Interface-Driven Software Requirements Analysis

Rais Aziz Ahmad, Mgr.
Department of Information Technologies
University of Economics, Prague, Czech Republic

doi: 10.19044/esj.2016.v12n30p40 URL:http://dx.doi.org/10.19044/esj.2016.v12n30p40

Abstract

Software requirements are one of the root causes of failure for IT software development projects. Reasons for this may be that the requirements are high-level, many might simply be wishes, or frequently changed, or they might be unclear, missing, for example, goals, objectives, strategies, and so on. Another major reason for projects' failure may also be the use of improper techniques for software requirements specification. Currently, most IT software development projects utilise textual techniques like use cases, user stories, scenarios, and features for software requirements elicitation, analysis and specification. While IT software development processes can construct software in different programming languages, the primary focus here will be those IT projects using object-oriented programming languages. Object-oriented programming itself has several characteristics worth noting, such as its popularity, reusability, modularity, concurrency, abstraction and encapsulation.

Object-oriented analysis and design transforms software requirements gathered with textual techniques into object-oriented programming. This transformation can cause complexity in identifying objects, classes and interfaces, which, in turn, complicates the object-oriented analysis and design. Because requirements can change over the course of a project and, likewise, software design can evolve during software construction, the traceability of software requirements with objects and components can become difficult. Apart from leading to project complexity, such a process can impact software quality and, in the worst-case scenario, cause the project to fail entirely.

The goal of this article is to provide interface-driven techniques that will reduce ambiguity among software requirements, improve traceability and simplify software requirements modelling.

Keywords: Object-oriented analysis and design, interface-based analysis

Introduction

The latest research, most notably the Standish report of 2015, shows that around 50% of IT projects carried out between 2011 and 2015 experienced difficulties in delivering their requirements (features and functions) on time and under budget, and around 18% of IT projects failed to meet their targets (Standish, 2015). The Standish Chaos Manifesto (Standish, 2013) specifies that IT projects delivered 74% of their itemised requirements in 2010, and only 69% in 2012. In addition, it also points out that half the requirements demanded are never used, and 30% of the requested features and functions are used only rarely. Requirements elicitation, analysis and implementation nevertheless remain the most difficult tasks in delivering successful IT projects. The Standish Chaos Manifesto (Standish, 2013) concludes that requirements management is the process of identifying, documenting, communicating, and tracking requirements and their evolution, and has to be maintained throughout the IT project life cycle. In other words, software requirements have their own life cycle, which exists parallel to the project life cycle and the software development life cycle.

The software development life cycle

In order to analyse the complexity of software requirements elicitation and analysis, it is necessary to look at these processes from the perspective of the software development life cycle. This enables us to understand how other processes depend upon them. The life cycle is a process that can be continuous or discrete, and can have a start and an end time. In an IT project, software development life cycles are discrete, because all activities in the process happen separately and can depend on each other's outputs. Therefore, software development life cycle processes can be organised and structured in different types, such as waterfall, spiral, incremental, agile, and so on.

Each process is transient in nature, (it has a definite beginning and end), and is aimed at creating a unique product, service, or result within the life cycle process as a whole: initiating, planning, executing, controlling and monitoring, and closing (PMI, 2013). The software development life cycle manages these temporary software development processes, including implementation processes. As defined by ISO/IEC 12207 (2008), these can be modelled as project sub-processes, as in the figure below:

