Interface-based software requirements analysis

Aziz Ahmad Rais, Rudolf Pecinovský
Department of Information Technologies
University of Economics, Prague
Prague, Czech Republic
aziz.ahmad.rais@gmail.com, rudolf@pecinovsky.cz

***Abstract* — Software requirements analysis is a critical phase in the software development life cycle [1]. It is usually carried out using one of the following: use cases [2], user stories [3] or business scenarios [4]. Use cases, user stories and business scenarios thus become inputs for the object-oriented analysis (OOA) of application software development. Most translating and mapping of software requirements from text to objects and classes creates problems with software acceptance. There is also a practical issue with object-oriented analysis, namely, that OOA and object-oriented design (OOD) both operate with the same objects and classes, and, in practice, it has not yet been possible to separate them from one another. The goal of this article is to provide a solution that will simplify software requirements analysis and separate such analysis from design, in order to give architects, developers and testers the ability to work independently through a contract (interface) that integrates their work.**

Keywords—Model-driven architecture, UML, Object-oriented analysis, Object-oriented design

I. INTRODUCTION

According to the ISO 12207 [1], there are two types of life cycle process: system-specific (system life-cycle) and software-specific (software life-cycle). The system-specific processes include activities that are not directly related to application software, such as, for example, project management related processes, and information system related processes, and so on. It is necessary to recognize that system life-cycle processes are important to software life-cycle processes. The most significant among these is the system requirements analysis process. The outcome of this last demonstrates how the whole organization is involved in the process that produces value and creates its core business.

Most organizations underestimate the value of enterprise-level processes [4], business requirements and business processes. Software developers must, therefore, clarify these business requirements before software requirements analysis, with the help of various business techniques, such as brainstorming, business rules, data dictionaries and glossaries, data diagrams, data modelling, decision analyses, functional decomposition, interviews, prototyping, risk analyses, scenarios and use cases, user stories, sequence diagrams, state diagrams, and so on [6]. The techniques listed by BABOK [6] can be used to elucidate business requirements at the enterprise level. This does not mean, however, that we have a well-defined idea of software requirements, which can equally be delineated using most of these techniques.

Each business requirements [6] (system requirement - ISO 12207 [1]) analysis technique is used for specific purposes, and distinguishes individual business requirements from other perspectives. The techniques used for functional software requirements analysis are as follows: scenarios and use cases, user stories and sequence diagrams. They are most often used here because of a confusion of business requirements analysis and software requirements analysis.

The techniques used for functional software requirements are written in natural language, so translating them to technical language or software items [1] such as objects, classes, domain objects, components, interfaces, messages, and so on, causes a loss in consistency and a break in the relation between requirements and software items. This break is caused by the fact that the software also has to meet software quality requirements [7]. Thus, software application components have to be organized in a way other than by defined and grouped use cases, or user stories. The translation issue is a result of difficulties in identifying the domain objects, business processes or functionality, validation or limitation of inputs, formats of outputs, and so on.

ISO 12207 divides software development life cycle processes (SDLC) [1] into software implementation processes, software support processes and software reuse processes. Object-oriented analysis and object-oriented design are included in software implementation processes.

Software implementation processes are executed by software development methodology in different ways, such as incremental [8], Agile [9], spiral, waterfall [10], and so on. Most such software development methodologies do not consider software support processes and software reuse processes as part of their software development life cycle. Though they model the software implementation processes in different ways, all software development methodologies basically utilize three practices: object-oriented analysis, object-oriented design, and use cases of processes such as the software requirements analysis process, the software architectural design process and the software detailed design process. Nevertheless, how these practices are used differs according to the software development methodology in question.

The paper was processed with contribution of long term institutional support of research activities by the Faculty of Informatics and Statistics, the University of Economics, Prague.

