Structural Analysis for Simple Games Source Codes Applied to Programming Learning

Elanne C. O. Santos[1,2]
Victor H. V. Sousa[1]
Instituto Federal de Educação, Ciência e Tecnologia do Piauí, IFPI[1]

Gleison B. Batista[2]
Esteban W. G. Clua[2]
Universidade Federal de Fluminense, UFF[2]

Abstract

Teaching programming and algorithms is a big challenge, not only in universities but also in schools and training centers. Many proposals for stimulating this process were made in the last years. Previously to this work we had developed JPlay. The JPlay framework was proposed and developed for teaching programming with the development of simple 2D games. In this paper we propose a heuristic based on the structural analysis of the behaviors of a JPlay program and, based on this heuristic, we developed a tool that makes analyzes of JPlay programs, guiding and teaching a student for a specific game development. The heuristic consists on a comparison approach between the student program and the model program and it has four levels of analysis: the sequential code pattern of the JPlay, standardization model, the comparison of similar classes and construction of behavior trees of similar variables. Thus, the comparison consists on searching behaviors of correspondence between pairs of classes among these programs. In this paper we also present a review of results of BrickBreak game based on the source code of the integrated high school students in the course of Computers.

Keywords: heuristic, programming, JPLAY, behavior, learning, games

Authors' contact:
elannecristina.santos@ifpi.edu.br
vhv.sousa@gmail.com
{gbatista, esteban}@ic.uff.br

1. Introduction

The teaching of programming and algorithms consists in a big challenge, not only in universities but also in schools and training centers. Studies point to the difficulty of teaching and learning the disciplines related to algorithms and programming, resulting in high dropout rates in computer courses [Pinheiro et al. 2007, Barbosa et al. 2011]. The main reason for this negligence is the difficulty in learning abstract concepts of programming [Santos and Rapkiewicz 2007]. Many proposals for stimulating this process were made in the last years [Allen et al. 2002; Kolling et. 2013; Traetteberg and Aalberg 2006; Allowatt and Edwards 2005].

In this sense, the JPlay framework was proposed and developed for teaching programming [Feijó et al. 2010]. JPlay is a framework for facilitating the teaching of programming, providing an algorithmic learning process related with the logic of simple 2D game development. JPlay does not interfere with the structure of basic programming necessary for a correct learning of algorithmic logic and does not introduce specific features of design patterns or stereotypes of games in the source code. The tool allows the students an easy way to draw and move images on a computer screen and provides methods and objects that help to create 2D games using the Java language.

Previously to this work we proposed a semantic analyzer based on behaviors comparison between two programs: a model program and a student program. We showed results using a comparison algorithm between the variables of the same type of each pair of classes, analyzing its game context instead of syntax details, based on a heuristic for guessing variables behaviors. The programs compared are simple 2D games developed by JPlay framework. Thus, in order to compare the behaviors of two programs we developed a comparison algorithm between the classes of the model program and student program, resulting in similar pairs of classes. Previously, we show how the algorithm combined the similar classes of the two programs [Santos et al. 2013]. In the present paper we will use a four levels heuristic, based on the structural analysis of the behaviors. These levels are: the sequential code pattern of the JPlay, standardization model, the comparison of similar classes and construction of behavior trees of similar variables. This analysis will show more specifics results using a behaviors comparison heuristic. Behaviors are identified through the analysis of its occurrence scope at the source code, such as an assignment, a loop or a conditional usage. Therefore, when behaviors differences are identified, the system makes suggestions about these differences found in the student program and gives clues that may indicate a possible semantic error. The teacher adds the suggestions in the form of comments in the source code of the model program, which are automatically captured by our solution. In this paper, we will also show some results of this comparison.

2. Related Work

During the process of learning programming different techniques can be used for students in addition of learning to program with the purpose of acquiring good programming practices. The techniques are classified as follows: tests based programming, programming pattern, automatic evaluator, programs diagnostic systems [Pinheiro et al. 2007].

