EPiC Series in Computing

Volume 69, 2020, Pages 98–111

Proceedings of 35th International Conference on Computers and Their Applications

A Bimodal Approach for the Discovery of a View of the Implementation Platform of Legacy Object-Oriented Systems under Modernization Process

Hamza Abdelmalek[1,3], Gino Chénard[2,] Ismaïl Khriss[1], and Abdeslam Jakimi[3]

[1] Département de mathématiques, d'informatique et de génie, UQAR, Rimouski, Québec, Canada

[2] Cégep de Rimouski, Rimouski, Québec, Canada

[3] Faculty of Sciences and Technics, UMI, Errachidia, Morocco

Hamza.Abdelmalek@uqar.ca, ginochenard@gmail.com, ismail_khriss@uqar.ca, ajakimi@yahoo.fr

Abstract

Organizations are highly dependent on their software in carrying out their daily activities. Unfortunately, the repeated changes that are applied to these systems make their evolution difficult. This evolution may be necessary to maintain the software, replace or upgrade it. In the case of complex and poorly documented legacy systems, modernization is the only feasible solution to achieving the evolution goals. The OMG (Object Management Group) consortium created the Architecture-Driven Modernization (ADM) initiative to cope with the challenges of modernization. This initiative proposes, among other things, modernization through model-driven engineering (MDE). In this context, the modernization of a legacy system, not developed in an MDE environment, begins with its migration towards this type of environment. This migration raises the problem of finding the models necessary for the use of MDE representing this system.

In this paper, we present a new bimodal approach to ADM modernization by enabling automatic and interactive modes to discover a view of the implementation platform of a legacy object-oriented system. Also, we present the key ideas of the algorithms behind this discovery process. Finally, we describe our prototype tool that implements our approach. This tool has been validated on several systems written in C# and Java languages.

1 Introduction

Organizations are heavily dependent on software in their daily activities. Unfortunately, the repeated changes that are applied to these systems make evolution difficult. This evolution may be necessary to maintain the software, to replace it, or to modernize it [1]. We are talking about a legacy system in the case of one where it has become obsolete in terms of architecture or platform and whose evolution can no longer meet the objectives of evolution [2]. In these cases, modernization is the only feasible solution to achieve these goals. Indeed, the purpose of modernization is to evolve a system when conventional practices no longer reach the desired goal [3]. It is, therefore, seeking to modify these systems to improve their maintainability.

Although modernization seems an ideal solution, it is a complicated task. In particular, the forecast for risks and costs is tricky. The OMG (Object Management Group) consortium created the Architecture-Driven Modernization (ADM) initiative to address this problem. The main goal of this initiative is to provide a set of standards to simplify the interoperability of modernization tools [4]. ADM proposes to carry out modernization using Model-Driven Engineering (MDE). MDE is a software development approach based on the representation by models of different aspects of the system to be developed and the use of model transformations including source code generation [5].

In the context of ADM, the modernization of a legacy system developed in a non-MDE environment begins with its migration to this type of environment. This migration poses the challenge of discovering the necessary models for MDE representing it. The discovery of low-level models is not a significant problem. Among these models in the context of MDA (Model Driven Architecture), an implementation of MDE proposed by OMG [6], we find the Platform Specific Model (PSM). This model faithfully represents the current implementation of the system. It contains both the elements of the implementation platform and the business domain. But the discovery of models with a higher level of abstraction is more difficult to achieve, whether automatically or manually. Among these models, there is the Platform Independent Model or PIM. This model represents the system from an independent point of view of any implementation platform. It contains only elements of the system's business domain.

Another problem with this type of approach is that the MDE process lacks precision. In particular, the form that must take its models and the description and application of transformations to move from one model to another. Thus, a significant difficulty is the definition of a Platform Description Model (PDM). This high-level abstraction model defines both the concepts of an implementation platform and how to apply them to a PIM to create a PSM reflecting this PIM and PDM. A concept is the formulation of a system requirement from the point of view of the domain of the implementation platform or the business domain.

Our modernization process of a legacy system within ADM is expressed in Figure 1. It is the migration of a system from a non-model-driven environment to a model-driven environment. This process includes two stages. The first stage uses reverse engineering to discover abstract models of the legacy systems. More specifically, we have:

- a Text-to-Model (T2M) discovery to get a PSM from the source code;
- a Model-to-Model (M2M) discovery to get a PDM from a PSM;
- an M2M discovery to get a PIM from a PSM and a PDM.

The second stage of the modernization process uses traditional forward engineering to obtain the source code of the new system using M2M and Model-to-Text (M2T) transformations.

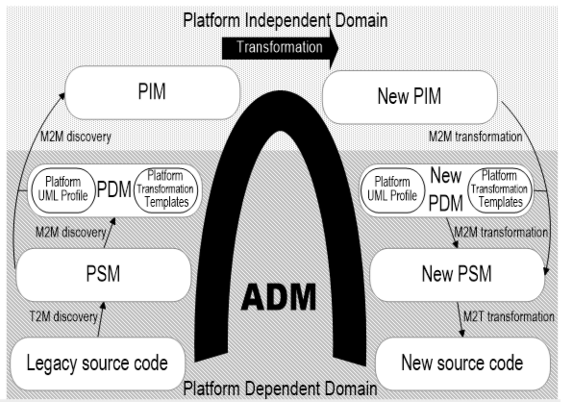

Figure 1: The modernization process within ADM

To overcome the problems associated with the PDM definition, we propose to describe a PDM in two kinds of views. The first view is a UML profile of the system's implementation platform. The second view is a set of transformation templates capturing the source code of the system's implementation platform. The most complex task to discover these models is the identification of the concepts present in the legacy systems and to distinguish those of the implementation platform from those of the business domain. To accomplish this task, we base ourselves on the hypothesis that the code linked to the platform and its vocabulary are repetitive or semi-repetitive. To identify this repetitiveness, we used four analysis techniques for discovering the first view of the PDM and the PIM of a legacy system: latent semantic indexing, clone detection, data flow analysis, and analysis of identifiers. Recall that Latent semantic indexing (LSI) is used to determine similarities between a set of textual documents [7]. One interest of this approach is to help to deal with synonymy and polysemy. The data flow analysis (DFA) examines a system to determine where and how data are defined and used [8]. DFA is used in several contexts, such as in code optimization during compilation.

