Proceedings of the Seventh International Conference on Research in Intelligent and Computing in Engineering pp. 75–81 DOI: 10.15439/2022R27 ISSN 2300-5963 ACSIS, Vol. 33

Automated Code Generation from Use cases and the Domain Model

1st Minh-Hue Chu
Hung Yen University of Technology and Education, Vietnam
Email: huectm@gmail.com

2nd Anh-Hien Dao
Hung Yen University of Technology and Education, Vietnam
Email: hienda@gmail.com

***Abstract*—In this paper, we propose a method to automatically generate source code files from a use case model and a domain class diagram named USLSCG (Use case Specification Language (USL) based Code Generation). In our method, a use case scenario is precisely specified by a USL model. The USL model and the domain class diagram then are used as inputs to generate source code files automatically. These source code files include classes following three-layer applications and a SQL script file to create a database and store procedures.**

***Index Terms*—Generate source code, USL, Use case, USLSCG**

I. Introduction

The software development life cycle is divided into some main stages. In the first stage, software requirements are documented in the SRS (Software Requirement Specification) document. These requirements are usually documented by UML (Unified Modeling Language) models and statements in the natural language. In the second state, design documents then are built from the SRS document. Models in design documents present different design views, for example, database designs, architecture designs, object designs, user interface designs, etc. Next state, the design models are implemented into the code source. Finally, the testing activity is performed to ensure the quality of software products [1]. The input of design and test stages are the software requirements in the SRS document that are usually documented by use case diagrams and textual use case descriptions in the template-based natural language [2]. Design models are then input for programmers to transform into source code files. These activities are usually performed manually by developers. Firstly, They will read software requirement specification documents which are typically several hundred pages to build analysis models, design models, and test cases. They then transform design models into source code files. However, in software development, requirements usually change during development. So, when the software requirements change, analysis models, design models, source code, and test cases must be rebuilt.

To reduce the time and cost of software development, automation solutions are proposed and developed. A major challenge for automation in software development is software requirements described in the natural language and modeled by models that are not precise enough. proposed a model To deal with this challenge, [3] proposed a DSML (Domain-Specific Modeling Language) named USL (Use case Specific Language) to precisely specify textual use case descriptions for automation aims in software development. In the research, we discussed abilities to generate analysis, design models, and test cases automatically from USL models. In the previous research [4], we also proposed a method named USLTG to generate test cases from the USL models. In another research [5], we also proposed a method named USLCG. The USLCG method allows generating design class diagrams automatically from USL models and the domain class diagram. In this paper, we focus on generating source code files automatically from USL models and the domain class diagram. These source code classes conform to design class diagrams generated in the previous research [5]. We named this method USLSCG. Firstly, functional requirements are captured by UML use case diagrams and USL models which specify use case descriptions precisely. In addition, the Entities of the system are captured by a domain class diagram in UML. Secondly, for each use case, USLSCG transforms the corresponding inputs above into source code classes of the use case. Besides, we also generate automatically a SQL script file containing T-SQL statements for creating a relational database and store procedures of the database.

To summarize, the main contributions of this paper are:

- the USLSCG method to generate automatically source code files from use cases and the domain class diagram;
- a set of rules to map action types into source code classes and methods of classes.
- algorithms to transform use cases into source code files
- a generator to realize the USLSCG method.

The rest of this paper is organized as follows. Section II introduces the background and motivation for developing USLSCG. Section III shows our proposed approach. Section IV explains how to generate source code files of the USLSCG method. Section V briefly discusses the tool support of USLSCG. Section VI discusses related works. The paper is closed with a conclusion and future works.

II. Background and Motivation

In this section, we first discuss the basic knowledge that we use in this research. We then present our motivations.

a) Use cases: A use case describes a sequence of interactions between a system and an external actor that results in the actor being able to achieve some outcome of value [6]. In the SRS documents, use cases are commonly used for capturing and structuring the functional requirements of software systems. Use cases are widely modeled by UML use case diagrams and each use case is loosely structured by textual descriptions following the structure as in [2]. Use case models are central models in software development. These models will be used as inputs to build different software artifacts including activity diagrams, class diagrams, sequence diagrams, source code, functional test cases, and so on. For example, Figure 1 shows a simplified use case model of an ATM system, Table 1 shows a specification of the use case Withdraw describing event flows of this use case. In this paper, we used the Withdraw use case for illustrative examples.

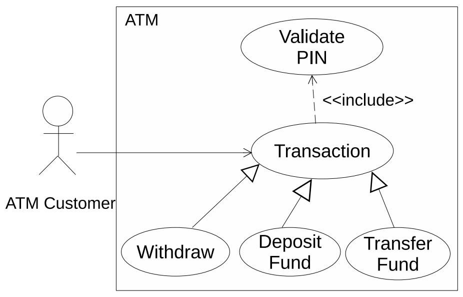


Fig. 1. A simplified use case model of the *ATM* system.

