

Extending UML Use Case Diagrams to Represent Non-Interactive Functional Requirements

Saqib Iqbal*, Issam Al-Azzoni*, Gary Allen**, Hikmat Ullah Khan***

**Department of Software Engineering and Computer Science, Al Ain University, Al Ain, UAE*

***Department of Computer Science, University of Huddersfield, UK*

****Department of Computer Science, COMSATS University Islamabad, Wah Campus, Pakistan*

saqib.iqbal@aaau.ac.ae, issam.alazzoni@aaau.ac.ae, g.allen@hud.ac.uk,
hikmat.ullah@ciitwah.edu.pk

Abstract

Background: The comprehensive representation of functional requirements is a crucial activity in the analysis phase of the software development life cycle. Representation of a complete set of functional requirements helps in tracing business goals effectively throughout the development life cycle. Use case modelling is one of the most widely-used methods to represent and document functional requirements of the system. Practitioners exploit use case modelling to represent interactive functional requirements of the system while overlooking some of the non-interactive functional requirements. The non-interactive functional requirements are the ones which are performed by the system without an initiation by the user, for instance, notifying something to the user or creating an internal backup.

Aim: This paper addresses the representation of non-interactive requirements along with interactive ones (use cases) in one model. This paper calls such requirements ‘operation cases’ and proposes a new set of graphical and textual notations to represent them.

Method: The proposed notations have been applied on a case study and have also been empirically evaluated to demonstrate the effectiveness of the new notations in capturing non-interactive functional requirements.

Results and Conclusion: The results of the evaluation indicate that the representation of operation cases helps in documenting a complete set of functional requirements, which ultimately results in a comprehensive translation of requirements into design.

Keywords: Use Case modeling UML Requirements Engineering Functional Requirements

1. Introduction

Software Engineering is concerned with developing software as per the stakeholders’ expectations (requirements). Gathering, eliciting and documenting these requirements is the most crucial phase of the software engineering process. During this phase, detailed software requirements specification (SRS) documents are developed to specify the system requirements. One of the most widely used requirements specification tools is use case modelling, which represents the system

users (actors) and their interactive requirements (use cases). Use case modelling was proposed by Jacobson [1] and was later adopted by the Unified Modelling Language (UML) [2]. The purpose of use case modelling is to represent requirements in such a way that all stakeholders (from a technical or non-technical background) could easily understand and review them [3]. Use case modelling has been considered as an effective tool by the academic research community [4, 5] and industry [6, 7] to specify and model functional requirements. There are, however, some functional

requirements which often are not represented as use cases as they are not initiated by an actor. Examples of such requirements in a simple ATM banking system would be ‘Check Cash Dispenser’, ‘Notify Bank About an Empty Cash Dispenser’ or ‘Display Promotional or Informational Messages’. These functions are triggered either in response to an interactive requirement (a use case), an event, or at a specified time according to the internal system clock. We may call these requirements non-interactive requirements as they are triggered by the system without users’ initiation. System specification is not complete without the representation of these requirements along with use cases.

A use case, as the name suggests, has always been considered as a case of usage of the system, a usage scenario in other words. The representation of a requirement that does not represent a usage scenario (a non-interactive requirement) as a use case would only lead to confusions. There is a need for a separate representation for such requirements which is graphically and textually different from a use case. Both types of requirements, however, are needed to be represented in one model because they are both functional requirements and their representation in one model would provide a single point of reference for a complete list of functional requirements.

To cater for this need, we propose a new construct called ‘*Operation Case*’. The Operation Case is a system function which is not initiated by an actor, rather is triggered either by a use case or is initiated in response to an event or system clock. Operation cases are modelled along with the use cases in the same subject (system or module) to represent a full set of functional requirements of the subject. The construct has been added to the use case models in addition to other constructs by introducing a new profile to UML. The proposed constructs have been applied on a case study to show the comprehensiveness of the approach in representing functional requirements. In addition, an empirical evaluation has been conducted. The evaluation focuses on addressing the hypothesis that the proposed constructs and method represent a comprehensive list of functional requirements which eventually

leads to a complete and consistent design. The empirical evaluation demonstrates that use case modelling without operation cases can lead to overlooking of key functional requirements.

The rest of the paper is organized as follows: Section 2 provides a review of the related literature. Section 3 outlines the problem and motivations behind the research. Section 4 describes operation cases in detail. Section 5 provides details of the implementation of the proposed concepts. Section 6 illustrates the application of the new notation via a worked case study. Section 7 reports on the results of a controlled experiment. To address possible internal validity threats to the conclusion of the controlled experiment, a second experiment was conducted and it is presented in Section 8. Section 9 concludes the paper with a discussion on the future work.

2. Related work

In [8], Glinz identifies and demonstrates several deficiencies of UML, with emphasis on use case models and system decomposition. Our work attempts to address two of the deficiencies mentioned. The first deficiency is the omission of active objects in UML use case diagrams. Inclusion of active objects in use case diagrams is needed to specify interaction requirements where the system itself initiates an interaction between the system and an external actor. The use of observers in our new notation fills this gap. The second deficiency is that UML use case models cannot express state-dependent system behaviour adequately. We address this issue by introducing operation cases, which can be specified to capture the system state.

Several papers have also presented problems and limitations of use cases. The paper by Génova *et al.* [9] identifies sources of ambiguity that exist in use case models. The paper by Metz *et al.* [10] looks at the problem of use case interleaving present in UML 1.3. In [11], the authors highlight major problems associated with the semantics of extension points and rejoin points, which are used as branching and return locations for a use case’s alternative interaction

courses. The paper by Simons [12] traces the unstable semantics of use cases from Jacobson [13] to UML 1.3.

The paper by Tiwari and Gupta [14] presents a systematic literature review that examines the evolution of use cases, their applications, quality assessments, open issues and the future directions. In addition, the paper identifies a total of twenty existing templates that are used to specify use cases. In [15], the authors investigate via empirical studies the comprehension and learnability aspects of these templates.

The paper by Misbhauddin and Alshayeb [16] proposes an extension to the UML use case metamodel. The extended metamodel captures both the structural and behavioural views of use cases. The aim is to exploit the extended metamodel for model composition, model evaluation, and model interchange.

In [17], the authors propose an extension to the UML metamodel for presenting a refinement relationship between two use cases. The authors discuss the differences between include and refine relationships. The refinement of a use case results in more detailed use cases. Refinement can be defined by decomposing a use case according to the parts that compose the object of that use case, or according to the activities that compose the use case being refined. In our new notation, we do not attempt to represent use case refinement, but rather we introduce the concept of operation case to model non-interactive requirements. In [17], both the refining and refined use cases represent external functionality of the system and thus they agree with the UML definition of use cases. In our work, an operation case is not a type of use case. Other work that builds on use case refinement is [18]. There the authors present an approach to decompose a use case model into models at several levels of abstraction. The authors extend the UML use case metamodel with a refine relationship between a use case and a Use-CaseModel. For each abstraction level, several use case diagrams are used to capture the use cases at that abstraction level.

Several authors have attempted to formalize use case notations. In [19], the control-flow semantics of use cases is described in terms of

control-flow graphs. The technical report by Hurlbut [20] presents a survey of approaches for describing and formalizing use cases. The paper by Metz *et al.* [21] provides definitions for different types of alternative interaction courses in the context of goal-driven requirements engineering. Stevens [22] explores how UML use case notations can be formalized. Savic *et al.* in [23] propose the idea of use case specification at different levels of abstraction: interaction, behaviour, and user interface levels. Each abstraction level extends the previous level. The interactions in a use case are specified using the SilabReq language, which is a textual domain specific language.