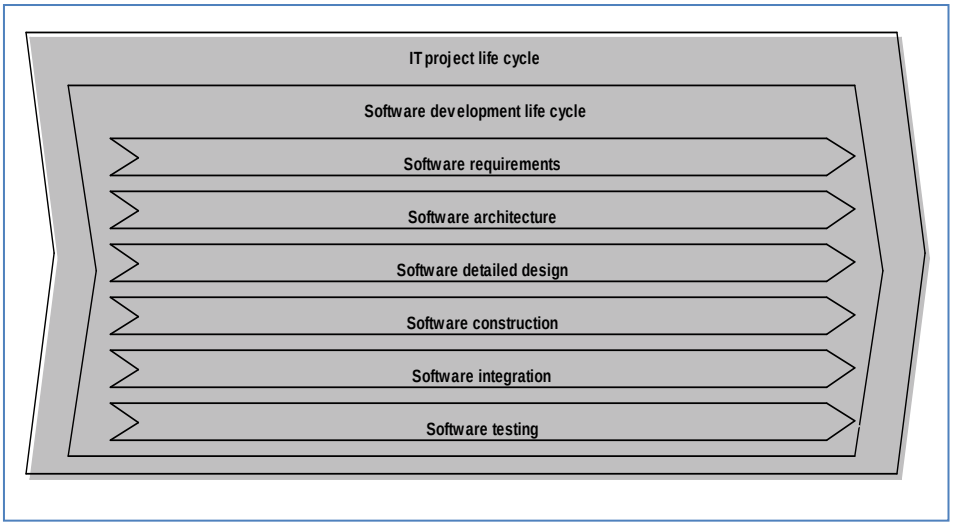

Fig. 0.1 Software development life cycle implementation processes (source: author)

Although the topic of this article is the complexity of software requirements analysis, it is necessary to include software architecture and detailed design here because these highlight the requirements and structure of software. They are therefore important factors when it comes to differentiating between the processes that clarify and affect our understanding of software requirements.

Software architecture design

Software architecture is a combination of the following: a set of architectural elements, for example, components; the relation between those elements; and the rationale for choosing them (Babar, 2014, pp.3-4). Babar also notes that software architecture is a means of achieving quality goals or meeting non-functional requirements. The software architecture process, which is high-level, and the detailed design process together constitute the **design processes** of the software development life cycle. High-level design defines the abstract level of functional and logical components based on the requirements, and identifies their external interfaces. As a result, these components can, in the detailed design process, be decomposed further into objects and classes, or software items and units.

Both levels of design can be executed using different methods. For high-level design, one example of an applicable approach is the Architecture Development Method (ADM) from TOGAF (2011), which provides a view of how software architecture will be used within an enterprise. Similarly, Multidimensional Management and Development of Information Systems (MMDIS) (Vorisek, et al., 2008), or Service-Oriented Architecture (SOA, Ref. Arch., 2011) can be applied to enterprise architecture as well as to

developing software architecture, as can the "4+1" View Model of Software Architecture (Kruchten, 1995), Model-Driven Architecture (MDA, 2014), and so on. The decomposition of high-level design into detailed design can be done using methods such as structured (functional) design, object-oriented design, component-based design, aspect-oriented design, data-structured design, or centred design (SWEBOK, 2014).

Software analysis and detailed design

Essentially, there are two basic types of method for detailed software analysis and design: function-oriented and object–oriented (Mall, 2004). All other design techniques, practices and methods are either an extension of one of these or a combination of both.

Function-oriented analysis and design decomposes the high-level view of the software system into detailed functions, which are then split further into sub-functions, and so on. The system state is then shared among all these functions. All the identified functions are grouped into components and represented in a function decomposition table for better tractability (Wieringa, 1998). The function-oriented methods are structured analysis (SA) and structured design (SD). Examples of SA and SD are the Structured Analysis and Design Technique (SADT), the Yourdon System Method (YSM), Specification and Description Language (SDL), Jackson System Development (JSD), and Jackson Structured Programming (JSP) (Wieringa, 1998).

Object-oriented analysis and design decomposes a software system into software units, objects, classes, and interfaces. Then, based on these analysed units, various logical and physical models are constructed using different notations, such as, for example, the Unified Modelling Language (Booch, et al., 2007, pp.42).

To some extent, the relation between analysis and design methods in programming is given historically. The rest of the time, this relation is established through their use of similar concepts. For example, the fundamental concepts in object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP) are objects, classes, and interfaces, and the interaction between these three. On the other hand, function-oriented analysis and design methods, including, for example, the Structured System Analysis and Design Method (SSADM), separate tasks (functions) and data, emphasise the software system's procedural aspects, and use tools such as data flow diagrams (DFD) and structure charts (decision tables and decision trees) (UAC, 2016). These procedures, along with variables, commands, and data abstraction, are the basic concepts of imperative programming (Watt, 2004). In other words, imperative programming is programming with a certain state and commands that modify

that state. Combining these attributes with sub-programs delivers procedural programming, which can, therefore, be considered a subset of imperative programming, modularising, as it does, the source code of the latter in the form of procedures (Kaisler, 2005). In object-oriented programming, "objects" encapsulates both data and operations (methods) that manipulate data, but procedural programming separates data from operations (procedures) (Weisfeld, 2009). The encapsulation of data and operations by objects can be seen as an improvement upon imperative and procedural programming.

