From Design to Code: An Educational Approach

Candice Eckert*, Brian Cham*, Jing Sun† and Gillian Dobbie†

*Department of Electrical and Computer Engineering
†Department of Computer Science
The University of Auckland, New Zealand
Emails: *{ceck002, bcha899}@aucklanduni.ac.nz, †{jing, gill}@cs.auckland.ac.nz

***Abstract*—Model Driven Engineering (MDE), despite having many advantages, is often overlooked by programmers due to lack of proper understanding and training in the matter. This paper investigates the advantages and disadvantages of MDE and looks at research results showing the adoption rates of design models. In light of the findings, an educational tool, namely Lorini, was developed to provide automated code generation from the design models. The implemented tool consists in a plug-in for the Astah framework aimed at teaching Java programming to students through UML diagrams. It features instantaneous code generation from three types of UML diagrams, code-diagram matching, a feedback panel for error displays and on-the-fly compilation and execution of the resulting program. Evaluation of the tool indicated it to be successful with unique educational features and intuitive to use.**

I. Introduction

The development of a software product includes many steps such as requirement analysis, design, coding and testing. The design phase usually involves formal design models. A more efficient alternative to manual coding is automated code generation from design models. This approach is known as Model Driven Engineering (MDE). The main advantages of MDE are its efficiency in terms of development time[1] and the reduction of human errors. This last property plays an important role in real-time embedded systems, where the slightest mistake could lead to lethal consequences[2]. However, the structure of auto-generated code is more complex than manual code, making it harder to understand and re-use[3]. It is also less efficient in terms of number of calls per executable statement, number of executable statements, and time taken by stack allocation/deallocation[1].

The main issue regarding MDE remains the low adoption rate. Only 11% of programmers often use formal design models, others using it scarcely, for communication purposes only, or not at all[4]. This is mostly due to a lack of understanding and training in the matter. An experiment conducted on a software design class showed that most students did not use MDE before the course, and 70% of them found MDE to be useful in order to understand software design[5]. By combining these research results, the emerging factor seems to be the lack of educational tools for programmers to learn formal design modelling. This has the unfortunate effect of reducing the adoption rate of MDE, even though many improvements could be introduced by using design models.

DOI reference number: 10.18293/SEKE2016-007

Giving a new direction to the research, it was found that teaching modelling before programming is feasible and even beneficial, considering that modelling is applicable to different disciplines and can favour a more structured learning[6]. It is suspected that complex object oriented concepts could be more easily grasped by students if they were introduced through design models. As a consequence of these findings, we decided to develop an educational tool for students to learn object oriented programming starting from formal design models, specifically learning Java from UML. An overview block diagram of the implemented tool is shown in Figure 1.

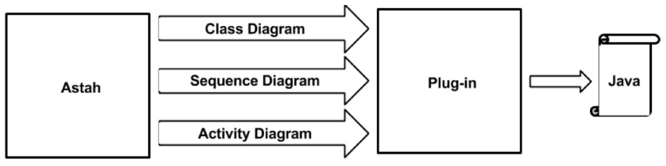


Fig. 1. Overall Approach to UML to Code Generation

The tool adopts the approach described by Rivera-Lopez et al.[7] in teaching the foundations of object oriented programming using UML. We make use of three types of UML diagrams, namely class, sequence and activity diagrams, and the following generation steps:

- Design the class diagram showing the relationships among the classes, and generate the Java class skeleton code from the class diagram.
- Design the sequence diagram describing the method calls from one class to another, and generate the main class (the main program) using the sequence diagram.
- Design the activity diagram corresponding to the code of each method, and generate the methods of the Java class from the activity diagrams.

It allows the automatic generation of executable Java program from UML design models, and visually highlights the relationships between the design model and the code. It is important to note that the implemented tool is not intended to be used in place of a programming course but rather as a supplementary learning aid aiming to facilitate the students' understanding of the subject. There are many different systems and plug-ins that perform code generation from design models.

