AN APPROACH FOR ARCHITECTURAL SOLUTIONS ESTIMATION

Olena Chebanyuk, Dmytro Povaliaiev

***Abstract**: Quick and effective estimation of architectural solutions that are represented as UML class diagrams allows obtaining quality information for further software development. One of the approaches to estimate class diagrams is verifying them according to SOLID design principles. Class diagrams, which follow the SOLID design principles, simplify the process of both architectural solutions and code reusing. Also such operations as class diagram refinement, merging, comparison, and analysis are performed more easily, too.*

An approach for estimation of class diagram according SOLID design principles is proposed in this paper. Entire information for class diagram is stored in XMI file that is created by software modeler environments. Estimation of class diagram correspondence to SOLID design principles is based on the automatic parsing of XMI file containing class diagram. Firstly class diagram classes and interfaces are identified from XMI file. Then relationships between them are defined. After recognizing class diagram structure queries for checking whether class diagram corresponds to every of SOLID principle are performed. Analytical foundation for forming such queries, based on predicate logic, was proposed in paper [Chebanyuk, 2016].

Then, architecture of proper software tool is described in the paper. Grounding for choosing technology for effective proceeding of XMI file content is represented. Also case study for using designed tool for class diagram estimation is represented.

***Keywords**: Abstract Syntax Tree (AST), Software Architecture, SOLID Design Principles, Class Diagram, XML Metadata Interchange (XMI), Object Constraint Language (OCL), LINQ query.*

***ITHEA Keywords**: D2 software engineering, D 2.0 Tools,*

***ACM Keywords**: Software and its engineering, Software system structure, Software system model, Model driven system engineering.*

Introduction

The more complex the software system becomes, the more complicated architecture it requires. The architecture is prominent for the software as it defines most of the technical aspects (maintainability, flexibility, extensibility etc.) that influence both the development process and the quality of the software system.

The most widespread approach of software development lifecycle management nowadays is Agile. Every operation in software development lifecycle management is performed by means of software model processing. Software models are represented as UML diagrams. They play both cognitive and communicative roles in collaboration between stakeholders and between stakeholders and customers [Chebanyuk and Markov, 2015 ICP].

As the customer may change software requirements, consequently all software development artifacts can be modified. Updated software models contain actual information about requirements, design, test cases, and other software artifacts.

Software models, reflecting architectural solutions, are modified too. They serve for many tasks, for example simplify a constant revision of the architecture of software being developed.

An important step in the architecture solutions designing is their verification. Unfortunately, performing of this step requires many facts that are complex to be formalized. In some companies this step is usually missed, therefore leading to mistakes in design and higher overall project cost. Lack of software tools, which allow to perform class diagram verification in automatic mode, become a motivation for authors fo provide this research.

This paper continues investigation, started in paper [Chebanyuk, 2016]. It was grounded using predicate logic for class diagram processing. Then corresponded predicate expressions for estimation class diagrams for accordance to SOLID design principles were proposed. Also criteria for estimation of analysis results were formulated.

Contribution of this paper is grounding choosing of technology to process class diagrams in automated mode and formulating LINQ queries for XMI file parsing that stores class diagram. Description of an approach for class diagram processing, which represent SOLID recondition algorithm and software architecture designing, is also proposed in this paper.

Related papers

Proper approach to software design process is based over understanding of some fundamental principles and guidelines. One can see a variety of articles describing benefits of sticking to SOLID design principles and drawbacks of neglecting them, for example [Ocampo J, 2009], [Lakhal F., 2012], [Chebanyuk, 2016]. These papers help to compose approaches for estimation of architectural solutions characteristics in accordance to SOLID design principles.

Author of book [Ocampo, 2009] investigates SOLID principles in details and proposes practical recommendations for verification class diagram and designing code. Many advices for flexible designing from the reusability point of view are listed in this book.

Question of architectural solution estimation are important for processes where it is necessary to obtain stable model of problem domain. Consider a paper, which touches question of estimating structural characteristics of software models, represented as UML diagrams.

Article [Sanchez et al 2015] considers aspect for designing stable component diagrams for adaptive systems. Authors introduce quality attributes of these systems as a their non-functional properties. Respectively, such properties are classes attributes and methods. Authors also touch a problem of reusing domain entities, transferring attributes of system, developed earlier to new ones.

Authors propose a framework for the specification, measurement and optimization of quality attribute properties expressed on top of feature models. Then it was shown how these properties can be specified by means of feature attributes and evaluated with quality metrics in the context of feature models. Structure of diagram, reflecting feature models modified by means of aggregate functions over the features is changed in the run time. For instance, if properties of component or its structure are changed, the system structure should be changed at run time and the system takes to reconfigure itself. Authors propose configuration selection algorithms for enrich obtained model by necessary amount of qualities to design quality model.

Consider researches of software tools designing for optimizing architectural solutions.

Other paper that manage quality of software models by means of analyzing interconnection between domain entities is devoted to modifying of problem domain profiles when borders of application domain are changed [Lakhal, 2012]. Consequently the structures of profiles are changed too.

The models have then to be fitted to the new profiles version. Implementing designed tool authors propose to refuse from manual adaptation of software models, using the combination of Eclipse plugins

to compare two software models. Then defined differences was a cause to make some changing in profiles and finally to adapt the models to the new version of the UML profile [EMF, 2011].

Paper [Sielis et al, 2017] describes the design, development and evaluation of a software prototype, named ArchReco, an educational tool that employs two types of Context-aware Recommendations of Design Patterns, to support users (students or professionals) who want to improve their design skills when it comes to training for High Level Software models. The tool's underlying algorithms take advantage of Semantic Web technologies, and the usage of Content based analysis for the computation of non-personalized recommendations for Design Patterns. The recommendations' objective is to support users in functions such as finding the most suitable Design Pattern to use according to the working context, learn the meaning, objectives and usages of each Design Pattern. Authors also design an environment for visualization of design of high level software diagrammatic models that obtained after recommendations gathering and processing. Design patterns are chosen according recommendations. These recommendations are proceeded involving semantic web technologies.

Nowadays there are lots of tools that allow us to automate many processes within software development life cycle, but they still pay not enough attention to keeping track of correspondence to SOLID principles. In fact there are no tools that can grant SOLID consistency.

Using an advanced modeling environments, as IBM Rational Software Architect [IBM, 2015] or Eclipse plugins modeling software, for example Papyrus [Eclipse, 2015], class diagram may include constraints to precise requirements of application domain.

The most widespread Object Constraint Language (OCL) [OCL, 2014] performs check of class diagram components for accordance requirements of application domain. Theoretically, OCL may be used for checking whether class diagram corresponds to SOLID design principles. But such operation should be performed manually. For example let us consider the procedure of checking whether class conforms to single responsibility design principle by method, proposed in [Chebanyuk, 2016]. Idea to do this, proposed by authors is to define the number of public methods in class. Using OCL it is necessary to type:

context class_name:OCL expression

for every class in class diagram. To observe class diagram for checking such feature of all its classes will take less time in comparison with composing OCL expression for every class.

The most convenient way for class diagram estimation will be creating of software tool or plugin for development environment to perform this task.

Investigation of class diagram storing format

Many modeling environments store information about UML diagrams in formats representing software model structure as a hierarchical tree. The key advantage of such modeling environments is using Xml Metadata Interchange (XMI) standard [XMI, 2015]. This allows to store relations between classes, and, this way, it allows user to design constraints to apply on these relations. However, there are few environments that support XMI. These are IBM Rational Software Architect, Papyrus, and UML Designer [Eclipse, 2015]. Other environments use their own not unified format for storing UML diagram data (JSON schemas, custom XML tag systems) [ASTM™, 2011].

