Scientific Research Publishing

Journal of Computer and Communications, 2025, 13(3), 86-102
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

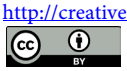
Reverse Engineering Approach for Analyzing and Transforming Graphical User Interface Source Code into Class Diagrams

Herifidy Malalaniaina Andrianaivo, William Germain Dimbisoa, Thomas Mahatody

School of Computer Science, University of Fianarantsoa, Fianarantsoa, Madagascar
Email: andrianaivoherifidy@gmail.com

How to cite this paper: Andrianaivo, H.M., Dimbisoa, W.G. and Mahatody, T. (2025) Reverse Engineering Approach for Analyzing and Transforming Graphical User Interface Source Code into Class Diagrams. *Journal of Computer and Communications*, **13**, 86-102.
https://doi.org/10.4236/jcc.2025.133007

Received: January 30, 2025
Accepted: March 23, 2025
Published: March 26, 2025



Abstract

In the field of software engineering, the design phase occupies a pivotal position, serving as a critical juncture in ensuring the quality, maintainability, and efficiency of software systems. The creation of software typically begins with the design phase, which is then succeeded by the production phase. In this context, model-driven engineering (MDE) represents a robust methodology for enhancing the development of complex systems by situating models at the core of the process. This method is concerned with the creation and transformation of models, progressing from abstract to concrete representations. To illustrate, in the context of MDE, models can be transformed from the Unified Modelling Language (UML) to a graphical interface. In contrast, reverse engineering such as MDE-based reverse engineering is a process that aims to understand, analyse and reconstruct design artifacts from an existing system. This approach involves transforming concrete models into abstract models. This paper proposes an innovative approach to reverse engineering based on software design using the graphical user interface model. More specifically, this project involves the transformation of a WIMP (Window, Icon, Menu and Pointer) graphical user interface into a class diagram. To do this, we will use a syntactic analysis method of the source code based on the Java language and regular expressions, and the Atlas Transformation Language (ATL) will be used to transform the graphical interface into a class diagram.

Keywords

MDE, Reverse Engineering, Source Code Parsing, Regex, ATL, Class Diagrams

1. Introduction

Design is a crucial phase in software engineering. It precisely defines the detailed

requirements, the functionalities to be implemented and the concepts of the software to be created. It allows potential problems to be identified and resolved early in the project, ensures the consistency and quality of the final product, and optimises resources and lead times.

From this perspective, MDE is emerging as a powerful approach to strengthen these aspects by placing models at the centre of the development process. MDE focuses on the creation and transformation of models, facilitating the transition from abstract to concrete representations, and thus from design to implementation. For example, this approach allows models derived from the UML to be transformed into functional graphical interfaces, illustrating the potential of MDE to improve the quality and efficiency of the development of complex systems.

Many researches in MDE focus on the generation of HCI from autonomous models [1]-[3]. The creation of HCI is therefore an essential stage in the development of modern software, as it determines the way in which users interact with computers. Currently, it is common to come across software projects where existing systems lack adequate documentation or present absolute interfaces. So, in the face of rapidly evolving technologies and user requirements, it is essential to redesign, update or reuse HCI.

Reverse engineering [4] is a method of analyzing an existing product or system to understand its structure, operation and behaviour. The process involves breaking down the product into its fundamental components and creating an abstract representation or documentation of the system based on this analysis. Reverse engineering can be used to modernize interfaces by transforming HCI models into class diagrams. Transforming an HCI [5] model into a class diagram is a crucial step, as the class diagram is a fundamental component of the UML. It plays an important role in software engineering for the design and documentation of software systems. In addition, in object-oriented programming, the dominant trend in software development, the class diagram helps to organize code into classes and objects. By visualizing classes and their relationships, developers can design systems that are more modular, reusable and easy to maintain.

Furthermore, transforming the graphic user interface (GUI) into a class diagram poses significant challenges in reverse engineering, as these two models are not compatible. Obtaining a class diagram, including the types of relationships between classes and multiplicities, from the GUI requires clear methods.

Given the incompatibility between the graphical interface model and the UML model [6], this research aims to create a methodical approach for reverse engineering graphical interfaces of the window, icon, menu and pointer types into UML diagrams. The aim of this research is to analyze the source code of graphical interfaces, examining elements such as buttons, panels, window names and events. This analysis will make it possible to transform the results obtained into a UML model, thus facilitating the conversion of the graphical interfaces into coherent and usable UML representations.

For this project, we will use the reverse engineering method. First, we will pro-

ceed the parsing of the GUI source code to extract the various elements needed to create the class diagram. This includes class names, attributes, methods, relationships between classes and multiplicities, among others. This analysis will be carried out using regular expressions to accurately identify and extract these elements. Secondly, we will transform the results of this syntactic analysis into an instance of the Ecore metamodel using the Java language. Finally, we will transform the graphical interface into a class diagram using ATL [7].

2. Methodologies

This section presents a review of previous research in the area of reverse engineering, parsing and transforming GUI source code into UML class diagrams. It discusses existing methods and situates the present work in relation to this existing body of knowledge.

2.1. Reverse Engineering

Reverse engineering is a methodology frequently employed in model-driven engineering projects. [8] presented an illustrative example of reverse engineering, which was employed with the objective of facilitating the development and maintenance of software comprising substantial user interface source code. The GUISURFER tool [8] [9], a reverse engineering tool that automatically extracts behavioural models from GUI source code, was employed in this instance. The tool is also capable of automating certain tasks associated with the analysis of these models; however, it is unable to generate a class diagram from the GUI.

[10] employs a model-based architectural approach to facilitate comprehension of complex software systems throughout their evolution and maintenance. In accordance with [10] methodology, the conversion of GUI source code into a diagram is feasible; however, the requisite tool for transforming source code into a UML model is currently unavailable.