TABLE I
A TEMPLATE-BASED DESCRIPTION OF THE *Withdraw* USE CASE

Use case name: *Withdraw* **Brief description:** The customer withdraw cash. **Primary actors:** Customer **Precondition:** The Insert Card use-case was success. **Postcondition:** If the use-case was successful, the system updates the balance, dispenses the cash, prints a receipt for the user. If not, the system displays an error message. **Trigger:** User selects the Withdraw function. **Special requirement:** There is no special requirement.
Basic flow
1. The customer enters the withdrawal amount. 2. The system validates that the ATM has enough funds in the user account. If the user account has not enough funds go to step 2a.1. 3. The system generate the withdrawal transaction information. 4. The system sends the withdrawal transaction information to the Bank system. 5. The Bank system gets the withdrawal transaction information. 6. The Bank system sends the withdrawal transaction approval to the ATM system. 7. The system gets the withdrawal transaction approval from the Bank system. If the bank do not approve the withdrawal transaction, then go to step 7a.1. 8. The system updates the balance of the user account; The system dispenses the cash in the cash dispenser. 9. The customer gets the cash from the cash dispenser. 10. The system records the withdrawal transaction information. 11. The withdrawal transaction ends.
Alternate flows
2a. If the user account has not enough money. 1. The system displays an error message on the customer console and go to step 1. 7a. The bank do not approve the withdrawal transaction. 1. The system displays an error message on the customer console. 2. The system records the withdrawal transaction information. 3. The withdrawal transaction end.

b) Domain model: A domain model captures entities in a system. In software development, this model is usually specified by a UML class diagram including three types of elements: (1) domain conceptual classes, (2) attributes, and (3) relationships among conceptual classes. Domain conceptual classes represent objects used in the system use cases. Figure 2 shows the ATM domain model is picked from the work [7]. shows the ATM domain model in UML. In this research, a domain model in the UML class diagram is called a domain class diagram.

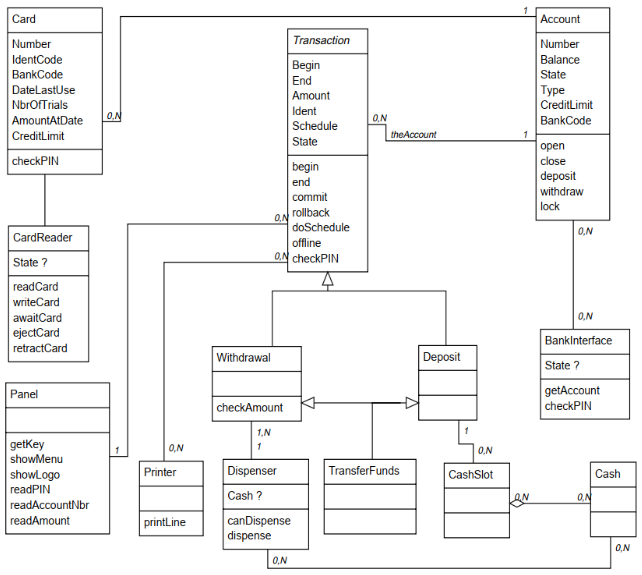


Fig. 2. The domain class diagram of the *ATM* system.

c) Three-layer architecture: Layers indicate the logical separation of components. A Layered architecture concentrates on grouping related functionality within an application into distinct layers that are stacked vertically on top of each other. Each layer has unique namespaces and classes. In the three-layer architecture, there are three layers. The first layer is the presentation layer where users can interact with the application. The second layer is the business logic layer. This layer is the middle layer - the heart of the application. It contains all the business logic of the application and describes how business objects interact with each other, where the presentation layer and data access layer can indirectly communicate with each other. The third layer is the data access layer. This layer enforces rules regarding accessing data, providing simplified access to data stored in persistent storage, such as SQL Server, and MySQL. It is noteworthy that this layer only focuses on data access instead of data storage. Besides, we have an extra layer called the business objects layer. This layer contains objects that are used in the application and common helper functions (without logic) used for all layers. In the three-layer architecture, the business objects layer is optional. However, as we follow the OOP, we should reduce the duplicate codes as much as possible. Therefore, using this layer to keep common codes instead of holding them in each layer is essential. In this article, we choose three-layer architecture to build template files for our solution

In software development, the source code of an application is transformed by programmers from design models manually. However, when software requirements are changed late, the design and source code must be rebuilt. To reduce efforts in software development, design models and source code need to be generated from software requirements automatically. A motivating question is how source code can be generated automatically from a use case what generated source code follows the design class models? This solution helps to semi-automate the implementation process. In order to automatically transform use cases and the domain model into source code files, we need to address the following main challenges.