After a brief analysis of thirty-five of them, a number of free, open-source tools were extracted and investigated in more depth. Some tools were found to be too complicated to use or difficult to install due to lack of documentations. With careful consideration, the Astah framework was elected as the best option. Its interface is user-friendly and easy to understand for intuitive users. A clear set of APIs is available, making it easily extendible. Moreover a free licence is dispensed to students, thus facilitating both the design and evaluation processes. Astah is available on Windows, Linux and Mac OS X, and the plug-ins are easy to install. The tools used were Eclipse IDE for code development and a Github repository for version control, helping with collaboration and synchronization. The outcome of the project was an interactive plug-in for the Astah modelling framework.

The rest of the paper was structured as follows. Section II discusses the findings of related research in the field, including surveys and existing solutions. In section III, we outline the objectives and technical decisions behind the plug-in to explain the rationale of the project. Section IV describes the implementation details of the plug-in to explain how it was developed. In section V, we list the criteria and conduct evaluations to assess the success of the plug-in. Finally, section VI concludes the paper and outlines the future work.

II. Literature Review

The development method of automatically generating software from design models is referred to as Model-Driven Engineering. In this practice, the design of software is fully specified in a standard modelling notation before executable program code is derived [2]. Often, changes in the design model or source code will result in an instant update of the other representation. In some implementations, the software may be changed in real-time while it is running by using metamodels, specifications for how one software model maps onto a different one[8]. To further understand the field and guide conceptual development of the project, four research questions were pursued as follows.

What are the key benefits of Model-Driven Engineering? Becucci, et al. [1] report that when implemented and used well, the automatic code generation process could aid development efficiency by reducing coding time. This in turn frees up time to spend on design, simulation, validation and testing. The use of implementation-independent design models ensures that software is interoperable between systems [1]. The conversion can also avoid human error and guarantee correctly functioning code, which is especially important for embedded systems and safety-critical applications [9].

What are challenges in Model-Driven Engineering? This question was investigated to identify key areas that the project may contribute to. Clarke et al. [5] educational experience highlighted some problems in the field. The most salient is the scarcity of software modelling education in the first place, which leads to low modelling skills in graduates. When they were teaching Model-Driven Engineering to university students, the main problems were related to the software infrastructure. They bemoaned the lack of dedicated pedagogical tools for teaching Model-Driven Engineering in particular. The majority of the students felt that the software they used (EMF), while effective, was too difficult to learn and needed more explanations, documentation and tutorials.

What are the most common design models used in industry? Gorschek et al. [4] survey of over three thousand developers showed that most respondents (76.2%) did not use any design models at all. This was followed by personal informal notation (10.9%), UML (7.9%), then all others (5%). Most software modelling was used only for communication, co-ordination or temporary brainstorming. Uptake of UML was unexpectedly low; respondents felt it was unnecessary, too big and too complicated. Even though Java is a widely used object-oriented program and can map directly to UML class diagrams, usage of UML was still low amongst Java developers too.

What other solutions exist? In order to gauge the benefits and drawbacks of current solutions, thirty-five programs and plug-ins were informally investigated by looking at their documentation and reviews. Out of those, eight were identified as free, open source and possibly extendible – Astah, AnyCode, Eclipse Epsilon, Open ModelSphere, RISE Editor, Sirius, TASTE and Yakindu Statechart. Each of these eight were personally installed and evaluated. The research and experience with these programs showed that most common problems were related to usability and convenience. Many could not be installed because of dependency issues or lack of installation instructions. Those that could be installed tended to have very complex interfaces, a scarcity of basic tutorials, vague error messages and no helpful documentation. Some even required learning a specialised language to use.

The general conclusion was that Model-Driven Engineering had intrinsic advantages but adoption was too low for industry to benefit from these. Developers generally do not see the benefits of software modelling, which may arise from a good introduction at the educational level. For the development of the project itself, Astah was identified as the easiest automatic code generation software to install, use and extend.

III. Technical Design

A. Project Scope

After research and discussion, the project scope was finalised as an educational plug-in to teach Object-Oriented Programming using automatic code generation with UML and Java, at the earliest stage of programming education. This was chosen for three main reasons:

Firstly, the consistent use of design models and UML in industry is very low. This has been attributed to a lack of good educational tools that can persuasively teach the usage and importance of software design modelling [5]. The evaluation of existing solutions revealed that they are only suited for those who are already deeply familiar with the practice of model-driven development, and do not cater for beginners.

