jCAB: Making Java Class Design Easier for Novice Programmers

Qian Liu
Mathematics and Computer Science Department
Rhode Island College
Providence, RI 02908
qliu@ric.edu

***Abstract*—This Research Full Paper introduces a tool called Java Class Auto Builder (jCAB) that helps students to design Java custom classes, helps them to understand the core concepts in the class design, and helps them to develop Object-Oriented (OO) ideas efficiently.**

When OO programming is introduced to students, its fundamental concepts are difficult for them to learn at first, and also impact them when they start to design their custom class templates. The existing UML (Unified Modeling Language) based tools could be served for educational purpose, but they are not suitable for novice Java students or programmers. Those tools generate class template code automatically from UML diagrams which must represent the class structures and functionalities clearly. That requires users to have a comprehensive view on the class structures. However, when they are first exposed to custom class design, novice programmers usually do not know what are necessary in the design and/or the relationships among classes if there are any. Therefore, it would be hard for them to develop UML diagrams which could represent class templates clearly. A tool that could help students in this stage is necessary.

The jCAB tool introduced in this paper is designed for Java CS1 course, and is a good start for novice programmers to learn how to implement custom classes. The tool divides the custom class design procedure into several steps in order to show students what are necessary in the class design clearly and how to do them in detail. Its GUI (Graphic User Interface) interacts with students, directs them step by step, and generates code in each step automatically, so that students will be aware of the class structure, the design procedures, and how each piece of code is related with OO concept, and they will have better grasp on the OO fundamentals and the custom class design procedures.

***Index Terms*-Java Custom Class; Object-Oriented; Graphic User Interface; Code Generation**

I. Introduction

In introductory Object-Oriented (OO) programming courses, the fundamental OO concepts are difficult for novice programmers to learn at the beginning [1], and also impact them when they start to design custom class templates. These concepts are called threshold concepts. In [2, 3, 4], several concepts in Computer Science and Programming areas are identified as thresholds, and many of them are OO related concepts, such as abstraction, encapsulation, inheritance, etc. In introductory OO programming courses, those concepts are obstacles that prevent students from moving forward.

This paper introduces a tool called Java Class Auto Builder (jCAB) that helps students in Java CS1 course to design custom class templates, helps them to understand those OO related threshold concepts, and helps them to develop OO ideas efficiently. jCAB is a good start for novice programmers to learn how to implement custom classes, and is a good start for novice programmers to connect OO concepts with code. For instance, the connections between Java keywords ("public", "private", and "protected") and OO concepts such as "encapsulation" and "inheritance". jCAB divides custom class design procedure into several steps in order to show students what are necessary in the design clearly and how to do them in detail. Its Graphic User Interface (GUI) interacts with students during the custom class design. It directs students step by step, and generates source code after each step automatically to give students detailed illustrations and connections between OO concepts and code parts. In this process, students will be aware of the class structure, the design procedures, and how each piece of code is related with OO concept, and eventually they will have better grasp on the OO fundamentals promptly and efficiently.

Existing approaches that can generate Java code automatically are UML (Unified Modeling Language) based tools [5, 6, 7, 8, 9, 10]. They generate source code from UML diagrams by translating the input diagrams based on some pre-defined symbols. For instance, symbols like "+", and "-" are used to indicate modifiers. Some of those tools [9, 10] are provided as plug-in functionalities for their IDEs (Integrated Development Environments), such as Eclipse Papyrus [9]. Those UML-based tools are excellent in areas of Model-Based Engineering and project management, and they could also be served as pedagogical tools for advanced OO courses. jCAB, however, is not an alternative to UML-based tools, it is designed for Java introductory course, and is much more suitable for novice programmers when compared to those UML-based tools because:

1) the UML diagrams must represent the class structures and functionalities clearly. This requires students to have a comprehensive view on the class structures at the beginning because users not only need to list all member variables and methods by using UML pre-defined symbols, but also need to indicate the

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

relationships among classes if there are any. Novice Java students, however, do not have a firm grasp on the OO concepts when those concepts are first introduced to them [11], and they may not have clear ideas on what are necessary in the custom class design. In this case, it would be difficult for them to develop an UML diagram that could represent a class structure and functionality clearly.