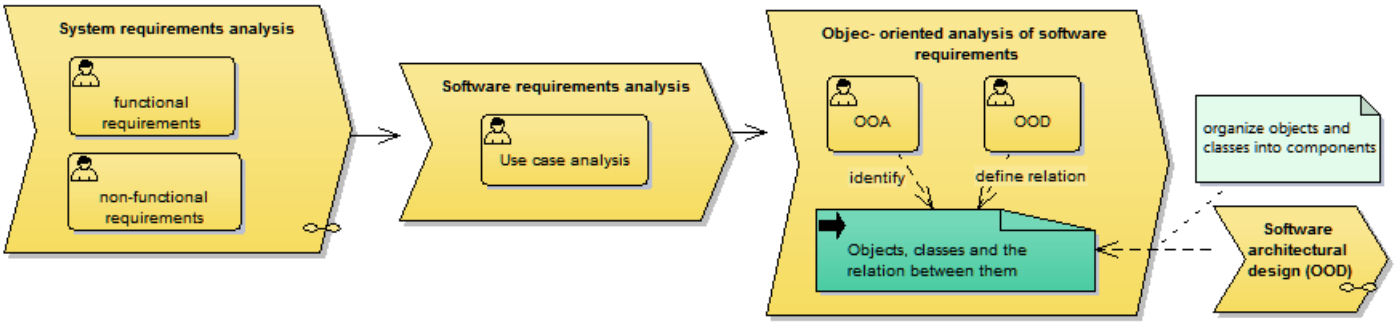

Fig. 1. A typical method of software requirements analysis (source: own)

The diagram at Fig. 1 shows a common usage of OOA and OOD.

Booch in [5] defines: *Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.*

Object-oriented analysis thus helps developers analyse software requirements written in text format (e.g. use cases) and natural language, and translate them into technical language: objects and classes. However, in order properly to understand these objects and classes, we need to place them in the context of business processes. In other words, we have to establish a relation between the objects, classes and model behaviour of relevant software.

Further Booch in [5] defines: *Object-oriented design is a method of design encompassing the process of objects oriented decomposition and a notation for depicting logical and physical as well as static and dynamic models of the system under design.*

ISO 12207 [1] defines software design as: *internal and external interfaces of each software item (e.g. objects, classes) and software unit (e.g. components); consistency and traceability are established between software requirements and software design.*

According to ISO 12207, the object-oriented design process has two aspects: the software architectural design process and the software detailed design process. The architectural design process may be considered conceptual design by identifying components and the relation between them. Software detailed design is about static and dynamic models composed of objects and classes. In other words, detailed design is very similar to object-oriented analysis. The only difference between them is that component and conceptual design ensures that detailed design is viewed simply and abstractly in order to organize the objects and classes in such a way as to meet non-functional application software requirements. As a result, many software development methodologies cannot adequately separate OOA from OOD process and so many organizations skip the design phase and move directly onto developing the software.

In order to analyse and design software, different modelling languages and tools may be used, the most common of which is UML. A modelling language is a model-driven architecture (MDA) concept [11]. Sometimes modelling is used to visualize software components, and so tools such as, for example, Microsoft Visio are put into practice to provide objects with images to show these components as natural words. The goal of such modelling reaches further than conceptualizing the software component as words, however, and also covers visualizing it within the software itself, making the model physically available as an item or unit. This means that the model can be transformed into a software item and vice versa.

Modelling is one of MDA concepts, and such transformation between the model and a software item is another one [11].

To use and interpret modelling only as the visualization of a concept makes it difficult to maintain and update such a model when application software changes during software support processes. For that reason, OOD process is skipped during software implementation processes and architecture design is done very abstractly.

There are tools available, like, for example, Spring Roo, that generate fully functional software based on domain objects [12]. Most of these tools work with domain object models, and generate Create, Read, Update and Delete operations for the domain objects. Such software is, however, usually limited in its ability to provide or cover all the functionalities required in business. Furthermore, other issues may arise with using this type of MDA to produce software, for example, the performance, maintainability or extensibility [7] of software items and units.

II. Interface-based software requirements analyses

Analysing software requirements requires an understanding of how software is composed of different software parts (e.g. layers), software units (e.g. components) and software items (e.g. objects or classes). Architecture and architecture layers are further MDA concepts. By applying these two concepts, software parts can be identified and divided into three layers independently of software requirements.

- **Consumer layer**: This layer describes how the client will interact with the software and use the service. If the client is human, then it will be either a graphical user interface or a command line interface. If the client is other software, it will be a service layer wrapper accessible remotely by consumer software.

- In use cases, the boundary of the software is not always clear. Sometimes use cases describe the end user interaction with the software, and at other times describe the system process; sometimes they describe the design. It can also be difficult to gain perspective on use case granularity.

