# Reducing the Gap between OOP Design and Development

**Minjie Hu**
**School of Business & ICT, UCOL**
**Palmerston North, New Zealand**
**m.hu@ucol.ac.nz**

**Aaron Steele**
**School of Business & ICT, UCOL**
**Palmerston North, New Zealand**
**a.r.steele@ucol.ac.nz**

## ABSTRACT

Unified Modelling Language class diagrams are commonly used as a "bridge" between user and developer as well as between problem and solution in teaching object-oriented programming (OOP). In order to reduce the gap between design and development in the teaching OOP in C#, this research focuses on exploring students' perspective of generating class code from class diagrams using a UML tool, Visual Paradigm. The findings suggest that it is beneficial to introduce class code generation from class diagrams into OOP course curriculum.

**Keywords**: object-oriented design, Unified Modelling Language (UML), Class Diagram, object-oriented programming (OOP)

## 1. INTRODUCTION

Model-driven development (MDD) or Model-driven architecture (MDA) is promising as a paradigm for developing object-oriented programming (OOP) quickly and effectively. Specifically, the Unified Modelling Language (UML) class diagram plays the role as a "bridge" between user and developer (Nikiforova, Sejans, & Cernickins, 2011). Subsequently, many educators introduced "*design first*" or "*modelling first*" using UML class diagram in teaching OOP (Moritz & Blank, 2005; Wei, Moritz, Parvez, & Blank, 2005; Georgantaki & Retalis, 2007). Furthermore, some researchers took advantage of UML tools to generate program code (Georgantaki & Retalis, 2007; Carlisle, 2009; Usman & Nadeem, 2009; Lethbridge, Mussbacher, Forward, & Badreddin, 2011). In these researches, there is a common consensus that using a UML tool makes it easy for students to understand and modify modelling diagrams. However, the usability of generated code is concerned in teaching practice.

In order to reduce the design and development gap in our teaching of OOP in C#, this research focuses on exploring students' perspective of generating class code from class diagrams using a UML tool, Visual Paradigm (VP). We believe that a picture is worth a thousand words. Subsequently, our hypothesis is that through the integration of class diagram and code generation students would learn the connection between the diagram and its relevant code through comparison of both of them as well as in the further implementation. Therefore, our research questions are:

*RQ1: How does the integration of OOP class design and program code development support student understanding of the concepts of class inheritance in C#?*

*RQ2: How to evaluate the integration of class diagram and program code supporting students' understanding of various concepts in class inheritance and program development?*

In the following sections, we firstly explore the literature relating to the use of UML tools in teaching. We then present how we use a UML tool in our course. Next, we discuss our findings through this case study. Finally, we make conclusion of this study.

## 2. LITERATURE REVIEW

This quality assured paper appeared at the 8th annual conference of Computing and Information Technology Research and Education New Zealand (CITRENZ2017) and the 30h Annual Conference of the National Advisory Committee on Computing Qualifications, Napier, New Zealand, October, 2-4, 2017. Executive Editor: Emre Erturk. Associate Editors: Kathryn MacCallum and David Skelton.

### 2.1 Why Using C# and Class Diagrams

Java has been widely used in teaching OOP in the last two decades. Through comparison, Raoufi and Maniotes (2005) believed that "*using C# as first programming language overcomes these and other difficulties associate with Java… provides a more proper language and environment to teach introductory course for beginning students*" (p. 6). Considering C# replacing Java, Reges (2002) suggested:

> *"C# has tremendous overlap with Java which means that as a language it will probably be as effective as Java in teaching CS1 and CS2. It fixes many of the shortcomings of Java that have been particularly difficult for novice programmers and it provides extra features that have the potential to enrich the course" (p. 5).*

The UML class diagram was proposed as the central component of model driven software development due to its representation of a solution domain in a platform independent manner (Nikiforova, et al. 2011). However, Nikiforova, et al. (2011) argued that UML class diagrams were not fully used in software development and developers did not spend enough time with them. Through the investigation of literature, they found that in theory we were ready to use code generation for development, but in practice modern CASE tools did not provide sufficient support for development because of the quality of generated code. They therefore concluded that the issue is not the UML class diagram itself, but the code generator. Since class diagrams were not widely used for code generation or documentation in software development, Sejans and Nikiforova (2011) expressed concern that class diagrams were losing their role as a "bridge" between problems and solutions.

