC contained fields for the UC name, actors involved, preconditions, postconditions and the actual description of the intended behavior. The description section of each UC stated a basic flow as well as alternative whenever applicable.	
2	"Abstract UCs must be realized with included or inheriting concrete UCs." **(Covered by a1. and a12.)**	0
3	"A concrete UC cannot include an abstract UC (unless it is realized)." **(Covered by a3., a9. and a12.)**	
	Analysis: There was no abstract UCs in the original MAPSTEDI UC model.	
4	"Extending UC relationships can only exist between concrete UCs." **(Covered by a4., a6. and a9.)**	0
	Analysis: The extend relationship only existed in two diagrams: (a) the Data Edits and (b) the Database Queries UC diagrams. The Data Edits had one extend relationship between two concrete UCs. Meanwhile, the extend relationships between all UCs in the Database Queries UC diagrams are concrete.	
5	"Use activity diagrams to show all possible scenarios associated with a UC."	N/A
6	"The definition of a UC must be consistent across all diagrams defining the UC."	
7	"Use sequence diagrams rather than collaboration diagrams to define one thread or path for a process."	
8	"Avoid realization relationships and artifacts in the analysis models."	
	Analysis: Only the UC model of the MAPSTEDI was available. Moreover, this case study focuses on comparing approaches that improve UC models early in the development cycle where only the UC model is available.	

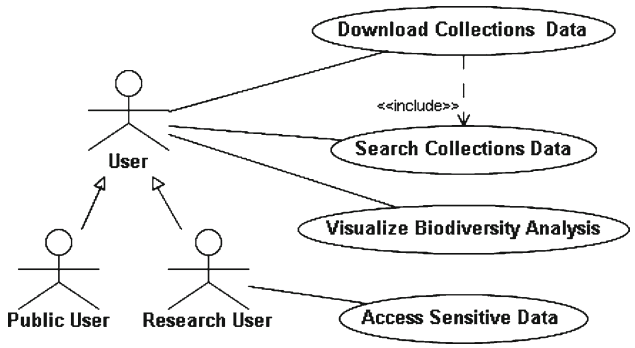


Fig. 8 The Database Access UC diagram after the first iteration

its adoption by analysts and minimize the requisite learning curve, while supporting the basic notational subset of UC models. Automation support for detecting antipatterns is provided via the tool ARBIUM, which is also discussed in this paper. ARBIUM provides (semi-) automation support to 23 antipatterns presented in the repository and allows analysts to define their own antipatterns.

The effectiveness of the approach was demonstrated upon the MAPSTEDI system. Before applying the proposed process, the MAPSTEDI UC model suffered from a number of quality degradation issues. Most issues (antipatterns) were detected automatically using ARBIUM. Most antipattern matches addressed resulted in changes; however, there were also a small number of antipattern matches that are considered false positives. This indicates that real-world UC models are highly vulnerable to poor modeling habits and design decisions and often require improvements. Many of these improvements were critical as they improved the correctness and consistency of the UC models. Others enhanced the understandability of the UC models and made them more analytical. The antipattern matches revealed the issues that existed in the original UC model that had been overlooked. The issues were addressed and resolved accordingly, resulting in a higher quality UC model.

Future work will initially be based around the creation of more UC modeling antipatterns; and improving the usability (e.g. the incorporation diagrammatic construct drawing package) of ARBIUM with respect to the construction of

Fig. 9 A merged view of the remaining three UC diagrams after the first iteration