Secondly, the usage of software modelling is very low for Java developers in particular, despite the potential benefits [4].

Teaching UML with Java as a well-known reference language may help users to understand the simple link between Java code constructs and the design model elements.

Thirdly, personal experience suggests that students find Object-Oriented Programming difficult to conceptualise when introduced. This may be remedied by starting Object-Oriented Programming education with the higher level of abstraction found in visual design models, which Starett [6] showed was feasible as early as high school.

B. Software Used

The project was developed in the form of a plug-in for an existing application. This was decided to avoid "reinventing the wheel", especially with a constrained time frame. The selected code generation application was Astah, because of the following factors identified in the solution investigations:

- Astah is free to download, meaning it is easy to obtain for the developers and users.
- Astah is compatible with all three major operating system families – Windows, Mac OS X and Linux.
- Astah is easy to install as a plug-in on any platform. This process requires only a single drag-and-drop operation.
- Astah is open source and has a free, fully documented API. This makes it possible to create new extensions of the application.
- Astah's API supports a myriad of potential features and data queries.

C. Key Features

Rivera-Lopez, et al. [7] outlined a successful educational methodology in teaching programming through design model, which formed the basis of the project. The rough steps were, in this order: 1) Design a class diagram from a description of the problem, 2) Design a sequence diagram to determine messages between the objects, 3) Design an activity diagram to specify the internal logic of objects' methods, 4) Design Java class code from the class diagram, 5) Design Java Main class from method calls in the sequence diagram, and 6) Design method based on activity diagrams. The student becomes aware of the simple, one-to-one relationship between the visual and textual languages.

Our project supports this educational approach by implementing a software tool for realising these steps. The UML diagrams and equivalent Java code are displayed simultaneously. The code automatically updates upon each change to the diagrams, in real-time. For ease of understanding, elements and changes in both views can be colour coded to visually establish the links between design model diagrams and source code. Constant feedback is available to guide the user. At the end, if the diagrams have been constructed correctly, the generated program can be compiled and executed. These desired features were all deemed to be possible with the Astah API which allows for access to diagram details, editing of diagrams and standard Java Swing components.

Throughout, it helps learners take their first steps by avoiding complex terminology (e.g. "polymorphism") or complex programming concepts. The interface complexity is also restricted to a minimum to avoid overwhelming or confusing new learners. The plug-in is not intended to function as a fully self-contained educational experience. It is fundamentally a flexible tool to be used in conjunction with customisable teacher exercises. It includes a simple tutorial that explains the usage, for any interested readers who wish to try it out[1].