In education, UML is used as a tool to design the model of a solution and a means of communication with students and student project team members. Specifically, UML class diagrams were commonly used in the name of "*design first*" or "*modelling first*" while teaching OOP (Moritz & Blank, 2005; Wei, et al., 2005; Georgantaki & Retalis, 2007; Eckert, Cham, Sun, & Dobbie, 2016). Sarkar, Lopez, Oliver, and Lance (2012) used UML in first-year Objects first programming course. Through analysing the performance from two groups of 221 students (n1 = 165 and n2 = 56) in 2009, 2010, and 2011, they found that there was a strong relationship between results in UML and programming from both groups. Based on interviews of 20 final year students, Boustedt (2012) suggested that using UML class diagrams as

documentation of program code is good for students to learn and modify the program code.

## 2.2 Program Code Generation

Automated code generation is the key feature of MDD. In recent years, many UML tools have been developed and used to support class diagram code generation in Java, such as Eclipse Modeling Framework (EMF), RAPTOR, and Umple. However, there have not been many tools that support the C# language.

We surveyed the literature of six studies using different case tools to generate Java code from UML diagrams (see Table 1). Two studies only generated a class code skeleton from UML class diagrams (Lethbridge, et al. 2011; Geogantaki & Retalis, 2007). One study included method implementation by using extra flowchart-like diagrams via a Method Editor (Carlisle, 2009). Inspired by generating Java code from UML class diagram, another study generated entire project code for web development in JS in teaching model-driven engineering (Cabot & Kolovos, 2016). In order to generate complete implementation in Java code for the entire project, the last two studies used three diagrams: class diagram, sequence diagram, and active diagram (Usman & Nadeem, 2009; Eckert, Cham, Sun, & Dobbie. 2016).

**Table 1: Literature of generating code from diagrams**

| No. | Author | Case Tool | Diagrams | Code |
|---|---|---|---|---|
| 1 | Lethbridge, et al. (2011) | Umple | Class Diagram | Java class skeleton |
| 2 | Georgantaki & Retalis, (2007) | ArgoUML | Class Diagram | Java class skeleton |
| 3 | Carlisle (2009) | RAPTOR | Class Diagram and Method Diagram | Java class code (property and method) |
| 4 | Cabot & Kolovos (2016) | EMF & WebRatio | Class Diagram | Fully functional web apps in HTML/ CSS/JS/ |
| 5 | Usman & Nadeem (2009) | UJECTOR | Class Diagram, Sequence Diagram, and Activity Diagram | Fully functional Java application |
| 6 | Eckert et al. (2016) | Lorini plug-in Astah | Class Diagram, Sequence Diagram, and Activity Diagram | Fully functional Java application |

Although some CASE tools only generate code skeletons from class diagram, two studies found positive response from surveys in teaching object-oriented programming. Through a survey of 30 students, Lethbridge, et al. (2011) found that no student disliked the ability to generate code in Java to represent UML design using Umple. In the survey of 18 students from an OOP seminar using ArgoUML, Geogantaki and Retalis (2007) discovered that "*although they appreciated the task to design class diagrams and create code skeletons for their applications using a CASE tool, they faced some usability problems when using this tool. Nevertheless, all students managed to submit their designs which most of them were correct*" (p121). Using class diagrams and method flowchart-like representation together in RAPTOR to generate both class skeleton and method details in Java code, Carlisle (2009) suggested that students valued the visual representation and were more successful in learning programming concepts.

Based on both UML class diagrams and the interaction flow modelling language (IFML), Cabot and Kolovos (2016) used a WebRatio[1] tool to automatically generate fully-functional web application in teaching model-driven engineering (MDE). However, through a survey of 29 students, they felt the approach was unsuccessful. Although students believed generating code is a good in itself, students were still concerned they were wasting time modify a lot of generated code. They suggested letting students work on several similar projects. Subsequently, students can modify classes and regenerate code and feel they were saving time due to MDE.

