Reverse engineering UML class diagrams from Java code

SOPHIE VAN DER DONG, University of Twente, The Netherlands

During software development, mistakes occur which causes code to become inefficient or contain bugs. The more of these mistakes that occur within a project, the more technical debt it is said to accrue. To improve the occurrence of this, in this paper higher-level technical debt such as design-level debt is improved upon by reverse engineering unified modeling language (UML) class diagrams from Java code within a project. By visualizing the project's development and generating a class diagram from each occurring commit within a project, software developers can understand the state of their project more easily in this higher abstract view. Allowing for a better understanding of how progress was made during the project's life cycle and improving on the shortcomings made throughout it. This resulted in a mapping of Java to XMI representing these class diagrams as a step to allow other software developers and researchers to further improve upon detecting and documenting such debt to improve the quality of developed software and reduce long-term maintenance costs.

Additional Key Words and Phrases: XMI, UML, XML, Class Diagrams, Technical Debt, Java

1 INTRODUCTION

Software companies have always wanted to decrease their costs and improve their product quality. While research has been made, all possibilities to enact these changes are still not fully explored. Technical debt is one of these aspects. It is related to the cost of needing to rework code, may it be monetary, time-based, or something else. Technical debt can be considered the gap that arises between the hypothesized state of the project cost, time, and quality compared to the actual state it achieves [11]. The term itself can be considered an all-encompassing term for the debts it represents. Among these, higher-level debts such as design, architecture, or domain debt, have been under-researched. Higher-level debts (e.g. designing phase) refer to the debts that can occur in the early stages of a project when planning and design occur. Comparatively, to lower-level debt (e.g.implementation debt) where it is easier to quantify and create programs that check mistakes in static or dynamic code. Higher-level debt is more difficult to identify and quantify as it has an overarching effect on the whole project's development.

If repaid in a timely manner, accruing debt can help to temporarily boost performance. When prolonged, debt can stagnate projects by destroying progress and productivity. Hence, it is important to develop methods that enable researchers and developers alike, to detect and quantify technical debt, especially the higher-level debt. However, despite this importance, research has shown that higher-level technical debt is often neglected with 78% of the academia covered being of implementation-level debt [10].

A widely applied method to improve both efficiency and quality in software engineering involves the designing of diagrams as it allows developers to have a more abstract overview of the structure of their project. Using this approach generally improves people's understanding and comprehension of the overall systems they have to implement and would decrease future costs that can emerge from having to restructure or redo certain developed systems. This means it would allow reducing the accumulation of technical debt throughout the project, as addressing poor design decisions early on can prevent higher-level technical debt from accruing [9].

A unified modeling language (UML) class diagram is a tool that allows easier visualization. The creation of one class diagram for each commit a project has made, could aid in showcasing a lack of proper design or planning. This could be noted by the extent to which the class diagrams change throughout the project or certain changes that occur. Although a class diagram is not the only UML diagram that could have been chosen, it is a well-known and used diagram by software developers [4, 12]. Therefore, believing that UML class diagram visualization could aid in showcasing a lack of proper design or planning to improve a project's development, this paper aims to answer the following research question: *How to reverse engineer UML class diagrams from Java code?*.

To answer this question, we developed a script to reverse-engineered UML class diagrams from Java code within a project to show the project structure throughout its development to potentially identify instances where poor diagram design, inadequate documentation, or a lack of context may have started to influence the project's design and structure, thereby contributing to technical debt. This script can help to improve software documentation and communication, which, in turn, can help clarify one's understanding, rectify misunderstandings, and enhance the overall development process, ultimately reducing technical debt and improving project outcomes. Moreover, the approach proposed can aid software developers and researchers in prioritizing the detection and documentation of higher-level technical debt to mitigate long-term maintenance costs. It can also contribute to advancing research about higher-level technical debt.

The rest of the paper unfolds as follows. Section 2 provides the relevant background knowledge related to UML and XML Metadata Interchange (XMI). Section 3 explores how the selection of repositories was made, which choices were applied to extract the necessary information from the Java files, and how applications including Javaparser aided in reverse engineering the Java code to the class diagrams. Section 4 explores the conversions that was applied to map the source code to the relevant UML elements. Section 5 denotes some related work that conducted similar studies to reverse engineering UML class diagrams. Finally, section 6 makes some final considerations and discusses the direction of future work.