Therefore, some analysis and design concepts of function-oriented methods can be utilised by object-oriented analysis and design thanks to the former's simplicity. However, because imperative programming languages do not have a concept of "object", it does not make sense to carry out object-oriented analysis and design, and then construct the software in an imperative (or procedural) programming language.

Software analysis and the detailed design process

Booch (2007, pp. 259) describes object-oriented analysis and design as one process, and explains it as the *transformation of the requirements into a design of the system, which serves as a specification of the implementation in the selected implementation environment*. However, in order to analyse the complexity connected with object-oriented analysis and design, we need to understand the difference between object-oriented analysis and object-oriented design. Khurana (2012) illustrates the object-oriented approach in Fig. 0.2.

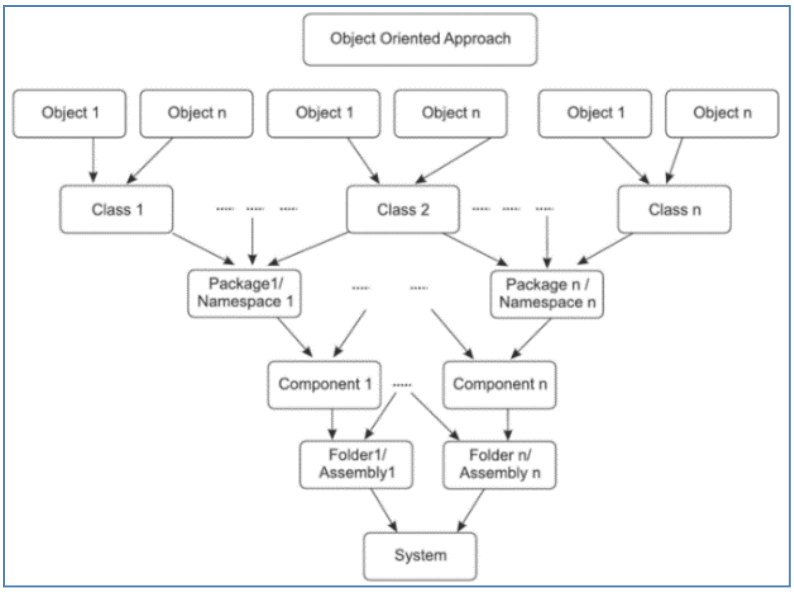

Fig. 0.2 A typical view of the object-oriented approach (source: Mala and Geetha, 2013)

The object-oriented analysis process is, according to Khurana (2012), composed of activities related to objects, such as, for example, object identification, object structure, object attributes, object association, and service definition based on the object specified. Consequently, the OOA process can be illustrated as in **Error! Reference source not found.**:

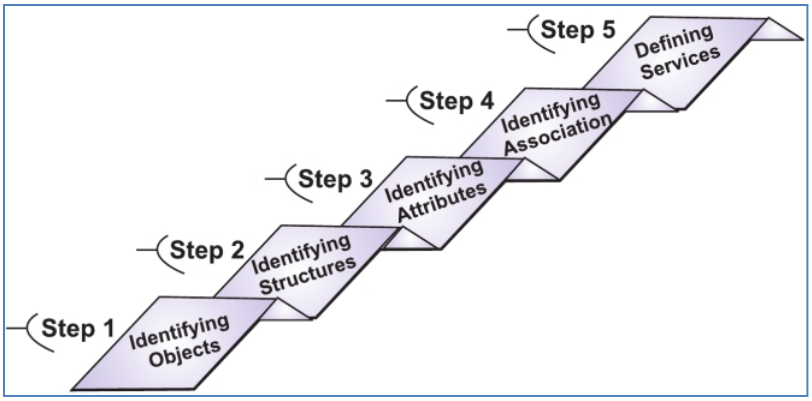

Fig. 0.3 Steps in object-oriented analysis (source: (KHURANA, 2012))

By comparing the activities in Fig. 2 and Fig. 3, we can see that object-oriented design starts with mapping objects to classes, structuring classes, and so on. In order to map each object, it is necessary first to analyse and identify the classes, and then to map the object to a specific candidate class. This also seems to be an analysis process. Therefore, object-oriented analysis can be seen to concern the identification of objects and candidate classes, and their respective attributes, before mapping each to a service definition.