Based on previous works [Delgado 2005; Pinheiro et al. 2007], we classify the techniques related to programming learning context as follows: programming based unit testing, proposals of programming environments, automatic evaluator, analysis of programming patterns and automatic depuration systems (intelligent tutoring systems and programs diagnostic systems).

In programming based unit the teacher provides a set of specific tests to solve a particular problem and the student must build a program that allows the achievement of expected results in the execution of all tests [Pinheiro et al. 2007; Traetteberg and Aalberg 2006]. The proposals of programming environments consist on the fact that some development tools were created in order to assist students in introductory programming, such as BlueJ and DrJava [Kolling et al 2013; Allen et al. 2002].

The automatic evaluator is used to help the teachers with tasks of activities corrections. The teacher can define acceptance tests to be automatically executed after the students deliver their programming activities and results of the tests can be used to compose the final score the student [Pinheiro et al. 2007]. We can quote the Web-Cat [Allowatt and Edwards 2005] as an automatic evaluation tool.

The analysis of programming patterns is based on research of programming learning suggestions that experienced developers solved when looking for previous solutions that are related to the new problem and that can be adapted to the ideal situation [Delgado 2005]. Thus, the concept of patterns is based on the fact that experienced programmers are able of solve new problems through the analysis of a previously solved problem. They can identify what structure to use, what types of data is involved, as well as other ways to solve the same problem, through previous experiences that identify solutions [Alexis and Deller 2013]. Previous experiments contain the basis for Programming patterns, which are solutions that often appear in solving computational problems [Alexis and Deller 2013]. Thus, patterns translating programming strategies created by experts can lead to good programming practices. We can quote the systems Proust [Johnson and Soloway 1984] and PROPAT [Delgado 2005] as systems that use the strategy of analysis of programming patterns.

An automatic depuration system is a system that uses techniques in order to find and classify components from a program. Based on the type of technique used, this may be classified as a programming intelligent tutoring systems and program diagnostic system.

Laura is one of the first attempts to build a tutoring system for teaching programming and is written in Fortran [Botelho 2010]. Its strategy is a comparison between two programs, the model and the candidate. The comparison is possible through the representation of the model and candidate programs by graphs, and its heuristic strategy to identify step by step the elements of the graphs [Adam and Laurent 1980].

We have previously developed a knowledge modeling system for semantic analysis of Games [Santos et al. 2013] and is based at learning objectives. It aims to find and to classify possible errors that happen in the program. Therefore it can be used in order to guide the student about these errors. It has a function of interpreting semantically and architecturally a Java program developed that uses the JPlay and return results of this examination to the programmer. The process consists on a comparison between the student program and model program. For this, the programmer must select, in his integrated development environment (IDE) tool, the model program that he wants to use as reference, previously available in a repository. Thus, the analyzer is able to interpret semantically the program that is being built by the student, may point out problems and suggest possible solutions.

The comparison is based on behaviors of the programs. Different behaviors between the model program and student program produce suggestions about possible errors in the student´s source code. Thus, in order to compare the behaviors of two programs we developed a comparison algorithm between the classes of the model program and student program, resulting in similar classes pairs. Previously, we show how the algorithm combined the similar classes of the two programs [Santos et al. 2013].

After the selection of similar classes, variables of each pair are compared and similar variables pairs are selected. Then, at this stage, the analysis shows more specifics results using a comparison algorithm between the variables of the same type of the pairs of classes. The comparison is based on variables behaviors. Behaviors are identified through the occurrence, at the source code, of assignments, loops or conditionals statements. Therefore, when behavioral differences are identified at the student program, the algorithm makes suggestions about. The teacher adds these suggestions in the form of comments in the source code of the model program.

A difference is that our proposal is based on a design pattern oriented to simple 2D game, following

the original purpose of JPlay [Feijó et al. 2010]. One more important difference concerns the comparison between similar variables. In the case of Laura, for example, two graphs of the model program and candidate program are built and compared. In our work we can build a data structure (behavior tree) starting from the behaviors of the variable, and compare each of these structures, thus obtaining a higher level of granularity in this heuristic strategy in order to identify behaviors differences between programs.