2 BACKGROUND KNOWLEDGE

In this section, the aim is to give a better understanding of the matters discussed within this paper.

TScIT 41, July 5, 2024, Enschede, The Netherlands

2.1 Unified Modeling Language (UML)

UML is a standardized modeling language developed to aid in Object-Oriented [OO] software development. Its current development and specifications are handled by the Object Management Group [OMG].[1] The UML consists of various diagrams as seen in Figure 1 that allow multiple stakeholders, including software developers to specify and visualize their software systems. It is a collection of practices proven to be beneficial in the development and maintenance of software systems [2, 5].

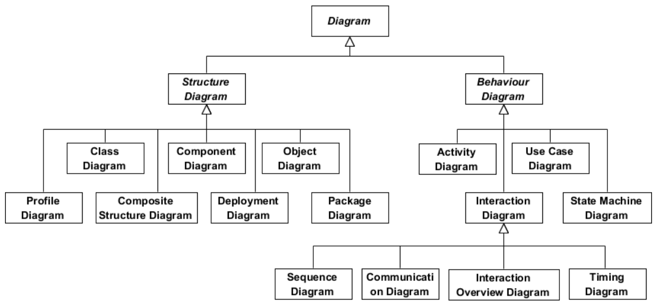


Fig. 1. UML Diagrams

As the purview of this study relates to the structure that the Java code creates, a structured diagram was selected. Comparatively, to behavior diagrams whose focus remains on the dynamic behavior occurring during the runtime of a program, structure diagrams focus on representing the static state of the systems with the classes, fields, methods, and their connections represented as objects, attributes, operations, and relationships. Class diagrams as one of the structured diagrams are considered one of the most important building blocks for modeling and designing object-oriented software systems [1]. It can provide a higher-level overview of the software system's design which aids in documenting its structure throughout multiple phases of the project life cycle. It consists of containers/classes that contain attributes and operations which for OO languages can be represented as fields and methods. These classes can be related to other classes through differing associations. However, as certain associations are logic-based, not all can be easily inferred from OO languages.

By using these features that class diagrams have, software developers can reduce their cost of production by determining their software systems structure and deciding where and which functionalities to implement, as it's easier and quicker to solve the earlier it occurs in a project's life cycle. On another note in regards to the paper, the chosen OO language from which the class diagrams are generated is Java, as it's a simple language that offers many libraries to peruse and most importantly is a strongly typed language. Determining the relationships between the classes is easier when the types are statically defined than during runtime.

2.2 XML Metadata Interchange (XMI)

Modeling software can be done with different levels of abstraction. Where UML is an abstract modeling language, the Extensible Markup Language [XML] is a language that aids software developers by creating rules that define the data and being a widely used standard for storing and transporting this data. Extending this, XML Metadata Interchange [XMI] is a markup language that can be utilized to exchange UML models across different tools and software programs. It has been developed to interchange objects and models by exchanging metadata information. To facilitate this exchange of information, XMI implements Extensible Markup Language [XML] which provides a set of rules for defining data and allows a specification standard for storing and transporting this data. XMI can be used for any meta-model that is expressed by the Meta-Object Facility [MOF] but is primarily utilized for exchanging UML models across different tools and software. This can therefore be considered a format that joins XML, UML, and MOF to support the development of OO systems [8]. Therefore, while supportive of the variety of diagrams UML consists of, the structured models that resemble OO systems tend to be better represented than the behavior models [7].

As XMI is an extension of XML, Document Type Definition [DTD] and XML Schemas can be used to validate its generated XML documents. While an XML document can be well-formed it can still contain errors such as its content. Both DTD and XML Schemas solve this by describing the XML document's structure, content, and constraints. When choosing this method of validation using an XML Schema is preferable as it's written in the same language, disallowing the need for learning another syntax, being extensible, and can depend on the context in which the element is used [3]. This definition of documents is of use due to the multiple structures and constraints that are possible for the same language version of XMI and UML.

3 METHODOLOGY

3.1 General Overview

Before attempting to resolve the research question, the reasoning and choices that were determined for it are expressed.