We presented respectively in [9] and in [10] an earlier version of the algorithms to discover the first view and the second view of a PDM from the source code of a legacy object-oriented system. This version, implemented in a prototype tool developed within the Eclipse Environment, proposed an automatic approach for the extraction and supported only legacy systems in Java language. The validation of this initial work showed that giving the possibility to the experts in the domain or the implementation platform to interact with the tool will improve the quality of the discovered models.

In this paper, we describe a new algorithm of the UML profile discovery process. The main contributions of this paper are:

- A revised version of the UML profile discovery process, allowing a bimodal approach to the discovery process: automatic and interactive.
- A support of legacy systems developed in C# language.
- A new prototype tool supporting the bimodal approach to the model-driven modernization of legacy object-oriented systems developed in C# and Java languages.

This paper is organized as follows. Section 2 reviews existing related work. Section 3 presents the metamodel of our approach and gives an overview of the UML profile discovery process through a running example. Section 4 discusses the validation of our work and presents our prototype tool. Finally, Section 5 provides some concluding remarks.

2 Related Work

In subsection 2.1, we review existing approaches in software modernization and particularly in model-driven migration. As our approach also falls in the research area of concept identification, we discuss this area in subsection 2.2. Note that our two areas of interest intersect, as we will see with our approach where our migration process involves identifying the concepts of the implementation platform. In fact, legacy system understanding is a prerequisite activity for the legacy system's migration.

2.1 Software modernization and model-driven migration

Kesserwan et al. [13] describe an approach to reverse engineer legacy software tests to a model-driven testing methodology. The legacy test procedures are translated to the TTCN-3 (Testing and Test Control Notation) language and then abstracted to test cases in TDL (Test Description Language) format. The latter is a formal language for expressing test cases.

Modisco [14] is an extensible framework to develop model-driven tools to support the modernization of legacy systems. Its architecture is organized in three layers to enable the reuse of components between modernization solutions: use-cases, technologies and infrastructure. The first layer aims at providing tools for a specific modernization use-case. The second layer offers specific components for one legacy technology, and the last layer provides generic components independent from any legacy technology.

Sadovykh et al. [15] report a real case of modernization using ADM from a legacy C++ system to Java. The creation of their PSM is based on human experts who have to decide which part of PSM is related to the problem domain and which is related to the solution domain. They claim that it is challenging to automate this distinction because it needs input from experts in the problem and solution domains.

Zhang et al. [16] present a methodology for a generic model-driven migration strategy to port legacy systems to SOA (service-oriented architecture). Their framework discusses different questions like how to extract legacy models from legacy systems and how to develop architectures suitable for SOA.

Favre [17] presents a formal MDA-based framework for modernizing systems. She proposes to translate meta-models and meta-model transformations into the NEREUS meta-model. This meta-model allows the expression of model-to-model transformations such as refinement and refactoring.

Fleurey et al. [18] present a semi-automatic approach for model-driven migration. They automatically extract a PSM from a legacy system and use transformations to generate a PIM and a PSM in the new target architecture. These transformations are defined manually.

Chen et al. [19] present an approach that analyzes assembler source code to extract a set of models representing the software components and its architecture described in a formal language. First, a PSM is created from the source code of the legacy system. Then, decomposition techniques are used to restructure the PSM. Next, abstraction rules are applied to extract a PIM from the restructured PSM. The approach presents two limitations. First, the abstraction rules have to be defined manually. Second, the extracted PIM cannot be reused since the names of its model elements are not meaningful as the source of extraction is an assembler source code.

Doyle et al. [20] propose a migration approach from a legacy system developed in fourth-generation proprietary languages specific to a problem domain to MDA by providing transformations from the legacy meta-model to UML models. In this case, the extraction of models or abstractions is not the issue as in our approach. The challenge is to find how to convert proprietary models to MDA models.

Qiao et al. [21] discuss the need to migrate a legacy system to a model-driven environment. Their approach extends their previous work on architecture recovery by a new phase for performing the migration. This phase is a translation into UML of architectural models obtained in the SADL (Simple Architecture Description Language) language.

2.2 Concept identification

The concept identification problem is an active research area since the early 1990s [22] for legacy system understanding. A concept is defined as a formulation of a system's requirement from the problem domain or the solution domain. A feature is a concept describing a functional requirement visible and accessible to users. Recently, with the popularity of aspect-oriented programming (AOP), concepts have taken different names like concerns and aspects. A concern is an area of interest or focus in a system. Aspects are a possible implementation of concerns. There is also a notion of code smell that is any characteristic in the source code of a program that possibly indicates a deeper problem[*].

Work on the area has one of the two following objectives:

- Identifying parts of a source code that implement a given concept. With this objective, the problem is frequently named the concept assignment (or location) problem.
- Identifying concepts that a source code implements.

Several approaches have tackled the problem. They use different techniques such as querying tools, fan-in and fan-out analysis, identifier analysis, dynamic analysis, clone detection, LSI (Latent Semantic Indexing), natural language processing of source code or a combination of techniques. In the following, we discuss the closest to our work. Refer to [23] for a broader discussion of approaches for legacy system understanding.