2) the UML-based approaches generate the whole source code at the end, novice programmers don't know how the "translations" from the diagrams to code parts take place. On the contrary, jCAB directs students and generates code step by step, so that students are aware of the class design procedure, and are aware of how each piece of code is related with OO concept.
3) before developing any UML diagrams, one has to learn the protocols in order to use those UML-based tools. This puts more burden on novice programmers. jCAB, contrarily, does not have any requirements. It is a self-directed and self-explanatory tool designed for Java CS1 course while UML-based tools are not. jCAB provides the easiest interface and its interactions with students are very straightforward and clear.

II. Design

The architecture of jCAB is shown in Fig. 1. It is built on top of JDK (Java Development Kit), and consists of four independent units. As stated before, jCAB divides custom class design procedure into several steps in order to show students what are necessary in the design and how to do them in detail. The GUI unit is the interface that interacts with students and directs them in each step. The Processor unit processes and analyzes user input from the GUI. If an user input is not valid, it notifies the GUI unit which then reports error information and gives suggestions to the user. If an user input can be taken, the Processor unit will update the record in the Symbol Table unit, and then the Code Generator unit will show the corresponding code part in the GUI window. Code part is generated in each step in order to give students detailed illustrations of the relationship between the code part and the current design step.

The jCAB is released to students as a JAR (Java ARchive) binary file, and has no requirement on the installation. It does not rely on any specific OS features, and is an OS independent, portable tool for PCs (Windows, Linux, and Mac OS).

The following subsections discuss the jCAB design in detail.

A. Class Name

Initially, there is only one button enabled on the GUI. As shown in Fig. 2, the "Create Class" button is used to create a class framework by naming a new class at first in the pop-up window shown in Fig. 3. After providing a class name, a

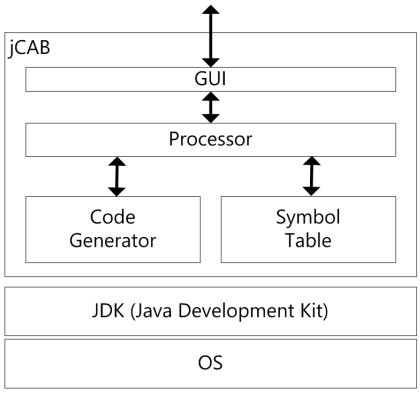

Fig. 1: The Architecture of jCAB

new class definition line will be displayed in the GUI code window (the window on the right hand side in Fig. 2), as shown in Fig. 4, if the new class name provided by student is a valid Java identifier. In the example shown in Fig. 4, a new class named "Item" is created. Its corresponding record will be created in Symbol Table unit (Fig. 1) for future reference. Afterwards, the button "Create Variables" is enabled, and the "Create Class" button is disabled, indicating that the next step should define class member variables (See next). This step shows students that the first step of a custom class implementation is to name a new class.

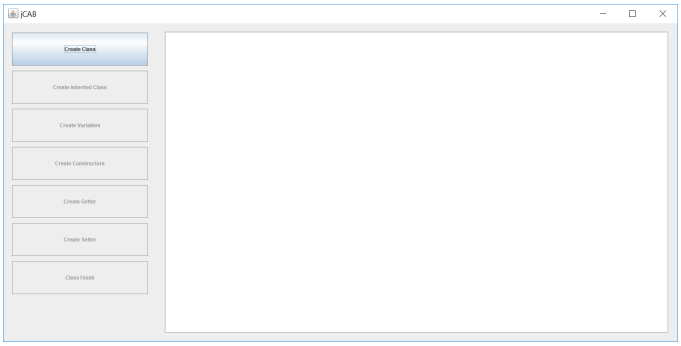

Fig. 2: The jCAB Main Interface

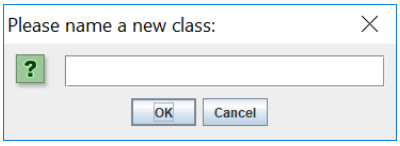

Fig. 3: Name a New Class

B. Member Variables

A class may or may not contain member variables, but jCAB requires students to define at least one member