As previously stated Java is a language containing useful features and similar to class diagrams is OO, which caused it to be selected for being reverse engineered and to not add any additional languages was also used for writing the parser that allowed the Java code to be converted to class diagrams in XMI. For this, IntelliJ was chosen as the environment wherein a Maven Project, a build automation tool that does dependency management, plugins and other functions was created.

For finding relevant Java projects to convert, several Java repositories were explored by using the search tool SEART [2] from GitHub to realize which requirements the repositories should have. Afterward, rather than making a list of the repositories that fulfilled certain conditions an already existing list was chosen. This list consisted of repositories under a size limit of 100mb and ample commits, this would ensure that the repositories aren't too large while still retaining enough Java code that has underwent changes as represented by the commits. These requirements were sufficient to do most of the initial conversion testing on several repositories from this file. Later, a repository was created to more specifically test how certain properties and types of classes translated to XMI, in addition to the class's

[1] https://www.omg.org/

[2] https://seart-ghs.si.usi.ch/

relationships, without having to search for their occurrence within other repositories. Classes, fields, and methods with varying modifiers were included with the main focus of testing the relationships to verify: No duplication, fields made from multiple classes have their types properly extracted(e.g. ArrayList<HashMap<OtherClass, HashSet<ExampleClass>>> someField;) and that the relationship would be found and correctly parsed, as the relevant class could be within the same package, another package or be treated as an external class. This was done in parallel with the BigUML extension within Visual Studio Code. The resulting XML files generated by the parser script would be compared to the XML files generated from the class diagrams, which contained the same objects and information as in the created repository. The specifications of the generated files are version 2.5.1 for the XMI and version 5.0.0 for UML. For the comparisons an xsd file (XML Schema) from the BigUML to validate the generated XML by parser would have been preferable to more easily and certainly determine the correctness, however, this was not retrievable from BigUML, necessitating an xsd file to be written if wanting to achieve better verification and easy validation. In Figure 2 the general process of what happens with the chosen repositories is shown, where the commits are obtained of the repositories found by jGit and converted to AST trees with Javaparser before converted to an XML file with the determined mappings found through the help of BigUML and specification files. A more detailed explanation is given below.

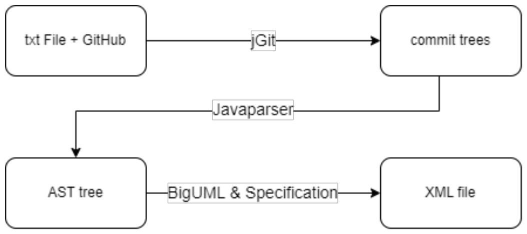


Fig. 2. Overview

3.2 Process

For obtaining and analyzing the repositories from GitHub, the jGit [3] library was used in addition to the text file that contained the unique repository path reference. When jGit retrieves the repositories, they are stored in a state that allows the important information regarding their structure, file content and commits to remain. When traversing through the commits tree like structure, only the java files are converted into an Abstract Syntax Tree [AST]. This was achieved using the Javaparser[4] library. This allows nearly no needed information to disappear of the Java objects themselves. Thus, the patterns found by using Javaparser would allow the AST to be converted to XMI. However, finding where the types within a class are referencing is not easy and occasionally unachievable.

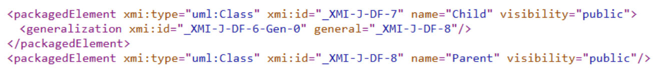
To properly convert the Java objects into XMI, two aspects needed to be determined first before the parsing occurs. This causes each commit to be traversed three times. In the first traversal, the indexing is done. Each UML object has a unique ID, including relationships, which also tend to contain the ID of the related objects. Therefore to be able to establish any connections between the objects, IDs have to be determined first (Table 1). The container objects (class, interface, and enumeration) are the objects whose IDs are established first and their IDs and absolute paths are stored within a HashMap. The creation of the IDs prevents mistakes from occurring for the packages or files containing the same name. These IDs have some differences that depend on which UML object they represent and from which class they are created as would be the case for the association XMI shown in Table 1. Additionally, during the second traversal when a relationship with an imported class is made, a new unique ID is established for that class and will be parsed as a class when all the classes within the project have their generated XMI code. The notable difference to this is that as an external class, its name will retain the full classpath. On the occasion that the external class was not found its name is noted as type+"_External". This occurs when the class originates from an import that imports multiple classes.