In their study, [11] employed reverse engineering to construct a class diagram from the graphical user interface, which was represented in the form of a screen capture. In this study, optical character recognition (OCR) [12] and Petri nets were employed to transform the graphical interface into a class diagram. This approach results in the generation of a class diagram; however, it lacks the requisite details pertaining to the recovery of the types of relationship between classes and stereotypes.

In this study, we put forth a novel methodology for parsing with regular expressions and transforming GUI source code into UML class diagrams. The method is distinguished by its flexibility and efficiency in source code transformation.

2.2. Parsing Analysis

Parsing analysis represents a fundamental stage in the processing and execution of programs. It is employed to ascertain whether the source code adheres to the grammatical conventions of the programming language and to transform this

code into an organised data structure for subsequent processing stages. The utilisation of decomposition and composition methodologies, in conjunction with tools such as Yacc [13], Bison [14] and ANTLR [15], facilitates the development of robust and high-performance parsers for an array of programming languages.

Parsing is the process of identifying the structure of a text, which is often a sentence in a natural language. It is also used for computer programs. In the context of computer science, parsing involves the examination of the contents of a text or file in order to ascertain its syntax or to identify the requisite elements. In the context of source code, [16] employed source code analysis and transformation techniques, with a particular emphasis on tools for the description, analysis and transformation of source code. This approach is concerned with the transformation of source code, employing techniques of syntactic analysis and code rewriting. However, it does not address the specific issue of transforming source code into UML class diagrams.

2.3. Regular Expressions

Regex are powerful tools that are used to identify and manipulate text patterns in character strings. Despite their occasionally intricate structure, regular expressions facilitate the formulation of precise and efficient text queries. They are a crucial tool in modern word processing and programming languages, including the retrieval of social media data using the Social Media Developers API and Regex [17].

Table 1 provides an illustration of the process of recovering a line of code in Java using regular expressions. In this example, a regular expression was employed to identify and extract the declarations of graphical components, including JButton, JTextField, JFrame, and JLabel. This approach automates the analysis of the source code, thereby facilitating the transformation of graphical elements into UML representations. Furthermore, the utilisation of regular expressions is highly advantageous for the processing of extensive code volumes, circumventing the potential inaccuracies inherent to manual analysis.

Table 1. Example of Regex-based code line retrieval in Java Swing.

Graphics Component	Line of code to be analyzed	Regular expressions
Boutons (Jbutton)	JButton myButton = new JButton("Click");	JButton\s+(\w+)\s*=\s*new\s+JButton\s*\((.*?)\)\s*;
Field (Jlabel)	JLabel myLabel = new JLabel("name: ");	JLabel\s+(\w+)\s*=\s*new\s+JLabel\s*\((.*?)\)\s*;
Text field (JtextField)	JTextField myTextField = new JTextField(20);	JTextField\s+(\w+)\s*=\s*new\s+JTextField\s*\((.*?)\)\s*;
Window (Jframe)	JFrame frame = new JFrame("Customer")	JFrame\s+(\w+)\s*=\s*new\s+JFrame\s*\((.*?)\)\s*;
Class name (public class)	public class Example	public\s+class\s+(\w+)\s*{?

Table 2 presents an example of a regular expression used in the analysis of web page source code, illustrating its capability to extract specific information beyond graphical components.

Table 2. Example of extracting web code lines using Regex.

Component/or test	Line of code to be analyzed	Regular expressions
Page Title	<title>MyTitle</title>	<title>(.*?)</title>
button	<button type="submit">Register</button>	<button(?:\s+[^>]*)?>(.*?)</button>
label	<label for="email">Email:</label>	<label for=\" (.*?)\">(.*?)</label>
Testing the presence of a price (in $ or €)	Price: $19.99	[$€]\\d+(\\.\\d+)?
To test for the existence of an email address	Exemple33@gmail.com	[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
To test for the presence of the word “name”	<h2 Id="name_product"> Status in Malagasy Art </h2>	<*=\"name
Redirection managed by HTML	Register	href=\" ([^"]*)\ "
Redirection managed by JavaScript	docu ment.getElementById('registerButton').onclick = function() { window.location.href = 'shop.html'; };	window\.location\.href\s*=\s*'([^']*)' or window\.location\s*=\s*'([^']*)'
To test for the presence of an image	<img src="mintour-banque-dimage-photo-336.jpg" alt="Product">	<img\s*[^>]*src\s*=\s*["'][^"']+\.(jpg|jpeg|png|gif)["'][^>]*>

2.4. Transformation of Source Code into a Class Diagram

The conversion of source code into a class diagram represents a fundamental aspect of systems design, particularly in the context of object-oriented systems. They play a pivotal role at various stages of software development, offering a visual representation of data structures and relationships between disparate classes within a system. The automatic transformation of source code into class diagrams facilitates a more comprehensive understanding, documentation and maintenance of software systems. This paper presents an overview of existing approaches to the automatic transformation of source code into class diagrams, an examination of the tools that are currently available for this purpose, and a discussion of the challenges that have been encountered. It then proposes a new method that is based on syntactic analysis and regular expressions.

Table 3. Existing and proposed approaches.

Methodology	Transformation tool	Graphical interface to class diagram
Approach of Saraiva *et al.*, (2012) [8]	GUISURFER	does not transform
Approach of Favre, (2012) [10]	NEREUS	only a theoretical approach, not a transformation
Approach of Muhairat *et al.*, (2011) [11]	OCR, Petri nets	contains the transformation, but the class diagram still needs to be completed
Proposed approach	Regex, Java and ATL	Transformation of the graphical interface into a class diagram

Table 3 illustrates the extant approaches, together with the tools used for model

transformation. The approach proposed by Saraiva *et al.*, (2012) [8] does not include the transformation of the graphical interface into a class diagram. The approach proposed by Favre (2012) [10] presents a theoretical framework but lacks a detailed explanation of the transformation process. Similarly, the approach put forth by Muhairat *et al.* (2011) [11] outlines the transformation but lacks a comprehensive methodology for identifying the types of relationships between classes and the associated stereotypes.