Shahbazian et al. [24] developed a technique called RecovAr that automatically recover design decisions during the software development process from the history of its development. This history information comes from maintenance repositories such as version control and change requests. They have formally defined the notion of an architectural design decision and developed an approach for tracing such decisions in the history of a software development project. They also classify whether decisions are architectural and map these decisions to source code artifacts. Their work is based on Architecture Recovery, Change, And Decay Evaluator (ARCADE) [25], a software workbench that proposes a set of architecture-recovery techniques, a catalog of architectural smell definitions, and a set of metrics for measuring different aspects of architectural change and decay. Note that ARCADE is also used by Langhammer et al. to extract the software's architecture, behavior, and usage models from the source code and test cases [26].

Van Der Spek et al. [27] propose an LSI-based approach to recover the architectural concept by clustering variable names based on their similarity. Like our approach, class operations are used as contexts for the LSI matrix. Every cluster represents an architectural concept.

Boer and Vliet [28] show in a proof of concept a similar approach to discovering architectural concepts. It is also based on LSI, but it requires manual intervention during the discovery process.

Greevy and Ducasse [29] search for infrastructure classes in a system's source code. Infrastructure classes implement low-level concepts, and they define them as involved in at least half of the features offered by a system. By analyzing execution traces, they detect an infrastructure class if it is present in at least half of the sets of traces. Clearly, many infrastructure classes will never be identified as such by this approach.

[*] https://en.wikipedia.org/wiki/Code_smell (2019-11-29)

3 Description of the approach

To illustrate our approach, we use a simple customer management system in C# language. Its PSM is described in Figure 2.

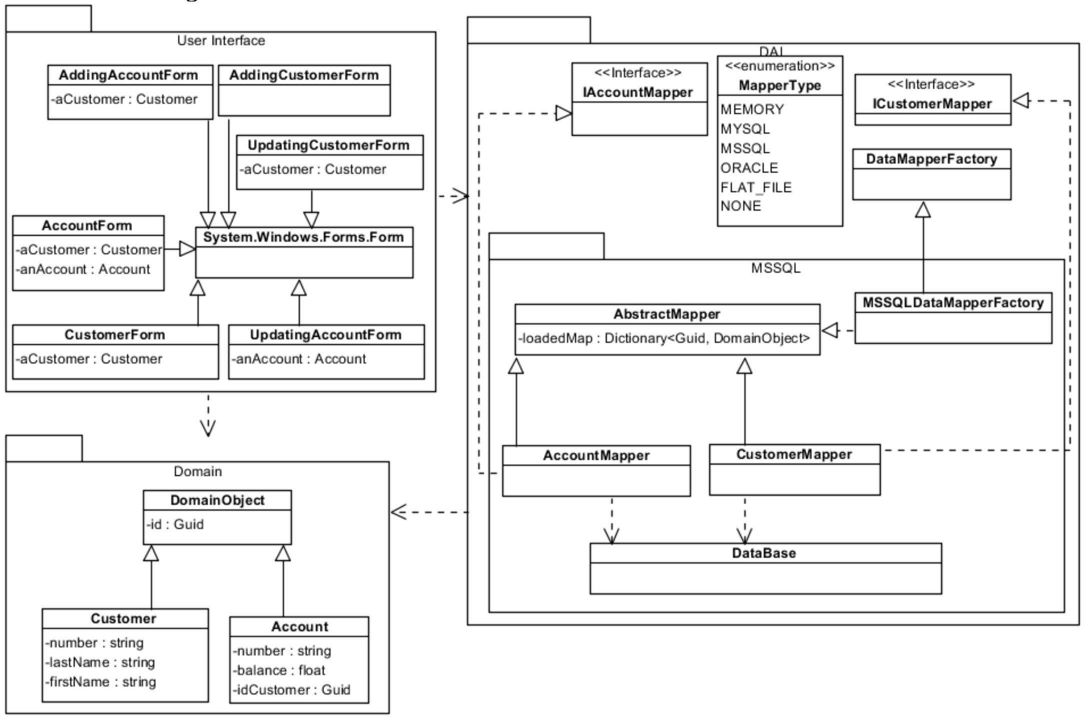

Figure 2: PSM of a simple customer management system[†]

The system implements an in-house software framework that uses a multilayered architecture: user interface, domain logic, and data access. The user interface layer contains six form classes: `AccountForm`, `AddingAccountForm`, `AddingCustomerForm`, `CustomerForm`, `UpdatingAccountForm` and `UpdatingCustomerForm`. All those forms inherit from .Net class `Systems.Windows.Forms.Form`. The domain logic captures the real-world business rules and contains two domain classes `Account` and `Customer`. They both inherit from the class `DomainObject` which is the result of the application of the *Identity field* enterprise pattern. The latter saves a database ID field in an object to maintain identity between an in-memory object and a database row [11]. The data access layer `(DAL)` simplifies the access to data stored in databases and implements a set of patterns such as *Data Mapper* [11] and *Abstract Factory* [12]. The `DAL` contains two interfaces `IAccountMapper` and `ICustomerMapper`, the enumeration `MapperType`, the class `DataMapperFactory`, and the subpackage `MSSQL`. The latter contains five classes: `AbstractMapper`, `AccountMapper`, `CustomerMapper`, `Database`, and `MSSQLDataMapperFactory`. This subpackage contains a specific code to interact with the data source SQL server.

[†] Due to lack of space, operations are not shown.

The system is simple to understand because it is tiny and especially if we know its implementation infrastructure. This simplicity is not the case when we are in legacy systems where the size is more substantial and we have very little information on how the system was implemented.