The paper of Al-alshuhai and Siewe [24] proposes an extension to the UML use case diagram with new notations to model context-aware applications. The proposed extension, called a use context diagram, allows the modelling of context-aware requirements in addition to the functional requirements of a software application. The new notations include new metamodel elements such as Context Sources and Use Contexts as well as a new utilise relationship between Use Contexts. This extension is useful to cater for the modelling and analysis of the requirements of context-aware applications. We note that our new notation can also be used to model context-aware requirements: an Operation Case can be used as a Use Context to model sequences of actions a system performs to acquire, aggregate, or infer context information. A Context Source can be represented as an Observer that measures context information, and a trigger relationship replaces the utilise relationship.

A systematic literature review on producing high quality use case models is presented by El-Attar and Miller [25]. In their work, twenty six anti-patterns are suggested. A use case anti-pattern explains a repeated pattern in use case models that may initially appear beneficial but ultimately may cause deficiencies [25]. Modellers can exploit these anti-patterns to improve the quality of their use case models.

Identifying use cases can be very useful for the subsequent phases in software development. For instance, Yue *et al.* have created a tool to automatically generate a UML analysis model

comprising class, sequence, and activity diagrams from a use case model [26]. The tool also supports the automatic generation of traceability links between model elements of the use case model and the generated analysis model. Wang *et al.* have proposed an approach for automatically generating executable test cases by exploiting the behavioural information described in use case specifications [27]. The use cases are assumed to be specified in a restricted form of use case specification called Restricted Use Case Modelling (RUCM) [28]. Kesserwan *et al.* present an approach for generating test artefacts from scenario models through model transformation [29]. In the proposed approach, the scenarios are deduced from use case specifications written in the Cockburn use case notation [30]. It is of interest to apply such work on operation cases as well.

In the book by Smialek and Nowakowski [31], the authors present a language specific for requirements modelling, called the Requirements Specification Language (RSL). RSL forms the basis for the framework of model transformation and code generation presented in the book. Functional requirements in RSL are defined mostly through use case models. Use cases in RSL are derived from UML, but several new and changed features exist. These changes are due to the ambiguous semantics of the use case models, as defined in the UML specification [31]. RSL is designed to be a comprehensive language to model use case scenarios while linking those scenarios to their respective domain model elements. We note that the authors define use cases in relation to outside actors. In their definition, a use case starts with the interaction of an outside actor with the system. Hence, RSL seems to capture interactive functional requirements only. The ability of RSL to model non-interactive requirements requires further investigation.

Use cases can also be useful for effort estimation in use case driven projects. Qi and Boehm [32] have proposed an effort estimation model based on a use case model that can be used to estimate project effort during the early iterations in system development. In their work, the size metrics are defined based on the artefacts of a use case model. For example, the Early

Use Case Points (EUCP) metric is a size metric that weights each use case with the number of scenarios identified from the use case description. We believe that operation cases can be dealt with in a manner similar to use cases and they can be useful in effort estimation as well. Use Case Points (UCP) is a software effort estimation technique based on the use case model. A review of effort estimation frameworks and tools based on UCP is provided in [33].

3. Problems and motivation

There have been a number of efforts to extend use case models for representing non-functional requirements [8, 34] along with use cases but there is no evidence in the literature of addressing representation of non-interactive requirements. These requirements are the functional requirements, which are not initiated by an actor rather are triggered by a use case, event or the system clock. For instance, ‘turn on power saving mode’ in a mobile phone operating system is triggered in response to the battery level dropping to a certain level. Similarly, messages to the user, such as ‘battery fully charged’, ‘new SMS received’, or ‘application update available’ are also triggered without the user’s involvement. We may call these requirements non-interactive requirement as they are performed by the system without interaction with the user.

To illustrate these non-interactive requirements in more detail, let us consider Library Management System with the use case diagram in Figure 1. The functional requirements of the system would be:

- R1:** The librarian shall be able to add items such as books, journals and magazines to the system.
- R2:** The librarian shall be able to issue a library item to a user.
- R3:** The librarian shall be able to return an issued item.
- R4:** The librarian shall be able to send a request for a new library item to a vendor.
- R5:** The user shall be able to search for a library item.

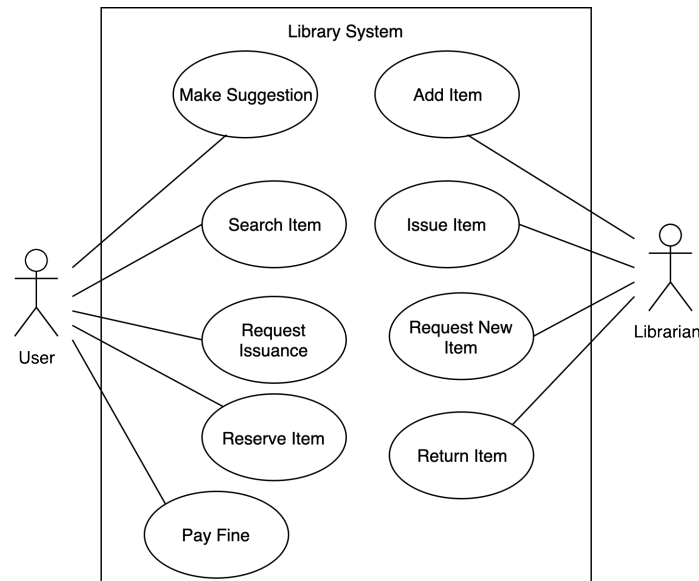


Figure 1. The use case diagram of Library Management System

- R6:** The user shall be able to request issuance of an item that is available.
- R7:** The user shall be able to reserve an item if the item is already issued to someone else.
- R8:** The user shall be able to pay fines where these have been incurred.
- R9:** The user shall be able to make suggestions for new library items.
- R10:** The system shall notify the user when a reserved book becomes available.
- R11:** The system shall calculate fine for late returns, with fines accruing each day after 15 days of issuance of an item.
- R12:** The system shall notify the user of any fines every 3 days.
- R13:** The system shall notify the suggestions provided by the users to the librarian.
- R14:** The system shall make backups at specified times.

The use case diagram of these functional requirements, shown in Figure 1, captures the interactive requirements, but is unable to capture the system requirements, R10 to R14. These requirements are as important for the complete functionality of the system as any other requirement, but they cannot be captured/represented by a use case model as they are not initiated by an actor.

The design artefacts extracted from the use case model would not include these requirements,

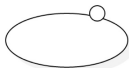

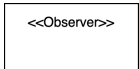
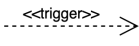
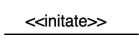
which would eventually lead to incomplete implementation. Representation of these requirements at use case modelling level would make the requirement specification complete and compliant with the system goals. More detailed discussion of this example is provided in Section 7.

4. Operation cases

We propose a new construct for the representation of non-interactive requirements, called ‘operation cases’. An operation case is an internal function which is initiated within the system either by a system timer, an event observer, or a use case. An operation case cannot initiate other operation cases, but can interact with the user, for instance, in case of a dialogue input or a message display to the user. The rationale behind the name of the operation case is that an operation case will represent a complete scenario of an internal operation. It has a separate notational representation to distinguish it from a use case. A complete use case model would include both use cases and operation cases representing all functional requirements identified during the analysis phase, which later will be translated into design mechanisms and design constructs.

A complete list of new notations and related associations is given in Table 1.