Through two detailed case studies of generating fully functional Java code, Usman and Nadeem (2009) explored using UJECTOR for code generation from three UML diagrams: 1) class diagram (for a class skeleton); 2) sequence diagram (for method code); and 3) activity diagram (for object manipulation and user interaction). The approach was effective in generating fully functional Java code, however, there was no empirical connection to the teaching of object-oriented programming. Recently, Eckert et al. (2016) developed a plug-ins, Lorini tool, for Astah framework in order to generate Java program from the above three UML designs. It also shows promise for the teaching of object-oriented programming. However, the evaluation of using this tool was only conducted once from eight volunteers. Further evaluations would be valuable in order to provide additional support for the approach.

Regarding code generation for C#, Sejans and Nikiforova (2011) had investigated the transformation of class diagrams into program code using Sparx Enterprise Architect (APARX). However, the code generated was very poor and unsatisfactory in corresponding to notations and model details. In a further relevant study, Nikiforova et al. (2011) also found poor results of C# code generation using Visual Paradigm (VP).

In Summary, C# is a suitable language for teaching object-oriented programming for CS1 and CS2. Although generating fully functional programming from three UML diagrams is promising, it is still not widely used in teaching practice. The most successful teaching practice that received positive feedback was at the stage of generating code from class diagrams. However, the generated code is in the Java language rather than in C# language. That is to say, in literature, there lacks sufficient research on the generation C# program code from UML class diagrams for the teaching of object-oriented programming.

# 3. TEACHING OOP WITH VP

At UCOL, object-oriented programming is mainly taught in a second year course (CS2) after initially being introduced in a first year course (CS1). There were 28 students enrolled in this course, studying in a software lab with a mixture of lecture and lab activities in three hours of class per week. Based on Reges (2002) that C# is an appropriate language for CS1 and CS2, we use C# in Microsoft Visual Studio (MS VS) in our course teaching class and file process, class inheritance, interface, and database applications. Regarding to CASE tool,

[1] http://ww.webratio.com

although Nikiforova et al. (2011) found unexpected results by using VP itself, we would explore the opportunities of success by embedding VP into MS VS to generate C# (see Figure 1). Based on the above literature study, specially Carlisle's (2009) assumption that students were in favour of visual presentation as well as Cabot and Kolovos's (2016) suggestion of using similar projects for refactoring, we decided to integrate VP into MS VS in our lab practice in order to reduce the gap between object-oriented design specifically in UML class diagram and object-oriented programming in C#. Therefore, students can design classes by drawing class diagrams in Visual Paradigm first and then generate C# code in MS VS from the class diagrams. We chose VP rather than MS VS UML tool because VP generates better quality of C# code, which requires less implementation time than those from the MS VS UML tool. Moreover, VP is not a completely new development environment to our students who had drawn Use Case, DFD and ER diagrams in their system analysis and design course. The teaching of integrating design and development in class inheritance was presented in the following three subsections.

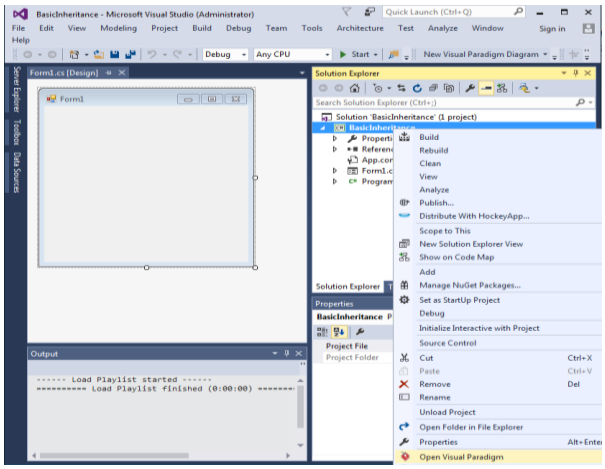

**Figure 1: Integration of Visual Paradigm and Visual Studio**

## 3.1 Introduction of Class Inheritance

Inheritance is one of the most challenging and the most crucial aspect in teaching object-oriented programming (Schmolitzky, 2006). In order to let students understand that subclass (e.g. Student class) can inherit all the properties from its base class (e.g. Person), we required students to draw a hierarchical class diagram (see Figure 2). Based on this Figure, we introduced that the Student class has three properties. Besides one property (course) in the Student class, there are two inherited properties (firstName and lastName), which were not repeated in the Student class.