Two techniques are usually used here in order to identify objects, classes, interfaces, and their interactions: the Vocabulary Approach and CRC Cards (Blaschek, 1994).

We can see that the difference between object-oriented analysis and object-oriented design is difficult to determine because both operate with the same software units. Thus, the complexity lies in determining which level of software unit detail should be handled by OOA, and which by OOD. In practice, it may also happen that the structure of classes and the relation between them change during OOP, which can cause conflicts in requirements specifications.

The complexity of requirements specification

Software analysis and design processes can start based on the outputs of software requirements specification processes. These latter are, according to (SWEBOK, 2014), composed of the following activities:

1. **Requirements elicitation**: this is the process of gathering requirements. At this point, the requirements are raw, meaning they need to be specified and detailed; duplicates, conflicts, and inconsistencies must be cleaned up; and those requirements whose implementation is more expensive than their value should be removed.
2. **Requirements analysis**: here, the requirements need to be classified as either functional or non-functional, and a conceptual model in the form of, for example, a sequence diagram, a use case diagram, or a state diagram must be built. Conflicts in the software features are also solved at this stage, duplicates are removed, the requirements' consistency with the overall goals and strategy is assured, and they are prioritised with the help of a cost-benefit analysis. Further, analysing the requirements at this point in the software life cycle also means defining the software domain or the problem domain. In other words, the requirements analysis is not, at this stage, about object-oriented analysis and design and specifying the logical and physical software model.
3. **Requirements validation**: this is the final validation of the requirements analysis specification with stakeholders.
4. **Requirements specification**: here, the result of the analysis is documented as the software requirements specification then used for OOA and OOD, and for the software's implementation.

Software requirements elicitation and analysis requires language and techniques that both businesspeople and IT specialists understand. As a result, most projects use the four textual techniques:

1. **User story**: user stories are meant primarily for communication with stakeholders, and for planning the release and estimate of work (Wells, 1999). According to Flower (2003), '*the user story and the use case have a complex correlation. Stories are usually more fine-grained because they have to be entirely buildable within one iteration (one or two weeks for XP). A small use case may correspond entirely to a story; however, a story might be one or more scenarios in a use case, or one or more steps in a use case. A story may not even show up in a use case narrative, such as adding a new asset depreciation method to a pop up list.*'
2. **Use case**: this is a traditional and common technique used for requirements elicitation and analysis, and to model the software requirements (Jacobson, Spence, and Bittner, 2011).
3. **Scenario**: scenarios, or, more precisely, usage scenarios, describe the steps and events in the interaction of people and the organisation with the system (Ambler, 2014). The goal of this technique is to migrate from use cases to sequence diagrams.
4. **Feature**: features alone are not sufficient for describing or analysing requirements. They must, therefore, be combined with use cases. A

feature may be derived from a requirement for supporting a solution, and can be described with a use case or with an alternative use case scenario (De Oliveira, 2013).

Oberg, Probasco, and Ericsson, (2000) defined a few **common problems** with requirements management, some of which can be seen to comply with the research overview given in the Introduction:

1. Tracking changes in requirements and features can be very time-consuming. Requirements are not always clear to begin with and, as the business evolves, they can also change and develop, resulting in many becoming obsolete or unnecessary, or being duplicated due to multiple stakeholder viewpoints. Thus, using only the textual techniques described above for requirements specification and object-oriented analysis can cause problems in tracking changes.

2. As mentioned above, requirements are not always obvious and may have many sources. The requirements of different stakeholders can be diverse, and sometimes conflict with each other, or they may be duplicated. As a result, it can be complicated clearly to define the problem domain.

3. Requirements may not be expressed in clear language and, as a result, remain ambiguous. This common problem is rooted in the textual techniques.

4. Requirements have different types and different levels of detail.

These four common problems, along with the complexity of object-oriented analysis and design mentioned in the chapter entitled “Software analysis and the detailed design process”, can lead to the following three major issues, which can in turn result in IT project complexity and even failure:

1. **Duplicated analysis**: we perform the same analysis twice: once as part of the software requirements specification process, and then again as part of the object-oriented analysis and design.