Table 1. Notations and Descriptions of New Constructs

Construct	Notation	Description
Operation Case		An <i>Operation Case</i> specifies a set of actions performed by its subjects, which may or may not yield an observable result that is of value for one or more <i>Actors</i> or other stakeholders of each subject. The actions in an <i>Operation Case</i> can only be triggered by an action in a <i>Use Case</i> or initiated by an <i>Observer</i> or a <i>Timer</i> .
Timer		A <i>Timer</i> represents an internal clock of the system or a specific interval of time represented in the implementing software.
Observer		An <i>Observer</i> represent a system component that initiates <i>Operation Cases</i> in response to an internal or external event.
Trigger		<i>Trigger</i> relationship defines that a <i>Use Case</i> triggers an <i>Operation Case</i> .
Initiate		<i>Initiate</i> relationship defines that an <i>Observer</i> or a <i>Timer</i> initiates an <i>Operation Case</i> .

5. Extension to use case modelling

The UML [1] is a widely used modelling language for representing and designing structural and behavioural properties of a system. It provides graphical models and notations that help in modelling internal and external behaviour of a system and representing the structural organization of system modules. Although UML is the most popular visual modelling language in software design, it only supports one paradigm of software design, which is object-oriented design. To counter this problem, the Object Management Group (OMG), the proprietary owner of UML, has proposed UML 2.0, which offers flexibility of extending UML diagrams and design notations. The extension is achieved through the introduction of profiles. A UML profile is an element of the UML; it is defined inside the UML metamodel [2]. Profiles are used to extend classes of the UML metamodel with additional stereotypes, tagged values, and constraints. The stereotypes are used to distinguish similar design notations representing different concepts; the tagged values are new attributes attached to a design construct; whereas constraints are used to introduce invariants and semantic-related limitations on a design diagram or a notation.

5.1. Operation cases profile definition

Since the operation cases introduce a new notational concept in the use case model, we intro-

duce a new profile, named *OperationCasesProfile*, to extend UML metaclasses. The new profile defines several stereotypes which extend standard UML metaclasses. Figure 2 shows the new operation cases profile. The new stereotypes are: *OperationCase*, *Trigger*, *Initiate*, *Observer*, and *Timer*. Their corresponding icons are shown in Table 1.

An *OperationCase* extends the UML metaclass *UseCase*. An *OperationCase* specifies some behaviour that a *subject* can perform. An *OperationCase* defines an offered behaviour of the *subject* with possible reference to its internal structure. Similar to a *UseCase*, an *OperationCase* may apply to any number of *subjects*.

An *OperationCase* may include or extend any number of other *OperationCases*, but may not include or extend any other *BehaviouredClassifiers* (i.e. *UseCases* or *Actors*). In addition, an *OperationCase* cannot be included or extended by a *UseCase*. The definitions of the *Include* and *Extend* relationships between *OperationCases* is the same as those between *UseCases*.

A new relationship added by the profile is the *Trigger* relationship. It is a relationship from a *UseCase* to an *OperationCase*. It specifies that a *UseCase* triggers an *OperationCase*. *Trigger* extends both *Include* and *Extend* metaclasses, such that the source is the triggering *UseCase* and the target is the triggered *OperationCase*. This indicates that the behaviour of the *OperationCase* is triggered while the behaviour of the *UseCase* is being executed. In UML profiles, if a stereotype

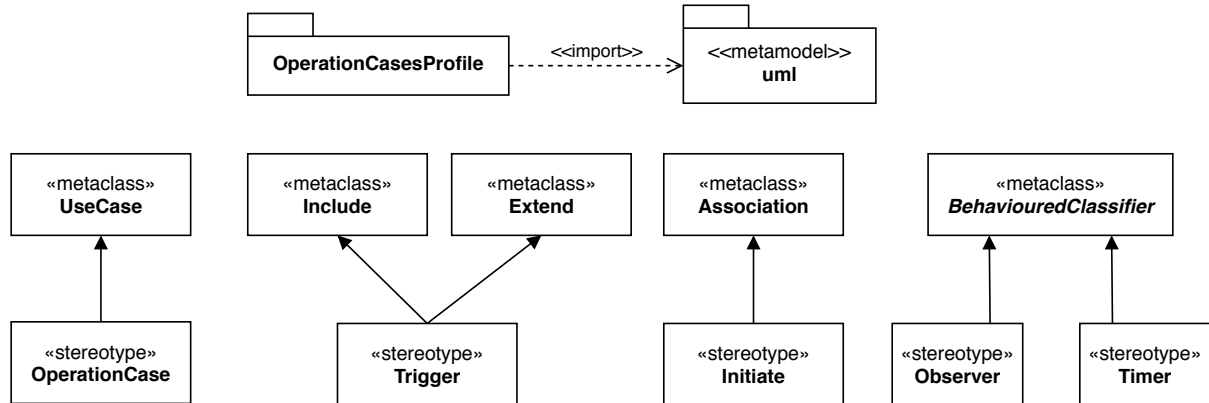


Figure 2. The operation cases profile

extends several metaclasses, it can only be applied to exactly one instance of one of those metaclasses at any point of time. The rationale for extending both *Include* and *Extend* metaclasses is that the behaviour of a triggered operation case can be inserted into the behaviour of the including operation case (in the case of *Include*), or can be added, possibly conditionally, to the behaviour of the extended operation case (in the case of *Extend*).

When an *OperationCase* applies to a *subject*, it specifies a set of behaviours performed by the *subject*. These behaviours can be triggered by an action in a *UseCase* or initiated by an *Observer* or a *Timer*. An *Observer* extends the UML metaclass *BehiouredClassifier*. Part of a subject, an *Observer* observes events and when an observed event occurs it causes (initiates) the execution of the behaviour of an associated *OperationCase*. A *Timer* also extends *BehiouredClassifier*. Part of a subject, a *Timer* initiates an operation case when its time interval expires. *Observers* are useful to model behaviours that are initiated by internal or external events. *Timers* are useful to model behaviours that are initiated at specified times according to an internal system clock. The *OperationCasesProfile* defines an *Initiate* stereotype which extends the UML metaclass *Association*. An *Initiate* association is between a *Timer* or an *Observer* on one end of the association and an *OperationCase* on the other end.

An *OperationCase* cannot be associated with *Actors*. Rather, it can only be associated with *Timers* or *Observers*. An *Actor* interacts with a subject through its associated *UseCases* which can indirectly trigger *OperationCases*.

We also add the following constraints to the operation cases profile:

- *OperationCases* can only be involved in binary associations.

```
context OperationCase
inv: Association.allInstances() ->
  forAll(a | a.memberEnd.type ->
    includes(self) implies
      a.memberEnd->size() = 2)
```

- An *OperationCase* cannot include *OperationCases* that directly or indirectly include it.
- ```
context OperationCase
inv: not allIncludedOperationCases()
 -> includes(self)
```

Here, the operation *allIncludedOperationCases()* returns the transitive closure of all *OperationCases* included by this *OperationCase*.