XMI representation corresponds to theoretical approach Abstract Syntax Tree (AST). An AST is a formal representation of the syntactical structure of software that is more amenable to formal analysis techniques than is the concrete or surface syntax of software. Construction of ASTs typically involves the use of parsing technologies. AST model structures permit the expression of compositional relationships to other language constructs and provide a means of expressing a set of direct and derived properties associated with each such language construct [ASTM™, 2011].

The data structures from which the abstract syntax trees are composed provide an exhaustive collection of formal compositional elements for a language. These language model elements (or constructs) are generally defined in a type (or class) hierarchy. There are many ways to define these ASTs. The AST may be derived from an analytical process that can be applied to the surface syntax of the software asset or may be captured through a process that involves the application of rewrite rules to other data structures. For instance, a common, or language-neutral, AST model might be generated by the application of rewrite rules that generate a language specific AST model of some application, or a generic AST model might be generated directly from a UML class diagram or action diagram by means of a series of refinement rules. An AST may be an invertible representation. In other words, it may be possible to traverse the AST and reconstruct the “surface syntax” of the legacy system or reconstitute it in textual form from the abstract structures. An AST may be augmented; it may be analyzed and updated using additional structures that describe other properties about the software. Common analyses that augment an AST with additional properties include constraint analysis, data-flow analysis, control flow analysis, axiomatic analysis, and denotational analysis. ASTs are generally augmented with additional analyses layers, such as type analysis, control-flow analysis, or data-flow analysis (to support code optimization). Augmentation may also support capture of software

engineering metrics and documentation. Having a standard metamodel to represent ASTs will facilitate interchange at a foundational level for all architecture-driven modernization work. Hence, the AST provides an appropriate formalism for the derivation of properties required for detailed knowledge discovery. Formally, in computer science, an AST is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators. Thus, the leaves have nullary operators, i.e., variables or constants. In computing, it is used in a parser as an intermediate between a parse tree and a data structure, the latter which is often used as a compiler or interpreter's internal representation of a computer program while it is being optimized and from which code generation is performed. The range of all possible such structures is described by the abstract syntax. An AST differs from a parse tree by omitting nodes and edges for syntax rules that do not affect the semantics of the program. The classic example of such an omission is grouping parentheses, since in an AST the grouping of operands is explicit in the tree structure. In contrast, an Abstract Semantic Graph (ASG) is a data structure used in representing or deriving the semantics of an expression a formal language (for example, a programming language). The scope of article includes only XMI-stored diagrams, considering that it is the most expressive and informative of the others [ASTM™, 2011], [Newcomb, 2005].

Abstract syntax tree helps to design XMI schemas. XMI schemas have hierarchical structure and tree serve to represent hierarchical structure of class diagram. Every XMI schema consists of the following declarations [XMI, 2015]:

1. An XML version processing instruction;
2. An optional encoding declaration that specifies the character set, which follows the ISO-10646 (also called extended Unicode) standard;
3. Any other valid XML processing instructions;
4. A schema XML element;
5. An import XML element for the XMI namespace;
6. Declarations for a specific model. Every XMI document consists of the declarations, unless the XMI is embedded in another XML document;
7. An XML version processing instruction.

Conclusion from the review

Review of software engineering standards, related to representing of information about software models shows the next:

- general approach to store software models is based on hierarchical representation of text information;
- rules for composing of this representation are based on abstract syntax tree;
- the format to represent hierarchical structure in readable manner is XML, namely the XMI;
- to choose software tool for effective (quick and strick) processing of such an information it is necessary to investigate text query based techniques.

Task

Task is to propose a technique and a software tool for estimation class diagram for accordance them to SOLID design principles. In order to accomplish this task it is necessary to do the following:

- investigate the format of class diagram storing (see previous chapters);
- ground choice of software techniques for class diagram verification;
- propose techniques for class diagram verification, that is based on analytical approach, proposed in paper [Chebanyuk, 2016];
- represent an algorithm for software tool working;
- describe a software architecture components.
- represent case study;
- verify case study results by means of analytical apparatus, proposed in [Chebanyuk, 2016].

Grounding of the choice of development environment and tools for XMI files processing

Concretely, each modeling environment tool stores and manages the models with its own internal format or its own “dialect,” even if a standard format is adopted. To improve interoperability between modeling tools, standardization bodies have defined specific model interchange languages.

The best-known model interchange language is XMI (XML Metadata Interchange), a standard adopted by OMG for serializing and exchanging UML and MOF models. Unfortunately, even if this has been devised as an interchange format, different tools still adopt it with different flavors.

Therefore, to provide a framework for operations on XMI-compliant diagrams, the information should be processed by the Model-to-Text transformation [XMI, 2015].

There are several ways to perform this operation and each of them provides a specific tool for extraction of information from the text. The most widely used is a regular expression (regex) engine that provides a special notation to complete this task (e.g. .”*name="\K.*?(?=".*)” to extract the name of the class). However, regular expressions are suitable for processing of natural texts and other non-structured information, but perform much worse when the task is to extract information from the text with strict and specific structure like the XML is. Other downsides include low level of extensibility and readability and therefore increased cost of maintenance.

The other possible solution is a Language INtegrated Queries (LINQ) featured in .NET platform:

class.Attributes().FirstOrDefault(f => f.Name.LocalName.ToLower() == “name”).

LINQ extracts the name of the class. Such queries allow easy manipulation of XML documents via elements of functional programming like Lambda expressions. By using this approach, one can benefit from easy-to-use and logical syntax of predicates.

Therefore, the LINQ was chosen for proceeding class diagram. It is a more robust technology that decreases the time the development cycle takes. This decision defines the overall technological stack of the project - the cross-platform and open source .NET Core framework.

Being based on a cross-platform technology, application of plug-in for class diagram verification may be used in many different ways and embedded into various different applications and systems. Therefore the core functionality should be implemented as a portable library in a .NET Standard format. It

provides a full-featured transformation and analysis Application Program Interface (API) for any types of applications: desktop, web, command-line.

Because of increasing role of web technologies in the modern software development industry, the reference system is implemented both in the form of REST-compliant web application and command-line tool. Web application allows it for user to easily analyze diagrams and manage the output of results. On the other hand, command line tool provides a simple and straightforward interface suitable for both desktop and server environment.

Designing of patterns for identification of class diagram components from XMI file

XMI stores the models in a tree-like structure where the root element is “XMI” and its descendants store information about UML entities such as classes, interfaces, and relations between them. According to XMI standard each entity should have a unique string ID that allow it to be referenced by the other entities. Self-sufficient such as class, interface, or relation are represented in the form of “packagedElement” XML elements with the corresponding “type” attribute. Their properties such as name shown or visibility level are specified with additional attributes. The embedded entities such as operations and attributes are represented as child elements of corresponding class and interface tags as “ownedOperation” and “ownedAttribute” respectively.

The generalization is represented in the form of attribute of the class that is a derived one (the same scheme is applied to interface realization). Such links are marked as “generalization” and “interfaceRealization” tags. Generalization stores a string ID of the parent class in its single “general” attribute, while the interface realizaition has three of them: (1) the “supplier” with identifier of the interface being implemented, (2) the “client” with ID of the class that implements it and (3) the “contract” with ID of the contract specified by the interface (suitable for contract programming, stores the same ID as “supplier” by default).