2. **Object analysis**: in object-oriented analysis and design, we have trouble identifying objects and classes from requirements written using textual techniques.

3. **Traceability**: tracking the relation of requirements to components, objects, and classes is also problematic.

Interface-driven requirements

The concept of interface-driven requirements analysis is about analysing the software system functions with interfaces. This means representing the objects, classes, methods, modules, and components using interfaces. Consequently, it is necessary to understand the interrelation of objects, classes, and interfaces.

Objects, classes and interfaces

The relation between objects and interfaces, as explained by Blaschek (1994), is that objects are structures with a hidden state (data) and behaviour (operation), that have precise interfaces specifying which messages they accept, and that can communicate with each other by means of messages in order to perform complex tasks.

According to Pecinovsky (2013), there are two types of interface: the first describes what a module, an entity, a method, an object, and a class know, and how they communicate with each other. The definition of the first type of interface allows us to conclude that everything has an interface. The second type of interface is the representation of the first type in a concrete programming language. For example, an interface in Java is *a contract between a class and the outside world. When a class implements an interface, it promises to provide the behaviour published by that interface* (Java tutorial, 2015).

To put it differently, the first type of interface is the specification of an entity, which can be used for conceptual analysis, and can be known as a **conceptual interface**. The second type of interface, depending on the programming language in question, can be termed a **contractual interface**. If a programming language does not support an interface feature, then the contractual interface will be represented by each specific feature, for example, a class, an abstract class, and so on.

Object-oriented programming languages are classified as pure, hybrid, or object-based. Pure object-oriented programming languages, including Java, Simula, Smalltalk and Eiffel, support all object-oriented concepts. Hybrid object-oriented programming languages, such as C++, Object Pascal and Turbo Pascal, support not only object-oriented concepts but also procedural programming or functional programming concepts. Object-based languages, for example, Ada, support only the concepts of data encapsulation, data hiding, and access mechanisms, automated initialisation and object clean-up, and operator overloading (Balagurusamy, 2014).

Classifying object-oriented programming languages is difficult because of the way they implement and support object-oriented concepts and features. From the categories available on the website DMOZ (AOL, 2016), namely pure, class-based, prototype-based, scripting, compiled, garbage-collected, interpreted, and aspect-oriented, we can see that some programming languages fit into more than one group. However, the majority of object-oriented languages are class-based (a class is an object template), and all the others support different features for creating objects (in Perl, for example, a class is created with the key word “package” (TP, 2014)). According to the TIOBE index (2016), the following class-based object-oriented programming languages, from a total of 100 such languages, cover

50% of the market: Java, C++, C#, Visual Basic.NET, Delphi/Object Pascal, Objective-C, Swift, Python, PHP, Perl, and Ruby.

The basic concept of interface-driven requirements analysis is to use the first type of interface for analysis, and a modelling language such as the Unified Modelling Language (UML), which is supported by the Meta Object Facility (OMG-MOF, 2015), to transform the UML model of interface-driven software requirements into either the second type of interface or the specific feature of a programming language. All ten of the most frequently used object-oriented programming languages support OOP's class feature, and some of them OOP's interface feature.

Interface-driven requirements analysis

In order to specify and analyse software requirements, it is first necessary to look at them from different perspectives: **functional** (what application software should do), **non-functional** (how the software should work), and **external interfaces**. The last of these includes the information system (how other business processes and functionalities are automated and set up in the organisation, and their relation to new business), the user (human consumer), the hardware, or low-level interface (how the application software communicates, and how it uses the services provided by the hardware, the operation system, the application server, and so on), and communication (which protocols are to be used to interact with the application software, and the specification of the external systems protocol) (IEEE Std 830, 1998).

The core software requirements are functional, while all the other perspectives aid in understanding, consuming and supporting this core functionality. As the tool used to help application software meet non-functional requirements is software architecture, it is necessary to envision where and how various software building blocks will implement these core functionalities. Examples of software architecture building blocks are components, layers, and tiers. Using a combination of these three types of building block, we can construct different architecture styles, including one of the most famous and frequently used architecture styles, service-oriented architecture (SOA).