3. JPLAY

Our proposal is based in the JPlay framework. JPlay was previously developed with the purpose of teaching computer science and algorithms based in game development.

We classify the first level of analysis as the sequential code pattern of the JPlay. A sequence code pattern of the JPlay is a code sequence in the program based on JPlay that must always happen when the program is correct.

In order to identify sequential patterns in JPlay architecture, we divide the JPlay diagram into three parts: the interaction between game and player, characters and output game.

The classes responsible for interaction between game and player are: Keyboard (define input data for the keyboard) and Mouse (define input data for mouse). The classes responsible for creating the characters of the game are: Animation (defines an animation. It must have a picture and their frames. A frame is a piece of the picture responsible for the movement of animation), Sprite (the Sprite class extends the Animation class. The Sprite class contains methods that can make the image move across the screen) and Body (the Body class extends the Animation class. Like Sprite, the Body class also contains methods that can move the image, and beyond these methods it adds methods to accelerate and decelerate the image across the screen). The classes responsible for outputs in the game are: Window (defines a window where all the game elements will be drawn), Time (defines a time counter), Sound (defines the sound that will be played in the game) and Collision (it is a static class, used to check if there was a collision between two objects. The occurrence of a collision can be verified using this method in all classes, or by the Collision static class).

4. A Heuristic Based on Behaviors Comparison of Programs

We propose in this work a heuristic based on behaviors comparison of the programs with the aim of analyzing the code being generated by the students. The system is composed by many stages and modules, which are illustrated in Fig. 1.

Basically, the systems start classifying tuples of classes that are taken from both the model class, which corresponds to the teacher´s program, and student code (1). The analyzer then checks if the pairs of classes are standardized according to the properties defined by the teacher for each exercise, which corresponds to the model program (2). Thus, in the standardization phase (2), the student´s code must be standardized according to the model program so that they can be compared later. Thus, a markers structure must be filled in each of the classes of the program model. Each marker indicates the characteristics that the class must have to implement their behavior. The teacher must inform through comments, in the source code of the model program, the values of the markers in each of the classes of the program model. Then, the analyzer identifies, in the student code, the values of the markers of each class. Some of the markers that are used are: inheritance, constructor, movex, movey, keyboard, mouse, method, game object, main and game loop. Although we define basic markers, more specific markers can be defined by the teacher. If the standardization is correct for each pair of classes, the analyzer checks for JPlay sequential pattern (3) and starts the comparison process (4), classifying variables pairs between pair of classes, otherwise the analyzer requests the student adaptations in the code and the process returns to the beginning.

5. Jplay Sequential Pattern

Our proposal is based on design patterns of sequence used in JPlay framework. A JPlay sequential pattern is a code sequence in the program based on JPlay that must always happen when the program is correct.

JPlay follows a typical game framework pattern: objects, also called as game objects, are initially defined. A loop is initiated (also called as a game loop) and each iteration corresponds to a frame being produced. In this loop all game objects are updated with their corresponding logic (coming from an AI algorithm, physic algorithm or even from the user interface sequence).

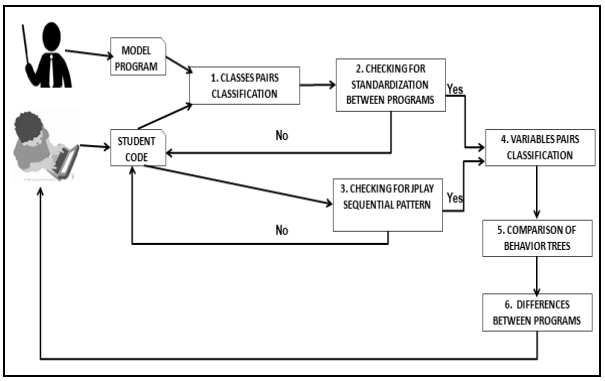

Figure 1. Stages of the Semantic Analyzer Process

Finally, all the elements of the game are drawn in the screen. Fig. 2 illustrates the typical sequence of activities of the JPlay framework:
A main method must be defined in the initialization of the program (1);

- In the body of the main method objects will be instantiated. One of these objects must contain a game loop (infinite loop). Finding a class that contains this loop means finding the loop execution of the program (2).