In order to achieve this, we put forward a syntactic analysis approach for source code, based on regular expressions, and a Java transformer to transform the analysis result into a UML class diagram. Figure 1 presents an overview of the methodology employed in this research project.

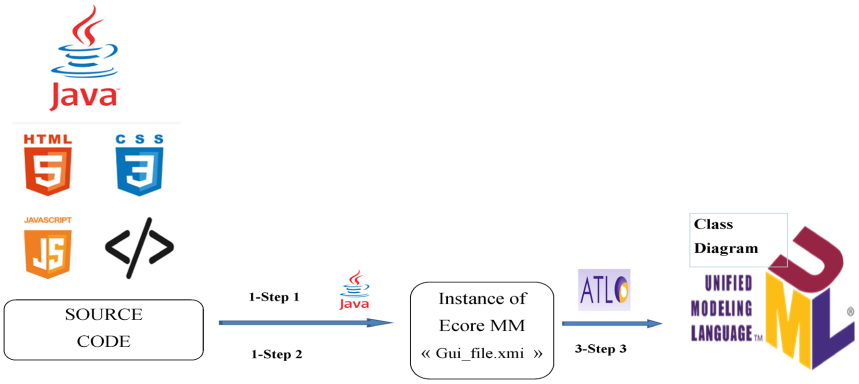


Figure 1. Overview of the proposed approach.

Step 1: Parsing source code with regex,

Step 2: creation of the Ecore metamodel instance,

Step 3: ATL transformation,

MM: Metamodel, ATL: Atlas Transformation Language, UML: Unified Modeling Language.

Figure 1 illustrates the methodology employed in this study. Initially, the syntax of the GUI source code is analyzed using regular expressions, a process conducted in Java. Subsequently, an instance of the Ecore metamodel is created in Java. Finally, the instance of the Ecore metamodel is transformed into a UML class diagram utilising ATL.

Figure 2 provides a more detailed explanation of the stages involved in the approach.

The three stages, as illustrated in Figure 2 will be discussed in greater detail below.

First step: The initial stage of the process is to parse the source code of the GUI using regex. In this phase, specific information from the GUI source code is extracted, including button names and window titles, through the use of regular expressions.

In the context of our project, regex functionality is employed for the purpose of identifying and subsequently retrieving the requisite information within the given

line of code. In order to detect the button, the following Java code with regular expressions is employed: String buttonRegex = "\\bJButton\\b\\s*\\w*\\s*=\\ w*\\s*=\\ w*\\s*=\\ s*JButton\\s*\\(";.

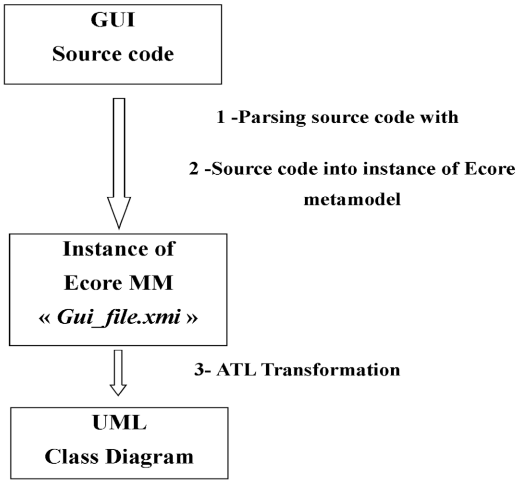


Figure 2. Three steps in the current approach.

Once we have extracted the line of code containing, for example, the declaration of a button, we can use the split() method to extract the useful information. Look at the following line of code: "JButton btn_Purchase = new JButton("Purchase");". To extract only the name of the "Purchase" button, we can use the split() method.

By employing the split function with the double quote character as the separator, specifically String[] parts = line.split("\""), an array is generated in which the elements between the quotes are isolated. Consequently, to retrieve the string between the quotes, the expression String buttonLabel = parts [1]; is utilized. Finally, the retrieved value can be displayed with the System.out.println("Button:" + buttonLabel.trim()) statement.

Second step: The second step is the creation of the Ecore metamodel instance. This stage involves the generation of instances of the Ecore metamodel based on the outcomes of the syntactic analysis conducted in the preceding stage. The final product will be a XML Metadata Interchange (XMI) file, in the .xmi format.

Third step: the third step is as follows, the graphical interface is transformed into a class diagram using the ATL transformation language.

ATL is a rule-based model transformation language [7] developed within the context of the Eclipse Modeling Framework (EMF) initiative. ATL enables the specification of transformations between models written in metamodels compliant with the Meta Object (MOF) Facility standard.

As part of this research, we use ATL to transform the result of the previous step, in particular the Gui_file.xmi file, into a class diagram. In fact, the Gui_file.xmi file obtained in the previous step is a metamodel that corresponds to the graphical interface model to be transformed. The transformation is then performed using this file as the source model.

The transformation of the graphical interface into a class diagram using ATL is shown in Figure 3.