- How do generate the source code of classes from a use case specification? Here, each generated class belongs to one of three layers (Presentation, Business Logic, Data Access). Besides, the domain class diagram is transformed into classes in the business objects layer.
- How are methods of source code classes defined? Operations of classes are defined based on the message passing among objects, i.e, objects collaborate together to realize a use case. We also define the parameters of the generated methods.

III. Overview of our approach

Our USLSCG approach is illustrated in Figure 3. This approach uses USL to specify each use case description by a USL model conforming to the USL metamodel as shown in Figure 4. We take a USL model, a UML class model

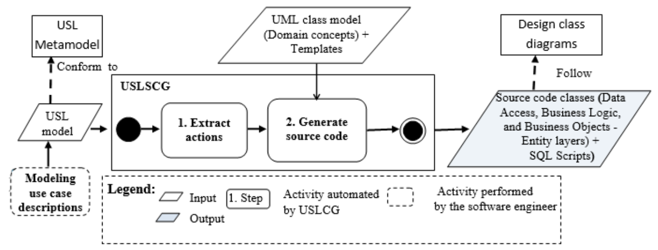

USL Metamodel
Conform to
USL model
Modeling use case descriptions
UML class model (Domain concepts) + Templates
USLSCG
1. Extract actions
2. Generate source code
Design class diagrams
Follow
Source code classes (Data Access, Business Logic, and Business Objects - Entity layers) + SQL Scripts)
Legend:
Input
Output
1. Step
Activity automated by USLCG
Activity performed by the software engineer

Fig. 3. Overview of the USLSCG Approach.

capturing domain concepts of the system, and source code templates as inputs to generate source code automatically. In SubSection III-A, we briefly explain the language USL. Our USLSCG method includes two main steps: Step 1 aims to extract actions with constraints from a USL model; Step 2 takes inputs including extracted actions, domain classes, and source code templates aim to generate source code classes and a SQL Script file. A detailed explanation of these steps is presented in Section IV.

A. Capturing Use Case descriptions in USL models

The USL language is a DSML to specify use cases precisely. This language was introduced in other research [3]. The USL is defined based on the metamodeling technique. The metamodel of the USL was determined as in Figure 4. The USL approach aims to specify use case descriptions as USL models that could be automatically transformed into other software artifacts, including analysis, design models, source code, and test cases. The following explains how to specify a use case description in natural language by a USL model:

- The use-case-overview field is specified by the *DescriptionInfo* object properties.
- Steps of the basic flow are specified by *FlowStep*s, that are connected by *BasicFlowEdge*s and *ControlNode*s. These steps are either *ActorStep*s or *SystemStep*s.
- Steps of each alternate flow are specified by *FlowStep*s that are linked by *ControlNode*s and *AlternateFlowEdge*s.
- An *ActorStep* can include one or more *ActorRequest*s and *ActorInput*s.
- A *SystemStep* can include one or more *SystemInput*s, *SystemDisplay*s, *SystemRequest*s, *SystemState*s, *SystemOutput*s, *SystemInclude*s, and *SystemExtend*s.
- Use case constraints, guard conditions, and actions in *SystemStep*s or *ActorStep*s are captured by *Constraint*s associated with *InitialNode*, *FinalNode*, *BasicFlowEdge*s, *AlternateFlowEdge*s, and *Action*s.

The USL model representing the use case *Withdraw* is shown as in Figure 5. This model captures the description fields of the use case *Withdraw* by the use-case-overview field, the basic flow by steps from *s1* to *s10*, the alternate flow *2a* by steps *s11*, the alternate flow *7a* by steps *s12* and *s13*, the actions in the steps by actions from *a1*...*a14*, the guard constraints to select between the flows by Constraints from *g1*...*g4*, and the postcondition of actions by Constraints from *p1*...*p3*. In particular, *s1*, *s5*, *s6*, *s9* are *ActorStep*s; the other *Step*s are *SystemStep*s; *a1* and *a6* are *ActorInput*s (the actor of *a1* is a person, the actor of *a6* is an external system); *a5* and *a10* (requesting object of *a10* is a device) are *ActorRequest*s; *a3* is a *SystemOperation*; *a4*, *a7*, and *a9* are *SystemRequests*; *a2*, *a8*, *a11*, *a14* are *SystemState*s; *a12* and *a13* are *SystemOutput*s.

IV. Transforming Use case into Source Code

In this section, we present about two steps of USLSCG as shown in Figure. 3 and explain how USLSCG can automatically generate source code files from a USL model and a UML domain class diagram.

A. Extraction of Actions

In the first step of USLSCG as in previous research, we develop an algorithm named *ExtActions* to extract a set of actions from a USL model, as shown in Algorithm 1. Algorithm *ExtActions* takes *D*, a USL model, as input. The output of this Algorithm is a set of Actions.