Java object	XMI ID
Subtree (folder/package)	"_XMI-P-Dir-"
Class / Interface / Enumeration	"_XMI-J-DF-" (classId)
Field	classId+"-A-"
Method	classId+"-M-"
Class implements / extends	classId+"-Gen-"
Class having or using another non primitive class	classId+"-C-"
External Class	"_XMI-Ext-"
leftover values of multiplicity	"_XMI-AsAt-
attribute of association	classId+"-AA-"
association	classId+"-C-A-"
dependency	classId+"-C-D-"

Table 1. XMI ID's for Java objects

In the second traversal of the commit the relationships between the classes are determined. When any link between classes is found this information is stored within a map along with the necessary IDs and names. The first connections determined are the class inheritances, this is represented as generalization objects within the class with "general=" referring to the inherited class as showcased in Figure 3. When an interface is inherited the inheritance that occurs is stored as a UML realization object, that is added to the list of external classes and relationships instead of being nested within the class.

[3]https://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit/6.9.0.202403050737-r
[4]https://mvnrepository.com/artifact/com.github.javaparser/javaparser-core/3.26.0



```
<packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-7" name="Child" visibility="public">
  <generalization xmi:id="_XMI-J-DF-6-Gen-0" general="_XMI-J-DF-8"/>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-8" name="Parent" visibility="public"/>
```

Fig. 3. class Child's generalization to Parent

For associations, when a class variable contains another class that isn't primitive, String, List, ArrayList, or HashMap it is represented as an association. The relevant classes receive an attribute association object in addition to a separate association object being generated. Compared to all other generated attributes, attributes generated from association contain the "LiteralString" UML type. Whilst different from the BigUML extension, this choice was consciously made to allow the use of the "*" to indicate many instances of the related object. This multiplicity of the attributes is determined by the variable being instantiated and whether it is within a list, which is determined by the existence of "<" and ">". When the variable is initially unsubstantiated the multiplicity refers to "0". Also, a constructor instantiating the variable upon construction is not considered for these associations, meaning the multiplicity would remain "0". The multiplicity is added to the generated attribute association for the current class, leaving the other unfinished to allow the other related class to generate the multiplicity if it contains a variable that relates to the current class. These associations are kept in separate HashMaps. When associations remain unfinished upon the full iteration of the project, the default attribute value "1" is added. When a class method uses another class in its parameters or return type, it is represented as a Dependency object. Dependencies are added to the list of external classes and relationships.

If the XMI was generated in the same traversal as the detection of the object relationships, it would be possible for an XMI of a class to be already generated while missing certain associations. Additionally, if the connections occur from an external import where the class is not part of the project, a new class will be made to represent this in the XMI with the full classpath. It is also possible for an external class to remain undetected if the import, imported multiple classes, indicative by "*".

In the third traversal, the XML files containing XMI and UML are generated. Whilst traversing the commits tree structure, the package structure is generated for the classes with the XMI of the classes nested within but as this script only considers Java files, package structures can contain no objects within them. When the class XMI is generated the modifiers (e.g. visibility, abstract, etc.) for the class, fields, and methods are considered and added as properties within the object-defining element tag. Then, the inheritance XMI is generated first within the object followed by all the attributes and methods. After, the relationships are added. When all classes of the project are generated the external classes and relationships are added, before being closed with the UML element tag.

4 TRANSFORMATION RESULTS

This section explains the resulting transformations from the mappings that were applied to the Java objects with the script to obtain the UML objects. These objects that make up a class diagram can be broken down into three parts: the containers, their features, and the relationships between them. A comprehensive overview is given for each of them below.

#	Java object	XMI
1	public class E1{}	<packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-1" name="E1"/>
2	public abstract class E2{}	<packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-2" name="E2" visibility="public" isAbstract="true"/>
3	public interface E3{}	<packagedElement xmi:type="uml:Interface" xmi:id="_XMI-J-DF-3" name="E3" visibility="public"/>
4	public enum E4{}	<packagedElement xmi:type="uml:Enumeration" xmi:id="_XMI-J-DF-4" name="E4" visibility="public"/>
5	package	<packagedElement xmi:type="uml:Package" xmi:id="_XMI-P-Dir-3" name="alsoPaper"/>

Table 2. Container Elements

4.1 Containers

The containers that can be identified are class, abstract class, interface, and enumeration. Packages are also considered containers within UML, however, they cannot be compared to any Java objects as it's equivalent to folders containing Java files. Their representation of the UML elements is shown in Figure 4 and representations of these elements within Java and their conversion to XMI are represented within Table 2.