3.1 The metamodel of the approach

The most complex task to discover the necessary models for a modernization process is the identification of the concepts present in the legacy system and to distinguish those of the implementation platform from those of the business domain. To accomplish this task, we base ourselves on the following hypothesis. The vocabulary of the source code of a system can be broken down into two vocabularies: independent (business) and dependent on the platform of implementation. The second consists of the keywords of the programming language and those of the implementation platform. Our hypothesis is that the code linked to the implementation platform and its vocabulary are repetitive or semi-repetitive. With this particularity, it is possible to identify this vocabulary and also to identify the business vocabulary by filtering vocabulary keywords of the implementation platform. Formally, the vocabulary (denoted by `V`) of a legacy system is the set of the words contained in its source code (comments excluded) and consists of: V_{PI} (the vocabulary introduced by the problem domain and therefore it is platform-independent), V_{PS} (the vocabulary introduced by the implementation platform and consequently, it is platform-specific) and V_{PL} (the vocabulary introduced by the programming language). We have $V = V_{PI} \cup V_{PS} \cup V_{PL}$.

A pattern of an identifier is a word (or a combination) contained as a substring in its name. We obtain patterns by splitting identifiers on delimiters like uppercase letters or underline symbols. For example, the identifier `AccountMapper` includes three patterns: `Account`, `Mapper`, and `AccountMapper`.

Let `V1` and `V2` be two variables of the source code of a system. `V2` is related by data-flow to `V1` if the DFA of the source code discovers that `V2` takes the value of `V1`.

A concept is a formulation of a system's requirement from the problem domain or the solution domain and is identified by its name. We distinguish two kinds of concepts: platform-independent concept (*PIC*) and platform-specific concept (*PSC*). A PIC is a concept from the problem domain, and a PSC is a concept from the solution domain. Figure 3 presents the metamodel of our approach.

A PIC belongs to V_{PI} and it can be one of the following types: classifier (class or interface), attribute, operation or parameter. It indicates a kind of element needed to support a requirement from the problem domain.

A PSC belongs to V_{PS} and it can be one of the following types: classifier, attribute, operation or parameter. It indicates a kind of element needed to support a requirement from the solution domain.

In our approach, we have a predefined set of PSCs to indicate elements of the PSM directly derived from elements of the PIM :

- *PICInterface* (respectively, *PICClass*) stating that the implementation platform requires the presence of at least one PSM interface (class, respectively) where its name belongs to V_{PI}.
- *PICAttribute* (respectively, *PICOperation*) stating that the implementation platform requires the presence of at least one attribute (respectively, operation) of a PSM classifier where its name belongs to V_{PI}.
- *PICParameter* stating that the implementation platform requires the presence of at least one parameter of a PSM operation where its name belongs to V_{PI}.

A concept may have one or several property concepts. A property concept (*PRP*) helps its parent concept in the formulation of a system's requirement, whether from the problem or the solution domain. A PRP has two attributes: *name* and *value*.

A group of classifiers (operations, respectively) is a set of classifiers (operations, respectively) grouped on a criterion. To ease the description of our approach, we assume that a group always contains at least two elements. A criterion, to create one of these groups may be one of the following:

- *LSI-based*, it allows the grouping of similar classifiers by LSI.
- *Clone-based*, it allows the grouping of operations that share cloned code.
- *Pattern-based*, it allows the grouping of classifiers or operations with a common pattern in their name.
- *Relationship-based*, it allows the grouping of classifiers that implement a common interface or inheriting a common classifier.

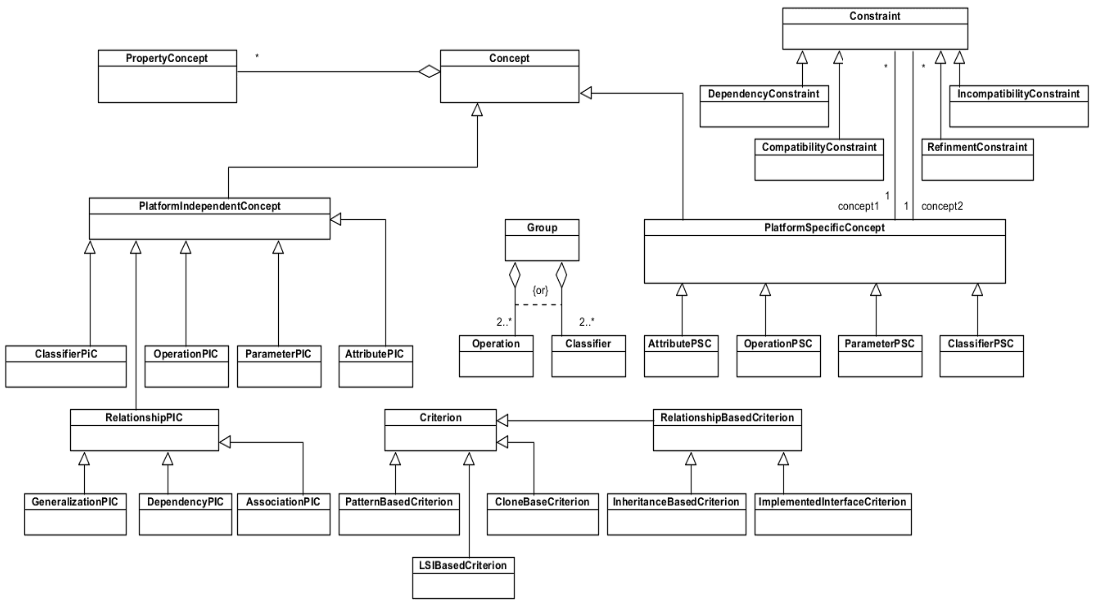

Figure 3: The metamodel of our approach

A group can be divided into subgroups according to one of the criteria defined above. Of course, an operation or a classifier can belong to different groups or subgroups.