- An *OperationCase* must have a name.
- ```
context OperationCase
inv: name -> notEmpty ()
```
- An *Observer* must have a name. The same is true for a *Timer*.

```
context Observer
inv: name -> notEmpty ()
```

- An *Observer* can only have Associations to *OperationCases*. Furthermore, these Associations must be binary. The same is true for *Timers*

```
context Observer
inv: Association.allInstances() ->
  forAll(a | a.memberEnd ->
    collect(type) -> includes(self)
    implies
      (
        a.memberEnd -> size() = 2 and
```

```

let
  observerEnd : Property =
  a.memberEnd -> any(type = self)
in
  observerEnd.opposite.class.
    oclIsKindOf(OperationCase)
)
)

```

5.2. Operation case template

Jacobson [1] introduced a use case template to represent and document the description of a use case. Due to the complexity and unneeded formalism within the template, several variations have been introduced [35–39]. Operation cases are represented in a similar textual template, as shown in Table 2. The template contains a description of constituent items of an operation case, its associations, and related details. The template also mentions the requirements which are represented by the operation case. This helps to improve documentation and traceability.

6. Application of new notations

We have selected a subset of functional requirements of a simple mobile phone system for the

sake of simplicity. The selected subset of requirements is summarised according to their classification below:

- **Interactive Functional Requirements:**
 - Make a phone call,
 - Receive a phone call,
 - Send a message,
 - Add a contact,
 - Set an alarm.
- **Non-Interactive Functional Requirements:**
 - Transmit data to the service provider,
 - Manage Contact Book (This requirement is concerned with system adding new contacts and placing them in alphabetical order),
 - Receive Push Notifications,
 - Turn on Power Saving Mode in the case that the battery is lower than a threshold,
 - Notify user of updates,
 - Make the phone ring on receipt of an incoming call,
 - Sound an alarm at the required time.

Figure 3 shows the traditional representation of these requirements in a use case diagram. The non-interactive requirements are missing from this model as they do not represent any usage scenario. This model is supposed to

Table 2. Operation Case Template

Operation Case ID:	
Operation Case Name:	
Requirement ID:	
Created By:	Last Updated By:
Date Created:	Date Last Updated:
Description:	
Pre-conditions:	
Post-conditions:	
Priority:	
Frequency of Use:	
Normal Course of Events:	
Alternative Courses:	
Exceptions:	
Includes:	
Triggered/Initiated By:	
Special Requirements:	
Assumptions:	
Notes and Issues:	

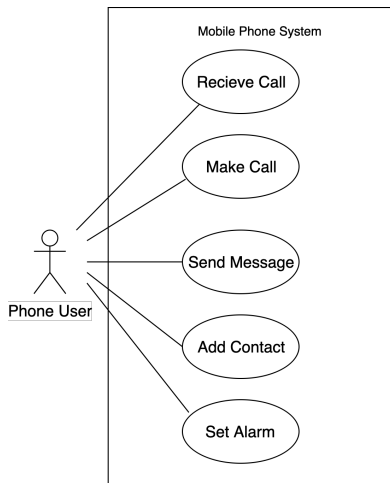


Figure 3. Use case diagram of a mobile phone system

be translated into design artefacts and models, but if the model is taken as a complete set of functional requirements, a number of critical requirements (non-interactive requirements) may be overlooked. Figure 4, on the other hand, shows a representation of a complete set of requirements. The requirements, such as ‘Turn on power saving mode’ or ‘Receive push notification’, are represented along with other interactive functional requirements in the same sub-system boundary.

As can be seen, the basic use cases remain the same, showing how the user will interact with the system. However, we can also see that:

- The “Receive Call”, “Make Call”, and “Send Message” use cases each trigger a “Transmit Data” operation case. In traditional use case modelling this could be modelled as a step in the primary path of each of the three separate use cases, but would not be shown on the diagram. By triggering an operation case the shared nature of this functionality becomes explicit, thus both simplifying the descriptions of the individual use cases and capturing the relationships between these functional concerns.
- The “Add Contact” use case triggers the “Manage ContactBook” operation case. It could be argued that the latter is simply a step in the primary path of the former, how-

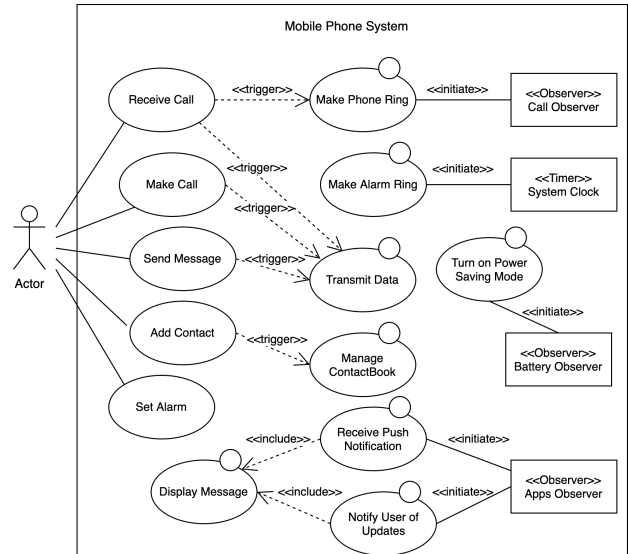


Figure 4. Revised use case diagram of a mobile phone system with operation cases

ever we would argue that our model is clearer and it allows for reuse of the “Update Contact Book” operation case. Additionally, the implementation of the operation case would need to deal with issues such as poor network connectivity and failure to connect to the server. This functionality would sit more sensibly in the “Update Contact Book” operation case than in the “Add Contact” use case.

- Three observers have been implemented, two of which monitor incoming connections (incoming calls and incoming push notifications), and one of which monitors the battery and turns on power saving mode when required. This latter example is a classic case of functionality that is difficult to represent clearly using a traditional use case model. There is no actor to drive the functionality, as it is not an interactive use case, instead being event driven. The use of the observer makes this internal functionality explicit.
- One timer driven event has also been implemented. This is the “Sound Alarm” operation case, which is initiated by a timer. Again, standard use case models do not allow for the representation of such functionality.

Note that, while a use case can trigger an operation case, the reverse is not true. Take as an example the “Make Phone Ring” operation case. It may, at first sight, appear that this could in

turn trigger the “Receive Call” use case. However, making the phone ring does not necessarily cause the user to answer it. The user may be away from the phone, the phone may be on silent, or the user may simply choose to ignore the incoming call. It is not, therefore, possible to assume that the operation case will cause the user to react. Similarly, the “Display Message” operation case may cause the user to read that message, but this is not certain, so the operation case cannot trigger a “Read Message” use case, nor can the operation case deliver data to the user via a directed association. In essence, an operation case may be triggered by use cases, but not vice-versa.

Having modelled the use cases and operation cases in a diagrammatic form, the next step is to write up detailed use case and operation case descriptions. Traditional use case modelling always includes this step, and several standards have been suggested for the layout of use case descriptions. These standards tend to be very similar, and the one chosen for use here is one of the selection that can be found at [40]. One example operation case description, for the Transmit Data operation case, is given in Table 3.

As can be seen, the primary, alternative, and exception paths of several of the use cases can be simplified by the use of operation cases, as their use helps to partition the system behaviour, and to identify and abstract out shared or common functionality. As an example, the Send Message use case in the traditional model contains the following steps in the primary path:

5. The phone handset connects to the mobile network and attempts to send the SMS message.
6. The message is sent successfully.

and the following alternatives or exceptions:

- 5.1. If the mobile network is unreachable then the phone will retry at intervals until successful.
- 5.2. If the receiver’s phone is unreachable (*e.g.* a wrong number or the phone is switched off) then an error is displayed to the phone user.

All of this behaviour can be abstracted out to the Transmit Data operation case, thus simplifying the use case description.