Example IV.1. Algorithm *ExtActions* will return thirteen Actions (*a1* ...*a13*) from the use case *Withdraw*. The extracted actions belong to one of nine action types as in the part (d) of Figure. 4.

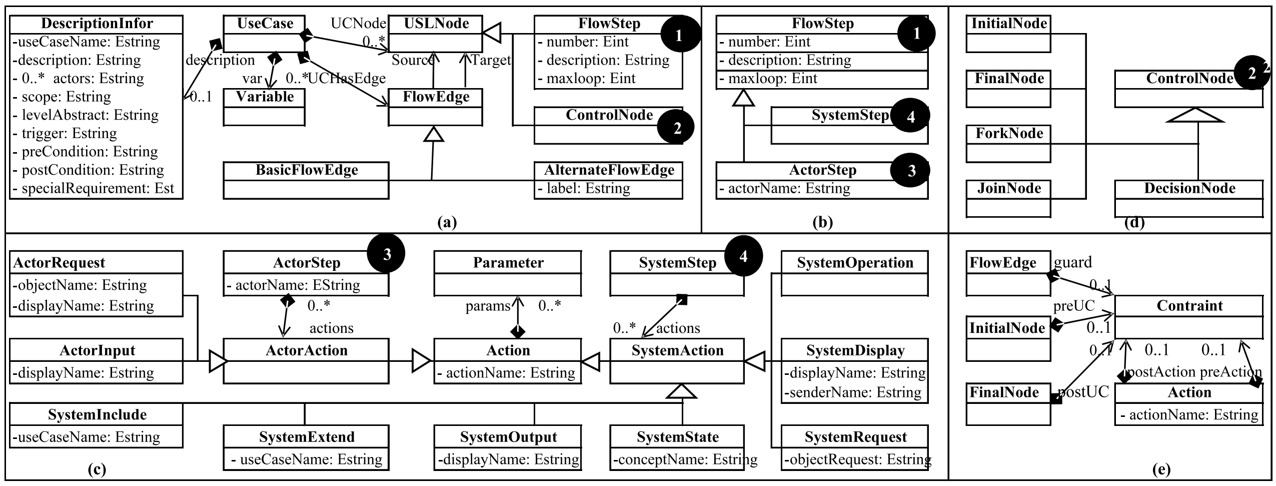


Fig. 4. The USL metamodel.