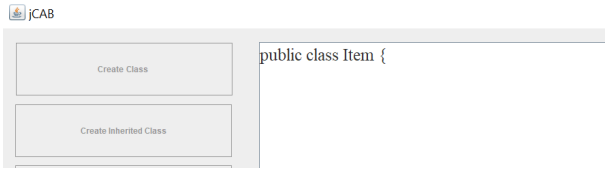

Fig. 4: Code Generated for New Class named “Item”

variable. This is why jCAB only enables “Create Variables” button after the previous step because we want students to define member variables first before member methods which usually access those member variables. Another reason we want students to declare member variables is that it would be easier to show further examples, such as: 1) each class object has its own copy of (non-static) member variables; 2) if one object calls class getter/setter method (discuss later), it has no impact on other objects.

Variables can be defined in the pop-up window shown in Fig. 5. The Processor unit would check and report any errors such as duplicate definition, invalid data type, etc. If a variable definition is acceptable, its information will be recorded in the Symbol Table.

Depending on the variable modifier selected in Fig. 5, GUI would report its scope information, in Fig. 6 for instance, giving student a quick summary of the variable, so that students are aware of the relationship between the modifier of a variable and its scope. Students are able to re-define a variable with another modifier in this step if the previous selected modifier is not desirable. Fig. 7 shows the generated code part if a new variable “price” is defined in this step.

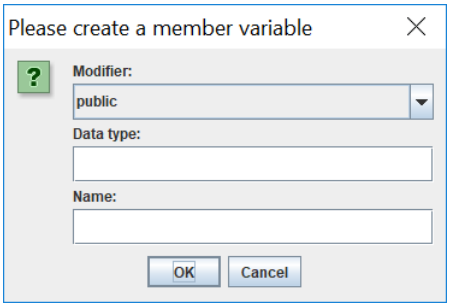

Fig. 5: Member Variable Creation Window

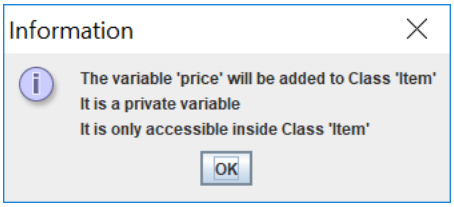

Fig. 6: Variable Scope Information Report

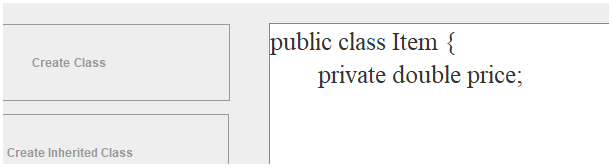

Fig. 7: Code Generated for Member Variable “price”

If there is at least one variable defined, buttons such as “Create Constructor”, “Create Getter”, etc, will be enabled, and the “Create Variables” button remains enabled, indicating that students could define more variables, and/or continue to work on constructors and other member methods.

This step places emphasis on the member variable definition for a newly defined class. It directs students to examine how to define member variables by using different Java modifiers. Usually modifiers “public”, “private”, and “protected” do not mean too much to students at the very beginning. The quick summary window (Fig. 6) in this step illustrates the primary differences among modifiers, and it helps students to understand concepts such as information hiding, encapsulation, and helps students to select the “appropriate” modifiers for their member variables. With the help of the scope report window, students have chance to use other modifiers to modify previously defined variable if they find out the variable scope is not what they want. The UML-based approaches, however, are using symbols like “+”, “-”, or “#” to represent modifiers. Although they are concise to represent class hierarchy in diagrams, they are not suitable for novice programmers. The jCAB places emphasis on the fundamental concepts when introducing custom class design, and it is much more suitable for students in introductory course.

C. Constructors

Constructor part is usually an error-prone part in Java. Theoretically they are methods, but they are different from the regular member methods because they have no return value. jCAB clearly illustrates the creation procedure of a constructor, including what a constructor does, and when a default constructor will be created automatically.