After completion of class diagram in VP, students can then use MS VS to generate C# code (see Figure 3). Although the generated C# code is part of project solution, it is only a class skeleton, comprising fully functional property code, but only skeleton of constructor and methods. Students have to implement them manually. However, since the class skeleton matches the structure of inheritance in UML class diagram, code generation provides a linkage for students to make transition from class diagram to outline of class code.

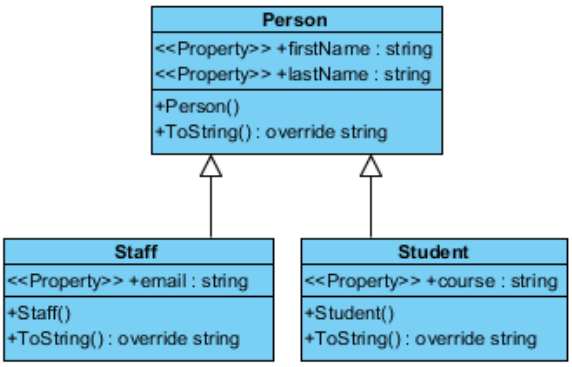


**Figure 2: Class Inheritance**

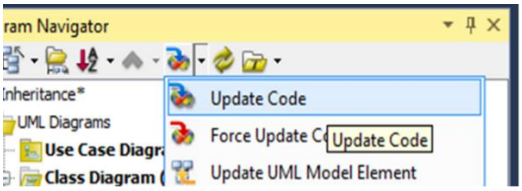


**Figure 3: Generate Code in MS Visual Studio**

## 3.2 Refactor Class Diagram and Code

It is easy for students to be confused by two major key words *abstract* and *virtual* in inheritance. Subsequently, we designed our teaching into three steps. Firstly, students reused their class diagram design from previous practice and updated their C# class code from modified UML class diagram (see Figure 4). Secondly, from modifying existing class diagram, students learned that 1) if a member is declared as abstract in a base class (e.g. *GetTitle(): abstract string* in Person class), this means it MUST be overridden in all the derived classes (e.g. in both Staff and Students classes); 2) If a member is declared as virtual in a base class (e.g. *GetDescription(): virtual string* in Person class), this means it is OPTIONAL to override this member in a derived class (e.g. in Student class, but not in Staff class). Finally, through implementation of the updated class skeleton, students followed the indication of the method skeleton to implement what they need to do in the derived classes (see Figure 5). Students also experienced taking advantages of refactoring from further development of reusing previous application.

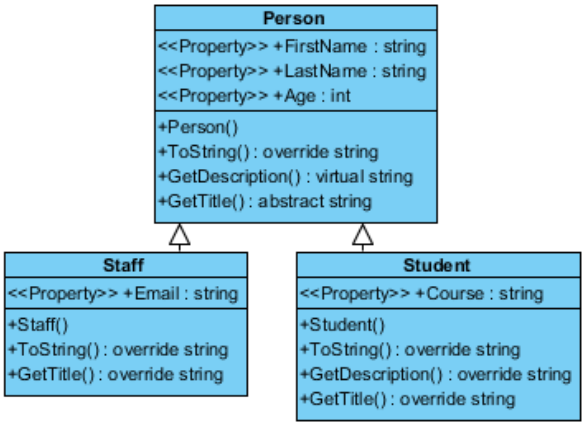


**Figure 4: Refactor of existing design**

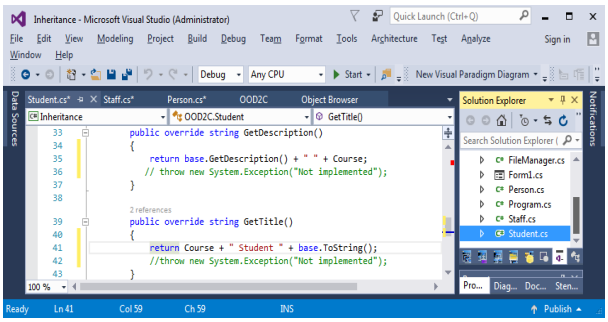

**Figure 5: Indication where to implement by updated skeleton**

## 3.3 Interface

An interface contains definitions for a group of related functionalities that a derived class can do, but not how to do it. By using interfaces, students learned that the use of the interface keyword allows the pseudo-inheritance from multiple interfaces as base classes. Instead of previously using one class (an interface, abstract class, or normal class) as base class, an example of using one interface (IClassification) and one normal class (Product) as base classes was introduced to generate C# code (see Figure 6). From the class diagram, they can also see the difference between generalisation (inheriting all the members of the base class) and realisation (providing the implementation of an interface). Although we emphasise not to forget implementing interface members in the derived class, Visual Paradigm automatically generates the property code defined in the interface (see Figure 7).

