УДОСКОНАЛЕННЯ ПРОЦЕСІВ ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

УДК 004.4:004.738.5(045)

Chebanyuk O.V., Povaliaiev D.
National Aviation University

SOFTWARE ARCHITECTURE VERIFICATION APPROACH

Verification is very important part of software architecture designing. In AGILE approach, architectural solutions are represented as static software models, namely UML diagrams.

Analytical foundations of class diagram verification, based on predicate logic, were proposed in paper. This research continues here, by proposing LINQ queries to define interconnection between class diagram elements.

The approach, proposed in this paper, is based on the automatic parsing of class diagram XMI file using suggested LINQ queries for every SOLID design principle.

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

Introduction

The most widespread approach of software development lifecycle management nowadays is Agile. In Agile every operation in software development lifecycle management is performed by means of software model processing. Software models are represented as UML diagrams.

When customer changes software requirements all software models are changed too. Static software models, that reflect software architecture, are changed too. According UML 2.5 standard static software models are class, component, and package diagrams.

Following Model-Driven Engineering principles, an important step proposing architecture solutions is their verification. It 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. The lack of tools, which allows performing class diagram verification in automatic mode, becomes a motivation for authors to propose considering approach. It is proposed to use it in Design phase of software development life-cycle (SDLC).

Planning
Analysis
Design
Implementation
Testing
Maintenance

Fig. 1. Structure of a general SDLC

Nowadays there are a lot of tools for automating many operations within software development lifecycle, but they still pay not enough attention to keeping track of architectural solutions verification. In fact there are no tools that can grant SOLID design principles consistency.

But, using an advanced modeling environments like Rational Software Architect [IBM, 2015] or Eclipse plugin 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 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.

Lack of automatized tools for class diagram verification becomes a precondition for formulating task of research. Research, performed in this article, was started in [Chebanyuk, 2016]. It was proposed to verify class diagram in accordance to SOLID design principles by means of predicate expressions. But rising of effectiveness of architecture verification process requires software to automate this operation.

Goal of the article

Propose LINQ queries which verify all the SOLID design principles, namely:

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

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

Related standards

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:

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 following declarations, unless the XMI is embedded in another XML document:
7. An XML version processing instruction [XMI, 2015].

Class diagram are stored in XMI format. XMI schema is composed using hieratical structure of XML tags.

Such 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].

For UML, the abstract syntax is defined as a MOF metamodel. The UML specification also defines additional constraints that the metamodel representation of a valid UML model is required to meet. These constraints are the equivalent of the static semantics of UML.

However, since these constraints can all be checked statically, they are not part of the execution semantics of UML. Indeed, any model that violates one or more of these additional constraints is not actually well formed. Such an ill-formed model cannot really be assigned any meaning at all.

In functional UML, static semantics are not considered to be part of the execution semantics to be specified. That is, any well-formed model is already presumed to have met all the constraints imposed on the abstract syntax as defined in the UML Specification. Semantic meaning will only be defined for models that are well formed in this sense.

The Action Language for Foundational UML (or “Alf”) is a textual surface representation for UML modeling elements [ALF™, 2017]. The execution semantics for

Alf are given by mapping the Alf concrete syntax to the abstract syntax of the standard Foundational Subset for Executable UML Models (known as "fUML"). The result of executing an Alf input text is thus given by the semantics of the fUML model to which it is mapped, as defined in the fUML specification.

A primary goal of an action language is to act as the surface notation for specifying executable behaviors within a wider model that is primarily represented using the usual graphical notations of UML. For example, this might include methods on the operations of classes or transition effect behaviors on state machines.

However, Alf also provides an extended notation that may be used to represent structural modeling elements. Therefore, it is possible to represent a UML model entirely using Alf, though Alf syntax only directly covers the limited subset of UML structural modeling available in the fUML subset [ALF™, 2017].

Review of software engineering standards shows that information about class diagrams stored in XMI format is represented as text organized as tree-like structure. To verify architectural solutions different software tools, for example IBM Rational Software Architect or Eclipce work with software profiles or problem domain metamodels. Profiles, represented as class diagram with constrains, expressed in OCL. IBM RSA and Eclipse engines proceed with text representation of class diagram, linking it with constraints. But OCL constraints are interconnected with naming of class. For using tools verifying class diagrams it is necessary to follow strict naming. From the other side, lack of tools for analyzing class diagram static semantic encourages authors to design and develop software tool for class diagram analysis.

Proposed approach

Investigation of XMI file regularities

XMI stores the software 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 allows it to be referenced by the other entities. Self-sufficient, such as class, interface, or relation, is represented in the form of "packagedElement" of 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 realization has three of them: the "supplier" with identifier of the interface being implemented, the "client" with ID of the class that implements it, and the "contract" with ID of the contract specified by the interface (suitable for contract programming, stores the same ID as "supplier" by default).

LINQ queries to extract information about class diagram components

Table 1 illustrates the LINQ queries proposed by authors for processing XMI file in which class diagrams are stored.