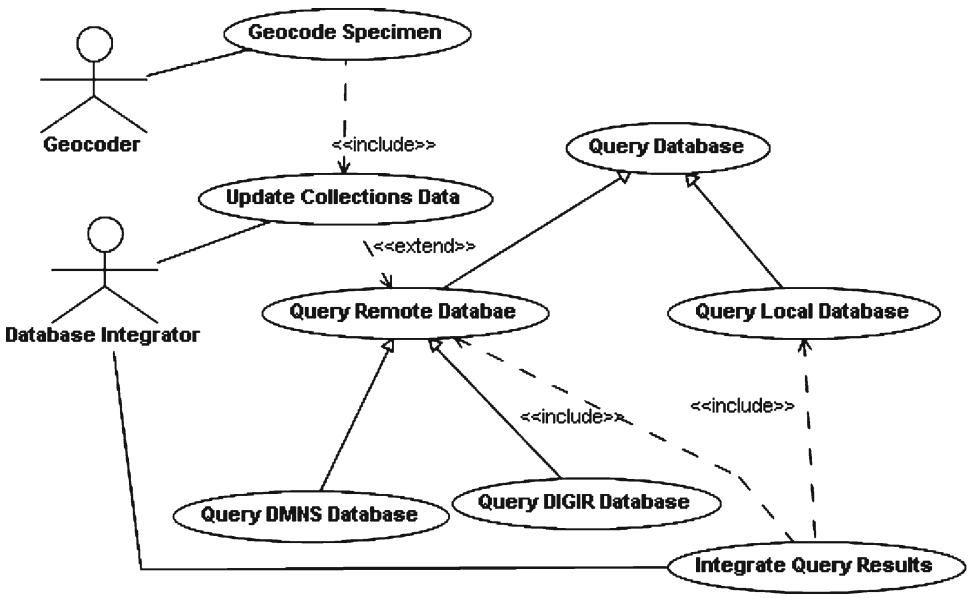
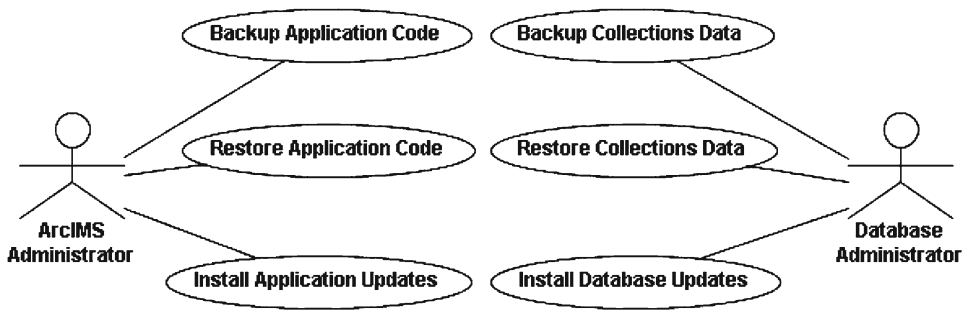


Fig. 10 The Administrative Process UC diagram after the first iteration



new domain-specific anti-patterns. ARBIUM can also be upgraded to perform limited textual analysis, making use of any structure that may exist in UC descriptions, such as the actual template. Another beneficial upgrade to ARBIUM is the implementation of transformation rules written in a model transformation language such as queries/views/transformation (QVT) [36] to formalize and automate changes applied to UC diagrams.

Other future work can be directed towards creating a hierarchy of antipatterns. The hierarchy will act as an antipatterns matching strategy for analysts to apply the proposed technique more efficiently. Analysts will be able to determine which antipatterns to look for first and when to start a new iteration. This will help reduce the effort and time required to apply the technique. The antipatterns matching strategy may then be implemented in ARBIUM to further automate the technique and reduce the analyst's workload.

Finally, it will be beneficial to improve the UC modeling notation in order to prevent the occurrence of many antipatterns. For example, while analyzing a large number of UC models and applying the proposed technique, it was discovered that many antipatterns matches existed due to a notational limitation in UC modeling. The *extend* relationship is used to model both exceptional behavior and optional behavior. One of the greatest advantages of UC modeling is that it contains a small notational set, allowing its ease of use. However, it may be advantageous to introduce two additional relationships that explicitly represent optional and exceptional behavior separately.