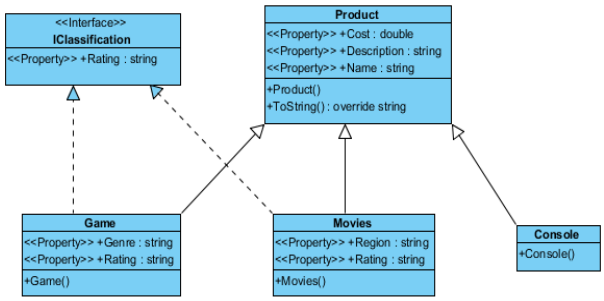


**Figure 6: Example of using interface**

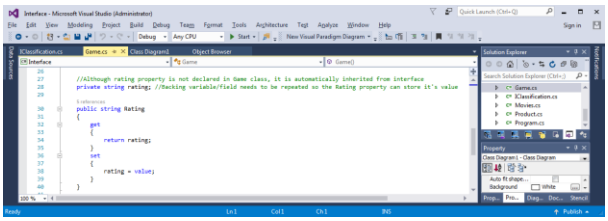

**Figure 7: Example of code generation according to interface**

In summary, in the hierarchical class diagram, we emphasise that properties and methods in the base class (super class or parent class) are inherited by the derived class (subclass or child class). The C# code of these properties and method signatures are automatically generated in Visual Studio based on the class diagram created in VP. Worked examples of using virtual and abstract methods as well as interfaces are used from class diagram to code generation in order to support students understanding of the connections between class diagrams and program code.

# 4. FINDINGS AND DISSCUSSION

After four weeks teaching and practice, we conducted a survey to twenty-one students about our new methods of drawing class diagrams and generating C# class code using nine Likert-scale type questions (see Table 2). We evaluated the survey results using the methodology described in Lethbridge et al. (2011), calculating a score for each question on the five-point scales from 1 to 5 according to response such as "1-Strongly disagree", "2-Disagree", "3-Undecided/neutral", "4-Agree", and "5-Strongly agree". In the survey, demographic questions were also included to identify beginners and advanced learners by asking if they had previously learned 1) programming, 2) object-oriented, and 3) CASE tool for programming. The survey results of average scores to each question from all students (whom were also divided into beginners and advanced students), beginners, and advanced students are listed in Table 2.

**Table 2: Survey Results**

| Q | Methods of drawing class diagram and generating C# class code | All | Beginners | Advanced |
|---|---|---|---|---|
| 1) | Help me to understand various concepts of class inheritance | 3.90 | 3.80 | 4.00 |
| 2) | Help me to understand the purpose of class design using a class diagram | 3.86 | 3.80 | 3.91 |
| 3) | Help me to reduce the gap between class diagram and program code | 3.71 | 3.70 | 3.73 |
| 4) | Help me to improve my object-oriented programming skills | 3.67 | 3.50 | 3.82 |
| 5) | Are better than those of directly writing class code in C# | 3.33 | 3.30 | 3.36 |
| 6) | Are easy to learn | 3.62 | 3.50 | 3.73 |
| 7) | Provide me an interesting learning experience | 3.52 | 3.50 | 3.55 |
| 8) | Add a value to this course | 4.00 | 3.70 | 4.27 |
| 9) | Should be used in this course next year | 3.86 | 3.70 | 4.00 |
| | Number of Students | 21 | 10 | 11 |

In general, the average scores in Table 2 are all more than 3. We also see the median values (with the cross sign "x") in Boxplot in Figure 8 are also above 3. This means all the twenty-one students generally agreed to nine questions in the survey. Regarding to Q1, both lower and upper quartiles are equal to 4 (agree) although there were outliers of extreme values (3 and 5) in Figure 8. It means that there are about 75% students agreed that our methods of drawing class diagram and generating C# class code helped them to understand concepts of class inheritance. Similarly to Q2, it also means that there are about 75% students agreed that our methods also helped them to understand the purpose of class design using a class diagram. For Q3, the lower and upper quartiles are between 3 and 4. Also, there is no lower whisker. It means that more than half of students agreed our methods helped them to reduce the gap between class diagram and program code. And no one disagreed so.