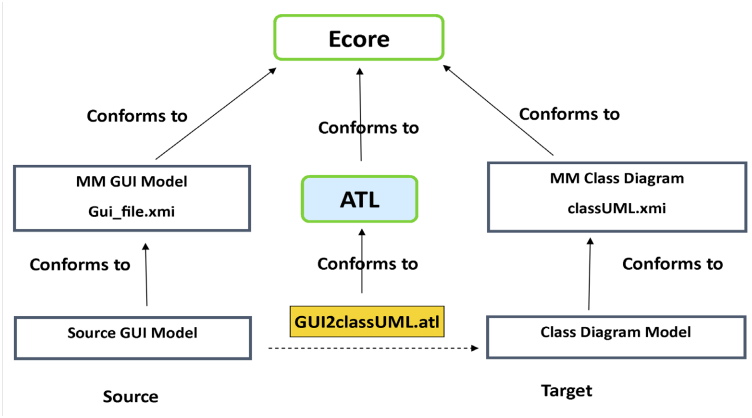


Figure 3. Transformation of the graphical interface into a class diagram using ATL. MM: Metamodel, ATL: Atlas Transformation Language, UML: Unified Modeling Language, GUI: Graphical User Interface.

Figure 3 illustrates that the source model aligns with the source metamodel, as represented by the Gui_file.xmi file. This modeling is based on the Ecore metamodel, which serves as the foundation for modeling within EMF.

2.5. Transformation Rules

Transformation rules are directives or criteria that are employed in order to facilitate the conversion of one model into another. In the context of ATL, transformation rules [18] are defined with the objective of establishing a consistent and precise association between the structures and properties of source models and those of target models. In this project, the transformation rule is typically applied during the parsing of the source code. ATL merely transforms the retrieved information into a UML class diagram.

In the case of this work, the source metamodel, Gui_file.xmi, contains parsing results obtained using regular expressions. The transformation rules comprise the recovery of elements from the source metamodel with a view to converting them into a class diagram.

The establishment of transformation rules is of paramount importance in the conversion of source models into target models that align with the requisite metamodels. In the process of transforming GUI source code into UML class diagrams, these rules serve to determine the manner in which elements extracted from the source code are represented in the UML diagram. In this context, the source model is the GUI source code, and the target model is the UML class diagram. The following section will provide a detailed overview of the transformation rules.

Rule 1: The names of the classes present on the graphical interfaces will be converted into class names on the product class diagram.

Rule 2: The textual content displayed on the JLabel of the graphical interfaces will be converted into class attributes on the product class diagram.

Rule 3: It is not possible to ascertain the attribute type either by examining the graphical interface or by parsing the source code. As a result, the attributes of the generated classes are defined in a systematic manner as follows:

- The Identifier (Id) and Quantity attributes will be of the integer data type.
- The Price attribute will be of the floating-point data type.
- All other attributes will be of the string data type.

Rule 4: The text on the buttons will be transformed into class methods on the product class diagram.

Rule 5: The relationships between classes in UML can be described as follows.

- Association: When one class contains a list of objects from another class, this indicates an association relationship. This means that instances of the first class have a reference to one or more instances of the second class.
- Aggregation: When a class contains a collection of objects from another class without managing their lifecycle, this relationship is called aggregation. In other words, the aggregated objects can exist independently of the aggregator object.
- Inheritance: When the "extends" keyword is used to define a class in terms of another class, this indicates an inheritance relationship. This means that the derived class inherits the attributes and methods of the base class.

Rule 6: If an "Add" button is present in the source code, as in the following example "JButton btnNewButton = new JButton("Add");", this generally indicates that the interface allows multiple items to be added to a collection or list within the class. Consequently, the multiplicity of these elements in the class can be inferred by "0…*". The line of code containing the button can be easily analyzed using regex. In addition, if a class contains a list (or other collection) from another class, this also indicates a multiplicity of "0…*" for the elements in that list. If neither of these conditions is met, the default multiplicity is 1.

2.6. Tools and Technologies

The main tools used to transform graphical interfaces into UML class diagrams are Java, Regular expressions and ATL.

Java is a high-level, object-oriented programming language that is widely used for the construction of applications. In this case study, the graphical interfaces utilized are Java Swing interfaces created with WindowsBuilder, an Eclipse plugin dedicated to the design of graphical interfaces. Java is employed to author the application source code, perform syntactic analysis of the GUI code and generate the Ecore metamodel instance.

A regular expression is defined as a sequence of characters that define a search pattern. They are employed extensively for the purposes of string matching and manipulation. In this context, regex is employed for the parsing of source code within graphical user interfaces, with the objective of extracting pertinent information such as class names, attributes, operations, relations and multiplicities.

ATL is a model transformation language and tool developed by the Eclipse Foundation. It permits the delineation and implementation of transformations between disparate models. In the present study, the objective is to transform the results of the syntactic analysis of the source code of graphical interfaces into UML class diagrams.

In addition to the aforementioned principal tools, we also employ the EMF [19] [20], the Eclipse Integrated Development Environment (IDE), and the Papyrus plugin for the purpose of visualizing the class diagrams that are produced.

3. Results

This study demonstrates the feasibility and efficacy of employing a combination of Java, regular expressions, and ATL to transform graphical user interfaces into class diagrams. This methodology contributes to the advancement of software systems engineering by providing a UML representation of graphical interfaces, which facilitates comprehension and maintenance of systems.

The transformation of the GUI source code into a class diagram is performed in three main steps: first, syntactic analysis of the GUI source code is performed using regular expressions; second, an instance of the Ecore metamodel is created; third, the transformation of the GUI into a class diagram is performed using ATL.

Syntactic analysis of the GUI source code is used to extract the elements required to construct the class diagram, which is shown in **Table 4**.

Table 4. Correspondence between graphical interface and class diagrams.

Graphical interface components	Class diagram elements
Class name or window title	Class name
Text on labels	Attributes
Button names	Methods
Correspondences between classes and between attributes	Relations
According to the transformation rules, existence of the "add" button	Multiplicity

Table 4 shows the correspondence between the graphical interface and the class diagram. The use of regular expressions to analyze the syntax of the source code proves effective in recovering the elements needed to construct the class diagram.