Table 1. Patterns and LINQ queries for XMI file processing

Aim of LINQ request	Fragment of XMI file	LINQ queries to parse XMI tags
Extract all classes	<packagedElement xmi:type="uml:Class" xmi:id="_qu0M8ASgEeeImrsnVV7-jw" name="Class2">	var class = diagramDoc.Descendants().Where(w => w.Name.LocalName.ToLower() == "packagedelement" && w.Attributes().Any(a => a.Name.LocalName.ToLower() == "type" && a.Value == "uml:Class"));
Get all operations of the particular class	<ownedOperation xmi:id="_HUTN4A-dEeeRAcEtZoXAZQ" name="Create"/>	var generalization = child.Descendants().FirstOrDefault(w => w.Name.LocalName.ToLower() == "ownedoperation");
Get all attributes of the particular class	<ownedAttribute xmi:id="_Ja4tMAZYEeeIgspGntkIFA" name="attribute1" visibility="private"> </ownedAttribute>	var generalization = child.Descendants().FirstOrDefault(w => w.Name.LocalName.ToLower() == "ownedattribute");

Aim of LINQ request	**Fragment of XMI file**	**LINQ queries to parse XMI tags**
Extract information about interface realizations of the particular class	<interfaceRealization xmi:id="_2WPxUA-cEeeRAcEtZoXAZQ" supplier="_u_OgcA-cEeeRAcEtZoXAZQ" client="_qu0bMA-cEeeRAcEtZoXAZQ" contract="_u_OgcA-cEeeRAcEtZoXAZQ"/>	var implementations = child.Descendants().Where(w => w.Name.LocalName.ToLower() == "interfacerealization");
Get information about attributes of the particular class representing ends of Associations' relation	<ownedAttribute xmi:id="_WaWnBA-dEeeRAcEtZoXAZQ" name="testClass2" visibility="private" type="_PE6DcA-dEeeRAcEtZoXAZQ" association="_WaWnAA-dEeeRAcEtZoXAZQ"> </packagedElement>	var associationAttributes= xmlClass.Descendants().Where(w => w.Name.LocalName.ToLower() == "ownedattribute" && w.Attributes().Any(a => a.Name.LocalName.ToLower() == "association")&& ! w.Attributes().Any(a => a.Name.LocalName.ToLower() == "aggregation"));

Aim of LINQ request	**Fragment of XMI file**	**LINQ queries to parse XMI tags**
Get information about attributes of the particular class representing ends of Aggregations' relation	<ownedAttribute xmi:id="_WaWnBA-dEeeRAcEtZoXAZQ" name="testClass3" visibility="private" type="_PE6DcA-dEeeRAcEtZoXAZQ" association="_WaWnAA-dEeeRAcEtZoXAZQ" aggregation="composite"> </packagedElement>	var aggregationAttributes= xmlClass.Descendants().Where(w => w.Name.LocalName.ToLower() == "ownedattribute" && w.Attributes().Any(a => a.Name.LocalName.ToLower() == "association")&& w.Attributes().Any(a => a.Name.LocalName.ToLower() == "aggregation"));
Extract all interfaces	<packagedElement xmi:type="uml:Interface" xmi:id="_u_OgcA-cEeeRAcEtZoXAZQ" name="TestInterface"/>	var interface = diagramDoc.Descendants().Where(w => w.Name.LocalName.ToLower() == "packagedelement" && w.Attributes().Any(a => a.Name.LocalName.ToLower() == "type" && a.Value == "uml:Interface"));
Extract all Association and Aggregation relations on class diagram	<packagedElement xmi:type="uml:Association" xmi:id="_WaWnAA-dEeeRAcEtZoXAZQ" memberEnd="_WaWnAQ-dEeeRAcEtZoXAZQ _WaWnBA-dEeeRAcEtZoXAZQ"/>	var xmlAssociations = diagramDoc.Descendants().Where(w => w.Name.LocalName.ToLower() == "packagedelement" && w.Attributes().Any(a => a.Name.LocalName.ToLower() == "type" && a.Value == "uml:Association"));

Description of class diagram estimation algorithm

The defined task consists of two parts: the class diagram importing and further verification. Steps for class diagram refinement algorithm are represented below:

1. Obtaining XMI representation of class diagram. Performing this step model to text transformation is done. This action is executed by software modeling environment.
2. Defining classes of class diagram.
3. Defining interfaces of class diagram.
4. Defining relations between classes and interfaces.
5. Checking class diagram to accordance to SOLID design principles.
6. Generating a report of class diagram accordance to SOLID design principles
7. Modification of class diagram by software architector in modeling environment (optional).

Points 2-6 are performed by designed software tool by means of LINQ queries proposed in previous points.

The general sequence diagram of the model import and verification is represented at the Figure 1.

To perform the model verification of SOLID principles compliance, it should be first transformed from the XMI-compliant text representation to the corresponding data structure. Parsing of XMI file, containing XMI class diagram representation is performed in several passes. At the first pass, a class diagram entities (classes, interfaces, use cases and actors) are extracted along with their attributes (e.g. operations and properties for class diagrams). Those attributes include class diagram components that are used in XMI scheme as ends of relations (e.g. inheritance or association) between corresponding entities. On the next pass, the information about the relations is extracted and the edges of the graph are established. The direction of edges is determined by the direction of the original relations on which they are built upon.

The developed application represents the class diagram (software model) as a directed graph, where the entities (classes and interfaces) are graph vertexes and nodes and relations between them (association, aggregation, composition, and inheritance) are the graph edges.