An attribute PSC has one predefined PRP, namely *initialValue,* which captures the initial value of a PSM attribute.

A constraint can exist between two PSCs `C1` and `C2` to guarantee the integrity of an implementation platform, and can be one of the following types:

- A *dependency constraint* occurs when the implementation of `C1` requires the implementation of `C2`.
- A *compatibility constraint* occurs when the implementation of `C1` is compatible with the implementation of `C2`.
- An *incompatibility constraint* occurs when `C1` and `C2` cannot be implemented together.
- A *refinement constraint occurs* when the implementation of `C2` presents different variations, and the implementation of `C1` is one of these variations.

3.2 Overview of the UML profile discovery process

Figure 4 presents the steps of the UML profile discovery process. We allow a bimodal approach to this process: automatic and interactive. As we will see that systems leveraging a less uniformly implemented architecture or even without a real architecture pose a real challenge to our discovery process. Therefore, we decided to add an interactive mode where a user is asked to validate the results of some strategic intermediate steps. Since the primary goal of our approach is to help to understand a legacy system, prior knowledge of its implementation platform is not necessary. Intermediate results and discovering the code in strategic places will help him understand and thus validate the findings.

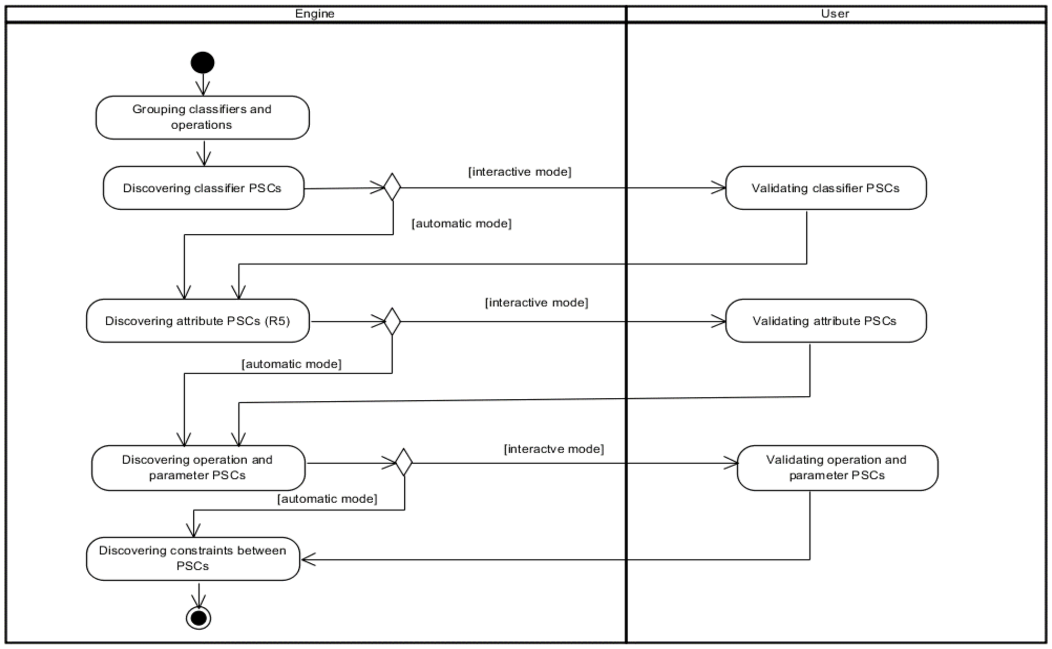
Engine
User
Grouping classifiers and operations
Discovering classifier PSCs
[interactive mode]
Validating classifier PSCs
[automatic mode]
Discovering attribute PSCs (R5)
[interactive mode]
Validating attribute PSCs
[automatic mode]
Discovering operation and parameter PSCs
[interactve mode]
Validating operation and parameter PSCs
[automatic mode]
Discovering constraints between PSCs

Figure 4 : Main steps of the UML profile process

Recall that our approach is based on the fact that it is possible to discover V_{PS}. We assume that the source code related to the implementation of PSCs has particular characteristics that distinguish it from others: it is repetitive or nearly repetitive. We can then assume that classifiers implementing the same PSC share a common vocabulary or cloned code. Therefore, we have chosen LSI and clone detection analysis techniques to partition the system into groups as the first step of our approach. The LSI technique groups classifiers sharing a common vocabulary while the clone detection technique groups operations sharing clones.

LSI-based groups of classifiers are extracted as follows. We build a matrix of the system. Columns are system's classifiers, and rows are verbs derived from the operation names of all classifiers (inherited operations are also considered for a classifier). The rationale behind using verbs comes from the fact that the verb in an operation name reflects best the behavior of the operation. The values in the matrix are then the frequency of verbs in classifiers after being normalized using tf–idf. The cosine between the corresponding vectors measures similarity between classifiers. The value of k is empirically determined for a system and the similarity threshold established to 0.995.

Clone-based groups of operations sharing a clone are extracted by using the tool CCFinder. First, the tool extracts clone pairs. Then, it generates clone groups (or clone classes) by chaining the clone pairs.

The second step discovers the system's vocabulary words likely to represent the classifier PSCs. They indicate which type of classifiers is needed to support an implementation platform requirement. The idea behind this step is that classifiers implementing the same PSC should share a common vocabulary or cloned code. Figure 5 illustrates this step with our simple system. The step isolates four classifier PSCs: `Adding`, `Form`, `Mapper`, and `Updating`. For example, all classifiers of the LSI-based group `{AccountForm, AddingAccountForm, AddingCustomerForm, CustomerForm, UpdatingAccountForm, UpdatingCustomerForm}` have in their name the common pattern `Form`. This pattern becomes then a classifier PSC. Note that the step does not identify some classifier PSCs such as `DomainObject` and `Database` because of the lack of repetitiveness in the implementation of these PSCs. In an interactive mode, a user can validate the resulting classifier PSCs. For our running example, the user can add the missing classifier PSCs.