- **Service layer**: This layer describes the core functionality implemented by software and provided as a service. Looking at software from the client perspective facilitates discussion regarding functional software requirements between developers and stakeholders, as the latter are not interested in the composition of the software, but rather in how to use it.
    - In use cases, identifying the core functionality and service that the software in question has to provide is combined and usually described in the form of a process. It is clear that such a process is composed of related sets of activities. Thus, each activity can be implemented in one or more services. As a result, use cases become difficult to map or translate into objects, classes and components.
- **Domain layer**: This layer describes the type of data or information required and/or provided by the service layer.
    - In use cases, identifying domain objects is a difficult task for most experienced software developers. There is simply no easy trick, tip or technique to teach developers how to identify domain objects, perform domain analysis and design the domain object model.

Software requirements can be divided into functional software requirements and non-functional software requirements, as is the case with business requirements [6]. This means that before analysing software requirements, business requirements must be available in the form of functional business requirements and non-functional business requirements.

In order better to analyse functional software requirements with regards to interface, let us first clarify the concept of interface.

The **Error! Reference source not found.** defines: *The interface of a given entity (= of a programs part – a module, a class, a method) specifies what the given entity knows and how to communicate with it. In case of an object it says which messages can be sent to it (to which messages the object understands) and how the object reacts to them. It is important to remember that an interface doesn't solve anything, it only promises, what the given entity can provide. We could say that the interface summarizes what the surrounding program should know about the given entity*.

This definition describes interface as physical characteristics of software items and units. However, 'interface' also has a second meaning:

Interface is used in term of the program's construction, which we might consider as syntax representation of interface and which behaves as a class without any implementation **Error! Reference source not found.**.

Abstraction is another foundational MDA concept, and "*deals with the concepts of understanding a system in a more general way"* [11]. Combining this idea with the second definition of interface means that we can use the latter to describe functional software requirements.

The notion of separation of concerns is a key MDA concept. By applying it to the software requirements analysis, we would be better able to identify and to manage dependencies between interfaces. Separation of concerns would make it possible to provide access to the stakeholders' perspective.

The easy way to apply separation of concerns to software requirements analysis would be to think about dividing the software architecture vertically. The result of a horizontal division of architecture is layers; with vertical division, on the other hand, we would be able to create modules of software requirements according to business concern. Please note that the vertical and horizontal division of software architecture performed during the software requirements analysis process is meant only conceptually and not physically. Physical

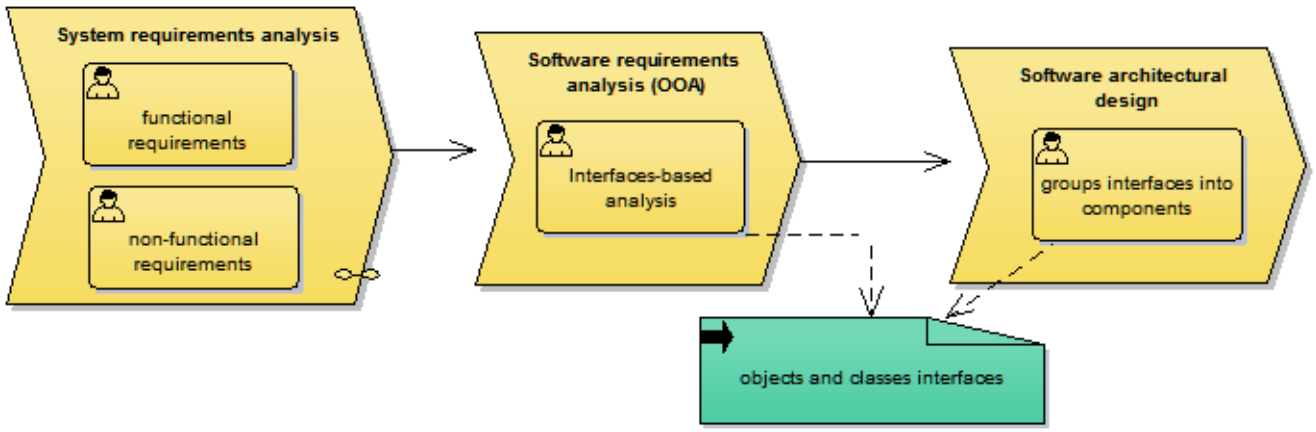

Fig. 2. Interface-based analysis (source: own)

separation is usually be done by software architecture design process.

The first goal of interface-based analysis is to reduce the complexity and difficulty of translating text into objects and classes by performing such analysis only once and remaining object-oriented. The second goal is to reduce the number of analysis phases and focus on the architecture. This means carrying out the analysis from an architectural perspective and not being limited by objects and classes that will change during implementation and synchronization between code, architecture and requirements.