However, from Q4 to Q7, there is a lower whisker for minimum value 2 (disagree). It reveals that although there were more than half of students agreed there were still some students who disagreed to these questions. For Q4, there was one student who disagreed that our methods can improve her/his object-oriented programming skills. For Q5, there were two students disagreed that these methods were better than those of directly writing class code in C#. For Q6, three students disagreed that these methods were easy to learn. And

for Q7, one student also disagreed that these methods provided an interesting learning experience.

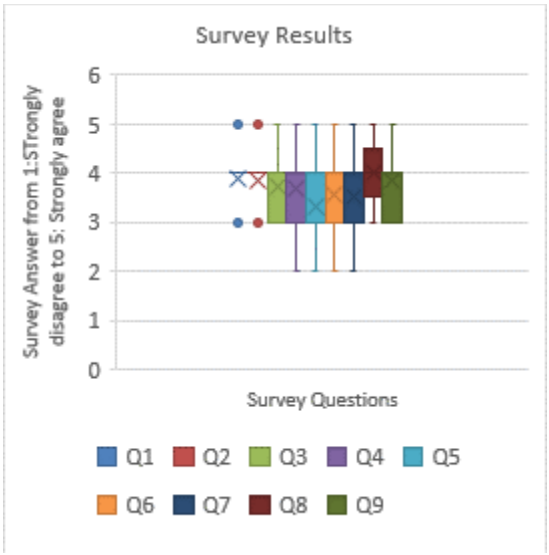


**Figure 8: Boxplot of Survey Results for nine questions**

Finally, for Q8, more than 75% students agreed that these methods added value to this course. And for Q9, more than half student agreed and no one disagreed that our methods should be used next year.

In Table 2, we also noticed that advanced students rated higher for each item in the survey than the beginners students did. Therefore, in order to follow up the survey results, an informal interview was conducted to focus groups in a review lab session. The focus groups were selected based on students' theory test results and divided into three groups (High, Medium, and Low), three students per group. Three questions were asked.

*Q1. Have you learned programming, especially OOP before?*

There were only two students in the high grade group who had learned OOP in C++ and C# before. The rest of them learned in the first year course. However, students from low grade group said that they learned a little bit of OOP in the first year study.

*Q2. Do you like using Class Diagrams to generate C# code? Why or why not?*

Students from high grade group commented that: "*For large project, yes; for small one, no*". "*I believe it is good for beginners*". "*It is extra work for me to do so*". Students from medium group all believed that it helps their learning. However, students from low group had different comments: "*It helps for class code. I can understand relations of classes, but I have difficulties in programming development, e.g. using text file management because I only learned a little bit before*". "*There is too much information in this course as we learned a little bit in the previous course*". "*No, I say no because I failed in the test*".

*Q3. Do you think that using Class Diagrams to generate C# code reduces the gap between design and development*?

Students from all three groups agreed. One student pointed that "*VP only generates part of code. I have to implement the uncompleted. No method details were generated*".

Based on students' feedback, we found that although students have different academic background, either novices or advanced learners, they all believe that using diagrams to generate code was beneficial for their learning, specifically reducing the gap between design and development in OOP. However, we noticed that beginners had issues to apply their previous knowledge into the new course while they are learning a lot of new information. This indicated that we should consider personalised learning by providing optional learning materials of previous knowledge online.

We also noticed that advanced students prefer to directly write code for current exercises although they believed generating code from class diagrams was helpful. This suggested that we should refactor our exercises to use more complex class diagrams so that students could take advantage of reusing design and code generation. Considering both beginners and advanced learners were working in complex multiple class projects, we may explore using reverse engineering tools from MS VS to regenerate Class Diagrams after implementation. In this semester, we required students to focus using VP to draw class diagram rather than to use MS VS tool regenerating class diagram after having generated code from VP.