IV. Implementation

A. Tool Overview

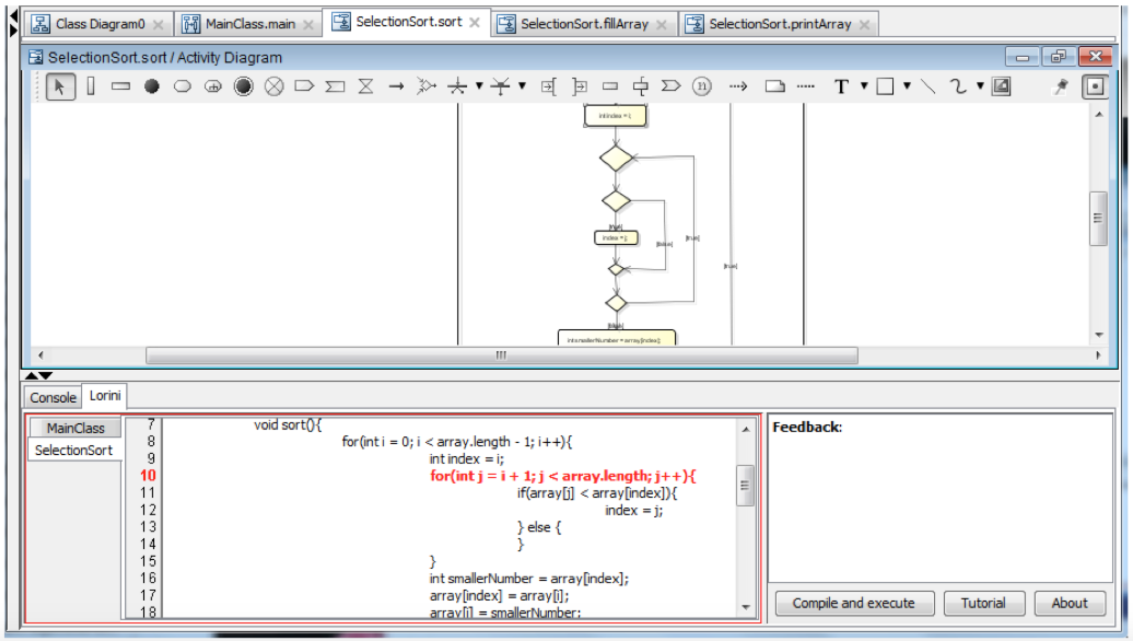
The tool development used Windows Command Prompt to build and launch the base Astah application, Eclipse IDE to code the plug-in itself and Github for back-ups and collaboration. The regular Astah interface includes a main diagram panel in the centre, a project panel on the left and a plug-in panel at the bottom. The project panel features a list of UML diagrams in the open project. Each one can be clicked to show the diagram in the main diagram panel, where they can be created, edited and deleted with reference to the underlying software model.

The implemented tool consists in a plug-in for Astah, as shown in Figure 2. It allows the automatic generation of Java code from UML diagrams, i.e., class, sequence and activity. The interface appears inside the plug-in panel at the bottom, which contains the interface of any loaded plug-in. The majority of the space is taken up by the text of the generated code. One class is shown at a time, and the user can navigate between classes using tabs. On the right is a small feedback section which lists errors to the user. In the corner is a button to compile and execute the code in a pop-up window, another to view the tutorial and another to view information about the plug-in itself. The plug-in automatically generates code from three types of UML diagram in the project – class diagrams for the class skeleton code, activity diagrams for the method contents (including if-branches and while-loops) and sequence diagrams for method calls between classes.

The code is generated as soon as the diagram is modified, providing the user with a fast, real time learning experience. Each Java file is represented by a tab displaying the class name. This allows for a clear and easy way to switch between files. The tool helps the user's understanding of the code by matching code and diagrams: when selecting an element in a diagram, the corresponding line of code is highlighted in red. Conversely, when clicking on a line of code, the corresponding diagram element gets selected. If the relevant file or diagram is closed, it is automatically opened in order to facilitate the transition. The code updates in real-time, i.e. every time something in the diagram changes.

At all times, any errors in the diagrams will be described in the feedback section, e.g. if an activity diagram is missing a final node, if a sequence diagram contains a call to a non-existent method, or if a class diagram contains an attribute with the reserved Java keyword "if". After creating a project using

[1]The developed tool, namely - Lorini, is available online for reviewing at https://briancham1994.wordpress.com/portfolio/lorini/.



```
void sort(){
        for(int i = 0; i < array.length - 1; i++){
                int index = i;
                for(int j = i + 1; j < array.length; j++){
                        if(array[j] < array[index]){
                                index = j;
                        } else {
                        }
                }
                int smallerNumber = array[index];
                array[index] = array[i];
                array[i] = smallerNumber;
```

Fig. 2. Graphical User Interface of the Lorini Tool

UML diagrams, the user can click on a button in a corner to compile and execute the generated code, and see the results or compilation errors if any.

A feedback panel on the left hand side of the code panel displays explanations about the errors made by the user, helping the user understanding their mistakes and correcting them. At the bottom of this feedback panel, a button is used to launch the compilation of the Java code. Upon compilation, the output of the program is displayed, or if need be, the compilation errors arisen.