3.1. Analysis and Transformation of Java Source Code into UML Class Diagrams

This case study examines the process of transforming a graphical user interface (GUI) into a class diagram. The GUIs used are simple forms created by WindowsBuilder, which are described in greater detail in **Figure 4**.

The graphical interfaces in **Figure 4** include three forms relating to suppliers, customers and products.

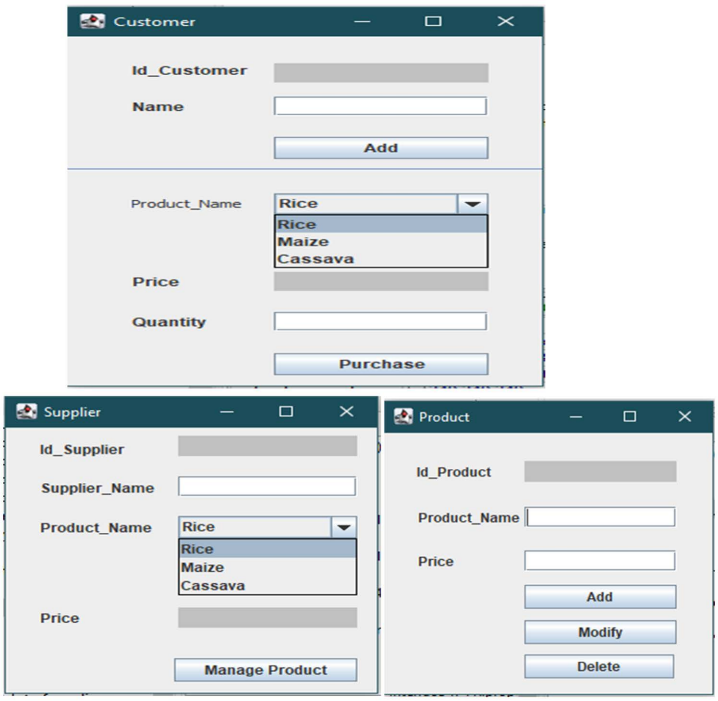


Figure 4. GUI to be analyzed.

Figure 5 illustrates the Java code employed for parsing the source code of the Customer.java GUI. And the result of utilizing regular expressions for the analysis of source code syntax is obtained through the application of the six aforementioned rules, as illustrated in Figure 6.

Customer.java ParsingClient.java

```
1  package Parsing;
2  import java.io.BufferedReader;
3  import java.io.FileReader;
4  import java.io.IOException;
5  import java.util.regex.Matcher;
6  import java.util.regex.Pattern;
7
8  public class ParsingClient {
9      public static void main(String[] args) {
10 String filePath = "C:\\Users\\Herifidy\\eclipse-workspace\\Project_Research\\src\\Interface\\Customer.java";
11
12          // expression régulière pour récupération de nom de classe java
13           String classRegex = "\\bclass\\s+(\\w+)|\\binterface\\s+(\\w+)|\\benum\\s+(\\w+)";
14           Pattern cpattern = Pattern.compile(classRegex);
15           try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
16               String textClass;
17               while ((textClass = reader.readLine()) != null) {
18                   Matcher matcher = cpattern.matcher(textClass);
19                   if (matcher.find()) {
20                     //  System.out.println("Button found at line " + lineNumber + ": " + textClass.trim());
21
22                    // Diviser la chaîne par le séparateur espace
23                       String[] parts = textClass.split(" ");
24
25                       // Si vous voulez récupérer uniquement la partie entre les guillemets
26                       if (parts.length > 1) {
27                           String className = parts[2];
28                           System.out.println("Class Name: " + className.trim());
29                       }
30
31                   }
32               }
33               } catch (IOException e) {
34               e.printStackTrace();
```

Figure 5. Example of Java code utilizing Regex for parsing Java source code.

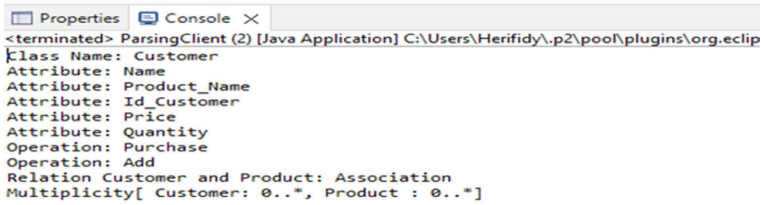


Figure 6. Result of parsing source code of the Customer.java GUI.

Figure 6 demonstrates that parsing the GUI source code is a pivotal step in transforming the GUI into a class diagram.

The creation of an instance of the ecore metamodel results in the generation of the GUI_file.xmi file. This file serves as a concrete representation of the GUI components and their relationships as defined by the metamodel. In particular, it encapsulates detailed information about the various GUI elements, such as buttons, text fields, and labels, along with their attributes. The structured format allows for the subsequent transformation and analysis processes, providing a standardized way to manage and manipulate the data extracted from the GUI source code.

The most intriguing outcome of this study is the class diagram, which is presented in Figure 7.