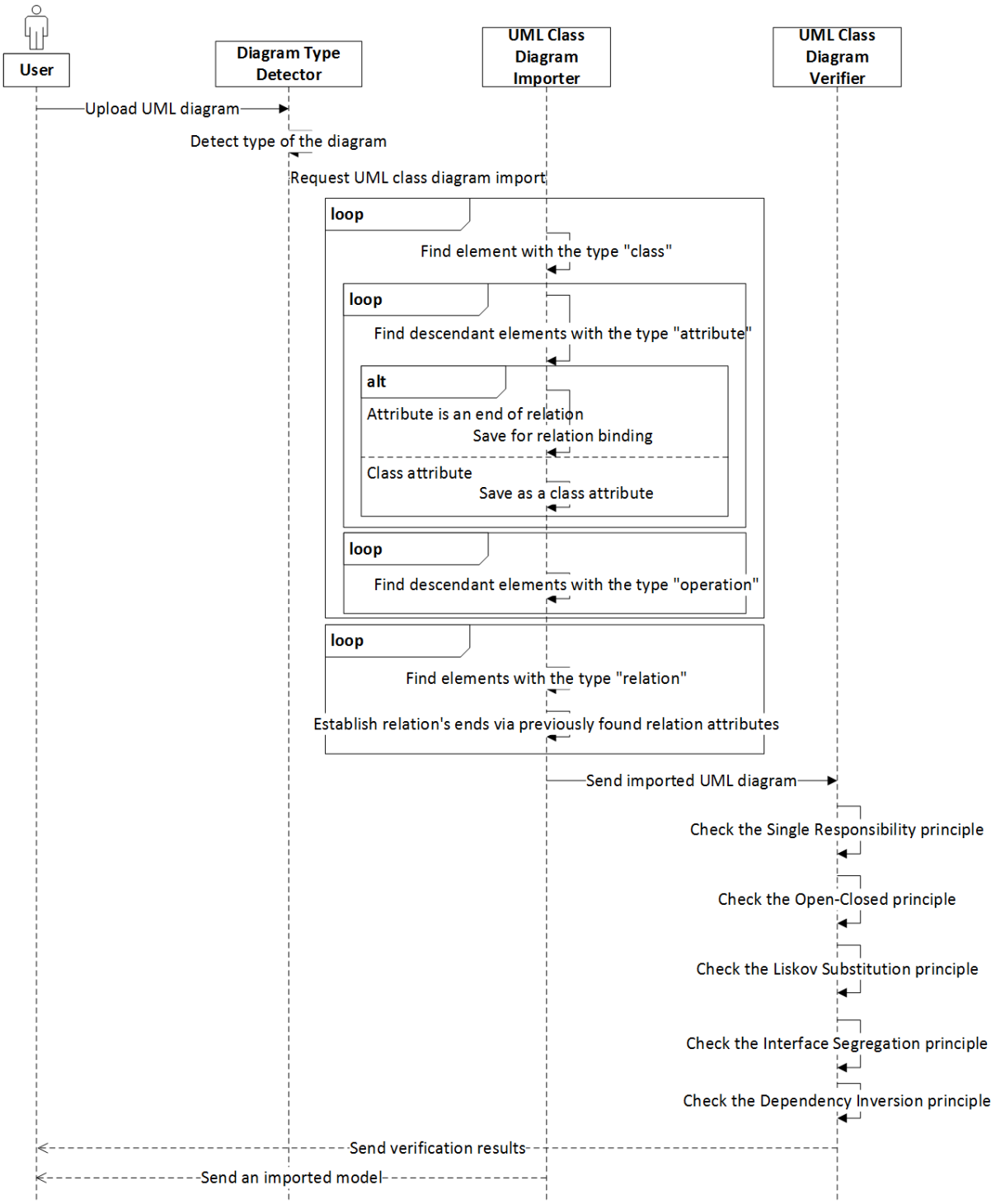


Figure 1. Sequence diagram of class diagram verification approach in accordance to SOLID design principles

The second task of designed approach is the software model verification to accordance to SOLID design principles. The verification module uses LINQ-to-Objects mechanism to provide the metrics of compliance using the syntax consistent with one used for extraction of information from the text representation of model.

Those LINQ queries verify all the SOLID design principles, namely:

- Single Responsibility;
- Open-Closed;
- Liskov Substitution;
- Interface Segregation;
- Dependency Inversion Principles.

Single Responsibility design principle is applied analyzing every class diagram entity separately. Other SOLID design principles are analyzed considering some of the interconnections between class diagram entities [Chebanyuk, 2016].

Software Product Architecture Description

The architecture of the software product is represented on Figure 2. The aim of designed software tool is to verify class diagram for accordance to SOLID design principles. Software architecture consists from several packages.

- "Model package" is used to prepare class diagram for further analysis;
- "Provider package" is used to extract the information from the XMI files and create the representation of the diagram on their basis;
- "Verification package" is used to provide verification tasks of model compliance to the SOLID design principles.

"Model package" contains classes for storing information about software model. These classes are UmlDiagram, UmlRelation, UmlStereotype. Also this package containes enumerations (UmlDiagramType, UmlRelationType and UmlVisibility). A list of classes that provide functionality of specific UML diagrams can be placed in separate sub-packages, namely UmlClass, UmlAttribute, UmlInterface, UmlOperation classes with IUmlClassMember interface for Class diagrams. They act as a software model entities and store properties like name of the entity, level of visibility etc. The collections

of elements like members of particular class or classes of the diagram are represented as a dictionary with XMI entity ID acting as a key to facilitate quicker access.

“Provider package” contains classes for class diagram import. It has importer XmiClassDiagramImporter which is responsible to handling import class diagram into the project. “Verification package” containes a single ModelVerifier class that provides methods for verification of the whole diagram or particular class. LINQ queries (Table 1) contained here allow flexible verification and provide results for frontend parts of the application.

To address the problem of verification, the product contains five methods of ModelVerifier class for SOLID compliance check, each addressing their own software design principle:

- IsSingleResponsibilitySatisfied checks the number of public operations of a particular UML class or interface. To mark the whole diagram as compliant, all classes and interfaces must be compliant;
- IsOpenClosedSatisfied checks if the number of classes participating in class hierarchy is big enough;
- IsLiskovSubstitutionSatisfied checks that there are no operations defined in the base classes or interfaces and not implemented on lower level of class hierarchy. It is the most complex one, utilizing algorithms such as cyclical graph traversal to perform verification;
- IsInterfaceSegregationSatisfied works specifically for the interfaces and verifies if the number of operations is not too big. Same as Single Responsibility, this principle must implemented by all interfaces to make the diagram compliant;
- IsDependencyInversionSatisfied enumerates all the classes on the lowest level of hierarchy and checks that there are no association or composition relations with them;

The results of verification of UML Class diagrams are combined with the lists of recognized entities and relations between them to facilitate the detection of non-well-formed entities and their correction.