In order to integrate Operation Cases fully into the requirements model a small number of amendments have been made to the use case de-

scription template. The template has also been adapted for the description of Operation Cases. The changes proposed are:

- A new section, “Triggers:” has been added to the Use Case Description template. This section lists the Operation Cases that can (optionally) be triggered by the use case.
- When describing Operation Cases, the “Use Case ID” and “Use Case Name” have been changed to “Operation Case ID” and “Operation Case Name”.
- When describing Operation Cases, the “Actor” section has been removed.
- When describing Operation Cases, a new section, “Triggered/Initiated By:” has been added to the template. This section is used to list the use cases, operation cases, observers, or timers that can trigger or initiate the operation case.

With these modifications and additions to the template, we have provided a clear mechanism for the description of all use cases and operation cases within the system model.

We can see that the operation case “Transmit Data” is triggered by the “Make Call”, “Receive Call”, and “Send Message” use cases. The clarity of this information aids understanding of the structure of the requirements, and helps software engineers identify shared and core functionality, in a way that traditional use case modelling is unable to support. This should in turn help software architects with the design of the software, and help project managers to prioritise the development of the system components.

7. A controlled-experiment based evaluation

This section reports on a controlled experiment that was conducted to test whether using use case diagrams extended with operation cases results in a comprehensive system design that incorporates both interactive and non-interactive functional requirements. Section 7.1 presents the research question and hypothesis. Section 7.2 presents the research design, and Section 7.3 presents and discusses the results.

Table 3. Transmit Data Operation Case Description

Operation Case ID:	MP_OC1		
Operation Case Name:	Transmit Data		
Requirement ID:	SR15, SR20, SR35		
Created By:	GA	Last Updated By:	GA
Date Created:	25 May, 2018	Date Last Updated:	29 May, 2018
Description:	This is a background process running on the phone which receives data from apps such as the dialler app and the messaging app and uploads those data to the mobile network.		
Pre-conditions:	The phone must be switched on and connected to the mobile network.		
Post-conditions:	None.		
Priority:	High – this is a core piece of functionality.		
Frequency of Use:	Variable, depending on the usage patterns of the user.		
Normal Course of Events:	<ol style="list-style-type: none"> 1. The operation case receives a request to upload data from another use case or operation case. 2. A connection to the mobile network is opened. 3. The data are uploaded. 4. The connection to the mobile network is closed. 		
Alternative Courses:	<ol style="list-style-type: none"> 2.1 A connection cannot be established. Try again. 3.1 The data does not upload correctly. Try again. 		
Exceptions:	If at any time the connection is lost and cannot be re-established within 1 second, then the operation case will return an error to the calling use case or operation case.		
Includes:	None.		
Triggered/Initiated By:	Triggered by use cases MP1a Make Call; MP2a Receive Call; MP3a Send Message.		
Special Requirements:	None.		
Assumptions:	None.		
Notes and Issues:	None at present.		

7.1. Research question and hypothesis

Our main proposition is that the UML use case diagrams fail to represent non-interactive functional requirements. Therefore, a systems analyst following an object-oriented development methodology that uses these diagrams to design the system, *i.e.*, construct the class diagram, will likely fail to include the necessary methods to implement the system's non-interactive functional requirements. The end result is a design

and an implementation of a system that do not implement all functional requirements and for which late changes are likely to be costly.

We carried out a controlled experiment to investigate the following research question:

Will using extended use case diagrams help systems analysts to not miss incorporating non-interactive functional requirements in system design?

The experiment was structured as follows. We developed a system story and asked the partici-

pants to draw the class diagram. We separated the participants into two equal groups: participants in the **UC group** were requested to draw the use case diagram for the system first and use it to draw the class diagram, while participants in the **OC group** were instructed on the use of operation cases and extended use diagrams and subsequently requested to draw the extended use case diagram and use it to draw the class diagram. More details on the participants are provided in Section 7.2.

In relation to our research question, we formulated the following hypothesis:

The number of correctly identified methods and classes related to non-interactive functional requirements will be higher in the OC group compared to the UC group.

The research variables are as follows. The independent variable is the diagram used: the use case diagram in the case of the UC group or the extended use case diagram in the case of the OC group. The dependent variable is the completeness of the class diagram with respect to incorporating the non-interactive functional requirements. This is captured in a score that ranges from 0 to 9. The scoring system is described in the next section.

7.2. Research design

There were 14 participants in the experiment. The participants were asked to fill a consent form before participating in the experiment. These participants were undergraduate students taking core courses in the program of Software Engineering in the College of Engineering at Al Ain University of Science and Technology. The OC group consisted of 7 students taking the course “Formal Specifications and Design Methods”, and the UC group consisted of 7 students taking the course “Software Measurement and Testing”. These courses were selected because they are advanced courses and the prerequisite course for both courses is “Software Requirements and Specification”. In this prerequisite course the students study capturing and representation of requirements. The students have completed courseworks and projects in which they have gathered and

represented requirements using use case modelling.

The experiment was conducted in the form of two voluntary quizzes; one in each course. The quizzes were conducted on separate days, and involved the participation of the first two authors. The students were given an incentive in the form of bonus points, which would be added to their final grade for the course. To encourage the students, the number of bonus points for each student were tied with the student’s score as follows: 3 points to scores of 7 or more, 2 points to scores of 4–6, and 1 point to scores of 3 or less. During the selection of students it was also ensured that the average Cumulative Grade Point Average (CGPA) of both groups was the same (UC group = 2.54/4, OC group = 2.62/4).

The students had been informed of the voluntary quiz and the bonus points in the earlier class. They were simply asked to show up in the same classroom at the regular class time. There were 10 students who attempted the quiz from the course “Formal Specifications and Design Methods”. On the other hand, only 7 students attempted the quiz from the course “Software Measurement and Testing”. Since the cumulative degree averages of both groups are different, we only scored a subset of the students who attempted the quiz in the “Formal Specifications and Design Methods” class. We ranked the 10 students in terms of their cumulative degree averages, and then we selected a sequence of 7 students such that the group’s cumulative degree average was close to the that of the group of the second course. The attempts by the other students were discarded; these attempts were never evaluated.

The experimental procedure was as follows: the first author gave a half-an-hour tutorial reviewing use case modelling. For both groups, the tutorial included a review of use cases and use case diagrams. An example system story was used in both tutorials. The instructor of the tutorial worked out an exercise developing a use case diagram that modelled the functional requirements presented in the example system story. The instructor presented how to create a class diagram based on the identified use cases.

In particular, the instructor reminded the students of the usefulness of use case scenarios in identifying the classes and their methods. The instructor encouraged the students to apply what they had learned in their earlier courses such as the use of sequence diagrams to model the use case scenarios and construct the class diagram. The instructor worked with the students on constructing the class diagram representing the initial design of the example system story.

The contents mentioned earlier were common in both tutorials. However, there were two key differences between the two. In the tutorial instructing the UC group, there was no mention of non-interactive requirements. The students were simply asked to apply what they already knew. On the other hand, operation cases and extended use case diagrams were introduced in the tutorial instructing the OC group. Non-interactive functional requirements were defined and discussed in this tutorial. These were also applied on the example system story. The students were recommended to use the identified operation cases in constructing the class diagram in a similar fashion to what they would do using use cases.

Figure 5 shows the system story. It describes the functional requirements of an online library management system. We selected this system since undergraduate students at this level are typically familiar with the services provided by the University's online library system. Therefore, they are familiar with library concepts such as searching for and reserving library items. The

figure includes the instructions handed to the UC group's students; for the OC group, the students were given the same instructions but were asked to first draw the extended use case diagram with operation cases rather than the standard UML use case diagram. The description includes a set of interactive requirements, such as requesting an item for issuance and requesting new items from vendors, in addition to a set of non-interactive requirements, such as the weekly back-up and the fine's calculation and notification requirements.