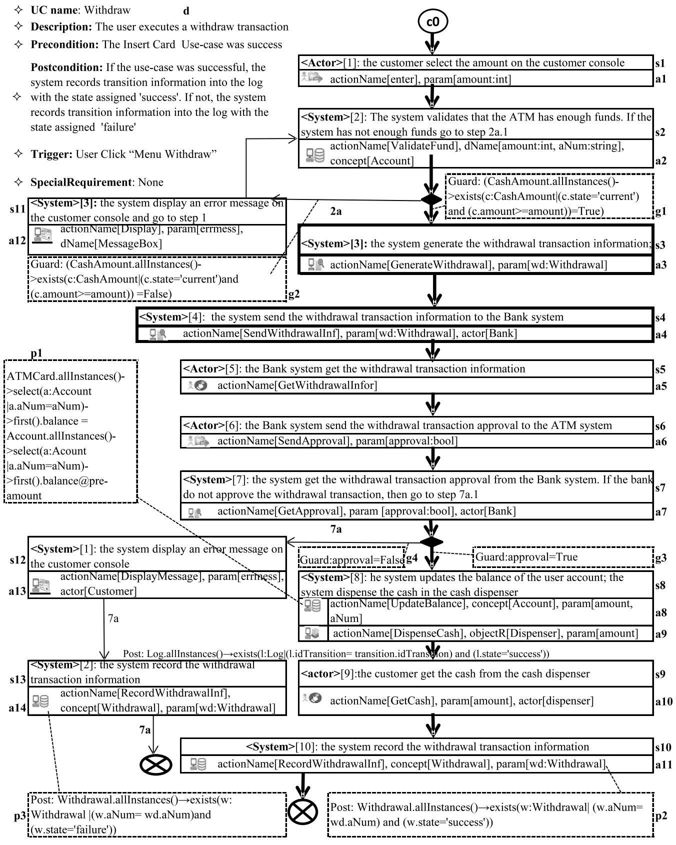

UC name: Withdraw
d
Description: The user executes a withdraw transaction
Precondition: The Insert Card Use-case was success
Postcondition: If the use-case was successful, the system records transition information into the log with the state assigned 'success'. If not, the system records transition information into the log with the state assigned 'failure'
Trigger: User Click "Menu Withdraw"
SpecialRequirement: None
c0
<Actor>[1]: the customer select the amount on the customer console
actionName[enter], param[amount:int]
s1
a1
<System>[2]: The system validates that the ATM has enough funds. If the system has not enough funds go to step 2a.1
actionName[ValidateFund], dName[amount:int, aNum:string], concept[Account]
s2
a2
Guard: (CashAmount.allInstances()->exists(c:CashAmount|(c.state='current') and (c.amount>=amount))=True)
g1
2a
<System>[3]: the system display an error message on the customer console and go to step 1
actionName[Display], param[errmess], dName[MessageBox]
s11
a12
Guard: (CashAmount.allInstances()->exists(c:CashAmount|(c.state='current')and (c.amount>=amount)) =False)
g2
<System>[3]: the system generate the withdrawal transaction information
actionName[GenerateWithdrawal], param[wd:Withdrawal]
s3
a3
<System>[4]: the system send the withdrawal transaction information to the Bank system
actionName[SendWithdrawalInf], param[wd:Withdrawal], actor[Bank]
s4
a4
p1
ATMCard.allInstances()->select(a:Account |a.aNum=aNum)->first().balance = Account.allInstances()->select(a:Acount |a.aNum=aNum)->first().balance@pre-amount
<Actor>[5]: the Bank system get the withdrawal transaction information
actionName[GetWithdrawalInfor]
s5
a5
<Actor>[6]: the Bank system send the withdrawal transaction approval to the ATM system
actionName[SendApproval], param[approval:bool]
s6
a6
<System>[7]: the system get the withdrawal transaction approval from the Bank system. If the bank do not approve the withdrawal transaction, then go to step 7a.1
actionName[GetApproval], param [approval:bool], actor[Bank]
s7
a7
7a
<System>[1]: the system display an error message on the customer console
actionName[DisplayMessage], param[errmess], actor[Customer]
s12
a13
Guard:approval=False
g4
Guard:approval=True
g3
<System>[8]: he system updates the balance of the user account; the system dispense the cash in the cash dispenser
actionName[UpdateBalance], concept[Account], param[amount, aNum]
actionName[DispenseCash], objectR[Dispenser], param[amount]
s8
a8
a9
7a
Post: Log.allInstances()→exists(l:Log|(l.idTransition= transition.idTransition) and (l.state='success'))
<System>[2]: the system record the withdrawal transaction information
actionName[RecordWithdrawalInf], concept[Withdrawal], param[wd:Withdrawal]
s13
a14
<actor>[9]:the customer get the cash from the cash dispenser
actionName[GetCash], param[amount], actor[dispenser]
s9
a10
7a
<System>[10]: the system record the withdrawal transaction information
actionName[RecordWithdrawalInf], concept[Withdrawal], param[wd:Withdrawal]
s10
a11
Post: Withdrawal.allInstances()→exists(w: Withdrawal |(w.aNum= wd.aNum)and (w.state='failure'))
p3
Post: Withdrawal.allInstances()→exists(w:Withdrawal| (w.aNum= wd.aNum) and (w.state='success'))
p2

Fig. 5. The USL model specifies the use case *Withdraw*.

Algorithm 1: EXTACTIONS

```
1  ExtActions(D)
   Input: D is a USL model
   Output: la is a set of actions extracted from D
2  BEGIN
3      la ← ∅;
4      foreach s in D.USLNodes do
5          if s is SystemStep then
6              foreach a in ((SystemStep)s).SystemActions do
7                  la ← la ∪ a;
8          if s is ActorStep then
9              foreach a in ((ActorStep)s).ActorActions do
10                 la ← la ∪ a;
11 END
```

transformation language M2T of the Eclipse framework [8]. Acceleo can automatically generate source code files by using templates such as illustrated Figure 6.

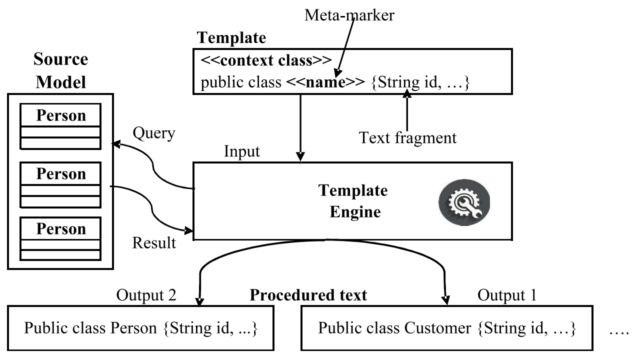


Fig. 6. Templates are used as inputs for the Accleo Project to automatically generate source code files[9].