## 5. CONCLUSION

From this case study, we concluded that although only the class skeleton was able to be generated from class diagram, students founded it useful for reducing the gap between design and development in teaching object-oriented programming. The class skeleton provides an outline for class development so that students can easily identify where they should focus in order to implement the details. If class diagrams bridges the gap between user and developer as well as between the problem and solution, we would argue specifically that teaching code generation can reduce the gap between design and development in teaching OOP. The contribution of this research is that we have explored the possibility of generating C# class code while integrating with VP. Teaching class code generation is helpful for students to learn OOP, which enriches our course curriculum. The research expands the literature of generating code from Java to the C# language in teaching OOP. Further research would explore using a UML tool to generate fully functional C# application code from Class Diagrams, Sequence Diagrams, and Activity Diagrams in teaching OOP.

## 6. REFERENCES


Aldaeej, A., & Badreddin, O. (2016, May). Towards promoting design and UML modeling practices in the open source community. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 722-724). ACM.

Boustedt, J. (2012). Students' different understandings of class diagrams. *Computer Science Education, 22*(1), 29-62

Cabot, J., & Kolovos, D. S. (2016, November). Human factors in the adoption of model-driven engineering: an educator's perspective. In *International Conference on Conceptual Modeling* (pp. 207-217). Springer International Publishing.

Carlisle, M. C. (2009). Raptor: a visual programming environment for teaching object-oriented programming. *Journal of Computing Sciences in Colleges, 24(*4), 275-281.

Das, N., Ganesan, S., Jweda, L., Bagherzadeh, M., Hili, N., & Dingel, J. (2016). Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems* (pp. 36-43). ACM.

Eckert, C., Cham, B., Sun, J., & Dobbie, G. (2016). From Design to Code: An Educational Approach. In *SEKE* (pp. 443-448).

Georgantaki, S., & Retalis, S. (2007). Using educational tools for teaching object oriented design and programming. *Journal of Information Technology Impact, 7*(2), 111-130.

Lethbridge, T. C., Mussbacher, G., Forward, A., & Badreddin, O. (2011). Teaching UML using umple: Applying model-oriented programming in the classroom. In *Software Engineering Education and Training (CSEE&T), 2011 24th IEEE-CS Conference* on (pp. 421-428). IEEE.

Moritz, S. H., & Blank, G. D. (2005). A design-first curriculum for teaching Java in a CS1 course. *ACM SIGCSE Bulletin*, *37*(2), 89-93.

Nikiforova, O., Sejans, J., & Cernickins, A. (2011). Role of UML Class Diagram in Object-Oriented Software Development. *Scientific Journal of Riga Technical University*. Computer Sciences, 44(1), 65-74.

Raoufi, M., & Maniotes, J. (2005). Why C# and Why .NET in The Undergraduate Information Systems Curriculum. *Information Systems Education Journal*, *3*(43), 3-6.

Reges, S. (2002, June). Can C# replace java in CS1 and CS2?. In *ACM SIGCSE Bulletin* (Vol. 34, No. 3, pp. 4-8). ACM.

Sarkar, A., Lopez, M., Oliver, R., & Lance, M. (2012). Relationships between Logic Depiction, UML Diagramming and Programming. In *Proceedings of the Computing and Information Technology Research and Education New Zealand (CITRENZ) 2012 Conference* (pp. 88-92), Christchurch, New Zealand.

Sejans, J., & Nikiforova, O. (2011). Problems and Perspectives of Code Generation from UML Class Diagram. *Scientific Journal of Riga Technical University*. Computer Sciences, 44(1), 75-84.

Stuurman, S., Passier, H., & Barendsen, E. (2016). Analyzing students' software redesign strategies. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 110-119). ACM.

Usman, M., & Nadeem, A. (2009). Automatic generation of Java code from UML diagrams using UJECTOR. *International Journal of Software Engineering and Its Applications*, *3*(2), 21-37.

Wei, F., Moritz, S. H., Parvez, S. M., & Blank, G. D. (2005). A student model for object-oriented design and programming. *Journal of Computing Sciences in Colleges*, *20*(5), 260-273.