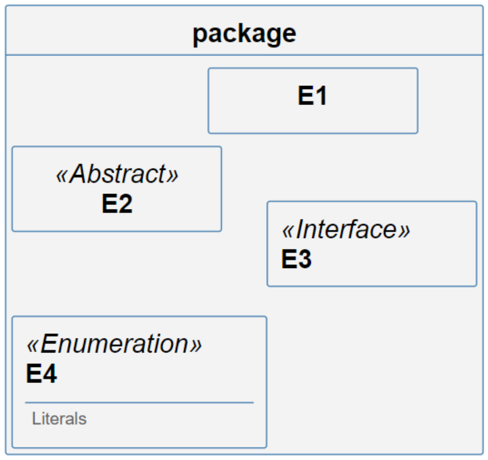


Fig. 4. UML container elements

As noted in # 2, the abstract class that exists within Java is not a UML element inherently and therefore contains a modifier "isAbstract" to denote this.

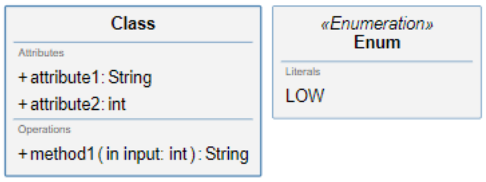


Fig. 5. feature elements

#	Java object	XMI
1	String attribute1;	<ownedAttribute xmi:id="_XMI-J-DF-1-A-0" name="attribute1" type="String"> ... </ownedAttribute>
2	int attribute2 = 3;	<ownedAttribute xmi:id="_XMI-J-DF-1-A-3" name="attribute2 " type="int"> ... </ownedAttribute>
3	String method1(int input){return "";}	<ownedOperation xmi:id="_XMI-J-DF-1-M-0" name="method1" type="String"> ... </ownedOperation>
4	LOW	<ownedLiteral xmi:id="_XMI-J-DF-31-0" name="LOW"/>

Table 3. Feature Elements

4.2 Features

Container UML elements can have multiple types of features; however, the enumeration container can only have literals. This is shown in Figure 5 and is represented by #4 in Table 3.

For an easier overview, the “lowerValue” and “upperValue” have been removed from the XMI column for #1 and #2 and the “ownedParameter” for #3. However, a part of an XMI-generated file can be viewed in Appendix B to view how the generated XMI is represented overall.

4.3 Relationships

Depending on the values and classes these features use, relationships are formed between the UML containers. Within Java files, this is represented by one file containing variables or methods of another file. In Table 4 the Java objects that indicate a relationship between classes is represented, with the additional relationship elements relevant to these objects being shown in Table 5. However, inner classes are not detected, meaning composition is not a defined relationship. Additionally, for converting Java to UML, aggregation is an association based on logical reasoning allowing its interpretation to vary. Due to this, the weaker relationship association is established when an attribute has another class. As noted by #6, if a method uses another object there is an even lesser relationship between classes and therefore only allows a dependency. Though, if an association already exists no dependency is added. Furthermore, for the relationships # 1 and # 6 exist as package elements outside of their related class. Including with this, for the association relationship, besides an attribute being nested within the two related classes, a separate package element is created that contains the association id and an attribute “memberEnd” which consists of the two attribute ids combined. For #3 this element would be represented as: <packagedElement xmi:type=“uml:Association” xmi:id=“_XMI-J-DF-20-C-A-0” memberEnd=“_XMI-J-DF-20-AA-0__XMI-J-DF-20-AA-3”/>