- In the class that contains the game loop (infinite loop) objects are declared (3a). At this point it is possible to check if all objects were also declared and have been instantiated (3b).

-Objects declared as Sprites, Animations and GameImages should be drawn in the window of the game; at this point it is necessary to verify if all declared objects of these types were draw (4).

- An object declared as Window should be updated in the game window; at this point it is necessary to verify if the Window object is updated. These objects must also call their update method (5).

6. Standardization model

In order to standardize the student´s code in accordance to the model program, we propose the usage of specific markers for each class at the model program. The markers must be declared at the beginning of each class of the model program. Each marker reports a value of behavior that the class should follow.

In order to correctly evaluate the class by the analyzer, it is necessary that the values of its markers are in agreement with the markers of its class pair in the program model. Some of the most important markers are:

- Inheritance: Identifies the super class.
- Constructor: The constructor of the class must be declared.
- Move X: A method to move the object on the x axis must be declared. The keywords used to find this behavior are called this.x (attribute used in JPlay) or movex (method used in JPlay).
- Move Y: A method to move the object on the y axis must be declared. The keywords used to find this behavior are called this.y (attribute used in JPlay) or movey (method used in JPlay).
- Keyboard: Identifies the use of the keyboard in this class. The keyword used to find this behavior is called keydown (method used in JPlay).

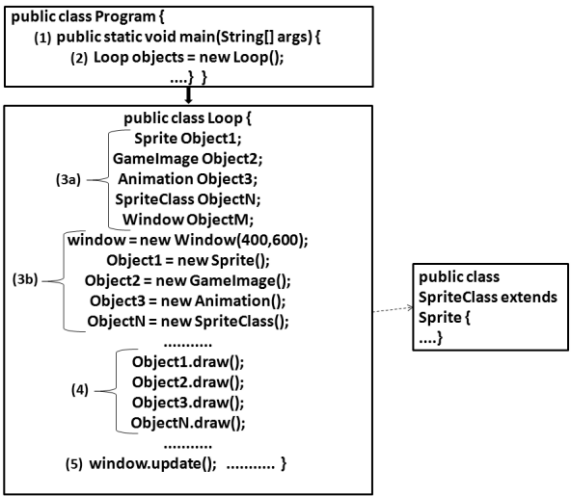

Figure 2. JPlay Sequential Pattern

- Mouse: Identifies the use of the mouse in this class. The keyword used to find this behavior is called isleftButtonPressed (method used in JPlay).
- Object Game: Identifies instantiated game objects in this class.
- Main: Identifies a main method in this class.
- Game loop: Identifies an infinite loop in this class.
- Comment: the first comment identifies the general behavior of the class. The others comments are specific and are associated with each marker. Just below of a marker follows a comment explaining the behavior associated. The comments are used to inform the student about the behaviors of the class.

The example illustrated on Fig. 3 shows the markers of the “Ball” class. According to Fig. 3, the “Ball” class must inherit from Sprite (inheritance), and it has three methods: the constructor (constructor), a method to move the ball in the x-axis (moveX) and a method to move the ball in the y-axis (moveY).

7. Classes Pairs Classification

Our proposal in Santos et al. [2013] affirm that since Badros approach allows the preservation of the source code and our method needs a subsequent semantic analysis, we initially convert all classes from a developed program into a XML representation, based on the proposed JavaML method. Due the increase of tag’s representation from JavaML 2.0, we ignored this update.

```
/**
 *
 @comment= "The Ball class must inherit from Sprite, must has two attributes of boolean type
           that control the movement of the ball to left and right.
           The class must has three methods: a constructor;
           a method to move the ball in the x-axis;
           and a method to move the ball in the y-axis."

 @inheritance=Sprite
 @comment= "The Ball class must inherit from Sprite"

 @method=3
 @comment="The class must has three methods: a constructor;
          a method to move the ball in the x-axis;
          and a method to move the ball in the y-axis."
 @constructor
 @comment="The class must has a constructor method"

 @moveX
 @comment="The class must has a method to move the ball in the x-axis"

 @moveY
 @comment="The class must has a method to move the ball in the y-axis"

 *
 **/
import jplay.Sprite;
public class Ball extends Sprite{
      /*
```