References

1. Adolph, S., Bramble, P.: Patterns for Effective Use Cases. Addison-Wesley, Reading (2002)
2. Anda, B., Sjøberg, D., Jørgensen, M.: Quality and understandability in use case models. In: Lindskov Knudsen, J. (ed.) Proceedings of the 15th European Conference on Object-Oriented Programming, pp. 402–428 (2001)
3. Anda, B., Sjøberg, D.I.K.: Towards an inspection technique for use case models. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, pp. 127–134 (2002)
4. Anderson, E., Bradley, M., Brinko, R.: Use Case and business rules: styles of documenting business rules in use cases. In: Addendum to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (1997)
5. Armour, F., Miller, G.: Advanced Use Case Modeling. Addison-Wesley, Reading (2000)
6. Ben Achour, C., Rolland, C., Maiden, N.A.M., Souveyet, C.: Guiding use case authoring: results of an empirical study. In: Proceedings of the IEEE Symposium on Requirements Engineering (1999)
7. Berenbach B.: The evaluation of large, complex UML analysis and design models. In: Proceedings of the 26th International Conference on Software Engineering, pp. 232–241 (2004)
8. Biddle, B., Noble, J., Tempero, E.: Essential use cases and responsibility in object-oriented development. Aust. Comput. Sci. Commun. **24**, 7–16 (2002)
9. Bittner, K., Spence, I.: Use Case Modeling. Addison-Wesley, Reading (2002)
10. Boehm, B.: Software Engineering—Economics. Prentice Hall, Englewood Cliffs (1981)
11. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
12. Butler, G., Xu, L.: Cascaded refactoring for framework evolution. In: Proceedings of the Symposium on Software Reusability, pp. 51–57. ACM Press, New York (2001)
13. Chandrasekaran, P.: How use case modeling policies have affected the success of various projects (or how to improve use case modeling). In: Addendum to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (1997)
14. Cockburn, A.: Structuring Use Cases with Goals, Tech. Report. Human and Tech., 7691 Dell Rd, Salt Lake City, UT 84121, HaT.TR.95.1, (1995)
15. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Reading (2000)
16. Constantine, L.L.: Essential modeling: use cases for user interfaces. In: ACM Interactions, vol. 2, pp. 34–46, Apr 1995
17. Fabbrini, F., Fusani, M., Gnesi, S., Lami, G.: The linguistic approach to the natural language requirements quality: benefits of the use of an automatic tool. In: Proceedings of the 26th Annual NASA Goddard Software Workshop, Nov., pp. 97–105 (2001)
18. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Application of Linguistic Techniques for Use Case analysis. In: Proceedings of IEEE Joint International Conference on Requirements Engineering, pp. 157–164 (2002)
19. Firesmith, D.G.: Use Case Modeling Guidelines. In: Proceedings of Technology of Object-Oriented Languages and Systems, vol. 30, p 184 (1999)
20. Gilb, T., Graham, D.: Software Inspection. Addison-Wesley, Reading (1993)
21. Gogolla, M., Bohling, J., Richters, M.: Validation of UML and OCL models by automatic snapshot generation. In: Proceedings of the 6th International Conference on the Unified Modeling Language (2003)
22. Gomaa, H.: Designing Software Product Lines with UML. Addison Wiley, Reading (2001)
23. Gomaa, H.: Use cases for distributed real-time software architectures. In: Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems, pp. 34–42 (1997)
24. Jaaksi, A.: Our Cases with Use Cases. J. Object-Oriented Program. **10**, 58–64 (1998)
25. Jacobson, I.: Use Cases—Yesterday, Today and Tomorrow. The Rational Edge (2003)
26. Kroll, P., Kruchten, P.: The Rational Unified Process Made Easy: A Practitioner's Guide To The RUP. Addison-Wesley, Reading (2003)
27. Kruchten, P.: Modeling component systems with the unified modeling language. In: Proceedings of International Workshop on Component-Based Software Engineering (1997)
28. Kruchten, P.: The Rational Unified Process: an Introduction, 2nd edn. Addison-Wesley Longman Inc., Boston (1999)
29. Kulak, D., Guiney, E.: Use Cases: Requirements in Context. Addison-Wesley, Reading (2000)
30. Lilly, S.: Use Case Pitfalls: Top 10 problems from real projects using use cases. In: Proceedings of Technology of Object-Oriented Languages and Systems (1999)
31. McBeen, P.: Use Case Inspection List. http://www.mcbreen.ab.ca/papers/QAUseCases.html (2007). Accessed Nov 2007
32. McCoy, J.: Requirements use case tool (RUT). In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 104–105 (2003)