The pop-up window shown in Fig. 8 will be used to create a constructor, if one clicks “Create Constructor” button. To illustrate the constructor concept better, jCAB aims to provide the easiest interface to students. It only asks for the type of the constructor one wants to create: the constructor without parameter (default constructor), or with parameter. Fig. 9 shows the code generated (including necessary comments for students) if one chooses to create a default constructor, which initializes all member variables student declared in the previous step to “0”s. If a variable is in a non-primitive data type, the constructor will automatically create an object for that variable to avoid any null dereference.

Because jCAB supports some well-known classes (String, Integer, etc), it is able to initialize variables in these types. In addition, because jCAB supports inheritance, it is also able to use appropriate constructors to initialize member variables in user-defined class types.

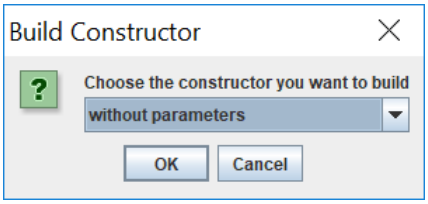

Fig. 8: Create Constructors Interface

```
public class Item {
    private double price;
    private String name;

    /* constructor has no return value */
    public Item() {
      /* initialize all member variables in constructor */
      price = 0.0;
      name = new String("");
    }
```

Fig. 9: Code Generated for default Constructor

To make one understand constructor easier, if student has already created a default constructor, when he/she clicks the "Create Constructor" button again, the pop-up window (Fig. 8) would only provide one option: the constructor with parameters. If one has already created these two constructors, the "Create Constructor" button becomes disabled to indicate that there is no need to declare more constructors for the current class. If one does not declare any constructor, the jCAB will automatically create a default constructor for students (discuss later). By using jCAB, students could understand the fundamental concepts in constructors easily and clearly. In the UML-based approaches, however, the default constructor creation and the "*super*" call are implicit, and they will not be shown in the code part, therefore, students have no clear ideas on these fundamentals when using UML-based tools to learn custom class design.

In jCAB, if student creates a constructor for an inherited class, the Code Generator unit will automatically insert a "*super*" call in the constructor of the inherited class to indicate that the constructor in parent class always gets called first. The easiest interface in this step shows the difference between a constructor and a regular member method, and it also illustrates the work a constructor should do: initialize all member variables appropriately.

D. Member Methods

Member methods generally consist of three parts: Accessor (getter), Mutator (setter), and other generic methods. jCAB provides interfaces for the first two parts (getter and setter) because we want to place emphasis on the class structure, class design procedure, and simple methods like getter/setter first. It would be easier to introduce generic methods once students have solid foundations.

When designing a member method, one common error novice programmers make is that they usually re-define a member variable in a member method. For instance, to set a new price value for member variable "price" which was declared previously, Fig. 10 shows an example of incorrect setter method novice programmer defines. Another kind of error, shown in Fig. 11, is that novice programmers ignore "this" keyword so that it is not possible to differentiate the formal parameters and member variables if they have the same name.

```
/* member variable price */
private  double  price;

/* constructors and other methods */

/* setter for price */
public void setPrice(double p) {
    double price;
    price = p;
}
```

Fig. 10: Common Error 1 Related with Getter/Setter

```
/* member variable price */
private double price;

/* constructors and other methods */

/* setter for price */
public void setPrice(double price) {
    price = price;
}
```

Fig. 11: Common Error 2 Realted with Getter/Setter

In order to illustrate how to make a general getter/setter method for a member variable clearly, jCAB provides the easiest interface to students, as shown in Fig. 12 when one clicks "Create Getter" or "Create Setter" button. It only lists the member variables declared previously, and lets users to choose one at a time to build a getter or setter for the chosen member variable. jCAB hides all other details, such as return value and method argument, which will be made up by the

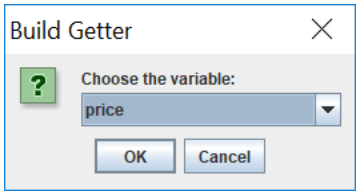

Fig. 12: Getter Creation Window

```
public class Item {
    private double price;
    private String name;

    /* constructor has no return value */
    public Item() {
        /* initialize all member variables in constructor */
        price = 0.0;
        name = new String("");
    }

    /* regular class member method has return value */
    /* public: public interface */
    /* class object will access private variable via public interface */
    public void setPrice(double price) {
        /* use method argument to set member variable belongs to */
        /* current class object, which is referenced by 'this' */
        this.price = price;
    }
```

Fig. 13: Code Generated for a Created Setter

Code Generator unit automatically. Fig. 13 shows the code part generated for setter method of member variable "price".