B. Technical Details

The code generation was implemented by retrieving information contained in the UML diagrams using the Astah's API and editing the diagrams as needed. In the case of class diagrams, the class name, attributes and methods are obtained and stored in string format. Each one in the open project is retrieved as an "IClassDiagram" object using the Astah API. These contain references to its name, attributes, methods, superclasses, subclasses and more. The details of these elements were extracted and put into the right locations in a String along with necessary brackets and tabbing. Each String represented the code contents of each class, and they were displayed in ScrollPanes. These were contained within a TabbedPane which allows users to switch between each class by clicking on a tab. These interface elements use regular Java Swing components, though the ScrollPanes have been extended to allow for line numbers.

In a sequence diagram, lifelines are the graphic representations of each instance of a class and messages correspond to method calls. The main method is generated from a sequence diagram by analysing the messages sent between the lifelines. The tool checks that an instance of the class has been created before calling its methods, and displays an error message if needed. In the case of synchronous messages, corresponding to non-void method calls, an error message is displayed if no return message is present. Sequence diagrams having been restricted to depicting the main method, an error is also displayed if a lifeline other than the class containing the main method is sending a message to another lifeline, i.e., trying to call a method from another class. Similarly, the following situations will result in an error being displayed:

- Multiple sequence diagrams created.
- Name of the diagram not following the convention "class.method".
- Lifeline missing for the main class.
- Multiple lifelines created for the main class.
- Invalid lifeline created.

The activity diagram was harder to convert to Java code because of the non-linear nature of the code structure, which can include if-else statements and loops. The first step was to retrieve the correct order of the statements, which was done by following the "flows", i.e. the arrows linking each node to the next, and storing the nodes in a tree. Then, each statement was extracted and stored to be printed. However it was necessary to insert lines of code such as "else {" or closing brackets "}" as well as re-ordering the if-else statement by inserting the "if" part before the "else". This was done by analysing the incoming and outgoing flows and looking for the "true" and "false" guards. Extra care had to be taken in the cases where one of the branches was empty.

One of the main difficulties was to be able to differentiate between a loop and an if-else statement, given that they both use the true and false guards in the exact opposite manner,

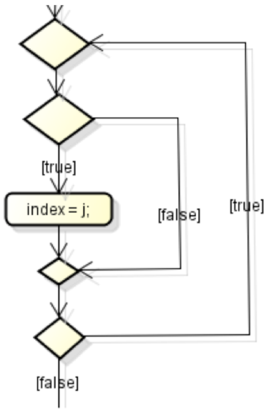


Fig. 3. Example of Loop Containing If Statement

as shown in Figure 3. Error checking had to be implemented to detect errors such as secluded nodes, if-else statements or loops missing a "true" or "false" guard and incorrect use of node types. Those errors are displayed in the feedback panel in order for the user to acknowledge and correct them, improving their learning experience.

Another example is the feedback section on the right which tells the user if there is anything wrong. This is implemented with an extended JScrollpane called "ErrorPane". It is highly encapsulated and any other class can add an error message with only a reference to the ErrorPane object, and without having to deal with formatting. It simply has to call the method "error (String s)" with the message to display, and ErrorPane will automatically append it to the end of a single formatted bullet list.

V. Evaluation

In order to evaluate the quality of the implemented tool, two types of evaluation methodologies were carried out, i.e., comparison with existing tools and user evaluation.

A. Tool Comparison

The implemented tool was compared to existing tools against the following criteria:

- Compatibility and portability
- Generating skeleton code from class diagrams
- Generating method code from activity diagrams
- Generating interaction code from sequence diagrams
- Code and diagram matching
- Instantaneous code generation
- Code compilation and execution
- Generating code in multiple programming languages
- Generating code from design models directly and automatically, without requiring the user to learn a dedicated language, diagram or syntax.

The implemented tool was found to be competitive on many aspects. Firstly, being portable and free for students, it is very accessible. It also benefits from an intuitive interface and uses only common standards such as Java and UML, avoiding the overhead of having to learn a specific language or diagram syntax. The educational aspect of the tool makes it unique compared to its competitors, allowing users to match diagrams and code with the help of highlighting and instantaneous code generation. Finally, the availability of feedback and compilation tools yields a better understanding of the process.