Figure 3. Examples of markers being applied at the "Ball" class in the model program

In order to semantically analyze a code under development, the programmer must select in a repository another program which will act as a model program for the comparison. The analysis consists in compare pairs of classes. Every class of the programmer code and from the program base will initially be transformed into XML by the parser. Each XML file will be read and interpreted by Java language using Document Object Model [DOM 2012].

After defining the pairs of classes, the comparisons between pairs are performed. The goal on this stage consists on identifying the pairs of classes that have higher similarity. The analysis of similarity between pairs of classes is based on rules previously established [Santos et al. 2013]. The matrix is first initialized with its similarity values, represented by the "weight" attribute, equal to 0 (zero) in all its elements. Other attributes are initialized with null.

There are specific rules to find the first and second pair. In order to classify the first pair of similar classes we establish as a single rule that the class contains the "main" method. When it finds two classes containing this method, the weight value is 2 and the "main" attribute value is true. After, in order to classify the second pair of similar classes we establish only one rule: if the class contains the game loop. When it finds two classes containing the game loop, the weight value is 2 and the "gameloop" attribute value is true. The others attributes are calculated according to the rules defined in [Santos et al. 2013] in order to be performed more analysis of the student program.

After the first and second pair of similar classes being classified, the analyzer will take the rest of the classes of the program to be processed. This classification is performed in levels. First, the analyzer will search pairs of classes that extend from the same super class. In Fig. 4, for example, the "MyBall" class belongs to student program and the "Ball" class belongs to model program. During the classification the analyzer verifies that the "MyBall" and "Ball" classes extend from the same Sprite class. The analyzer verifies the list of variables of each one of the classes of the program that extend the same super class. In Fig. 5, for example, the "MyBall" and the "MyBar" class belong to the student program and extend Sprite class, the "Ball" and "Bar" class belong to model program and also extend Sprite class. Then, all the combinations between these classes of the two programs will be analyzed with the goal of finding the pairs of classes more similar. In this step the combinations are performed by comparing the lists of variables of each class from the programs, as shown in Fig. 6. Each variable list contains the variable type and the number of variables of each type. The algorithm compares each pair of lists and calculates the difference between the values of variables of same type, after the similarity weight assigned the value of the sum of the results. Then, the pair having the lowest weight is rated as the most similar pair. Generally the comparison between the lists of variables each class does not get full precision. Thus, the results may be close to reality, but not entirely correct. In order to obtain greater accuracy in the result, the algorithm performs a second comparison based on lists of behaviors. At the next comparison, the algorithm generates lists of behaviors containing the type of behavior and the value of that type of behavior. The algorithm compares each pair of lists again and then the pair containing the smallest weight is classified as most similar pair. At this level, all pairs of similar classes are defined, and the pairs classified should be compared.

8. Variables Pairs Classification And Comparison Of Behavior Trees

After defining all pairs of similar classes, variables of the same type in each of the classes of the pair should be compared according to their behavior and then pairs of variables should be classified. The algorithm compares the variables of the same type using a list of behaviors and then the pair containing the smallest weight is rated as the most similar pair of variables.