An expert solution in the form of a class diagram was created by the first author and checked by the second author. The class diagram includes five classes, including the class *Library* which is used as a system class implementing the methods that are necessary to realise some non-interactive requirements. A total of 19 methods were identified, including 6 methods to realise the non-interactive requirements. The form used in the evaluation of each student's work is presented in Figure 6. The six methods realising the non-interactive requirements are shown in bold.

Since the experiment is concerned with non-interactive requirements, we developed a scoring method for these requirements only. A student gets one point for each correctly identified non-interactive method, *i.e.*, one of the six methods realising the non-interactive requirements. If a student places a method in an incorrect class, they are not rewarded the point. The justification for this is that the use case diagram (or its ex-

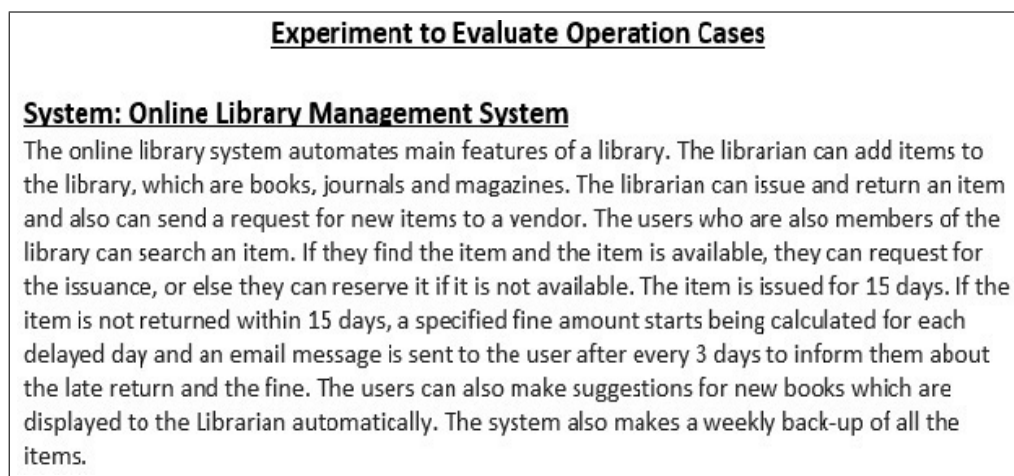


Figure 5. The description of Library Management System used in the evaluation

<u>Evaluation Form</u>	
Student ID:	
Completeness - Identification of Classes	
Library	<input type="checkbox"/>
LibraryItem	<input type="checkbox"/>
User	<input type="checkbox"/>
Librarian	<input type="checkbox"/>
Vendor	<input type="checkbox"/>
Score:	
Completeness – Correct Allocation of Methods to the Classes	
<u>Library</u> notifySuggestion() <input type="checkbox"/> notifyLateReturn() <input type="checkbox"/> notifyFine() <input type="checkbox"/> createBackUp() <input type="checkbox"/> <u>LibraryItem</u> searchItem() <input type="checkbox"/> reserveItem() <input type="checkbox"/> return() <input type="checkbox"/> issue() <input type="checkbox"/> calculateFine() <input type="checkbox"/> notifyAvailability() <input type="checkbox"/> <u>Vendor</u> supplyItem() <input type="checkbox"/>	<u>User</u> reserveItem() <input type="checkbox"/> returnItem() <input type="checkbox"/> makeSuggestion() <input type="checkbox"/> payFine() <input type="checkbox"/> requestIssuance() <input type="checkbox"/> <u>Librarian</u> • addItem() <input type="checkbox"/> • orderItem() <input type="checkbox"/> • issueItem() <input type="checkbox"/> • returnItem() <input type="checkbox"/>
Score:	
No. of Methods Identified:	
No. of Non-Interactive Methods Identified:	

Figure 6. The evaluation form used in evaluating a student's work

tended diagram variant) should help the systems analyst in building a complete and sound design. The only class that is critical to implement the non-interactive requirements is the system class *Library*. Four out of six non-interactive methods are in this class. Given its relevance, we assigned the weight of three points for identifying the *Library* class. Thus, the maximum score is nine points, including three points for the *Library* class and one point for each correctly identified non-interactive method. The presented scoring method is similar to [41], but it only considers non-interactive requirements.

The first author who presented the tutorials also evaluated the students' class diagrams. These were subsequently checked by the second author. Since the expert solution is not the only correct one, the evaluators were tolerant of

class diagram variations as long as the identified classes and methods were in line with the criteria mentioned earlier. For example, a student may use different method and class names and/or place a method in a different, but correct class.

7.3. Results and discussion

To address our research question presented in Section 7.1, we tested the following hypothesis which is similar to hypotheses in [41–43]:

H_0 : There is no difference between the scores of the UC and OC groups.

We tested the research hypothesis using the Mann–Whitney U test, as in [42, 44]. Mann–Whitney U test is a nonparametric test for the difference in two means [45]. The results of the test were obtained using XLSTAT which is a statis-

tical analysis tool for Microsoft Excel [46]. The findings indicate that the score of the OC group is significantly higher than the score of the UC group (p -value = 0.027), and therefore hypothesis H_0 is rejected. Note that a significance level of $\alpha = 0.05$ is chosen as the level of significance. On average, participants in the OC group scored significantly higher ($\bar{X} = 4.286$, $SD = 2.498$) than participants in the UC group ($\bar{X} = 1.286$, $SD = 2.215$; $p = 0.027$) (see Figure 7).

Below, we consider the four categories of threats to validity:

1. **Conclusion validity:** A study has conclusion validity if the results are statistically significant using appropriate statistical tests [47, 48]. We used the Mann–Whitney U test to analyse the results. The assumptions of using this test have been checked. In order to increase the reliability of measures [48], all student evaluations performed by the first author were checked by the second author. All solutions were checked against an expert solution that was constructed prior to the evaluation and checked for correctness and completeness by the authors.
2. **Internal validity:** Internal validity refers to the cause and effect relationship between the independent and dependent variables. One factor affecting this kind of validity is having any prior significant difference between
- the groups. In the design of our experiment, there was no significant difference between the groups with respect to the cumulative degree average. In addition, we believe that the exercise of identifying operation cases causes the systems analyst to identify non-interactive functional requirements, and thereby not miss incorporating them in system design. This is our rationale for why the independent variable would affect the dependent variable. One could argue that the participants in the OC group received direct training on identifying and modelling non-interactive requirements while the participants in the UC group did not. This could represent a threat to the internal validity of the experiment. To address this potential threat, we conducted a second experiment (see Section 8) that demonstrates that practitioner software engineers who typically create use case diagrams and follow the unified process of constructing use case diagrams first and using them to create the analysis and design level class diagrams are expected to miss some non-interactive requirements. This is because these non-interactive requirements are not emphasized (in fact, they are neglected) by the standard use case notations.
3. **Construct validity:** Construct validity concerns the use of measures that are relevant to the study. One factor affecting construct