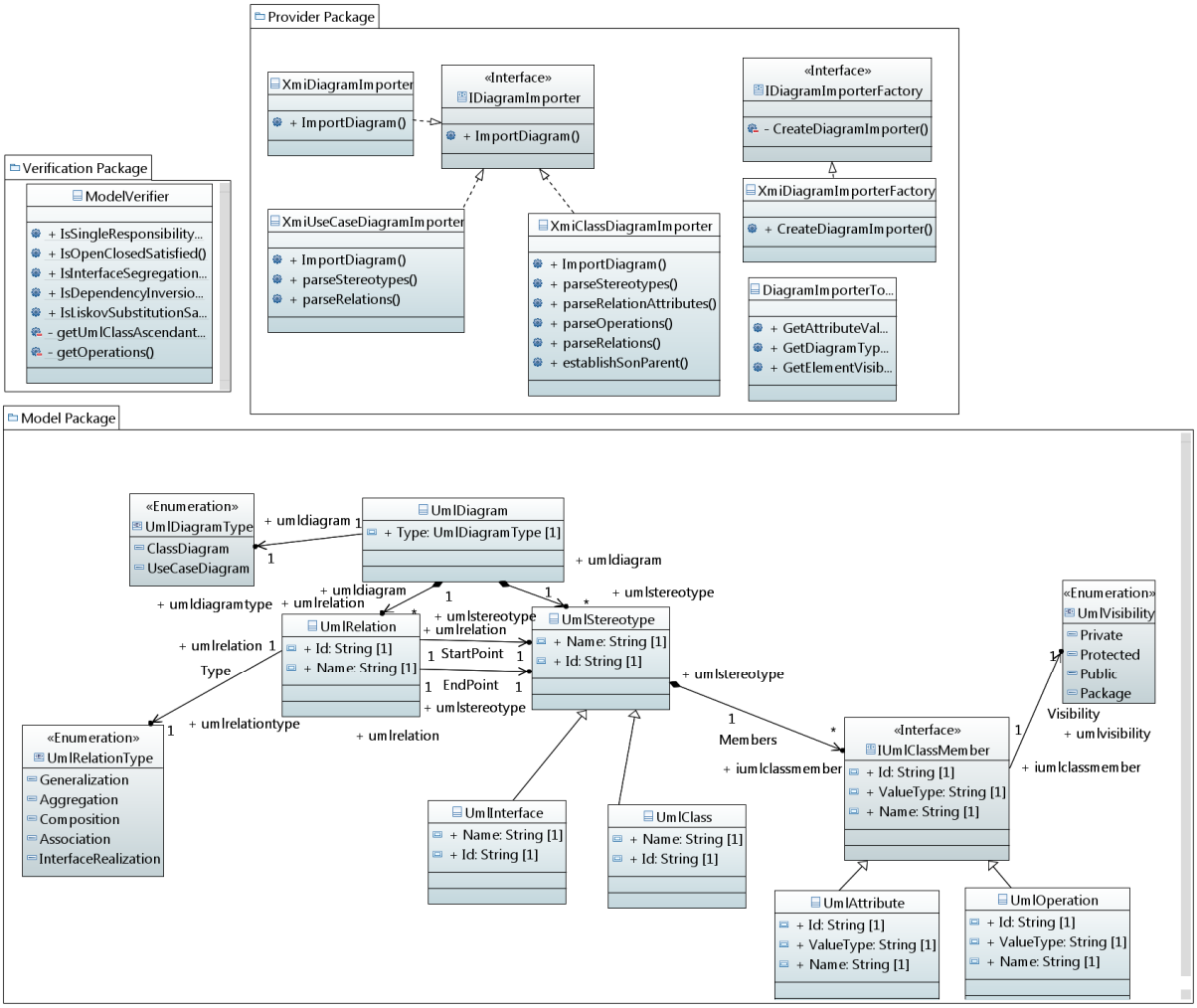


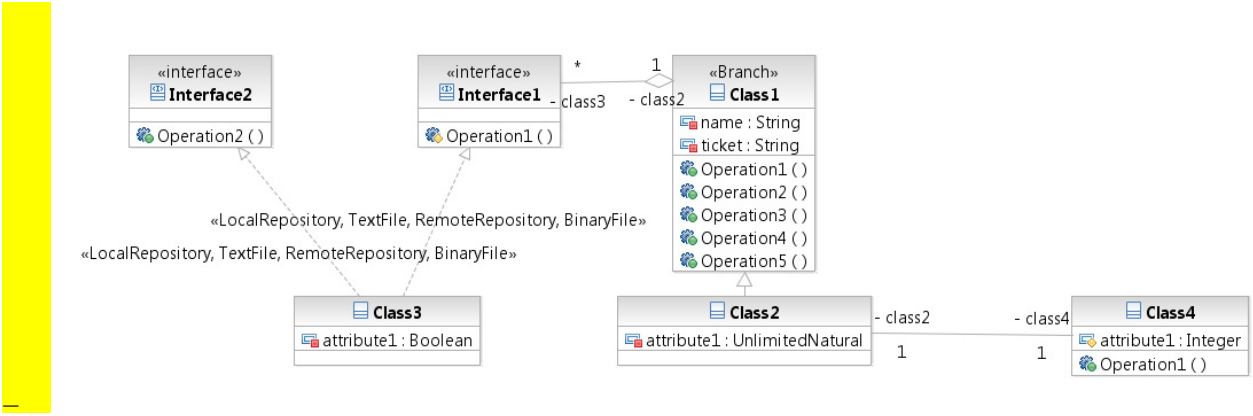
Figure 2 Architecture of software product for class diagram refinement

Case Study

To perform the case study, an example Class Diagram was created (Figure 3).

It contains several classes:

- Class1 contains a big amount of operations and several attributes;
- Class2 is a descendant of Class1 with one own attribute and is linked via aggregation with the Class3;
- Class 3 implements interfaces Interface1 and Interface2 each containing one operation;
- Class4 is connected with the Class3 via association.



— *Figure 3 Example of Class Diagram*

After selecting the correct type of diagram (XMI produced by IBM Rational Software Architect) and uploading the file (figure 4), the application begins the process of diagram verification.

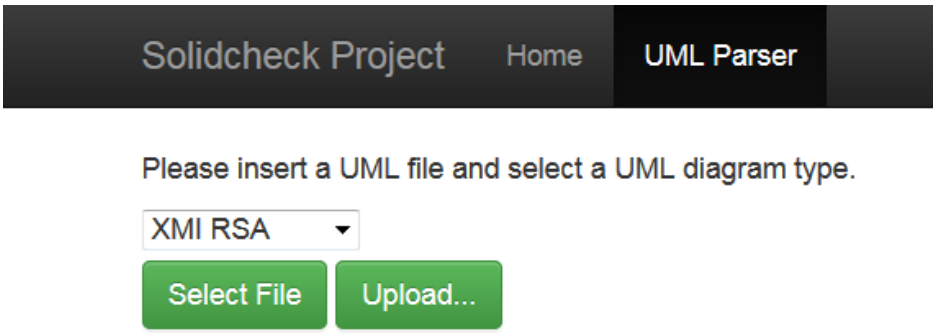


Figure 4 Diagram Upload Page

The result of class diagram estimation is represented on figures 5, 6 and 7.

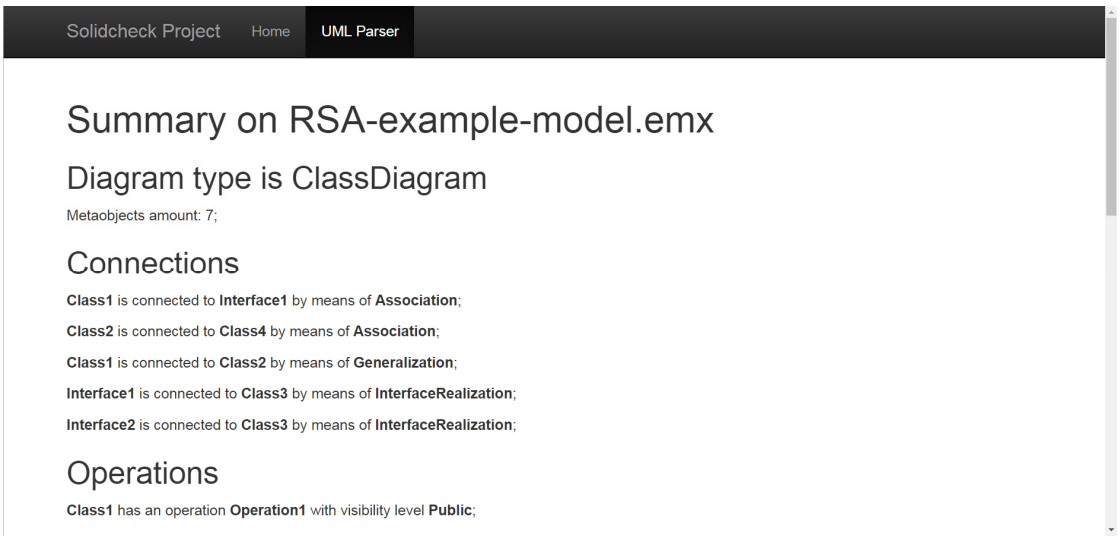


Figure 5 Results of Diagram Verification (Part 1)

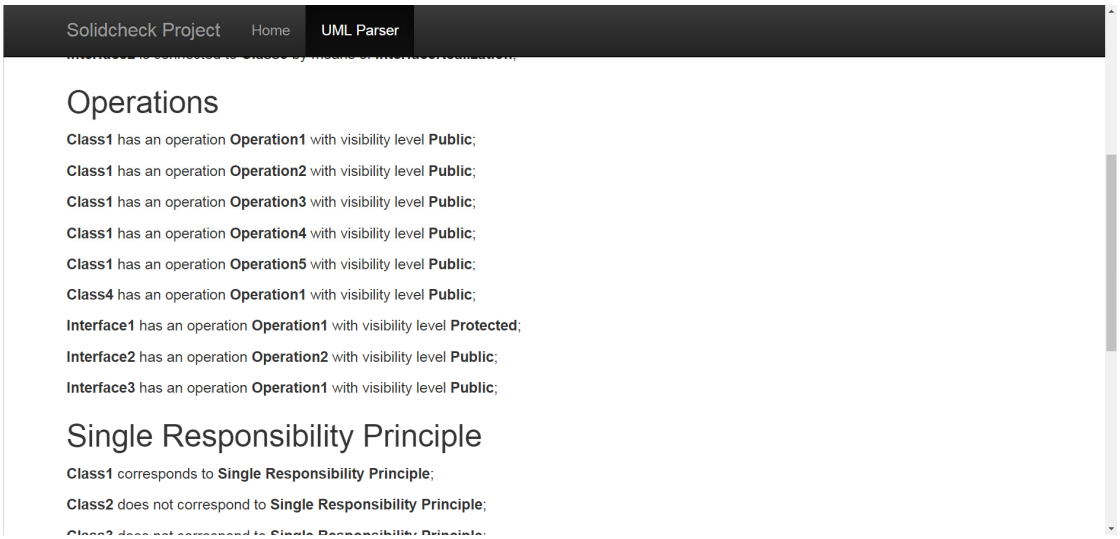


Figure 6 Results of Diagram Verification (Part 2)