#	Java object	XMI
1	public abstract class E2 implements E3{}	<packagedElement xmi:type="uml:Realization" xmi:id="_XMI-J-DF-17-Imp-0" client="_XMI-J-DF-17" supplier="_XMI-J-DF-18"/>
2	public class E1 extends E2{}	<packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-16" name="E1"> <generalization xmi:id="_XMI-J-DF-16-Gen-0" general="_XMI-J-DF-17"/> </packagedElement>
3	E1 variable1;	<ownedAttribute xmi:id="_XMI-J-DF-20-A-0" name="variable1" type="E1"> ... </ownedAttribute>
4	E1 variable2 = new E1();	<ownedAttribute xmi:id="_XMI-J-DF-20-A-3" name="variable2 " type="E1"> ... </ownedAttribute>
5	ArrayList<E1> variable3 = new ArrayList<>();	<ownedAttribute xmi:id="_XMI-J-DF-20-A-6" name="variable3 " type="ArrayList<E1>"> ... </ownedAttribute>
6	private Some m5(){ return new Some(); }	<ownedOperation xmi:id="_XMI-J-DF-5-M-7" name="m5" type="Some"/>

Table 4. Relationship Causing Elements

#	XMI
3	<ownedAttribute xmi:id="_XMI-J-DF-20-AA-0" name="E1" type="_XMI-J-DF-16" association="_XMI-J-DF-20-C-A-0"> <lowerValue xmi:type="uml:LiteralString" xmi:id="_XMI-J-DF-20-AA-1" value="0"/> <upperValue xmi:type="uml:LiteralString" xmi:id="_XMI-J-DF-20-AA-2" value="1"/> </ownedAttribute>
4	<ownedAttribute . . . > <lowerValue . . . value="1"/> <upperValue . . . value="1"/> </ownedAttribute>
5	<ownedAttribute . . . > <lowerValue . . . value="1"/> <upperValue . . . value="*"/> </ownedAttribute>
6	<packagedElement xmi:type="uml:Dependency" xmi:id="_XMI-J-DF-5-C-D-0" client="_XMI-J-DF-5" supplier="_XMI-J-DF-8"/>

Table 5. Relationship Elements

5 VALIDATION

This paper did not manage to implement a strong form of validation, however, if a form of validation were to be implemented, it would be through the usage of an XML schema. This is also referred to as XML Schema Definition (XSD) with a .xsd file name extension.

The creation of the XML schema would use UML and XMI namespaces represented by the xlmns attribute within its root element <schema>. By declaring these namespaces, conflicts can be avoided where element names are the same and indicate where the elements and data types come from. From there, a schema would be made to represent the XMI structure that is contained within the XML files. For actually validating the XML files a validator would be programmed with an error handler to validate the schema. By using an error handler with the validation process, the error can be visualized in an easier to overview format and can contain necessary information such as the location of the error in a more readable format.

6 RELATED WORK

Xin Wang et al. [13] use an AST based approach with the visitor pattern to build UML model elements where they found that the AST visitor can query any properties needed to build a design model, resulting in the AST based approach being more precise than other conventional existing approaches. Similarly, this paper also used an AST based approach for the precision, yet some dissimilarities between these studies are their use of an eclipse based plugin to allow a more efficient binding between classes for generalization and no mapping for defining associations between classes.

Fauzi et al. [6] use an AST based approach to assist in their reverse engineering to sequence diagrams. They found that by revealing the sequence of statements within the source code, AST works well for converting it to the sequence diagram. As they indicated this to be a functional process to generate a behavior type of UML diagram, they suggested to make the process of reverse engineering more portable by modifying the generated AST into intermediate formats such as XML, which this paper by using XMI manages to accomplish.

7 CONCLUSION AND FINAL REMARKS

This paper presents a script that generates XMI-formatted XML files from Java through the process of AST trees. By generating the UML class diagrams with XMI, it is represented at a higher level of abstraction that is easier to visualize while also supporting the exchange of design documents for UML diagram design tools. The mapping itself contained the main UML class diagram elements represented from any given Java code, which includes: (i) the mapping to the UML classes from Java classes (Class, Interface, Enumeration) except annotation and inner classes; (ii) the defining of associations and dependencies between classes from their first occurrence within a class; and (iii) the modifiers that are applied to Java objects represented as attributes within the UML elements. This all allows for a project's state to be better visualized at an abstract level, aiding developers' understanding of their projects and improving their future choices to reduce potential technical debt, determined from the changes they can note in the multiple generated XML files.

Future works could improve on extending the quality of the mapping regarding the constructors and multiple occurrences of a class within another class represented as a factor that influences multiplicity. Additional improvements could be made by allowing the validation of the XMI through the usage of an XML Schema. Finally, by analyzing changes that occur throughout the project with the visualization of the class diagrams, notes could be taken on the changes and patterns that would have or had a negative impact on the progress of the project.