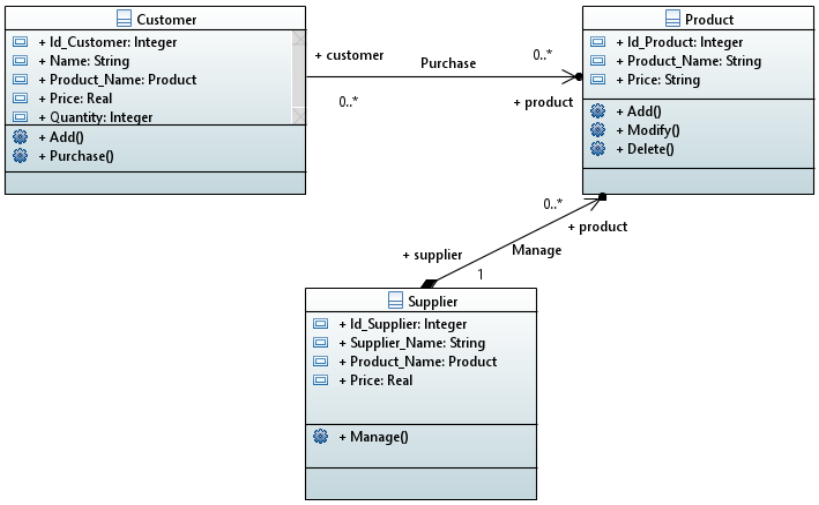


Figure 7. Result of transforming the graphical interface into a class diagram.

As illustrated in Figure 7, the diagram demonstrates the transformation of the graphical interface into a structural model with clarity. The classes identified, along with their attributes and methods and the relationships between them, are described in precise detail, thereby providing a complete overview of the system. The class diagram presented in Figure 7 was visualised using Papyrus, a UML-based modeling tool integrated into the EMF [19].

3.2. Analysis and Transformation of Web Source Code into UML Class Diagrams

The second case study, dedicated to the analysis and transformation of HTML

source code into UML class diagrams, demonstrates that the employed interfaces, designed using HTML and CSS, are not limited to simple forms, as illustrated in Figure 8.

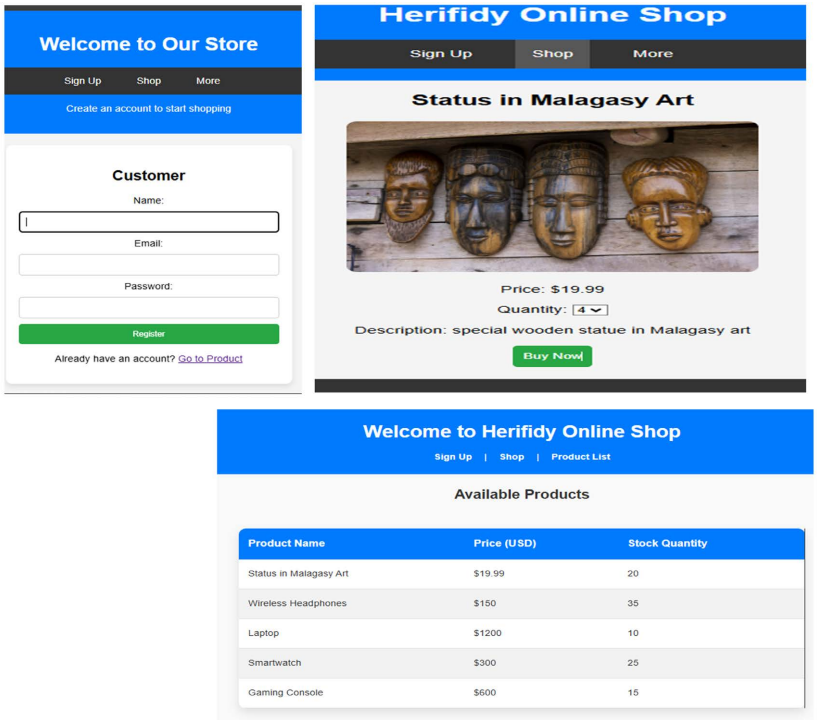


Figure 8. Web Graphical User Interface to be analyzed.

The three relevant web interfaces are dedicated to online sales and respectively address aspects related to clients and products. These graphical user interfaces comprise three pages: a registration page, a page dedicated to product purchases, and a page displaying the quantity of stock available in the database.

Parsing_html.java

```
public class Parsing_html {
    public static void main(String[] args) {
        try {
            // Lire le contenu du fichier HTML
            String html = readHtmlFile("C:\\Users\\Herifidy\\eclipse-workspace\\Parsing_html\\src\\Page\\shop.html");
            String html2 = readHtmlFile("C:\\Users\\Herifidy\\eclipse-workspace\\Parsing_html\\src\\Page\\signup.html");
            // Détection du nom de la classe (title)
            String className = extractData(html, "<title>(.*?)</title>");
            String className2 = extractData(html2, "<title>(.*?)</title>");
            // Extraction des labels
            List<String> labels = new ArrayList<>();
            // Regex pour capturer les labels du formulaire
            Pattern pattern = Pattern.compile("<label for=\"(.*?)\">(.*?)</label>");
            Matcher matcher = pattern.matcher(html2);

            // Détection des attributs
            boolean hasPrice = containsPattern(html, "[$€]\\d+(\\.\\d+)?"); // Verify $ ou €
            boolean hasName = containsPattern(html, "<*=\"name"); // Verify the presence of Name
            boolean hasqty = containsPattern(html, "<*=\"quantity"); // verify presence of Quantity
            boolean hasimg = containsPattern(html, "<img\\s*[^>]*src\\s*=\\s*['\"][^'\"]+\\.(jpg|jpeg|png|gif)['\"][^>]*>");
            // Détection des opérations
            String OperationName = extractData(html, "<button>(.*?)</button>");
            String OperationName2 = extractData(html2, "<button(?:\\s+[^>]*)?>(.*?)</button>");
            // Affichage des résultats
            System.out.println("Class name : " + className);
            if (hasName) System.out.println("Attribute : Name");
            if (hasPrice) System.out.println("Attribute : Price");
            if (hasqty) System.out.println("Attribute : Quantity");
            if (hasimg) System.out.println("Attribute : Image");
            System.out.println("Operation :" + OperationName.trim().replace(" ","_"));
            // Affichage des résultats 2
            System.out.println("--------------------------------");
            System.out.println("Class name2 : " + className2);
            while (matcher.find()) {
                labels.add(matcher.group(2).trim().replace(":", "")); // Capture le texte du label
            }
            for (String label : labels) {
                System.out.println("Attribute : " + label);
            }
```

Figure 9. Example of Java code utilizing Regex for parsing web page source code.