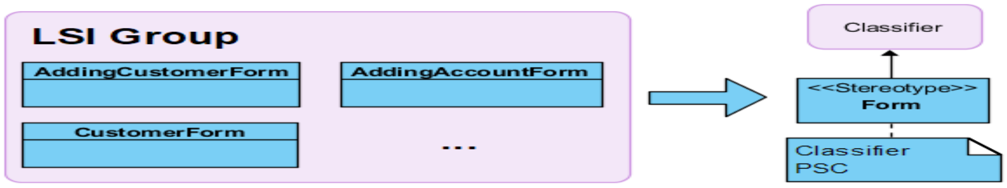

Figure 5: Illustration of classifier PSCs discovery step

The third step discovers the system's vocabulary words representing attribute PSCs. They indicate an abstract representation of the attributes needed by a classifier to support an implementation platform requirement. The intuition behind this step is that attributes present in any subgroup of classifiers implementing the same classifier PSC are probably related to this classifier PSC. Note that we check in any subgroup and not in all classifiers implementing the PSC since the latter may present different implementation variations. Figure 6 illustrates this step with our running example. In our case, all classifiers implementing the classifier PSC `Form` have the attribute `components`. Then this attribute represents an attribute PSC. Again, a user has the opportunity to correct the results of this step in an interactive mode. For example, the step incorrectly identifies `number` as an attribute PSC because it is in the `Form` classifier group. This error is mainly due to the small size of our toy system.

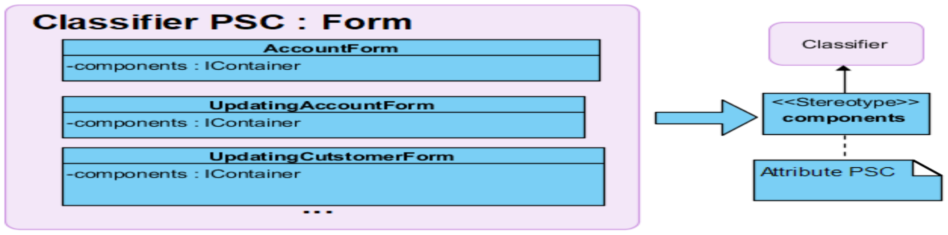

Figure 6: Illustration of attribute PSCs discovery step

The fourth step is divided into two sub-steps. The first sub-step discovers the system's vocabulary words representing operation PSCs. They indicate an abstract representation of the operations needed by a classifier to support an implementation platform requirement. The intuition behind this step that an operation containing an occurrence of a common vocabulary's word in its name or a common clone probably represents an operation PSC. Figure 7 illustrates this step with our running example. In our example, the operation `Insert` becomes an operation PSC because all classifiers of the `Mapper` group have this operation.

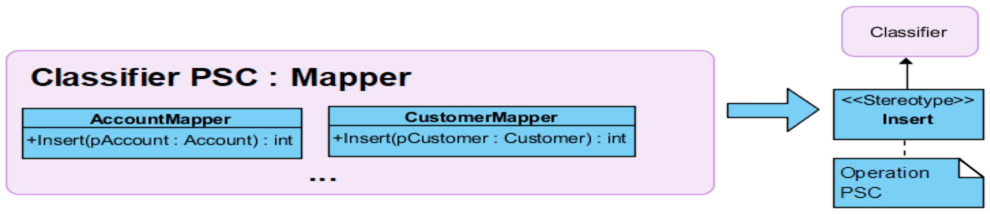

Figure 7: Illustration of operation PSCs discovery step

The second sub-step discovers paramater PSCs. A PSM parameter represents a parameter PSC if it is not linked to a PIC. Detection of this link by data flow analysis (DFA) or by identifier analysis.

The fifth step highlights constraints existing between the PSCs. First, this step exploits the relationships between classifiers and their attributes and operation to identify the constraints. For instance, as all `Mapper` classifiers in the PSM have the operation `Insert`, we can conclude that there is a dependency constraint that exists between the classifier PSC `Mapper` and the operation PSC `Insert`.

Figure 8 presents a part of the UML profile for our simple system.