SOA is composed of two horizontal layers and four vertical layers. The horizontal layers are the logical (comprising the Service Component Layer, the Services Layer, the Business Process Layer, and the Consumer Layer), and the physical (the Operational Systems Layer). The four vertical layers are the Governance Layer (covering availability, registries and repositories), the Integration Layer, the Quality of Service Layer (administration, monitoring and management), and the Information Layer (business events) (SOA Ref. Arch., 2011). As the service consumer can be either human or other application software, this layer describes the user

interface requirements of application software. Meanwhile, SOA's service layer is the core functionality provided by the application software, and is managed by the business process layer. The business process layer in turn determines which service is invoked when the application software receives a request from the consumer. Lastly, SOA's information layer is the equivalent of the domain layer in software architecture, because the business data are represented by domain objects in application software.

Thus, using SOA layers and software architecture in describing and analysing software requirements can be illustrated as a UML model in the following manner:

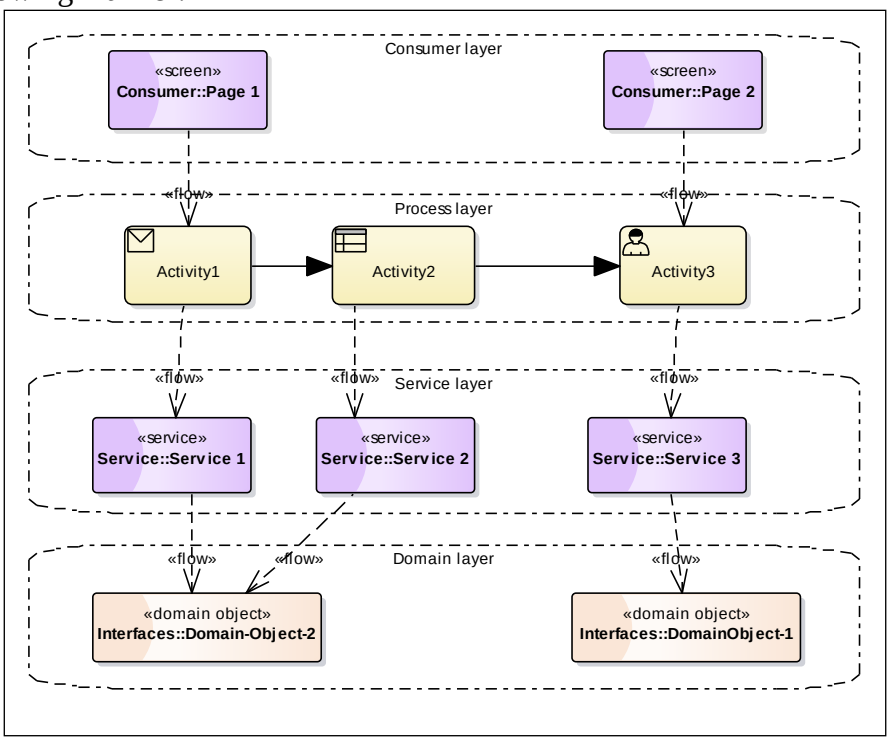

Fig. 0.1 An example model of interface-driven requirements analysis (source: author)

The Fig. 4 service layer is modelled in BPMN OMG Standard (OMG BPMN, 2011) for business process modelling, which can illustrate different types of activity, for example, user tasks, manual tasks, service tasks, and so on. The user task activity can be used to identify and analyse user interfaces in the consumer layer. The service task activity can be used to identify the software's core functionality, and can thus be mapped to the service interface. Similarly, the script task can be used for business rules, and so on. Every service accepts business information as a message (in the form of a user request or other software request) that can be represented as a domain object. As a result, we are equally able to use such business process models

to identify and analyse the application software's domain model. Using UML stereotypes, each type of interface can be marked in order better to transform the conceptual model. Lastly, so as to have small modules and clear diagrams, a business process can be split into different modules.

The next step in software requirements analysis might be to decompose the service layer interfaces in order to identify all the necessary operations. SOA separates operational interfaces (the service layer) from informational interfaces (the domain layer), so the service layer interface represents all the business functionalities that the software is going to implement. Of course, these core functionalities may be supported by technical functionalities such as, for example, data manipulation functions, using data access objects to access the database, and so on. All these technical functionalities can be identified and grouped into interfaces according to the interface-driven software requirements. The pages identified in the consumer layer can be enhanced by adding attributes and by providing the description to the user-interface designer for graphical visualisation. The domain objects can be mapped to the data model.