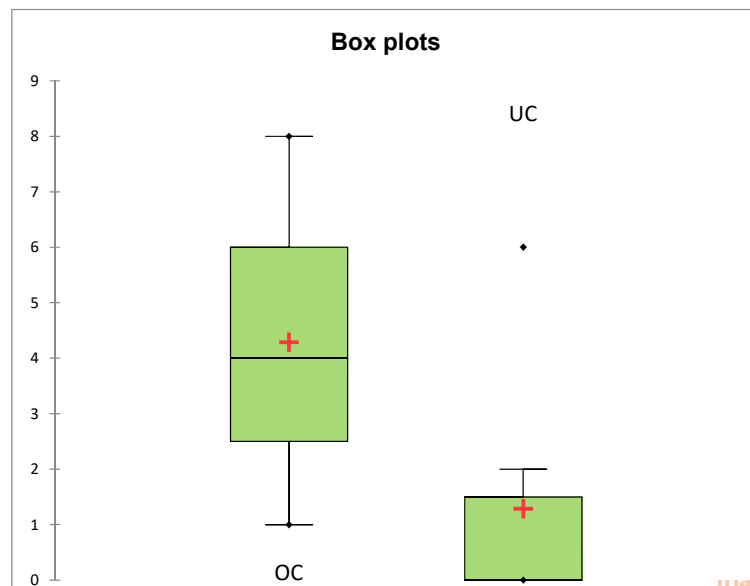


Figure 7. The box plots depicting the scores of the OC and UC groups

validity is how much the experimental setting differed from a real-world setting. The participants were not involved in a real-world system with real clients and users. However, with regards to the limitation of use case diagrams in modelling non-interactive requirements, the experimental setup highly resembles real-world conditions. A systems analyst cannot capture non-interactive requirements using a use case diagram only; and this has a significant impact on the completeness of system design and implementation. Another relevant factor is the use of meaningful measures of the completeness of design models with respect to non-interactive requirements. We used the same measure for completeness in terms of the number of identified methods in the class diagram as in [41]. We believe that this is a relevant measure since missing a method implies a design and an implementation that do not implement all functional requirements.

4. **External validity:** External validity refers to the generalisability of the results. One factor related to external validity concerns the fact that all participants were students. However, the study in [49], as noted in [42], found that there are minor differences between software engineering students and professional software developers suggesting the use of students instead of professional developers in software engineering experiments is valid under certain conditions. A second factor is related to the number of participants which is relatively small. Given the resources available on hand, this is the best subject population we could find. A third factor is related to the size of the task which is relatively small and does not reflect the typical work by a systems analyst. However, most software engineering experiments use such small tasks due to the inherent difficulty of measuring attributes of large and complex tasks [41–43, 47].

8. An Empirical evaluation using a case study

To address the threats to the internal validity of the controlled experiment presented in Section 7,

a second experiment was conducted. The experiment was in the form of a case study analysing the performance of 30 graduate students in the Master of Science in Computer Science program at COMSATS University Islamabad (Wah Campus) on identifying non-interactive requirements of a system. All of the students who participated in the study had a bachelor's degree and had completed at least one undergraduate-level course on object-oriented analysis and design. The majority of the students were part-time students who were actively working in industry. They were taking the course Advanced Topics in Object-Oriented Software Engineering at the time of the experiment. The experiment was conducted during regular class hours. All students completed the tasks during the regular class hours, although they had been informed that extra time would be given in case it was needed.

This experiment used the same system story as the one in the controlled experiment. The students were handed the description of the Library Management System shown in Figure 5. The task was identical to the task performed by the UC group in the controlled experiment, however no tutorial was provided on use case modelling. Information on each participant was collected with the submission, including the student id number, number of years since graduation, and current employment if any. The class diagrams developed by the students were collected and evaluated following the same scoring procedure as in the controlled experiment.

The same evaluation form was used as in the controlled experiment (shown in Figure 6). With respect to the number of non-interactive methods identified, the average score was 1.1 out of 6. On a 95%-confidence level, the confidence interval for the average score was (0.66, 1.54). The analysis was done using XLSTAT. This shows that many non-interactive requirements were missed and therefore not incorporated in the class diagram. Furthermore, there were only 20 participants who had identified at least one non-interactive method. Of these, 13 participants (*i.e.*, 65%) incorrectly represented the system itself as an actor in the use case diagram. This shows that the standard use case notation as practised might be inadequate in capturing non-interactive requirements.

Since the task done by the participants in this experiment is identical to that of the UC group in the controlled experiment, this experiment is likely to suffer from the same threats to validity as in the controlled experiment. However, the number of participants is much higher in this experiment. In addition, the participants have higher proficiency on use case modelling since they are graduate students with the majority working in industry. This experiment demonstrates that systems analysts who typically create use case diagrams and follow the unified process of constructing use case diagrams first and using them to create the analysis-level class diagrams are expected to miss some non-interactive requirements. This is because these non-interactive requirements are not emphasized (in fact, they are neglected) by the standard use case notations.

9. Conclusion and future work

The representation and documentation of functional requirements at the analysis phase is the most crucial activity in the software development life cycle. Use case modelling solves this problem by providing both textual and graphical methods, which makes it the most widely-used methodology. Use cases are designed into implementable modules in the next phase of the development life cycle. The problem, however, is the exclusion of some of the functional requirements in the use case models. These requirements are often not represented as use cases as they are not initiated by a user, and are thus known as non-interactive requirements. Such requirements are often directly addressed in the design phase without having any backward tracing to the use case models. It is evident from the available literature and existing software development practices that use case models are considered as a complete representation of all functional requirements of the system. Due to this practice, non-interactive requirements are often overlooked and consequently result in implementation of an incomplete system. To represent and document a complete system, non-interactive requirements need to be comprehensively represented and documented along

with interactive requirements (use cases) in the analysis phase. This paper addresses this problem and proposes an extension to use case models to accommodate non-interactive requirements. These requirements have been named as Operation Cases and are represented with a new set of graphical notations and textual templates. The paper presents a new profile to extend UML's use case notation with operation cases and their related constructs. The addressing of operation cases at the analysis phase allows analysts and designers to comprehensively document and trace the functional requirements effectively.

We applied operation cases in modelling a (partial) Mobile Phone operating system. For the sake of keeping it concise, only a few relevant functional requirements of the case study are discussed. In the case study, we showed that using use case models alone cannot represent internal non-interactive requirements of the system. Representation of operation cases solves this problem and makes use case models a more comprehensive graphical representation of the functional requirements of the system. A controlled experiment was also conducted to investigate the hypothesis that using operation cases results in more comprehensive designs than when using traditional use cases only. The results of the experiment confirmed our hypothesis.

Acknowledgement

This work has not received any funding.

References

- [1] I. Jacobson, "Object-oriented development in an industrial environment," in *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*, 1987, pp. 183–191.
- [2] "Unified modeling language," 2015, [Accessed September 2019]. [Online]. <http://www.omg.org/spec/UML/2.5>
- [3] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [4] B. Anda, K. Hansen, and G. Sand, "An investigation of use case quality in a large safety-critical