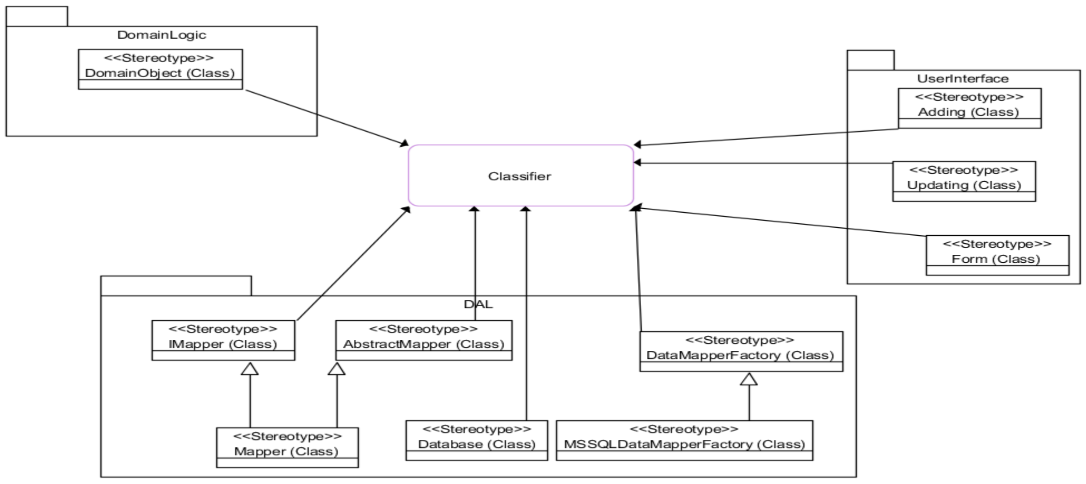

Figure 8: Part of the UML profile's simple system

4 Validation of approach

In [10], we presented the results of the validation of an earlier version of our approach. This version provided an automatic mode and supported legacy systems in Java language. As our approach is based heavily on the idea that a PSC belonging to an implementation platform is implemented repeatedly or semi-repeatedly in the source code of a system that adopts this platform, we discovered that with systems leveraging a less uniformly implemented architecture or even without a real architecture is more challenging. Repeatability can mean two things: consistency and plurality.

The lack of plurality may come from two reasons. The first reason is that the concept can be found uniquely in the source code of an implementing system. A common example of the case is the utility classes as the façade class `Database` in our running example. The second reason maybe because it is the nature of the analyzed system (for example, if the PIM is small), which ensures that the implementation is made only once. These two reasons lead us to think that with an extensive system, the impact would be less than what we had with our test systems.

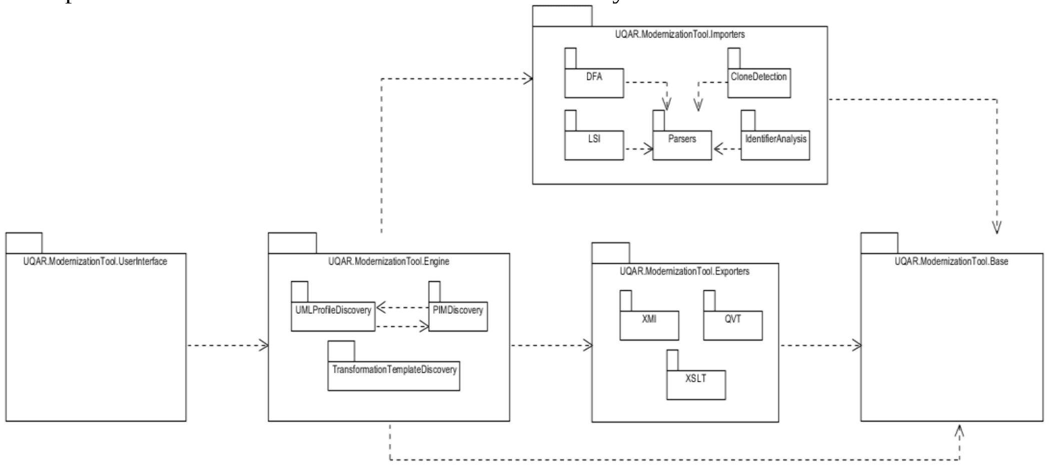
UQAR.ModernizationTool.Importers
DFA
CloneDetection
LSI
Parsers
IdentifierAnalysis
UQAR.ModernizationTool.UserInterface
UQAR.ModernizationTool.Engine
UMLProfileDiscovery
PIMDiscovery
TransformationTemplateDiscovery
UQAR.ModernizationTool.Exporters
XMI
QVT
XSLT
UQAR.ModernizationTool.Base

Figure 9: Architecture of our prototype tool

A legacy system with a consistent architecture shows a uniformity in the manner a PSC is implemented in the system. A variation in the implementation may come from a difference in the naming of the PSC or its implementation. The use of techniques as LSI and clone detection can handle inconsistent architecture to some extent. Therefore, we decided to add an interactive mode in a new version of our prototype tool where a user is asked to validate the results of some strategic intermediate steps in the discovery process. Moreover, this version supports the C# language. Figure 9 shows the architecture of our prototype tool. It is composed of five packages: `ToolUserInterface`, `ToolEngine`, `ToolImporters`, `ToolExporters`, and `ToolBase`. The `ToolUserInterface` package contains the different forms allowing a user to interact with the tool. The `ToolEngine` contains three sub-packages `UMLProfileDiscovery`, `PIMDiscovery`, and `TransformationTemplateDiscovery`. These sub-packages include the classes implementing the algorithm of our discovery processes (PIM discovery and Transformation template discovery processes are not described in this paper). The `ToolImporters` package contains five sub-packages: `DFA`, `LSI`, `CloneDetection IdentifierAnalysis`, and `Parsers`. The `DFA` package contains the source code implementing data flow analysis. We implemented our DFA by using the library offered by Soot[‡] for analyzing java's source code. For C# source code, we used the framework Roslyn[§]. The `LSI` package contains our own implementation of LSI analysis by using the library offered by ParallelColt[**] for linear algebra. We use the tool CCFinder[††] to create the groups of cloned operations. This code is in the `CloneDetection` package. The `Parsers` package implements the parsers for supporting C# by using the framework Roslyn and Java by using the framework SableCC[‡‡]. The `ToolExporters` package contains classes implementing the serialization of discovered models. PIM, PSM and UML profile models can be exported in XMI, and transformation models (not discussed in this paper) can be exported in QVT (Query/View/Transformation) or XSLT (eXtensible Stylesheet Language Transformations). The `ToolBase` package contains classes implementing the metamodel.