33. Medvidovic, N., Rosenblum, D., Redmiles, D., Robbins, J.: Modeling software architectures in the Unified Modeling Language. ACM Trans. Softw. Eng. Methodol. **11**, 2–57 (2002)
34. Museum and EPOB, University of Colorado, MAPSTEDI UC Model (2008). http://mapstedi.colorado.edu/documents/Mapstedi_High_Level_Use_Case_Model.pdf. Accessed
35. Object Management Group, UML Superstructure Specification (2005). http://www.omg.org/docs/formal/05-07-04.pdf, Version 2.0 formal/05-07-04. Accessed Dec 2005
36. Object Management Group, "MOF 2.0 Query/Views/Transformations RFP," Dec. 2002, http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf.
37. Overgraad, G., Palmkvist, K.: Use Cases Patterns and Blueprints. Addison-Wesley, Reading (2005)
38. Ren, S., Rui, K., Butler, G.: Refactoring the scenario specification: a message sequence chart approach. In: Proceedings of 9th Object-Oriented Information Systems, pp. 294–298 (2003)
39. Ren, S., Butler, G., Rui, K., Xu, J., Yu, W., Luo, R.: A prototype tool for use case refactoring. In: Proceedings of the 6th International Conference on Enterprise Information Systems, pp. 173–178 (2004)
40. Rosenberg, D., Scott, K: Use Case Driven Object Modeling with UML. Addison-Wesley, Readings (1999)
41. Rosenberg, D., Kendall, S.: Top Ten Use Case Mistakes (2007). http://www.sdmagazine.com/documents/s=815/sdm0102c/. Accessed Nov 2007
42. Rui, K., Butler, G.: Refactoring Use case models: the metamodel. In: Oudshoorn, M. (ed.) Proceedings of 25th Computer Science Conference, pp. 4–7 (2003)
43. Ryndina, O., Kritzinger, P.: Improving Requirements Specification: Verification of UC Models with Susan. Tech. Report CS04-06-00, Department of Computer Science, University of Cape Town (2004)
44. Schneider, G., Winters, J.: Applying Use Cases—a Practical Guide. Addison-Wesley, Reading (1998)
45. STEAM Laboratory website, University of Alberta, "Use Case Modeling Antipatterns (2008). http://www.steam.ualberta.ca/main/research_areas/Use%20Case%20Antipatterns%20Website.htm. Accessed Aug 2008
46. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Reading (1998)
47. Wikipedia: The Definition of Antipatterns (2007), http://c2.com/cgi/wiki?AntiPattern
48. Wohlin, C., Korner, U.: Software Faults: Spreading, Detection and Costs. Softw. Eng. J. **5**, 33–42 (1990)
49. Xu, J., Yu, W., Rui, K., Butler, G.: Use case refactoring: a tool and a case study. In: Proceedings of 11th Asia Pacific Software Engineering Conference, pp. 484–491 (2004)

Author Biographies

Mohamed El-Attar is Ph.D. candidate (Software Engineering) at the University of Alberta and a member of the STEAM laboratory. His research interests include Requirements Engineering, in particular with UML and use cases, object-oriented analysis and design, model transformation and empirical studies. Mohamed received a B.Eng in Computer Systems from Carleton University. Contact him melattar@ece.ualberta.ca

James Miller received the B.Sc. and Ph.D. degrees in Computer Science from the University of Strathclyde, Scotland. During this period, he worked on the ESPRIT project GENEDIS on the production of a real-time stereovision system. Subsequently, he worked at the United Kingdom's National Electronic Research Initiative on Pattern Recognition as a Principal Scientist, before returning to the University of Strathclyde to accept a lectureship, and subsequently a senior lectureship in Computer Science. Initially during this period his research interests were in Computer Vision, and he was a co-investigator on the ESPRIT 2 project VIDIMUS. Since 1993, his research interests have been in Software and Systems Engineering. In 2000, he joined the Department of Electrical and Computer Engineering at the University of Alberta as a full professor and in 2003 became an adjunct professor at the Department of Electrical and Computer Engineering at the University of Calgary. He is the principal investigator in a number of research projects that investigate software verification, validation and evaluation issues across various domains, including embedded, web-based and ubiquitous environments. He has published over one hundred refereed journal and conference papers on Software and Systems Engineering (see www.steam.ualberta.ca for details on recent directions); and currently serves on the program committee for the IEEE International Symposium on Empirical Software Engineering and Measurement; and sits on the editorial board of the Journal of Empirical Software Engineering.