Most importantly, the core functionality should be concentrated in one layer, thus improving the application software's reusability. These interfaces can physically exist as interfaces, abstract classes, or other programming language-specific features in application software code that contains no implementation. Using the object-oriented programming concept of inheritance, they can then be extended, implemented, and tested.

Conclusion

The goals of this article have been to reduce ambiguity among software requirements, minimise the problem of analysis duplication, improve the traceability of requirements, and simplify software requirements modelling. It has provided the relation between high-level and low-level design for the purpose of proving that the former can already be used during software requirements analysis in order to simplify understanding of the requirements. It has likewise demonstrated that detailed design uses methods that increase complexity in the separation of analysis and design activities, and that traceability is made difficult due to possible changes in software requirements and software implementation. This problem might be better solved by working with objects', classes', and components' interfaces, instead of with classes and components that are meant more for implementation than for specification and contracts. The model in Fig. 4 illustrates all the characteristics of software requirements, so there is no need for either a second object-oriented analysis, or a software requirements analysis using textual techniques. Tools and techniques such as interviews,

user stories, scenarios, observation, workshops, brainstorming, and so on, are sufficient to elicit software requirements.

The advantage of using interfaces with object-oriented concepts and programming is that the architect, testers, and developers can work and use features that provide them all with the same perspective. This can improve both the testability and the traceability of the requirements and software.

This approach will not limit the software developer in implementation, and, in the case of functionality re-factorisation, and thus of interfaces, we can immediately see the change's side effects. Further, the software quality assurance team can begin to write test cases and automate the testing before implementation. Interface-driven software requirements analysis is a good solution for supporting test-driven development.

References:

1. Ambler, S. W., 2014. *Usage Scenarios: An Agile Introduction.* Retrieved from http://www.agilemodeling.com/artifacts/usageScenario.htm
2. America Online (AOL), 2016. *DMOZ - Computers: Programming: Languages: Object-Oriented.* Retrieved from https://www.dmoz.org/Computers/Programming/Languages/Object-Oriented/
3. Babar M. A., Brown A. W., Mistrik I., 2014. *Agile Software Architecture.* Elsevier Inc. ISBN 978-0-12-407772-0
4. Balagurusamy E, 2008. *Object Oriented Programming With C++*. Retrieved from https://books.google.cz/books?isbn=0070669074
5. Blaschek Günther, 1994. *Object-Oriented Programming with Prototypes*. Springer-Verlag, Berlin, ISBN: 3-540-56469-1
6. Booch G., et al., 2007. *Object-Oriented Analysis and Design with Applications*. Third Edition, Addison-Wesley. ISBN 0-201-89551-X
7. Computer Society IEEE (SWEBOK), 2014. *Guide to the Software Engineering Body of Knowledge*. Version 3.0. A Project of the IEEE Computer Society. New York: Pierre Bourque, Richard E., ISBN-13: 978-0-7695-5166-1
8. Computer Society (IEEE Std 830), 1998. *IEEE Recommended Practice for Software Requirements Specifications.* IEEE Std 830-1998
9. De Oliveira, R. P., et al., 2013. *A Feature-Driven Requirements Engineering Approach for Software Product Lines.* DOI 10.1109/SBCARS.2013.11
10. Fowler, M., 2003. *UseCasesAndStories.* Retrieved from http://martinfowler.com/bliki/UseCasesAndStories.html