REFERENCES

[1] Fahad Alhumaidan. [n. d.]. A Critical Analysis and Treatment of Important UML Diagrams Enhancing Modeling Power. 04, 5 ([n. d.]), 231. https://doi.org/10.4236/iim.2012.45034 Number: 05 Publisher: Scientific Research Publishing.

[2] Maria Teresa Baldassarre, Danilo Caivano, Simone Romano, and Giuseppe Scanniello. [n. d.]. Software Models for Source Code Maintainability: A Systematic Literature Review. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2019-08). 252–259. https://doi.org/10.1109/SEAA.2019.00047

[3] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. [n. d.]. Inferring XML Schema Definitions from XML Data. ([n. d.]).

[4] Brian Dobing and Jeffrey Parsons. 2006. How UML is used. *Commun. ACM* 49, 5 (2006), 109–113.

[5] Wojciech J. Dzidek, Erik Arisholm, and Lionel C. Briand. [n. d.]. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. 34, 3 ([n. d.]), 407–432. https://doi.org/10.1109/TSE.2008.15 Conference Name: IEEE Transactions on Software Engineering.

[6] Esa Fauzi, Bayu Hendradjaya, and Wikan Danar Sunindyo. [n. d.]. Reverse engineering of source code to sequence diagram using abstract syntax tree. In *2016 International Conference on Data and Software Engineering (ICoDSE)* (2016-10). 1–6. https://doi.org/10.1109/ICODSE.2016.7936137

[7] Holly A. H. Handley, Wael Khallouli, Jingwei Huang, William Edmonson, and Nadew Kibret. [n. d.]. Maintaining the Consistency of SysML Model Exports to XML Metadata Interchange (XMI). In *2021 IEEE International Systems Conference (SysCon)* (2021-04). 1–8. https://doi.org/10.1109/SysCon48628.2021.9447105 ISSN: 2472-9647.

[8] F. Ruiz, A. Vizcaino, F. Garcia, and M. Piattini. [n. d.]. Using XMI and MOF for representation and interchange of software process. In *14th International Workshop on Database and Expert Systems Applications, 2003. Proceedings.* (2003-09). 739–744. https://doi.org/10.1109/DEXA.2003.1232109 ISSN: 1529-4188.

[9] Mohamed Soliman, Paris Avgeriou, and Yikun Li. [n. d.]. Architectural design decisions that incur technical debt — An industrial case study. 139 ([n. d.]), 106669. https://doi.org/10.1016/j.infsof.2021.106669

[10] Harald Störrle and Marcus Ciolkowski. [n. d.]. Stepping Away From the Lamppost: Domain-Level Technical Debt. https://doi.org/10.1109/SEAA.2019.00056

[11] Edith Tom, Aybuke Aurum, and Richard Vidgen. [n. d.]. A CONSOLIDATED UNDERSTANDING OF TECHNICAL DEBT. ([n. d.]).

[12] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. *Journal of Systems and Software* 86, 6 (2013), 1498–1516.