[‡] https://sable.github.io/soot / (2019-11-29)

[§] https://docs.microsoft.com/en-us/dotnet/standard/analyzers/ (2019-11-29)

[**] sites.google.com/site/piotrwendykier/software/parallelcolt, (2019-11-29)

[††] www.ccfinder.net (2019-11-29)

[‡‡] http://sablecc.org/ (2019-11-29)

5 Conclusion

In this paper, we proposed a bimodal approach to the discovery of a view of a legacy object-oriented system's implementation platform. The view is given as a UML class diagram representing its UML profile. Allowing a bimodal approach, where the interactive mode invites a user to validate intermediate results, enables our approach to give better support to legacy systems leveraging a less uniformly implemented architecture.

Since the primary goal of a practical modernization approach is to help to understand a legacy system, the user has not to have prior knowledge of its implementation platform. By allowing him to view intermediate results and discover the source code in strategic places will help him in this understanding process.

References

[1] N. W. Weiderman, D. B. Smith and S. R. Til, "Approaches to Legacy System Evolution," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa, 1997.

[2] A. Alam and T. Padenga, Application Software Re-Engineering, Pearson, 2010.

[3] R. C. Seacord, D. Plakosh and G. A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices, Addison Wesley Professional, 2003.

[4] OMG, "Architecture-Driven Modernization Task Force. Architecture-Driven Modernization scenarios," 2006.

[5] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer,* vol. 39, no. 2, pp. 25-31, 21 February 2006.

[6] J. Miller and J. Mukerji, "MDA Guide Version 1.0.1," Object Management Group, 2003.

[7] S. Deerwester, s. T. Dumais, G. W. Furnas, T. K. Landauer and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science,* vol. 41, no. 6, pp. 391-407, September 1990.

[8] U. Khedker, A. Sanyal and B. Sathe, Data Flow Analysis: Theory and Practice, CRC Press, 2009.

[9] G. Chénard, I. Khriss and A. Salah, "Towards the Discovery of Implementation Platform Description Models of Legacy Object-Oriented Systems," in *Workshop on Processes for Software Evolution and Maintenance (WoPSEM 2010) IEEE*, 2010.

[10] G. Chénard, I. Khriss and A. Salah, "Chénard, G., Khriss, I. and Salah, A. Towards the Automatic Discovery of Platform Transformation Templates of Legacy Object-Oriented Systems," in *Models and Evolution (ME) 2012 workshop a satellite event at MoDELS 2012*, Insbrusck, Austria, 2012.

[11] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.

[12] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[13] N. Kesserwan, R. Dssouli and J. Bentahar, "Modernization of Legacy Software Tests to Model-Driven Testing," in *AFRICATEK 2017*, Nader Kesserwan, Rachida Dssouli, Jamal Bentahar, 2017.

[14] H. Bruneliere, J. Cabot, J. Frédéric and M. Frédéric, "MoDisco: a generic and extensible

framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010.

[15] A. Sadovykh, L. Vigier, A. Hoffmann, J. Grossmann, T. Ritter, E. Gomez and O. Estekhin, "Architecture Driven Modernization in Practice - Study Results," in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, 2009.

[16] W. Zhang, A. Berre, D. Roman and H. Huru, "Migrating Legacy Applications to the Service Cloud," in *Proceedings of Systems, Languages, and Applications*, 2009.

[17] L. Favre, "Modernizing Software & System Engineering," in *Processes Proceedings of the International Conference on Systems Engineering*, 2008.

[18] F. Fleurey, E. Breton, B. Baudry, A. Nicolas and J.-M. Jézéquel, "Model-Driven Engineering for Software Migration in a Large Industrial Context," in *Proceedings of the international conference on Model Driven Engineering Languages and Systems*, 2007.

[19] F. Chen, H. Yang, B. Qiao and W.-C. Chu, "A Formal Model Driven Approach to Dependable Software Evolution," in *Proceedings of the International Computer Software and Applications Conference*, Chicago, United States.

[20] D. Doyle, H. Geers, B. Graaf, A. Deursen and J. Favre, "Migrating a domain-specific modeling language to MDA technology," in *Proceedings of the International Workshop on Metamodels*, Genoa, Italy, 2006.

[21] B. Qiao, H. Yang, W. Chu and B. Xu, "Bridging Legacy Systems to Model Driven Architecture," in *Proceedings of the International Conference on Computer Software and Applications*, Washington, United States, 2003.

[22] T. Biggerstaff, b. Mitbander and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the international conference on Software Engineering*, Baltimore, United States, 1993.

[23] A. Sivagnana Ganesan, T. Chithralekha and M. Rajapandian, "A Formal Model for Legacy System Understanding," *International Journal of Intelligent Systems and Applications*, vol. 10, no. 10, pp. 27-41, 2018.

[24] A. Shahbazian, Y. Kyu Lee, Y. Brun and N. Medvidovic, "Recovering Architectural Design Decisions," in *IEEE International Conference on Software Architecture*, 2018.

[25] Computer Science Department, University of Southern California, "Arcade Manual: Software Architecture Recovery, Smell Detection and Visualization," 2017.

[26] M. Langhammer, A. Shahbazian, N. Medvidov and R. H. Reussner, "Automated Extraction of Rich Software Models from Limited System Information," in *13th Working IEEE/IFIP Conference on Software Architecture*, 2016.

[27] P. Van Der Spek, S. Klusener and P. Van de Laar, "Towards Recovering Architectural Concepts Using Latent Semantic Indexing," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, Athens, Greece, 2008.

[28] R. C. de Boer and H. v. Vliet, "Architectural knowledge discovery with latent semantic analysis: Constructing a reading guide for software product audits," *The Journal of Systems and Software*, vol. 81, p. 1456–1469, 2008.

[29] O. Greevy and S. Ducasse, "Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces," in *Proceedings of International Workshop on Object-Oriented Reengineering*, Glasgow, U.K, 2005.