By using the easiest interface provided by jCAB, students can focus on the code to comprehend the correct way to define a getter/setter method for a member variable. By using the generated code, students can realize that there is no need to declare a member variable again in its getter or setter method because the previously defined member variable is visible in member methods. In this case, error in Figure 10 can be prevented. In addition, the generated code intentionally declares the argument of the setter by using the same name of the member variable in order to illustrate the usage of keyword "this" to students. This is used to rectify error shown in Figure 11. Table I in section III lists the detailed comparison before and after using jCAB among students.

In this step, jCAB illustrates member method design procedures clearly to students, such as how to define a member method, and what a (getter/setter) method should do. It further illustrates that member variables should be accessed through member methods in order to avoid data misuse. This step, in addition, helps students to understand and develop the OO "abstraction" idea, and places emphasis on the encapsulation and information hiding concepts: what features should be open to class/object users, and what units should be invisible to the outside world.

E. Class Finish

Button "Class Finish" is used to conclude a class implementation. First, it checks whether student has defined at least one constructor in previous steps, if not, it will create a default constructor automatically, and put it to the code window, indicating that there will always be at least one constructor defined for a class. Then, it implements *toString()* and *equals()* methods to indicate that every class inherits these two methods from base class *Object*. Students could modify these implementations if needed.

When a new class definition is finished, its record in the Symbol Table unit is complete. Students could design more classes in jCAB by using "Create Class" button, which is re-enabled. In addition, button "Create Inherited Class" is enabled for students to extend the existing class.

III. Results

jCAB tries to build connections between OO fundamental concepts and code parts in order to illustrate those concepts by simple pieces of code. Concept discussion is the first "remembering" stage, according to Bloom's taxonomy [12], but understanding those concepts does not mean students can code. Without code, concepts are abstract. We use jCAB to combine concepts and code parts in order to place emphasis on the Bloom's "understanding" and "applying" stages. As stated before, jCAB divides custom class design procedure into several steps, and shows students correct code and concept explanation in each step to illustrate why the code part is designed in such a way, and to make connections between code parts and concepts. By participating in each step actively, students get familiar with custom class design procedures along with coding patterns, so that students will be on the right track when designing other custom classes.

We introduced jCAB to 150 undergrad students in Java CS1 courses through 3 semesters. We first summarize the common errors students usually make when they begin to design custom classes, and then we examine whether they make any progress after using jCAB. Table I lists the percentage of students that makes those common errors before and after using jCAB. It points out that students are making significant progress in all aspects after using jCAB. They have better grasp on the custom class design procedures, and have clear understanding on what should be done in custom class design. The overall quality of their programs is also improved.

We let student to evaluate jCAB from several aspects. Fig. 14 shows their overall evaluations on jCAB (1 means "strongly disagree"; 5 means "strongly agree"). After surveying students for the past 3 semesters, they have identified

that jCAB is helpful for them to understand OO concepts better at the beginning in Java CS1 course, and it illustrates the custom class design procedures clearly.

TABLE I: Percentage of Students Makes Common Errors Before and After Using jCAB

Common Errors	Before	After
Variables/methods defined without modifiers	53%	11%
Does not provide any Constructors or Constructor has return type	40%	17%
No getter or setter method	31%	8%
Declare but not use member variables in methods, such as: return local variable in getter method, or try to modify local variable in setter method	57%	17%
Repeat base class's work in inherited class	26%	6%
Misuse of super()	13%	4%