11. International Organization for Standardization (ISO/IEC 12207), 2008. *System and software engineering – Software life cycle processes*.
12. Jacobson I., Spence I., Bittner K., 2011. *USE-CASE 2.0: The Guide to Succeeding with Use Cases.* Retrieved from https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf
13. Kaisler Stephen H., 2005. *Software Paradigms.* John Wiley & Sons, inc, ISBN 0-471-48347-8, pp. 22-23. Retrieved from https://books.google.cz/books?isbn=0471703575
14. Khurana, R. 2012. *Software Engineering (WBUT).* 2nd Edition, VIKAS publishing house PVT LTD. Retrieved from https://books.google.cz/books?isbn=8125953035 *pp. 64*
15. Kruchten, P., 1995. *Architectural Blueprints—The "4+1" View Model of Software Architecture.* Retrieved from https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf
16. Mall, R., 2004. *Fundamentals of Software engineering.* Third Edition, Prentice-Hall of India, ISBN-81-203-3819-7, pp. 162, Retrieved from https://books.google.cz/books?isbn=8120338197
17. Mala, D J., Geetha S., 2013.*Object Oriented Analysis and Design Using UML*. McGraw Hill Education (India) Private Limited, Retrieved from https://books.google.cz/books?isbn=1259006743 *pp. 20*
18. Object Management Group (MDA), 2014. *Model Driven Architecture (MDA) MDA Guide rev. 2.0.* [Online] OMG Document ormsc/14-06-01. Available at: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf, http://www.omg.org/cgi-bin/doc?ormsc/10-09-06.pdf
19. Oberg R., Probasco L., and Ericsson M., 2000. *Applying Requirements Management with Use Cases*.published: Rational Software Corporation, Rational Software White Paper, Retrieved from https://www.dimap.ufrn.br/~jair/ES/artigos/appreqmanucases.pdf
20. Object Management Group (OMG-MOF), 2015. *Meta Object Facility.* Retrieved from http://www.omg.org/spec/MOF/2.5/PDF
21. Object Management Group (OMG BPMN), 2011. *Business Process Model and Notation*. Retrieved from http://www.omg.org/spec/BPMN/2.0/
22. Oracle (Java tutorial), 2015. *Object-Oriented Programming Concepts.* Retrieved from https://docs.oracle.com/javase/tutorial/java/concepts/

23. Pecinovsky R., 2013: *OOP – Learn Object Oriented Thinking and Programming*. Publishing: Eva & Tomas Bruckner, 2013. ISBN 80-904661-8-4.
24. Project Management Institute (PMI), 2013. *A Guide to the Project Management Body of Knowledge (PMBOK® Guide) – Fifth Edition*. Project Management Institute, Inc. ISBN: 978-1-935589-67-9
25. Standish Group, 2015. *CHAOS Report 2015 [online]*. Retrieved from http://www.infoq.com/articles/standish-chaos-2015
26. Standish Group, 2013. *CHAOS Manifesto 2013.* Retrieved from https://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf
27. The Open Group Standard (TOGAF), 2011. *The open group architecture framework. Version 9.1, Document Number: G116. ISBN: 978-90-8753-679-4*
28. The Open Group Standard (SOA Ref. Arch.), 2011. *SOA Reference Architecture.* Document Number: C119. ISBN: 1-937218-01-0
29. TIOBE, 2016. *TIOBE Index for June 2016.* Retrieved from http://www.tiobe.com/tiobe_index?page=index
30. Tutorialspoint (TP), 2014. *Object Oriented Programming in PERL.* Retrieved from http://www.tutorialspoint.com/perl/perl_oo_perl.htm
31. The Open Group Standard (SOA Ref. Arch.), 2011. *SOA Reference Architecture.* Document Number: C119. ISBN: 1-937218-01-0
32. Uni Assignment center (UAC, 2016). *Overview Of Structured Systems Analysis Information Technology Essay.* Retrieved from http://www.uniassignment.com/essay-samples/information-technology/overview-of-structured-systems-analysis-information-technology-essay.php
33. Vorisek, J., et al., 2008: *Principy a modely řízení podnikové informatiky*. Praha:Oeconomica. ISBN 978-80-245-1440-6. pp 118.
34. WATT David A, 2004. *programming language design concepts.* John Wiley & Sons Ltd. ISBN 0-470-85320-4, pp. 265
35. WEISFELD Matt, 2009.*The Object-Oriented Thought Process.* Third Edition, Addison-Wesley. ISBN 978-0-672-33016-2, pp. 10
36. Wells Don, 1999. *User Stories*. Retrieved from http://www.extremeprogramming.org/rules/userstories.html
37. WIERINGA Roel, 1998. *Survey of Structured and Object-Oriented Software Specification Methods and Techniques.* Retrieved from http://www.diku.dk/OLD/undervisning/2002f/datV-system/structured_methods.pdf