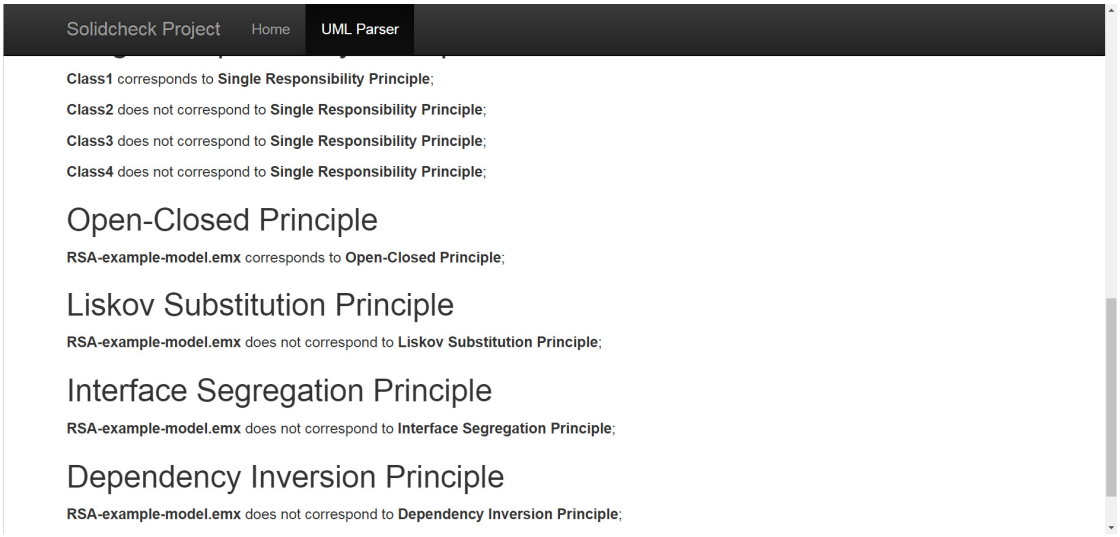


Figure 7 Results of Diagram Verification (Part 3)

Textual representation:

Summary on RSA-example-model.emx

Diagram type is ClassDiagram

Metaobjects amount: 7;

Connections

Class1 is connected to Interface1 by means of Association;

Class2 is connected to Class4 by means of Association;

Class1 is connected to Class2 by means of Generalization;

Interface1 is connected to Class3 by means of InterfaceRealization;

Interface2 is connected to Class3 by means of InterfaceRealization;

Operations

Class1 has an operation Operation1 with visibility level Public;

Class1 has an operation Operation2 with visibility level Public;

Class1 has an operation Operation3 with visibility level Public;

Class1 has an operation Operation4 with visibility level Public;

Class1 has an operation Operation5 with visibility level Public;

Class4 has an operation Operation1 with visibility level Public;

Interface1 has an operation Operation1 with visibility level Protected;

Interface2 has an operation Operation2 with visibility level Public;

Interface3 has an operation Operation1 with visibility level Public;

Single Responsibility Principle

Class1 corresponds to Single Responsibility Principle;

Class2 does not correspond to Single Responsibility Principle;

Class3 does not correspond to Single Responsibility Principle;

Class4 does not correspond to Single Responsibility Principle;

Open-Closed Principle

RSA-example-model.emx corresponds to Open-Closed Principle;

Liskov Substitution Principle

RSA-example-model.emx does not correspond to Liskov Substitution Principle;

Interface Segregation Principle

RSA-example-model.emx does not correspond to Interface Segregation Principle;

Dependency Inversion Principle

RSA-example-model.emx does not correspond to Dependency Inversion Principle;

As it can be seen, most of the classes does not correspond to the Single Responsibility principle because of number of their operations being too low to successfully perform even a single function. No other SOLID design principles except Open-Closed principle are satisfied.

Verifying case study results by means of analytical apparatus

Prove that considered class diagram corresponds to SOLID design principles.

Consider C as a set of classes in class diagram.

$$C = \{class1, class2, class3, class4\} \quad (1)$$

1. Single responsibility design principle

Number of public methods of class $c \in C$ is denoted as follows: $n(B_c^{public})$ where - B_c^{public} is a set of public methods of class $c \in C$ [Chebanyuk, 2016]. Consider that for performing one task classes should have no more than four public methods, namely $c \in C$, $n(B_c^{public}) \in \{3,4,5,6,7,8,9\}$,

$$class1 \in C, n(B_{class1}^{public}) = 5, class2 \in C, n(B_{class2}^{public}) = 0,$$
$$class3 \in C, n(B_{class3}^{public}) = 0, class4 \in C, n(B_{class4}^{public}) = 1 \quad (2)$$

Consider predicate expression for checking single responsibility design principle.

$$P(c, n(B_c^{public})) == c \ has \ n(B_c^{public}) \ public \ methods$$

$$P(class1, n(B_{class1}^{public})) = class1 \ has \ 5 \ public \ methods = true,$$
$$P(class2, n(B_{class2}^{public})) = class2 \ has \ 0 \ public \ methods = false,$$
$$P(class3, n(B_{class3}^{public})) = class3 \ has \ 0 \ public \ methods = false, \quad (3)$$
$$P(class4, n(B_{class4}^{public})) = class4 \ has \ 1 \ public \ methods = false.$$

Expressions (3) mach results of software tool according single responsibility design principle. (Figure 5).

2. Open-Close SOLID design principle

Class diagram, represented in Figure 8 has structural characteristics that match to design pattern “Abstract Factory”. Consider OCL Constraints of this pattern [ICER, 2015]. Class diagram is also taken from [ICER, 2015].

Correspondence of abstract factory participants to elements of class diagram presented at the Figure 8 is described in Table 2.