Partial comparison results [2] against a set of common UML tools can be found in Figure 4. Overall, the comparison highlighted the factors that make the project unique amongst similar tools:

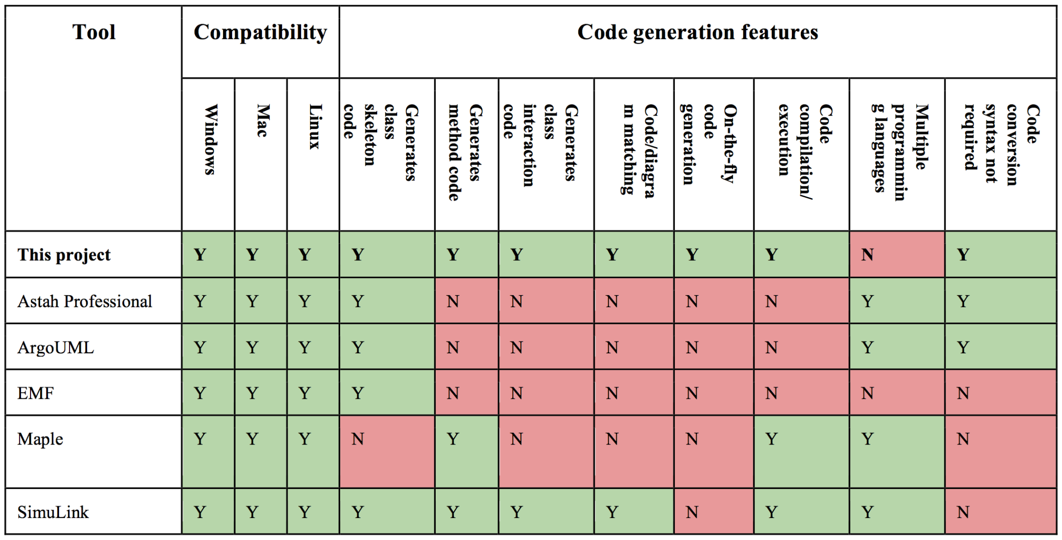
- **Convenient** – It is free of charge and compatible on all platforms.
- **Intuitive to use** – It does not require learning any special syntax or diagrams to perform the code generation. It uses standard UML and Java, not a dedicated language or format.
- **Educational** – It performs automatic highlighting of match between code and diagram, and vice-versa, which constantly update to reflect each other (i.e. on-the-fly code generation). This allows users to understand and explore at their own pace. Other tools assume that the user is already familiar with this relationship and does not help them to learn it. The ones that include these features either have limited functionality, or use their own formats instead of standards like UML and Java, limiting their educational use.
- **Feedback** – It tells the user basic details of anything wrong with the software model. It can also compile and check results of the code.

B. User Evaluation

User testing was performed in order to get a first-hand feedback on the implemented tool. The eight volunteers were aged 18 to 25 and included both males and females, experienced and neophytes in terms of programming and UML design. One of them had previous experience with Astah. Participants were asked to perform some basic tasks with the plug-in such as designing a "Hello World" program using the three supported types of UML diagrams. They were presented with a set of Likert-scale based questions[10]. For an easier analysis, the Likert items were converted to a numerical scale, 1 corresponding to "strongly disagree" and 5 corresponding to "strongly agree". The results were then averaged and are presented in Figure 5.

In average, the plug-in was considered very intuitive and uncluttered. Users found the code-diagram matching useful in order to understand the design process. However, some bugs were uncovered and some of the feedback messages were judged to be ambiguous and confusing. Moreover, the need for a tutorial was noticed. Following the evaluation results, improvements were made to the plug-in. A full testing session was carried out to identify and fix a plethora of remaining bugs. All problematic feedback messages were rewritten until users found them clearer. Thorough testing was performed and

[2]More detailed tool comparisons are available on the Lorini web page.