B. Generating automatic source code files

In order to generate source code files automatically, We use the Acceleo project. Specifically, Acceleo is a model

Firstly, USLSCG uses a *Business Object template file*, a *SQL Script template file* as inputs for the Acceleo project to generate source code classes at the business objects layer, and a SQL script file automatically. Specifically, each class in the UML domain class diagram is mapped to a business object class, and each attribute of a UML class or association relationship between two UML classes is transformed into a property of the code class. Each generalization relationship is mapped into the inheritance relationship between the child and parent class. In addition, the generated SQL script file is a file script including statements to create a database, tables, and store procedures.

Secondly, USLSCG uses *Data Access template files* as inputs for the Acceleo project to generate classes automatically at the data access layer. In particular, each class in the UML domain class diagram is mapped to a code interface and class at the data access layer. To use in different situations and use cases, classes at the data access layer can be generated full of methods. We use a *Data Access frame template* shown as List 1 to generate classes at the data access layers. In addition, each method has one template to replace *DataAccess* part of the frame template, for example, List 2 is a template of the Delete method.

```
using System;
using System.Data;
using System.Collections.Generic;
using Example.Common;
using {{ProjectName}}.DataModel;
using System.Linq;
namespace {{ProjectName}}.DataAccess {
    public partial class {{tableName}}Repository : I{{tableName}}Repository
  {
    private IDatabaseHelper _dbHelper;
    public {{tableName}}Repository(IDatabaseHelper dbHelper)
    {
      _dbHelper = dbHelper;
    }
    {{DataAccess}}
    }
}
```

Listing 1. DataAccess Template

```

/// <summary>
/// Delete records in the table {{tableName}}
/// </summary>
/// <param name="json_list_id">List id want to delete</param>
/// <param name="updated_by">User made the deletion</param>
/// <returns></returns>
public List<{{tableName}}Model> Delete(string json_list_id,Guid updated_by)
    {
            string msgError = "";
            try
      {
        var dt = _dbHelper.ExecuteSProcedureReturnDataTable(out msgError, "sp_{{tablename}}_delete_multi",
                "p_json_list_id", json_list_id,
                "p_updated_by", updated_by);
            if (!string.IsNullOrEmpty(msgError))
        {
                throw new Exception(msgError);
            }
            return dt.ConvertTo<{{tableName}}Model>().ToList();
        }
      catch (Exception ex)
      {
                throw ex;
            }
        }
```

Listing 2. Template of Delete method in the Data Access class

Finally, *Business Logic* and *Presentation template files* are used as inputs for USLSCG to generate classes at the business logic and presentation layer automatically. We develop Algorithm 2 to create these classes automatically. Note that the article only generates the source code class at the presentation layer and does not discuss the interface design based on the domain classes because this problem has been handled very well by editors supporting the programming languages.

Algorithm 2: GENBP

```
GenBP(la, tems)
Input: la, a set of actions;
tems are template files;
Output: lcbps, classes at business and presentation layer
BEGIN
Create a business class following templates corresponding to the use case
  named BC;
foreach a in la do
    switch $TypeOf(a)$ do
        case $ActorInput$ do
            if Business(a.ActorName) don't exit then
                Create a presentation class corresponding to
                  a.ActorName;
            CreatePresentationMethod(a);
        case $ActorRequest$ do
            if Presentation(a.ActorName) don't exist then
                Create a presentation class corresponding to
                  a.ActorName;
            if a.RequestObjectType is system then
                CreatePresentationMethod(a);
        case
          $SystemOperation$ or $SystemExtend$ or $SystemRequest$
          do
            CreateBusinessMethod(a);
        case $SystemDisPlay$ or $SystemOutput$ do
            CreatePresentationMethod(a);
        case $SystemState$ do
            CreateBusinessMethod(a);
END
```

Algorithm *GenBP* takes the set of constrained actions *la* which are extracted from Algorithm *GenActions* as input. The output of *GenBP* is classes conforming to class diagram *ACD* in the research [5].

Firstly, Algorithm *GenBP* generates a business class called *BC* and a presentation class called *PC* for each USL model. Nextly, Algorithm *GenBP* travers actions in *la* and transforms each action to corresponding methods of the *BC* or PC. Algorithm *GenBP* employs the functions Presentation(a.ActorName), CreatePresentationMethod(a), CreateBusinessMethod(a) that are explained below.

- The function *Presentation (a.ActorName)* checks whether a presentation class corresponding to the actor of action *a* exists. this function will return true if this class exists.
- The function *CreatePresentationMethod(a)* generates a method for the presentation class *PC* corresponding to

action *a*. The generated method is based on properties *ActionName* and *Parameters* of action *a*.

- The function *CreateBusinessMethod(a)* generates a method for the business class *BC*. The generated method is based on the properties *ActionName* and *Parameters* of action *a*.