Table 2 Correspondence of abstract factory participants to elements of class diagram

classes of class diagram on figure3	abstract factory participants (figure 8)
Class1	AWindows
Class2	motifWindow
Class3	PMwidgetFactory
Interface1	WidgetFactory

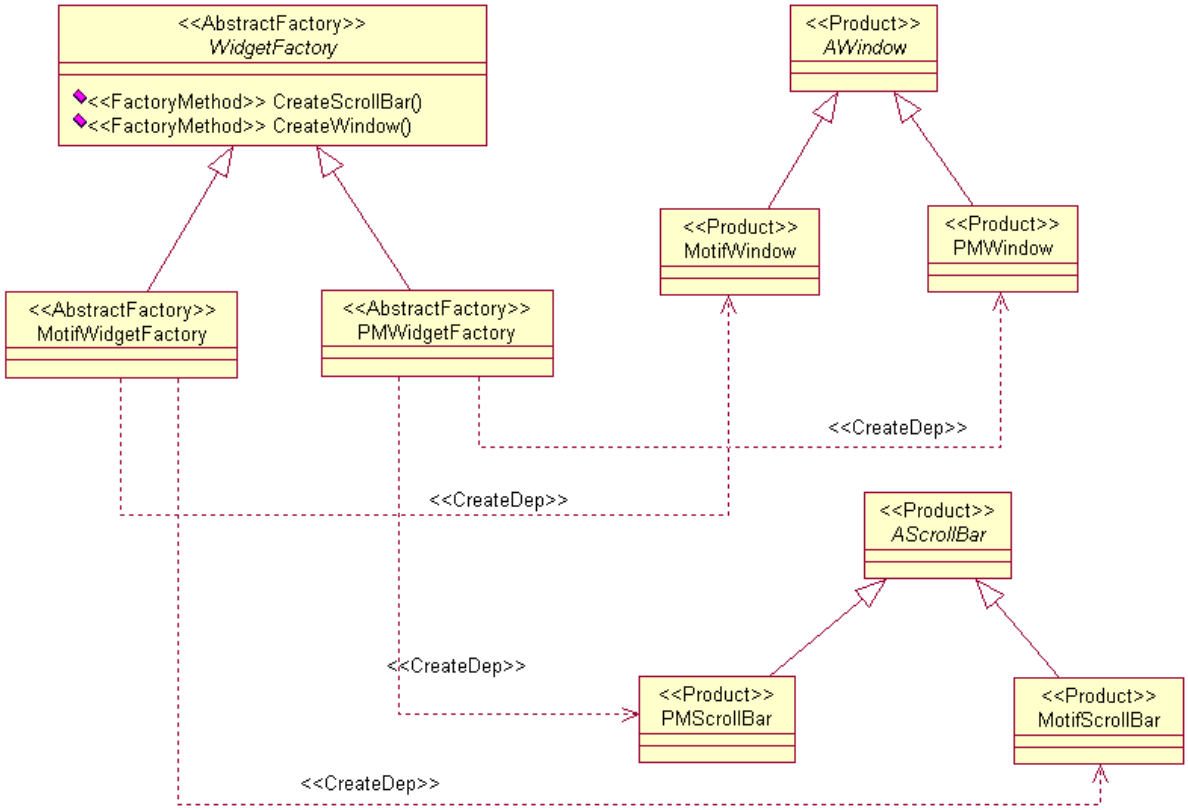


Figure 8. Abstract factory class diagram [ICER, 2015]

Rewrite OCL constrains from [ICER, 2015] to class diagram, represented on Figure 8:

context class1
inv: self.isRoot implies (not self.isLeaf and self.isAbstract)

inv: self.isAbstract implies (self.specialization->size() > 0 and
self.specialization->forAll (c | c.child.oclIsKindOf(class1)))

inv: not self.isAbstract implies self.createDep->size() = 1

context WidgetFactory
inv: self.isRoot implies (not self.isLeaf and self.isAbstract)

inv: self.isAbstract implies (self.specialization->size() >= 1 and
self.specialization->forAll (c | c.child.oclIsKindOf (WodgetFactory)))

inv: not self.isRoot implies (self.createDep->size() > 0)

Authors of [ECIR, 2015] formulate necessary structural characteristics of Abstract Factory design pattern. Represent characteristics, that match to classes in Figure 8, i.e. each abstract factory class

hierarchy has an abstract base class, which needs to have at least two sub-classes (to support multiple families of products, one family per abstract factory).

In paper [Chebanyuk, 2016] set of operation *OPER* for interconnection between classes is defined.

$$OPER = \{inh, ass, aggr, comp\} \tag{4}$$

where: *inh* - inheritance, *ass* - association, *aggr* – aggregation, *comp* - composition.

Refer to the term "functionality" from paper [Chebanyuk, 2016]. Consider $c, c^{|} \in C$. General idea of spreading class functionality $F(c)$, when c and $c^{|}$ are connected by means of $oper \in OPER$, is denoted as follows:

$$F(c)^{oper} = F(c) \cup F(c^{|}) \tag{5}$$

Sign $\cup$ depicts that functionality of class *c* ($F(c)$) is extended with functionality of class $c^{|}$ ($F(c^{|})$).

The same is true when functionality of $c \in C$ is spreading by inheritance or including reference to $i \in I$. It is denoted as follows:

$$F(c)^{oper} = F(c) \cup i \tag{6}$$

At the figure 3 *class2* inherits *class1.* Denote it by the following:

$$\begin{aligned} &\exists class1 \in C, \exists class2 \in C, \\ &F(class2)^{inh} = F(class2) \cup F(class1) \\ &P(F(class2)^{inh}) = true \end{aligned} \tag{7}$$

The root of each product tree must be abstract.

At diagram (Figure 3) *class3* inherits interfaces. Thus, the functionality of this class $F(class3)$ is represented by the following:

$$\begin{aligned} &\exists class3 \in C, I = \{\text{int}\, erface1\}, \\ &F(class3)^{inh} = F(class3) \cup \text{int}\, erface1 \\ &P(F(class3)^{inh}) = true \end{aligned} \quad (8)$$

Expressions 7 and 8 show that to make some changes to class diagram it is not necessary to modify code inside classes. As class3 and class2 depend upon abstractions one can modify class diagram not performing changes to existing methods of the classes. The same conclusion was donw by the software tool (figure 5). Thus class diagram correspond to Open-Close Design principle.

3. Interface Segregation

In order to follow interface segregation principle it is necessary to define the next:

- number of methods in interfaces, that are parent for classes should be less than five;
- class, that inherits interfaces should not contain empty overridden methods

$$\begin{aligned} &\exists c \in C, \exists i \in I, \\ &F(c)^{inh} = F(c) \cup i \\ &B_c^{public} = \{\beta_c^{public} \mid \beta_c^{public} \neq \varnothing\} \end{aligned} \quad (9)$$

Consider interface1 and interface2 from class diagram. They are following to the first condition, namely, every interface contains one public method.

Consider class3, inheriting interface1 and interface2. The number of public methods in it is zero. Also there are no other methods in this class. The second condition for interface segregation principle is not performed.

$$\begin{aligned} &\exists class3 \in C, \exists \text{int}erface1 \in I, \\ &F(c)^{inh} = F(c) \cup \text{int}erface1 \\ &B_{class_3}^{public} = \{\beta_{class_3}^{public} \mid \beta_{class_3}^{public} = \varnothing\} \end{aligned} \quad (10)$$

Expression (10) contradicts to expression (9). That why, class diagram does not follow interface segregation design principle.

4. Liskov Substitution design principle.

According to [Chebanyuk, 2016] represent the necessary conditions for Liskov Substitution SOLID design principle.

a) $\exists c \in C$ has a reference to another class diagram class $c^{|} \in C$. These classes are not connected by inheritance relationship. Then:

$$\begin{aligned} &\exists c, c^{|} \in C \quad F(c)^{acc} = F(c) \cup F(c^{|}) \\ &P(F(c)^{acc}) = F(c)^{acc} = true \end{aligned} \quad (11)$$

b) $c^{|} \in C$ has at least two inherited classes. Denote a set of such inheritors as $Cl\,(c^{|})$ .Then:

$$\begin{aligned} &\exists c^{|} \in C, \exists c_1^{|} \in C, \quad F(c_1^{|})^{inh} = F(c_1^{|}) \cup F(c^{|}) \\ &P(F(c_1^{|})^{inh}) = F(c_1^{|})^{inh} = true \\ &Cl(c_1^{|}) = \{c_1^{|} \mid P(F(c_1^{|})^{inh}) = true, c_1, c_1^{|} \in C\} \mid Cl(c^{|}) \mid > 2 \end{aligned} \quad (12)$$

Expression (12) should be applied for other classes' inheritors in $c^{|} \in C$ hierarchy, namely.

c) $c_1^{|} \in Cl(c^{|})$ has non empty overridden methods. Denote a set of these methods as

$$B_{c_1}^{\ override} \subset B_{c_1}^{\ public} \quad (13)$$

Then:

$$\beta_{c_1}^{override} \neq \varnothing, \quad \beta_{c_1}^{override} \in B_{c_1}^{override} \tag{14}$$

Expression (14) also should be applied for other classes' inheritors in $c^{|} \in C$ hierarchy.