The analysis of the web source code of these interfaces using Regex yielded the results presented in Figure 9 and Figure 10.

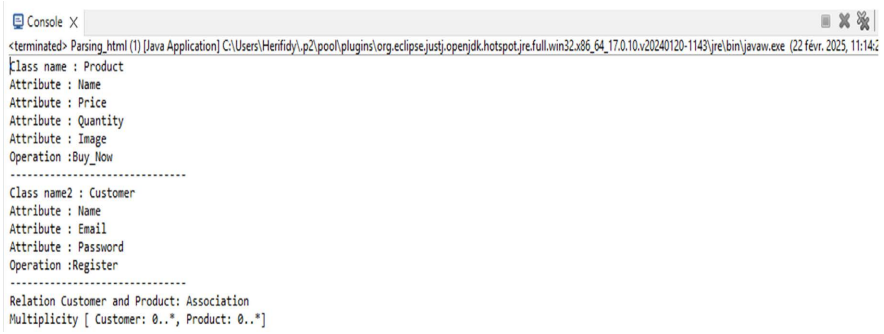


Figure 10. Results of web page source code analysis using Regex.

The results presented in Figure 10 highlight the elements necessary for the extraction of the class diagram, which is illustrated in Figure 11.

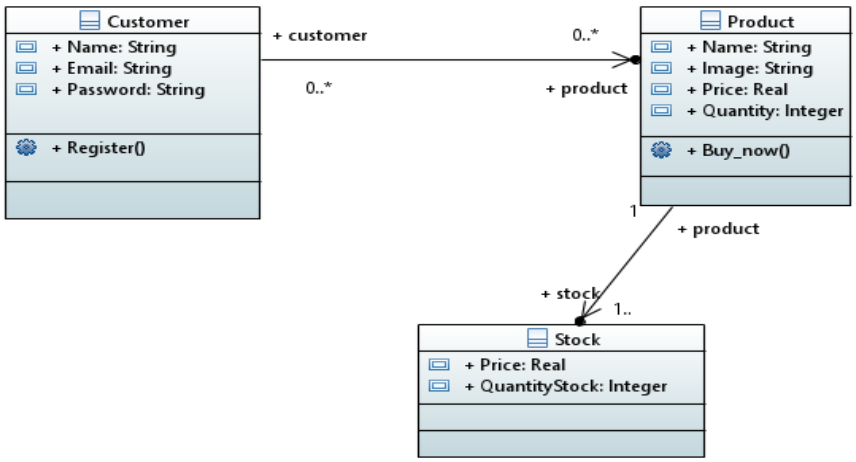


Figure 11. UML class diagram generated from web page analysis.

Similar to the analysis and transformation of Java source code presented in the previous case study, Regex facilitates the analysis of web page source code and its transformation into a class diagram. This process employs a principle analogous to ATL, yielding the class diagram illustrated in Figure 11.

4. Discussions

The GUI source code into a UML class diagram presents a number of advantages and challenges that warrant discussion. This case study examines the integrated utilization of Java, regex and ATL for model transformation. This approach is employed for the automation of the transformation of textual models into structural models, thereby facilitating a more comprehensive analysis of the software.

The utilization of Java in conjunction with regex to extract data from the GUI has been demonstrated to be an effective approach. The use of regex enables the precise searching and manipulation of text patterns, which is a crucial aspect for

the identification of pivotal interface components (such as buttons, text fields, and menus) and their associated properties. Furthermore, the ATL transformation language provides an exemplary of a straightforward transformation rule for the conversion of an interface component into a UML class. This rule can be extended to include additional properties, methods, and relationships between classes, thereby providing a comprehensive and accurate representation of the system's structural model.

This reverse engineering principle for the transformation of graphical components was implemented in the approach proposed by M. I. Muhairat (2011) [11], using Optical Character Recognition (OCR) and Petri nets. The approach [11] offers an advantage in the structured recognition of interface graphical components in image mode, through OCR, and identifies the correspondences of graphical components by means of Petri nets. However, this approach [11] is limited with respect to the identification of relationship types between classes and their multiplicities for the resulting class diagram. Nevertheless, the present approach offers the possibility to extract class diagrams from source codes written in various languages (Java, HTML, etc.), taking into account the types of relationships between classes and their multiplicities. The use of Regex allows for the analysis of graphical components within the source codes, such as buttons, labels, and text fields, and is not limited to simple forms.

In fact, the second case study, focusing on the analysis and transformation of HTML source code into UML class diagrams, highlights the power of Regex. These allow for the testing and extraction of information from elements present in web pages, such as, titles (<title> tag), meta-descriptions, the presence of monetary symbols (€ or $), regular expressions for testing email addresses and images, page redirections via buttons, database-linked table lists, and labels, among others.

The limitations of Regex in source code analysis can arise when there are no longer indices available to extract certain elements of the class diagram, even with highly sophisticated regular expressions. For example, it is possible that the attribute "Name" cannot be extracted from an HTML code if no "name" index is found, neither in the meta-description nor in other potential indices. This leads us to consider the utilization of Artificial Intelligence to extract class diagram elements from the context and the correspondence between terms present in the code and potential class diagram elements.

5. Conclusions

This study has demonstrated the efficacy of a methodology combining Java, regular expressions and ATL in the transformation of graphical interfaces into UML class diagrams. The integration of these tools and techniques enabled the automation and streamlining of the transformation process, thereby facilitating a more comprehensive understanding and documentation of the structure of software systems. The use of Java and regular expressions allowed for the precise extraction of graphical user interface components, thereby facilitating their transformation

into structured models. Subsequently, ATL afforded considerable flexibility in specifying transformation rules, ensuring the resulting models' compliance with system requirements.