Specifically, methods CreatePresentationMethod(a), CreateBusinessMethod(a) will use a template corresponding to the type of action *a*. For example, List 3 shows a template corresponding to the action *SystemOperation* mapped to the update method. List 4 shows the generated business class of use case Withdrawal.

```
using {{ProjectName}}.Common;
using {{ProjectName}}.Common.Caching;
using {{ProjectName}}.DataAccess
namespace {{ProjectName}}.Business {
    public partial class {{tableName}}Business : I{{
    tableName}}Business {
        private I{{tableName}}Repository _res;
    private ICacheProvider _redis;

        public {{tableName}}Business(I{{tableName}}
    Repository {{tableName}}Res, ICacheProvider
    redis)
    {
            _res = {{tableName}}Res;
      _redis = redis;
        }

    {{BusinessLogic}}
    }
}
```

```
/// <summary>
/// Update information in the table{{tableName}}
/// </summary>
/// <param name="model">the record updated</param>
/// <returns></returns>
 public bool Update({{tableName}} model)
    {
            return _res.Update(model);
        }
```

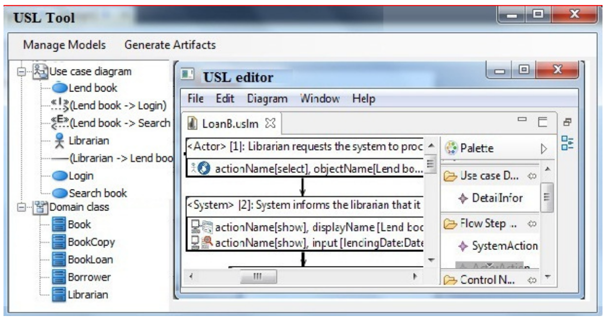
Listing 3. Update method template of business class

```
using System;
using System.Collections.Generic;
using ATM.BusinessObject;
using ATM.DataAccess;

namespace ATM.BusinessLogic
{
    public class WithdrawalBusiness :
    IWithdrawalBusiness
    {
        private IWithdrawalRepository _res;
        public EmployeeBusiness(
    IWithdrawalRepository res)
        {
            _res = res;
        }

        /// <summary>
        /// Validate the record Withdrawal
        /// </summary>
        /// <param name="model">the record validated
     </param>
        /// <returns></returns>
        public bool ValidateFund(Withdrawal model)
        {
            return _res.Validate(model);
        }

        /// <summary>
        /// Create a new record Withdraw
        /// </summary>
        /// <param name="model">The record is
    recorded </param>
        /// <returns></returns>
        public bool GenerateWithdraw(Withdrawal
    model)
        {
            if (model.Withdrawal_id == null || model
    .Withdrawal_id == Guid.Empty.ToString())
            { var c_guid = Guid.NewGuid().ToString()
    ; model.Withdrawal_id = c_guid; }
            return _res.Create(model);
        }

        //...

    }
}
```

Listing 4. The business class is generated for use case Withdraw

V. Tool Support

We have added a code generation feature named USLSCG for the USL tool in the research [3] as depicted in Figure 7. The USL tool allows the integration of use cases into model-driven software engineering (MDSE). Firstly, this tool reads use case diagrams and a UML domain class diagram. Secondly, we specify each use case description as a USL model by the USL Editor tool. Finally, the code generator USLSCG reads the source code template files and transforms the UML domain class diagram into source code classes at the business objects layer, data access layer, and a SQL script file and it also transforms each use case into source code classes in presentation and business layer by using model transformation languages Model to Text (M2T).



<Actor> [1]: Librarian requests the system to proc
actionName[select], objectName[Lend bo...
<System> [2]: System informs the librarian that it
actionName[show], displayName [Lend boc
actionName[show], input [lendingDate:Date

Fig. 7. The support tool USL contains the source code generation feature.

The architecture of the USLSCG generator is shown as in Figure 8. This tool takes as input a USL model and a UML domain class diagram and source code templates. The output of this tool is classes at presentation, business logic, data access, business object layers; a SQL script file to create Database and store procedures. In order to build the USLSCG generator, we implement two modules, Module (1) and Module (2) as shown

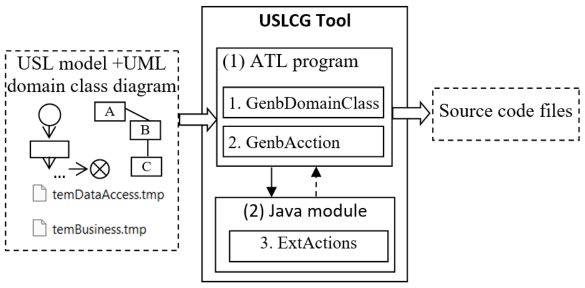


Fig. 8. The architecture of the generator USLSCG.

in Figure. 8. Module (1) is implemented by an Acceleo project. Module (1) aims to parse the USL model in Java and Its output is taken as input for Module (2). Module (2) extracts actions from the USL model and returns a list of actions for Module (1). Finally, Module (1) transforms the inputs into source code files by transformations in Acceleo. Firstly, Module (1) reads the UML domain class diagram and template files as inputs for Module (1) functions. Function 1.GenbDomainClass gets inputs including the UML domain class diagram, a SQL script template file, an object template file, and data access template files to generate a SQL script file, classes in the business objects layer, and classes at the data access layer, respectively. Next step, Module (1) read each USL model to pass Module (2) written in java to extract actions from the USL model and return a list of actions for Module (1). Function 2.GenbAcction reads presentation template files, business template files and actions returned from Module (2) to generate classes at the presentation and business logic layer.