Necessary condition for corresponding to Liskov substitution design principle is that class diagram should contain inheritance. UML diagram contains inheritance relationship between class1 and class2. Expression (12) shows this. But $|Cl(class1^{|})|=1$. That is why class diagram on figure 3 does not follow to Liskov Substitution design principle.

5. Dependency Inversion design principle

Represent the last SOLID design principle: Dependency Inversion design principle. Dependency Inversion Design Principle requires only one structural characteristic: functionality of $c \in C$ should be extended by means of one of the three variants, namely: $c^a \in C$, $c^{|} \in C$ (classes that are related on a top of hierarchy) or $i \in I$.

$$F(c)^{acc} = \begin{cases} F(c) \cup F(c^a) \\ F(c) \cup i \\ F(c) \cup F(c^{|}) \end{cases} \tag{15}$$

Consider class3 and class2

$$\begin{gathered} F(class3)^{inh} = F(class3) \cup \text{int}erface1 \cup \text{int}erface2 \\ F(class2)^{inh} = F(class2) \cup F(class1) \end{gathered} \tag{16}$$

$$F(class4)^{ass} = F(class4) \cup F(class3) \quad (17)$$

Class diagram (figure 3) does not follow to Dependency inversion design principle. Functionality of class4 is spread by class3 (17) that is not spread by means of abstractions.

Conclusion

An approach for class diagram estimation is proposed in this paper.

To refine class diagram it is necessary to check whether it corresponds to SOLID design principles. It is done in automatic way by means of proposed software tool. Description of this tool architecture is represented in the paper.

Contribution of this paper is a representation of approach and techniques for defining constituents of class diagram. Having information about class diagram content stakeholders may solve many important tasks in software development life cycle process more effectively. “Provider package” extracts initial information about class diagram. This information is used for all operations of software model processing, namely model comparison, merging, refinements, refactoring, and versioning.

Let's describe advantage of the proposed approach in respect to procedure of class diagram estimation by means of IBM RSA.

Using of IBM RSA, for estimation of class diagram, tools, supporting OCL, are involved. To estimate class diagram in accordance to SOLID design principles, OCL expressions should correspond with predicates for verifying class diagram, proposed in [Chebanyuk, 2016]. But in OCL expression context should be mentioned. There are no tools, allowing changing context automatically. That why software architect spends the same time to rewrite contexts for all OCL expressions of class diagrams or to analyze them mentally.

The tool, represented in this paper, allows to estimate automatically class diagrams according to SOLID design principles and to reuse them many times when changes to software are performed.

Further Research

It is planned to design an approach and software tool for Model-to-Model transformation, using collaboration of existing tools and environments, proposed in paper [Chebanyuk, 2017], namely IBM Rational Software Architect 7.5.5.2, Medini QVT, and Eclipce plug-ins. It is planned to obtain

information about class diagram components using LINQ queries (Table 1). Then modify QVT scripts for supporting Many-to-Many transformation. Source and target software models are visualized in IBM Rational Software Architect.

Acknowledgement

Authors thank to Marco Brambilla for discussing ideas of paper [Chebanyuk, 2016] at the conference MODELSWARD 2016.

This discussion was a starting point to start research represented in this article.

Bibliography

[ASTM™, 2011] Architecture-Driven Modernization™ (ADM™): Abstract Syntax Tree Metamodel™ (ASTM™) http://www.omg.org/spec/ASTM/1.0

[Chebanyuk, 2016] Chebanyuk E. and Markov K. An Approach to Class Diagrams Verification According to SOLID Design Principles.In Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD (2016), ISBN 978-989-758-168-7, pages 435-441. DOI: 10.5220/0005830104350441 access mode

http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=HASwCJGMcXc=&t=1

[Chebanyuk, 2017] Elena Chebanyuk and Kyryl Shestakov An Approach for Design of Architectural Solutions Based on Software Model-To-Model Transformation International Journal “Information Theories and Applications”, ISSN 1310-0513 Vol. 24, Number 1, 2017, pages 60-84

[Eclipce, 2016] Eclipce desctip IDE https://eclipse.org/ide/

[EMF, 2011] EMF Compare, 2011: Available from World Web: http://wiki.eclipse.org/index.php/EMF_Compare

[IBM, 2015] IBM, Rational software architect designer http://www-03.ibm.com/software/products/ru/ratsadesigner

[ICER, 2015] Western Michigan University project https://www.cs.wmich.edu/~OODA/patterns/AbsFactory.html

[Lakhal, 2012] Lakhal F., Dubois H. and Rieu D. (2012). P²E: A Tool for the Evolution Management of UML Profiles.In Proceedings of the 7th International Conference on Software Paradigm Trends -

Volume 1: ICSOFT, ISBN 978-989-8565-19-8, pages 211-217. DOI: 10.5220/0004081002110217 access mode http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=jbI8fcFzMFo=&t=1

[Newcomb, 2005] Philip Newcomb. Chief Executive Officer Abstract Syntax Tree Metamodel Standard ASTM Tutorial 1.0 access mode: http://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/T-3_Newcomb.pdf

[Ocampo, 2009] Joe Ocampo, Jason Meridth, Chad Myers, Sean Chambers, Ray Houston, Jimmy Bogard, Gabriel Schenker, and Derick Bailey, Pablo's solid software development, 2009 e-book access mode : https://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf

[OCL, 2014] Object Constraint Language Version 2.4 OMG standard http://www.omg.org/spec/OCL/2.4/PDF

[Papyrus, 2012] Papyrus, 2012. Available from World Web: www.papyrusuml.org. Unified Modeling Language (UML), 2011. Available from World Web: http://www.omg.org/spec/UML/2.3/

[Sanches, 2015] Sanchez, L.E., Diaz-Pace, J.A., Zunino, A. et al. An approach based on feature models and quality criteria for adapting component-based systems Journal Software Engineering Research and Development (2015) 3: 10. doi:10.1186/s40411-015-0022-1

[Sielis, 2015] Sielis, G.A., Tzanavari, A. & Papadopoulos, G.A. ArchReco: a software tool to assist software design based on context aware recommendations of design patterns Journal Software Engineering Research and Development (2015) 2. doi:10.1186/s40411-017-0036-y

[XMI, 2015] XML Metadata Interchange access mode http://www.omg.org/spec/XMI/

Authors' Information

Olena Chebanyuk *– Software Engineering Department, National Aviation University, Kyiv, Ukraine,*

Major Fields of Scientific Research: *Model-Driven Architecture, Model-Driven Development, Software architecture, Mobile development, Software development,*

e-mail: chebanyuk.elena@ithea.org

Dmytro Povaliaiev *– Software Engineering student, National Aviation University, Kyiv, Ukraine,*

Major Fields of Scientific Research: *Software Architecture, Software Development, Model-Driven Architecture, Model-Driven Development,*

e-mail: dmytro.povaliaiev@gmail.com