- software development project,” *Information and Software Technology*, Vol. 51, No. 12, 2009, pp. 1699–1711.
- [5] S. Tiwari and A. Gupta, “Does increasing formalism in the use case template help?” in *Proceedings of the 7th India Software Engineering Conference*, 2014, pp. 6:1–6:10.
- [6] D. Parachuri, A.S.M. Sajeev, and R. Shukla, “An empirical study of structural defects in industrial use-cases,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 14–23.
- [7] M. Ivarsson and T. Gorschek, “A method for evaluating rigor and industrial relevance of technology evaluations,” *Empirical Software Engineering*, Vol. 16, No. 3, 2011, pp. 365–395.
- [8] M. Glinz, “Problems and deficiencies of UML as a requirements specification language,” in *Proceedings of the International Workshop on Software Specification and Design*, 2000, pp. 11–22.
- [9] G. Génova, J.L. Morillo, P. Metz, R. Prieto-Díaz, and H. Astudillo, “Open issues in industrial use case modeling,” *Journal of Object Technology*, Vol. 4, No. 6, 2005, pp. 7–14.
- [10] P. Metz, J. O’Brien, and W. Weber, “Against use case interleaving,” in *Proceedings of the International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 2001, pp. 472–486.
- [11] P. Metz, J. O’Brien, and W. Weber, “Specifying use case interaction: Clarifying extension points and rejoin points,” *Journal of Object Technology*, Vol. 3, No. 5, 2004, pp. 87–102.
- [12] A.J.H. Simons, “Use cases considered harmful,” in *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, 1999, pp. 194–203.
- [13] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-oriented software engineering – A use case driven approach*. Addison-Wesley, 1992.
- [14] S. Tiwari and A. Gupta, “A systematic literature review of use case specifications research,” *Information and Software Technology*, Vol. 67, 2015, pp. 128–158.
- [15] S. Tiwari and A. Gupta, “Investigating comprehension and learnability aspects of use cases for software specification problems,” *Information and Software Technology*, Vol. 91, 2017, pp. 22–43.
- [16] M. Misbhauddin and M. Alshayeb, “Extending the UML use case metamodel with behavioral information to facilitate model analysis and inter-change,” *Software and Systems Modeling*, Vol. 14, No. 2, 2015, pp. 813–838.
- [17] S. Azevedo, R.J. Machado, A. Bragança, and H. Ribeiro, “The UML «include» relationship and the functional refinement of use cases,” in *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, 2010, pp. 156–163.
- [18] E.F. Cruz, R.J. Machado, and M.Y. Santos, “On the decomposition of use cases for the refinement of software requirements,” in *Proceedings of the International Conference on Computational Science and Its Applications*, 2014, pp. 237–240.
- [19] K. van den Berg and A.J.H. Simons, “Control-flow semantics of use cases in UML,” *Information and Software Technology*, Vol. 41, No. 10, 1999, pp. 651–659.
- [20] R.R. Hurlbut, “A survey of approaches for describing and formalizing use cases,” Department of Computer Science, Illinois Institute of Technology, Tech. Rep., 1997.
- [21] P. Metz, J. O’Brien, and W. Weber, “Specifying use case interaction: Types of alternative courses,” *Journal of Object Technology*, Vol. 2, No. 2, 2003, pp. 111–131.
- [22] P. Stevens, “On use cases and their relationships in the unified modelling language,” in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2001, pp. 140–155.
- [23] D. Savic, A.R. da Silva, S. Vlajic, S. Lazarevic, V. Stanojevic, I. Antovic, and M. Milic, “Use case specification at different levels of abstraction,” in *Proceedings of the International Conference on the Quality of Information and Communications Technology*, 2012, pp. 187–192.
- [24] A. Al-alshuhai and F. Siewe, “An extension of the use case diagram to model context-aware applications,” in *Proceedings of the SAI Intelligent Systems Conference*, 2015, pp. 884–888.
- [25] M. El-Attar and J. Miller, “Constructing high quality use case models: a systematic review of current practices,” *Requirements Engineering*, Vol. 17, No. 3, 2012, pp. 187–201.
- [26] T. Yue, L.C. Briand, and Y. Labiche, “aToucan: an automated framework to derive UML analysis models from use case models,” *ACM Transactions on Software Engineering and Methodology*, Vol. 24, No. 3, 2015, pp. 13:1–13:52.
- [27] C. Wang, F. Pastore, A. Goknil, L.C. Briand, and M.Z.Z. Iqbal, “Automatic generation of system test cases from use case specifications,” in *Proceedings of the International Sympos-*

- sium on Software Testing and Analysis, 2015, pp. 385–396.
- [28] T. Yue, L.C. Briand, and Y. Labiche, “Facilitating the transition from use case models to analysis models: Approach and experiments,” *ACM Transactions on Software Engineering and Methodology*, Vol. 22, No. 1, 2013, pp. 5:1–5:38.
- [29] N. Kesserwan, R. Dssouli, J. Bentahar, B. Stepien, and P. Labrèche, “From use case maps to executable test procedures: a scenario-based approach,” *Software and Systems Modeling*, 2017.
- [30] S. Adolph, A. Cockburn, and P. Bramble, *Patterns for Effective Use Cases*. Addison-Wesley Longman Publishing Co., 2002.
- [31] M. Smialek and W. Nowakowski, *From Requirements to Java in a Snap – Model-Driven Requirements Engineering in Practice*. Springer, 2015.
- [32] K. Qi and B.W. Boehm, “A light-weight incremental effort estimation model for use case driven projects,” in *Proceedings of the IEEE Software Technology Conference*, 2017.
- [33] M. Saroha and S. Sahu, “Tools and methods for software effort estimation using use case points model – A review,” in *Proceedings of the International Conference on Computing, Communication and Automation*, 2015, pp. 874–879.
- [34] M. Grossman, J.E. Aronson, and R.V. McCarthy, “Does UML make the grade? insights from the software development community,” *Information and Software Technology*, Vol. 47, No. 6, 2005, pp. 383–397.
- [35] D. Kulak and E. Guiney, *Use Cases: Requirements in Context*. ACM Press, 2000.
- [36] D. Liu, K. Subramaniam, B. Far, and A. Eberlein, “Automating transition from use cases to class model,” in *Proceedings of the Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology*, 2003, pp. 831–834.
- [37] P. Kruchten, *The Rational Unified Process: An Introduction*, 3rd ed. Addison-Wesley, 2003.
- [38] S.S. Somé, “Supporting use case based requirements engineering,” *Information and Software Technology*, Vol. 48, No. 1, 2006, pp. 43–58.
- [39] J. Kettenis, “Getting started with use case modeling: White paper,” Oracle Corporation, Tech. Rep., 2007.
- [40] “40 use case templates and examples,” [Accessed September 2019]. [Online]. <http://templatelab.com/use-case-templates/>
- [41] B. Anda and D.I.K. Sjøberg, “Investigating the role of use cases in the construction of class diagrams,” *Empirical Software Engineering*, Vol. 10, No. 3, 2005, pp. 285–309.
- [42] D. Beigel and E. Kedmi-Shahar, “Improving the identification of functional system requirements when novice analysts create use case diagrams: the benefits of applying conceptual mental models,” *Requirements Engineering*, 2018.
- [43] F. Ricca, G. Scanniello, M. Torchiano, G. Reggio, and E. Astesiano, “Assessing the effect of screen mockups on the comprehension of functional requirements,” *ACM Transactions on Software Engineering and Methodology*, Vol. 24, No. 1, 2014.
- [44] M. Dahan, P. Shoval, and A. Sturm, “Comparing the impact of the OO-DFD and the use case methods for modeling functional requirements on comprehension and quality of models: a controlled experiment,” *Requirements Engineering*, Vol. 19, No. 1, 2014, pp. 27–43.
- [45] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers, 6th Edition*. John Wiley and Sons, 2013.
- [46] “XLSTAT,” [Accessed March 2019]. [Online]. <https://www.xlstat.com/en/>
- [47] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, 3rd ed. CRC Press, Inc., 2014.
- [48] C. Wohlin, P. Runeson, M. Hst, M.C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, 2012.
- [49] M. Höst, B. Regnell, and C. Wohlin, “Using students as subjects—a comparative study of students and professionals in lead-time impact assessment,” *Empirical Software Engineering*, Vol. 5, No. 3, 2000, pp. 201–214.