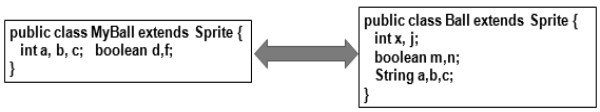

Figure 4. Example of "MyBall" and "Ball" classes code that extend the same Sprite super class

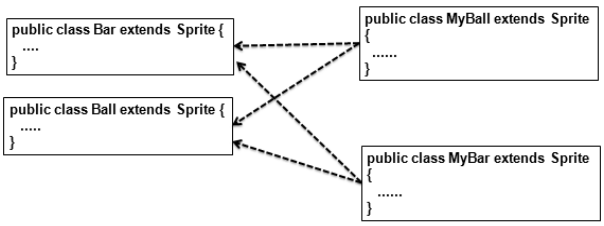

Figure 5. Example of classes from student program and model program that will be combined

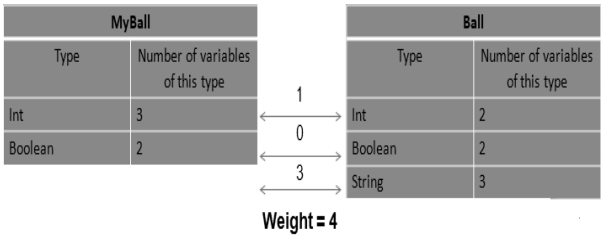

Figure 6. Example of the comparison between "MyBall" and "Ball" classes

After that, similar pairs of variables are defined and then analyzed. In order to compare these variables, we propose a tree structure, called behavior tree. This structure contains all the behaviors of a variable, which in our case we defined as 3 possible types:

- Assignment: It is identified when happens an assignment statement in the class;
- Conditional: It is identified when happens a conditional command in the class;
- Loop: It is identified when happens a loop command in the class.

The example of the Fig. 7 shows an assignment behavior of the "left" variable in the constructor of the "Ball" class.

We build a behaviors tree to each variable, and the analyzer compares the behavior tree of a variable of a class that belongs to the student program with the behavior tree of its similar variable from model program. The behavioral trees of both the variables are compared and when behaviors differences occur it is possible that the student program has an error.

In this point, when behaviors differences are identified in the trees, the analyzer makes suggestions about these behaviors differences found in the student program and give clues that may indicate a possible semantic error. The suggestions are defined as comments in the source code of the program model. Each comment must be previously edited predicting possible suggestions to the student. For example, in Fig. 7, if the assignment behavior is not identified in the behavior tree of the similar variable in the student program, the following comment associated with the behavior will be captured and suggested to the student:

- *"In the class constructor, initialize the attribute that controls the initial movement of the ball to the left or right."*

The example in Fig. 8 shows the behaviors tree of the "left" variable in the "Ball" class of the model program. The example in Fig. 9 shows the behavior tree of the similar "carry2" variable of the student program, where in this case it is found a difference (one more assignment behavior).

```
*/
public Ball(String Imagem){
      super(Imagem);

      /*
      In the class constructor, initialize the attribute that controls the initial
      movement of the ball to the left or right;
      */
      left=true;
```

Figure 7. Example of the assignment behavior of the variable "left" in the "Ball" class of model program

Since the analyzer found a difference in the assignment behavior at the comparison between the behaviors trees of the "left" and "carry2" variables, thus all comments related to the assignment behavior in the model program will be suggested to the student:

- *"In the class constructor, initialize the attribute that controls the initial movement of the ball to the left or right."*
- *"Modify the movement of the ball if the ball position at the X axis is less than minimum limit of the game window and it is going to the left."*
- *"Modify the movement of the ball if the ball position in the X axis is greater than maximum limit of the game window and it is going to the right. Consider this case the width of the ball. Example: maximum limit - width of the ball."*

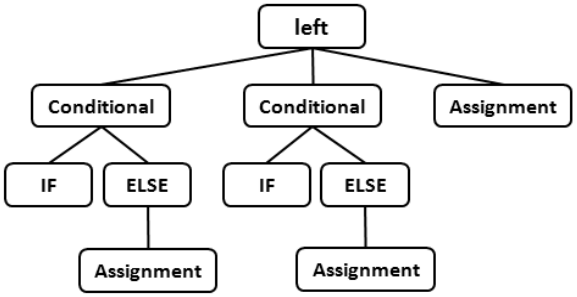

Figure 8. Example of the behaviors tree of the variable "left" in the "Ball" class of model program