Tool	Compatibility			Code generation features							
	Windows	Mac	Linux	Generates class skeleton code	Generates method code	Generates class interaction code	Code/diagram matching	On-the-fly code generation	Code compilation/ execution	Multiple programming languages	Code conversion syntax not required
This project	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**N**	**Y**
Astah Professional	Y	Y	Y	Y	N	N	N	N	N	Y	Y
ArgoUML	Y	Y	Y	Y	N	N	N	N	N	Y	Y
EMF	Y	Y	Y	Y	N	N	N	N	N	N	N
Maple	Y	Y	Y	N	Y	N	N	N	Y	Y	N
SimuLink	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	N

Fig. 4. Tool Comparison Results

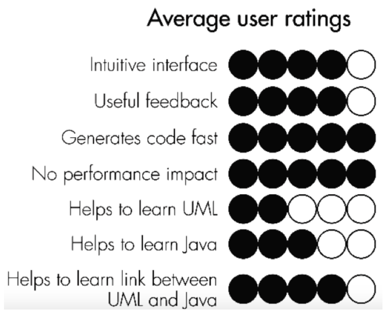


Fig. 5. User Evaluation Results

all uncovered bugs were fixed. The full tutorial was embedded into the plug-in to help with the understanding of the tool.

VI. Conclusion

While Model Driven Engineering is proven to be time-efficient and less prone to human errors, research showed that design models are scarcely used, mostly due to lack of education and training in the matter. In parallel, it has been shown that teaching design modelling to students before programming is feasible and even beneficial. These formed the basis of an educational approach that uses automatic code generation to teach Object-Oriented Programming and its relationship with design models. The implemented tool is a plug-in for Astah able to automatically generate Java code from three types of UML diagrams, namely activity, sequence and class diagrams. The tool includes the following features:

- Instantaneous code generation
- Code-diagram matching
- Feedback panel displaying user errors
- Compilation and execution of the code

Evaluation of the tool indicated it to be competitive in many aspects, mainly due to its unique educational features, as well as being easy to install and very intuitive to use.

In the future, we plan to conduct a large scaled user evaluation and focus more on measuring the education aspects of the developed tool. On the technical side, we would like to extend the current implementation to support the code generation in different programming languages, and from other types of UML diagrams.

References

[1] M. Becucci, A. Fantechi, M. Giromini, and E. Spinicci, "A comparison between handwritten and automatic generation of c code from sdl using static analysis," *Software: Practice and Experience*, vol. 35, no. 14, pp. 1317–1347, 2005.

[2] G. Nisha, "A model driven approach for design and development of a safety critical system," in *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, vol. 4, April 2011, pp. 15–18.

[3] H. Zhu, J. Sun, J. S. Dong, and S.-W. Lin, "From verified model to executable program: the pat approach," *Innovations in Systems and Software Engineering*, vol. 12, no. 1, pp. 1–26, 2015.

[4] T. Gorschek, E. Tempero, and L. Angelis, "On the use of software design models in software development practice: An empirical investigation," *J. Syst. Softw.*, vol. 95, pp. 176–193, Sep. 2014.

[5] B. Tekinerdogan, "Experiences in teaching a graduate course on model-driven software development," *Computer Science Education*, vol. 21, no. 4, pp. 363–387, 2011.

[6] C. Starrett, "Teaching uml modeling before programming at the high school level," in *Advanced Learning Technologies, 2007. ICALT 2007. Seventh IEEE International Conference on*, July 2007, pp. 713–714.

[7] R. Rivera-Lopez, E. Rivera-Lopez, and A. Rodriguez-Leon, "Another approach for the teaching of the foundations of programming using UML and Java," in *Proceedings of the 3rd WSEAS International Conference on Computer Engineering and Applications*, ser. CEA'09. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2009, pp. 279–283.

[8] F. Krichen, B. Hamid, B. Zalila, M. Jmaiel, and B. Coulette, "Development of reconfigurable distributed embedded systems with a model-driven approach," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1391–1411, 2015.

[9] M. Hinchey, J. Rash, and C. Rouff, "Requirements to design to code: Towards a fully formal approach to automatic code generation," Technical Report TM-2005-212774, NASA Goddard Space Flight Center, Greenbelt, MD, USA, Tech. Rep., 2004.

[10] O. Laitenberger and H. M. Dreyer, "Evaluating the usefulness and the ease of use of a web-based inspection data collection tool," in *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, Nov 1998, pp. 122–132.