The principal benefits of this methodology are the precision of data extraction facilitated by regular expressions, the adaptability of ATL for transforming extracted data into comprehensive class diagrams, and the automation of the transformation process, which reduces the time and effort required compared with a manual approach.

Our case studies are focused on the analysis of Java and web source codes; however, due to the power of Regex, which is programming language-independent, we foresee the potential for analyzing source codes from other languages.

Beyond class diagram elements, by the flexibility of Regex, it is possible to extract data. As demonstrated in our second case study, we can extract: product names, prices, stock quantities, and product descriptions, which are data that can be processed by other analyses such as data analysis, for example.

And the limitation of Regex when there are no longer indices available to extract certain elements of the class diagram, leads us to resort to other techniques such as Artificial Intelligence to infer the correspondence between terms present in the code and potential elements of the class diagram, according to the contexts.

Acknowledgements

I would like to extend my sincerest gratitude to all those who contributed to this work.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Dimbisoa, W.G., Mahatody, T. and Razafimandimby, J.P. (2018) Creating a Metamodel of UI Components in Form of Model Independant of the Platform. *International Journal of Conceptions on Computing and Information Technology*, **6**, 48-52.

[2] Crampes, J.-B. and Ferry, N. (2008) SNO: High Level Model for IHM Design and Mock-Up. *Electronic Journal of Information Technology*, **5**, 1-18.

[3] Ferry, N. (2008) Formalisation of MACAO Method Models and Development of a Software Engineering Tool for the Creation of Man-Machine Interfaces. Ph.D. Thesis, University of Toulouse.

[4] Telea, A.C. (2012) Reverse Engineering—Recent Advances and Applications. IntechOpen. https://doi.org/10.5772/1850

[5] Mahfoudhi, A., Bouchelligua, W., Abed, M. and Abid, M. (2006) Towards a New Approach of Model-Based HCI Conception. *Proceedings of the 6th WSEAS International Conference on Multimedia, Internet & Video Technologies*, Lisbon, 22-24 September 2006, 117-125.

[6] Lucas, F.J., Molina, F. and Toval, A. (2009) A Systematic Review of UML Model Consistency Management. *Information and Software Technology*, **51**, 1631-1645. https://doi.org/10.1016/j.infsof.2009.04.009

[7] Jouault, F., Allilaire, F., Bézivin, J. and Kurtev, I. (2008) ATL: A Model Transformation Tool. *Science of Computer Programming*, **72**, 31-39. https://doi.org/10.1016/j.scico.2007.08.002

[8] Saraiva, J.A., Campos, J.C., Silva, J.C. and Silva, C. (2012) GUIsurfer: A Reverse Engineering Framework for User Interface Software. In: Telea, A.C., Ed., *Reverse Engineering—Recent Advances and Applications*, IntechOpen.

[9] Silva, J.C., Silva, C.C., Gonçalo, R.D., Saraiva, J. and Campos, J.C. (2010) The GUISurfer Tool: Towards a Language Independent Approach to Reverse Engineering GUI Code. 2*nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, Braga, June 2010.

[10] Favre, L. (2012) MDA-Based Reverse Engineering. In: Telea, A.C., Ed., *Reverse Engineering—Recent Advances and Applications*, IntechOpen. https://doi.org/10.5772/32473

[11] Muhairat, M.I., AL-Qutaish, R.E. and Athamena, B.M. (2011) From Graphical User Interface to Domain Class Diagram: A Reverse Engineering Approach. *Journal of Theoretical and Applied Information Technology*, **24**, 28-40.

[12] Mithe, R., Indalkar, S. and Divekar, N. (2013) Optical Character Recognition. *International Journal of Recent Technology and Engineering* (*IJRTE*), **2**, 72-75.

[13] Johnson, S.C. (1986) YACC: Yet Another Compiler-Compiler. Bell Laboratories.

[14] Levine, J.R. (2009) Flex & Bison: Text Processing Tools, O'Reilly Media, Inc.

[15] Latif, A., Azam, F., Anwar, M.W. and Zafar, A. (2023). Comparison of Leading Language Parsers—ANTLR, Javacc, Sablecc, Tree-Sitter, Yacc, Bison. 2023 13*th International Conference on Software Technology and Engineering* (*ICSTE*), Osaka, 27-29 October 2023, 7-13. https://ieeexplore.ieee.org/abstract/document/10366650 https://doi.org/10.1109/icste61649.2023.00009

[16] Vinju, J. (2005) Analysis and Transformation of Source Code by Parsing and Rewriting, UvA-DARE (Digital Academic Repository). University of Amsterdam.

[17] Dewi, L.C., Meiliana, and Chandra, A. (2019) Social Media Web Scraping Using Social Media Developers API and Regex. *Procedia Computer Science*, **157**, 444-449. https://doi.org/10.1016/j.procs.2019.08.237

[18] Tisi, M., Martínez, S. and Choura, H. (2013) Parallel Execution of ATL Transformation Rules. In: Moreira, A., Schätz, B. and Gray, J., Eds., *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 656-672. https://doi.org/10.1007/978-3-642-41533-3_40

[19] Brun, C. and Pierantonio, A. (2008) Model Differences in the Eclipse Modelling Framework. *UPGRADE, the European Journal for the Informatics Professional*, **9**, 29-34.

[20] Arendt, T. and Taentzer, G. (2013) A Tool Environment for Quality Assurance Based on the Eclipse Modeling Framework. *Automated Software Engineering*, **20**, 141-184. https://doi.org/10.1007/s10515-012-0114-7