It remains to compare user stories with interface-based analysis. Because user stories are a decomposition of use cases, they can, in fact, be regarded as mini use cases [3].

The Fig. 3 depicts how interface-based analysis different views from software architecture, software developer, software tester and functional business requirements perspective.

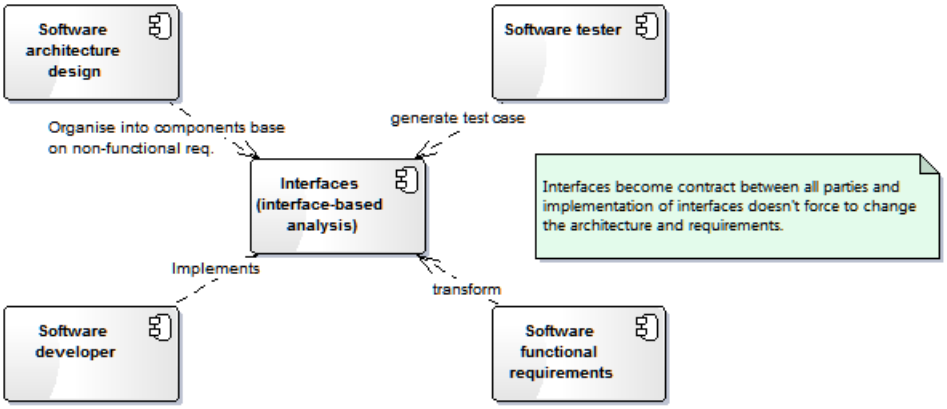

Fig. 3. Different views of interface-based analysis (source: own)

The UML model of interface-based analysis could be transformed to, for example, Java source code. After the transformation of UML model we can get service layer interfaces that can be implemented. All changes in requirements thus become easily traceable and the software developer has the freedom to implement them without limitation. The domain model can be converted into a class model, and the ATM user layer (consumer layer) can be converted into the model components of the MVC design pattern. The actors in interface-based analysis will be left out of the diagrams and instead there will be provided a matrix of interfaces and actors list and rights.

III. Example

In order to illustrate how interface-based analysis can work, it is useful to compare a use-case analysis model with an interface-based one. The goal is to see that what can be done with use cases can also be done with interfaces in an object-oriented way.

The example will analyse the following functionalities automated teller machine (ATM) bank:

- Customer is able to check the bank balance.
- Customer can withdraw money from ATM.
- Customer can give an order for transferring money.
- Customer can deposit funds into his bank account.

ATM machine need to be repaired and maintained thus customer should be able to use ATM machine.

A. *Use case way*

Use case at Fig. 4 diagram describes the functionalities required from ATM banking without scenarios.

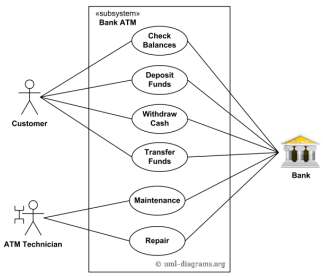

Fig. 4. A use-case analysis model (source: http://www.uml-diagrams.org)

B. *Interface-based way*

The ATM banking functionalities analysis with interface-based method will be divided into two business modules according to the separation of concern concept of MDA. The modules are:

- The operational module at Fig. 5 describes how customer performs cash operations with ATM interface.
- The second module depicted at Fig. 6 is maintenance module that will show change the ATM banking state into maintenance, so that the machine can be repaired