Table 1.
LINQ queries for defining class diagram elements

SOLID Design Principle Name	LINQ query for defining SOLID design principle

Single Responsibility design principle	*var publicOpsNumber = umlClass.Members.Values.Where(w => w.GetType() == typeof(UmlOperation) &&*
	w.Visibility == UmlVisibility.Public).Count();
	return publicOpsNumber >= 3 && publicOpsNumber <= 7;

Open-Closed design principle	var descendantsNumber = diagram.Stereotypes.Values.Where(w => d*iagram.Relations.Values.Any(a =>* *(a.Type == UmlRelationType.Generalization ||* *a.Type == UmlRelationType.InterfaceRealization) &&*

SOLID Design Principle Name	LINQ query for defining SOLID design principle

	(a.StartPoint == w || a.EndPoint == w))).Count();
	if ((double)descendantsNumber / (double)diagram.Stereotypes.Values.Count() >= 0.7) {
	return true;
	}
	return false;

Liskov Substitution design principle	*foreach (var umlClass in umlDiagram.Stereotypes.Values)*
	{
	var ascendantStereotypes = getUmlClassAscendants(umlDiagram, umlClass);
	if (!ascendantStereotypes.Any())
	{
	continue;
	}
	var ascendantsQueue = new Queue<UmlStereotype>();
	foreach (var stereotype in ascendantStereotypes)
	{
	ascendantsQueue.Enqueue(stereotype);
	}
	var currentClassOps = getOperations(umlClass);
	while (ascendantsQueue.Any())
	{
	var currentAscendant = ascendantsQueue.Dequeue();
	var ascendantOps = getOperations(currentAscendant);
	if (ascendantOps.Any(op => !currentClassOps.Contains(op)))
	{
	return false;
	}
	if (!umlDiagram.Relations.Any(relation => (relation.Value.StartPoint == currentAscendant || relation.Value.EndPoint == currentAscendant) &&
	relation.Value.Type != UmlRelationType.Generalization && relation.Value.Type != UmlRelationType.InterfaceRealization))
	{
	return false;
	}
	ascendantStereotypes = getUmlClassAscendants(umlDiagram, umlClass);
	foreach (var stereotype in ascendantStereotypes)
	{
	ascendantsQueue.Enqueue(stereotype);
	}
	}
	} return true;

Interface Segregation design principle	*var umlInterfaces = umlDiagram.Stereotypes.Values.Where(w => w.GetType() == typeof(UmlInterface));*
	foreach (var umlInterface in umlInterfaces)
	{
	var publicOps = getOperations(umlInterface).Where(w => w.Visibility == UmlVisibility.Public);
	if (publicOps.Count() > 5)
	{
	return false;

SOLID Design Principle Name	LINQ query for defining SOLID design principle
	} var descendants = umlDiagram.Relations.Values.Where(relation => relation.Type == UmlRelationType.InterfaceRealization && relation.StartPoint == umlInterface).Select(s => s.EndPoint); foreach (var descendant in descendants) { var descendantOps = getOperations(descendant).Where(w => w.Visibility == UmlVisibility.Public); if (publicOps.Any(op => !descendantOps.Contains(op))) { return false; } } }
Dependency Inversion design principle	*var lowestHierarchyStereotypes = umlDiagram.Stereotypes.Values.Where(w => umlDiagram.Relations.Values.Any(a => (a.Type == UmlRelationType.Generalization || a.Type == UmlRelationType.InterfaceRealization) && a.EndPoint == w) && !umlDiagram.Relations.Values.Any(a => (a.Type == UmlRelationType.Generalization || a.Type == UmlRelationType.InterfaceRealization) && a.StartPoint == w)); return !umlDiagram.Relations.Values.Any(a => a.Type != UmlRelationType.Generalization && a.Type != UmlRelationType.InterfaceRealization && lowestHierarchyStereotypes.Any(lhc => a.StartPoint == lhc || a.EndPoint == lhc));*

Conclusions

In this article, the structure of XMI file, used to save class diagrams was investigated. Then LINQ queries for class diagram verification in accordance to SOLID design principles were proposed. Software tool for analyzing class diagrams, based on these LINQ queries, allows avoiding OCL limits [OCL, 2014]. Obtained information about class diagram components allows providing further flexible analysis in software modeling environments [Papyrus, 2012], [IBM, 2015], for example software model transformation techniques [Chebanyuk, 2017].

Futher researches

Using designed LINQ queries 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;

– 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.

References

1. Architecture-Driven Modernization (ADM): Abstract Syntax Tree Metamodel™ (ASTM) [Electronic resource] – Access mode: http://www.omg.org/spec/ASTM/1.0

2. Action Language For Foundational UML, Version 1.1. [Electronic resource] – Access mode: http://www.omg.org/spec/ALF/1.1/

3. Chebanyuk E., 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 – Vol. 1: MODELSWARD. – 2016. – P. 435-441. DOI: 10.5220/0005830104350441 [Electronic resource] – Access mode: http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=HASwCJGMcXc=&t=1

4. Chebanyuk E., Shestakov K. An Approach for Design of Architectural Solutions Based on Software Model-To-Model Transformation. // International Journal «Information Theories and Applications» – Vol. 24. – 2017. – P. 60-84.

5. Eclipce desctip IDE [Electronic resource] – Access mode: https://eclipse.org/ide/

6. IBM, Rational software architect designer [Electronic resource] – Access mode: http://www-03.ibm.com/software/products/ru/ratsadesigner

7. Object Constraint Language Version 2.4 OMG standard [Electronic resource] – Access mode: http://www.omg.org/spec/OCL/2.4/PDF

8. Papyrus, 2012. [Electronic resource] – Access mode: www.papyrusuml.org.

9. Unified Modeling Language (UML), 2011. [Electronic resource] – Access mode: http://www.omg.org/spec/UML/2.3/

10. XML Metadata Interchange, [Electronic resource] – Access mode http://www.omg.org/spec/XMI/

Information about authors:

Chebanyuk Olena Viktorivna – Associate Professor of Software Engineering Department of the National Aviation University. Scientific interests: Model-Driven Architecture, Model-Driven Development, Software architecture, Software development.

E-mail: chebanyuk.elena@ithea.org

Povaliaiev Dmytro – student of Software Engineering Department of the National Aviation University, Kyiv, Ukraine. Scientific interests: Software Architecture, Software Development, Model-Driven Architecture, Model-Driven Development.

E-mail: dmytro.povaliaiev@gmail.com