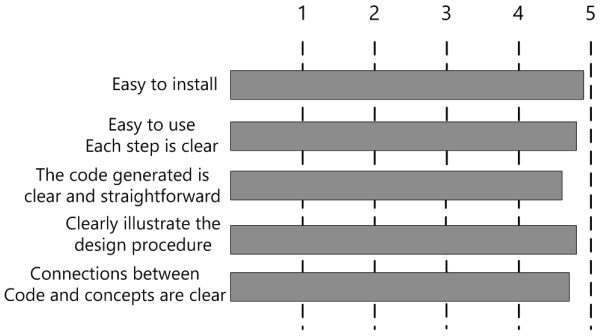

Fig. 14: The Result of Student Evaluations on jCAB

IV. Summary

Students in general do not have a firm grasp on the Object-Oriented (OO) fundamental concepts when those concepts are first introduced in CS1 course. This would impact students when they start to design custom classes. The jCAB introduced in this paper gives students structural ideas in (Java) custom class design and implementation, and it helps students to understand the fundamental OO concepts in the class design procedure. Even though it may not be difficult for all students to design custom classes in CS1 course, using jCAB is easier for students to understand custom class design, and it is also easier for teachers to illustrate class design procedures (member variables, constructors, getter/setter, etc) to students.

The UML-based tools "translate" UML diagrams to code in a convenient way based on some pre-defined symbols and rules, and they need users to have a comprehensive view on the class structures. However, novice programmers are not familiar with custom class design at the beginning, and they don't have a clear view on the class structures. It would be hard for them to develop UML diagrams clearly. jCAB, on the contrary, is a good start for novice programmers when they begin to learn and implement custom Java classes. It divides custom class design procedure into several steps, and directs students step by step through its GUI window with straightforward and clear interactions. jCAB generates code in each step automatically, so that students will be aware of how each piece of code is related with OO concept, and they will have better understanding on the OO fundamentals and custom class design procedures.

With jCAB, it would be easier to teach and demonstrate OO fundamentals in Java introductory course, especially when teaching custom class design. We have positive feedback from students. They have clear understanding on the Java custom design procedures after using jCAB. They have identified that jCAB helps them to go through the connections between the custom class design procedures and the fundamental concepts, and it also prepares them better for advanced OO courses in the future.

References

[1] J.H.F. Meyer and R. Land. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. In *Improving Student Learning - Ten Years on*, Jan 2003.

[2] K. Sanders, J. Boustedt, and et.al. Threshold Concepts and Threshold Skills in Computing. In *ICER'12 - Proceedings of the 9th Annual International Conference on International Computing Education Research*, Sep. 2012.

[3] K. Sanders and R. McCartney. Threshold Concepts in Computing: Past, Present, and Future. In *Koli Calling '16 Proceedings of the 16th Koli Calling International Conference on Computing Education Research* , Nov. 2016.

[4] J. Boustedt, A. Eckerdal, and et.al. Threshold Concepts in Computer Science: Do they exist and are they useful. In *SIGCSE '07 Proceedings of the 38th SIGCSE technical symposium on Computer science education* , Mar. 2007.

[5] P. Veisi and E. Stroulia. AHL: Model-Driven Engineering of Android Applications with BLE Peripherals. In *Aimer E., Ruhi U., Weiss M. (eds) E-Technologies: Embracing the Internet of Things. MCETECH 2017. Springer*.

[6] A. G. Parada, E. Siegert, and L. B. de Brisolara. Generating Java Code from UML Class and Sequence Diagrams. In *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*, Nov. 2011.

[7] A. Armentrout. The Tool for Designing Java Programs with UML. In *ITiCSE '99 Proceedings of the 4th annual SIGCSE/SIGCUE*, Jun. 1999.

[8] BlueJ, A free Java Development Environment. https://www.bluej.org. Last Accessed: Apr. 2018.

[9] Papyrus Software Designer. https://www.eclipse.org/papyrus/. Last Accessed: Apr. 2018.

[10] EasyUML Diagram Tool. http://plugins.netbeans.org/. Last Accessed: Apr. 2018.

[11] K. Sanders, J. Boustedt, and et.al. Student understanding of object-oriented programming as expressed in concept maps. In *SIGCSE '08 Proceedings of the 39th SIGCSE technical symposium on Computer science education* , Mar. 2008.

[12] L. W. Anderson, D. R. Krathwohl, and B. S. Bloom. *A taxonomy for learning, teaching, and assessing : a revision of Bloom's taxonomy of educational objectives*. New York: Longman, 2001.