VI. Related work

We position our work in the automatic generation of source code classes from use cases and a UML domain class diagram. Within this context, source code classes are often manually built from design class diagrams of use cases. In order to automatically generate source code classes from a use case model to reduce cost when software requirements are changed, several approaches [10], [11], [12], [13] have been proposed.

Sunitha *el al.* [11] proposed a methodology to automatically code generate from state chart diagrams. This paper presents a method to convert hierarchical states, concurrent and history states to Java code with a design pattern-based approach. Particularly, each state of the system (or object) will be mapped to a source code class. However, in the object-oriented approach, each object type is abstracted into a class, and each state is usually an obtainable value of a class property.

Francu *el al.* [10] presented a method that allows generating an implementation of a system from the use cases written in a natural language. This paper did not show the results of the generator for Entity Managers which are used services for classes of the above layers. Additionally, this paper does not handle one-to-many or many-to-many associations among the classes in the domain model. Besides, extracted verbs do not classify so arguments of the procedures do not be generated precisely.

Fatolahi *el al.* [12] proposed a semi-automated method for the generation of web-based applications from use cases. However, this approach is not fully automatic, they need to interact with the developer to obtain the appropriate value of required user parameters. Compared with all the works above, our approach allows generating classes containing full parameters following three layers architecture. Besides, a script file is also generated which creates the database and its store procedures. This file is not created in all of the above research.

VII. Conclusion

In this work, we proposed an automatic method for generating source code files from a use case specification represented as a USL model, a domain class diagram, and template files. Generated source code files contain classes of the application, a SQL script file to create the database, and store procedures for the application. We also developed the generator USLSCG to realize our method. In the future work, we will improve the generator USLSCG to generate various kinds of applications. Furthermore, generated source code files in different programming languages will also be supported in our next version of the generator USLSCG.

References

[1] I. Sommerville, *Software Engineering*, 10th ed. Boston: Pearson, Mar. 2015.

[2] A. Cockburn and H. a. Technology, "WRITING EFFECTIVE USE CASES," *Addison-Wesley*, p. 113, 2001.

[3] C. Hue and D.-H. Dang, "USL: A Domain-Specific Language for Precise Specification of Use Cases and Its Transformations," *Informatica*, vol. 42, Sep. 2018.

[4] C. Hue, D.-H. Dang, and N. Binh, *A Transformation-Based Method for Test Case Automatic Generation from Use Cases*, Nov. 2018, pages: 257.

[5] M.-H. Chu and D.-H. Dang, "Automatic Extraction of Analysis Class Diagrams from Use Cases," in *2020 12th International Conference on Knowledge and Systems Engineering (KSE)*, Nov. 2020, pp. 109–114, iSSN: 2164-2508.

[6] K. Wiegers and J. Beatty, *Software Requirements, 3rd Edition*, ser. 3rd Edition, Aug. 2013.

[7] G. Ksters, H.-w. Six, and M. Winter, "Validation and verification of use cases and class models," 04 2001.

[8] Laurent Goubet and Laurent Delaigue, "Acceleo/Getting Started - Eclipsepedia." [Online]. Available: https:// wiki.eclipse.org/Acceleo

[9] a. M. W. Marco Brambilla, Jordi Cabot, *Model-Driven Software Engineering in Practice*, 2nd ed. Morgan & Claypool, 2017.

[10] J. Francu and P. Hnetynka, "Automated Code Generation from System Requirements in Natural Language," *e-Informatica*, vol. 3, pp. 72–88, Jan. 2009.

[11] S. E. V. and P. Samuel, "Automatic Code Generation From UML State Chart Diagrams," *IEEE Access*, vol. 7, pp. 8591 – 8608, Jan. 2019.

[12] A. Fatolahi, S. S. Somé, and T. C. Lethbridge, "Towards A Semi-Automated Model-Driven Method for the Generation of Web-based Applications from Use Cases," 2008.

[13] H. Ikeda, H. Nakagawa, and T. Tsuchiya, "Towards Automatic Facility Layout Design Using Reinforcement Learning," Sep. 2022, pp. 11–20. [Online]. Available: https://annals-csis.org/proceedings/2022/drp/25.html