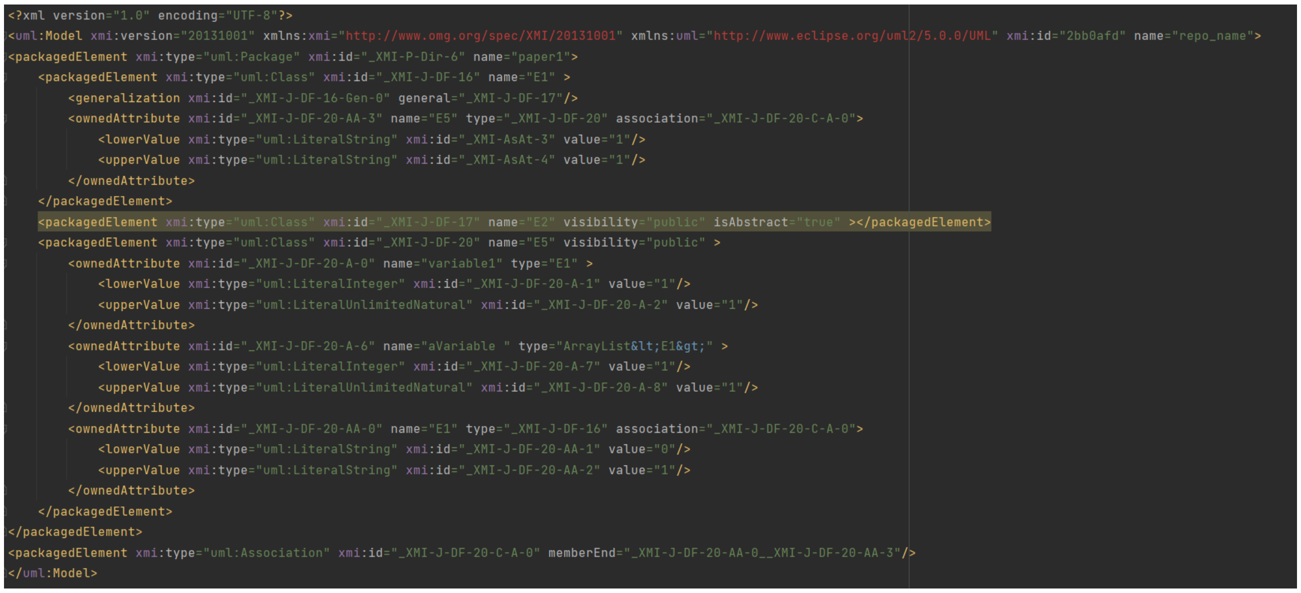
[13] Xin Wang and Xiaojie Yuan. [n. d.]. Towards an AST-Based Approach to Reverse Engineering. In *2006 Canadian Conference on Electrical and Computer Engineering* (2006-05). 422–425. https://doi.org/10.1109/CCECE.2006.277552 ISSN: 0840-7789.

8 APPENDIX

8.1 Appendix A Acknowledgments

To improve the readability, this work used the help of Grammarly as a technical assistant for spellchecking. By using this tool, the author takes full responsibility for the content of the work. Secondly, I would like to acknowledge the technical assistance from Tiago Prince Sales.

8.2 Appendix B Relationship example



```xml
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001" xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML" xmi:id="2bb0afd" name="repo_name">
<packagedElement xmi:type="uml:Package" xmi:id="_XMI-P-Dir-6" name="paper1">
    <packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-16" name="E1" >
        <generalization xmi:id="_XMI-J-DF-16-Gen-0" general="_XMI-J-DF-17"/>
        <ownedAttribute xmi:id="_XMI-J-DF-20-AA-3" name="E5" type="_XMI-J-DF-20" association="_XMI-J-DF-20-C-A-0">
            <lowerValue xmi:type="uml:LiteralString" xmi:id="_XMI-AsAt-3" value="1"/>
            <upperValue xmi:type="uml:LiteralString" xmi:id="_XMI-AsAt-4" value="1"/>
        </ownedAttribute>
    </packagedElement>
    <packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-17" name="E2" visibility="public" isAbstract="true" ></packagedElement>
    <packagedElement xmi:type="uml:Class" xmi:id="_XMI-J-DF-20" name="E5" visibility="public" >
        <ownedAttribute xmi:id="_XMI-J-DF-20-A-0" name="variable1" type="E1" >
            <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_XMI-J-DF-20-A-1" value="1"/>
            <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_XMI-J-DF-20-A-2" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:id="_XMI-J-DF-20-A-6" name="aVariable " type="ArrayList&lt;E1&gt;" >
            <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_XMI-J-DF-20-A-7" value="1"/>
            <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_XMI-J-DF-20-A-8" value="1"/>
        </ownedAttribute>
        <ownedAttribute xmi:id="_XMI-J-DF-20-AA-0" name="E1" type="_XMI-J-DF-16" association="_XMI-J-DF-20-C-A-0">
            <lowerValue xmi:type="uml:LiteralString" xmi:id="_XMI-J-DF-20-AA-1" value="0"/>
            <upperValue xmi:type="uml:LiteralString" xmi:id="_XMI-J-DF-20-AA-2" value="1"/>
        </ownedAttribute>
    </packagedElement>
</packagedElement>
<packagedElement xmi:type="uml:Association" xmi:id="_XMI-J-DF-20-C-A-0" memberEnd="_XMI-J-DF-20-AA-0__XMI-J-DF-20-AA-3"/>
</uml:Model>
```

Fig. 6. Relationship Example