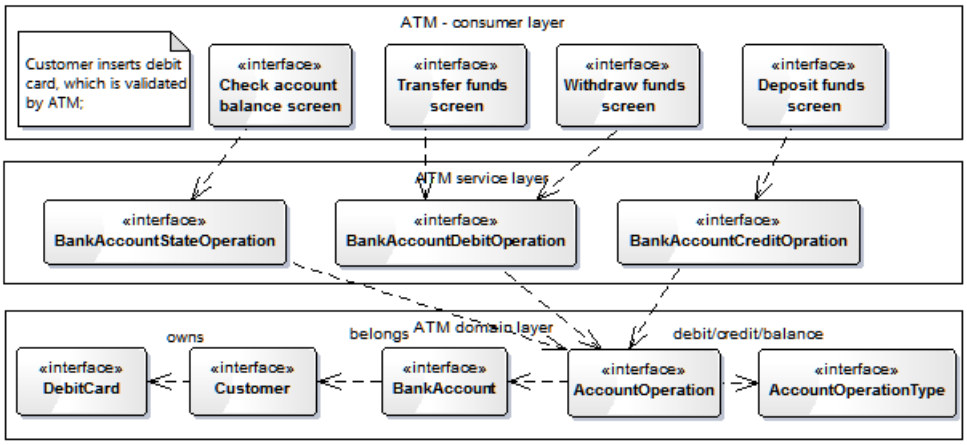

Fig. 5. An interface-based analysis of the operational module of an ATM (source: own)

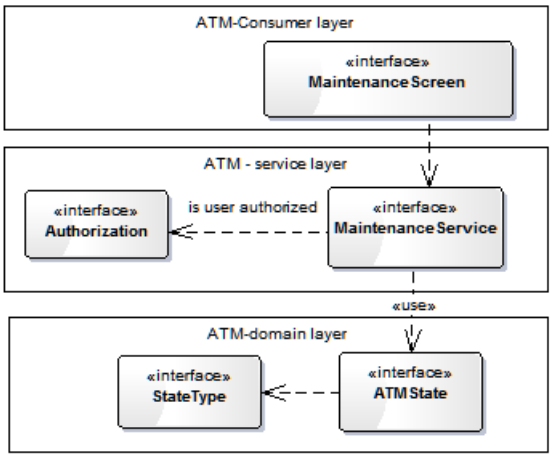

Fig. 6. An interface-based analysis of the maintenance module of an ATM (source: own)

Access rights	Customer	ATM-Technician
BankAccountDebitOperation	Read-Write-Execute	0 - 0 - 0
BankAccountCreditOpration	Read-Write-Execute	0 - 0 - 0
BankAccountStateOperation	Read-Write-Execute	0 - 0 - 0
MaintenanceService	0 - 0 - 0	Read-Write-Execute

Table 1. interface-based actors list and rights (source: own)

IV. Conclusion

Use cases are not object-oriented, and analyses are performed twice in most software development methodologies (see Figure 1). With interface-based analysis, we can perform object-oriented analyses of software requirements, and such analyses need only be performed once. Interfaces are physically available in many programming languages, such as, for example, Java or C#. Therefore, we can transform interfaces from the analysis phase into a physical interface, and can force development to become agile and modular. With the use of interfaces, we can clearly define analysis and design. Design will be easier with interfaces because architecture design works with components, and every component can have both provider and consumer interfaces. As a result, the mapping between architecture and requirements analysis will become clearer and easier.

References

[1] ISO/IEC 12207. System and software engineering – Software life cycle processes, 2008.

[2] KRUCHTEN Philippe. *Rational Unified Process, an Introduction*, Third Edition. Addison Wesley, December 19, 2003. ISBN: 0-321-19770-4

[3] COHN Mike. *User Stories Applied: For Agile Software Development*. Addison Wesley, 2004., ISBN-10: 0321205685

[4] The Open Group. *TOGAF 9.1*. 2011. ISBN: 978-90-8753-679-4

[5] BOOCH Grady. Object-Oriented Analysis and Design with Applications, Third Edition. Addison-Wesley, April 2007. ISBN 0-201-89551-X

[6] IIBA. *A Guide to the Business Analysis Body of Knowledge (BABOK Guide)*, Version 2.0. ISBN-13: 978-0-9811292-2-8

[7] ISO/IEC 25030. *Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Quality requirements*. 2007

[8] HUNT John. *Guide to the Unified Process featuring UML, Java and Design Patterns*. Springer, 2003. ISBN-10: 1852337214

[9] HAZZAN Orit and DUBINSKY Yael. Agile Software Engineering. Springer, 2008. ISBN-10: 1848001983

[10] O'REGAN Gerard. Introduction to Software Quality. Springer, 2014 .ISBN 978-3-319-06106-1

[11] OMG. *Model Driven Architecture (MDA) MDA Guide rev. 2.0*. OMG Document ormsc/2014-06-01.

[12] LARMAN Craig. Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process. Prentice Hall; October 30, 2004. ISBN-13: 978-0131489066

[13] PECINOVSKÝ R.: *OOP – Learn Object Oriented Thinking and Programming*. Eva & Tomas Bruckner Publishing 2013. ISBN 80-904661-8-4.