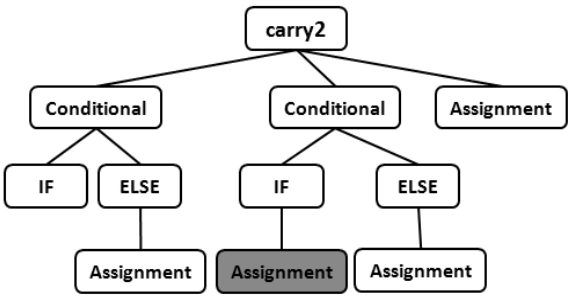

Figure 9. Example of the behaviors tree of the "carry2" variable in the "MyBall" class of student program

In this example, one more assignment behavior in the student program does not necessarily change the behavior of "MyBall" class, depending on the value that was assigned to "carry2" variable, so the analyzer

does not have to give a totally accurate result for an inappropriate behavior in "MyBall" class of student program but it can give a suggestion closer to the truth based on the difference found by comparing the trees.

9. Results

In this paper we propose a novel heuristic that checks a JAVA code and guide a student for a specific game development, giving clues of possible semantic failures, within a game oriented framework.

We developed a validation scenario in a classroom with 10 students of the integrated high school of the Informatics course. The students were proposed to develop a "BrickBreak" game, using JPlay framework. In this paper, we analyzed the "Ball" and "Bar" classes of the 10 codes developed, based on the model program developed by the teacher. The results analysis presented here are related with two levels of analysis (according to Fig. 1): Checking for standardization between programs (2) and Comparison of the behavior trees (5).

We evaluated the results according to behaviors expected for each class. The "Ball" class has two basic specifications: it must move in the x and y axis and collide with objects (bar and block). The "Bar" class also has two specifications: it should move through the keyboard control, for both right and the left sides and collide with the ball.

In case of the standardization model the results were marked as positive when the analyzer does not detect differences between the student program and the standardization model and negative when the analyzer detects differences between them.

For the behavioral trees, we evaluated the results as false-positive when the analyzer does not detected differences, but the behavior of the object is not correct and false-negative when the analyzer detects differences but the behavior of the object is correct. It is considered, negative when the analyzer detects differences that really it exists and positive when the analyzer does not detect differences and the behavior of object is correct.

Table I summarizes the results for the evaluation strategy for the "Ball" class example.

Students 1 and 4 used one method more than requested in the standardization of model. The analyzer checks the difference between the students program and standardization model and prints the following suggestion for the student:

- *"You probably defined more methods than the necessary."*

About the comparison of behavior trees, there are differences in the program of the students 1, 3, 4, 5, 6, 7, 9 and 10, according to Table I. The differences in students 1, 3, 4, 5, 6 are defined as false-negative because these differences don't modify the behavior of the program. Most of the differences in this class refer to assignment behaviors and are false-negative results. For example, the student 1 defines the "movex" and "movey" variables. These variables are compared with "left" and "up" of the "Ball" class of the model program. The results of comparisons between "movey" and "left" and "up" doesn´t show differences (the values are equal to 0). However, the results of comparisons between "movex" and "left" and "up" show differences (the values are to 2), according to Table II and Table III. The results of comparisons between the variables are not exact, and ties happen, thus the "movex" variable is combined with "left" and "up" variables, and all the comments relating to assignment behaviors of the "left" and "up" variables are suggested to the student. The student 1 result, in Table I, is false-negative because the difference found don't modify the general behavior of the student program. The same result happens with students 3, 4, 6, 5, 7 and 9. The results of the comparison of behavior trees show that the analyzer is not able to be totally accurate, but is able to make a suggestion closer to the truth to the student. The results of analysis according with the standardization between programs are most accurate, how much more standardized the student program, will be found less inaccuracy on comparison of the behavior trees. In case of the student 10, the program is totally incorrect according to the expected general behavior, the program is not according to the standardization and it is not possible compare variables, form pair of variables and compare the behaviors trees. Thus all the comments relating to behaviors of the "Ball" class are suggested to the student.

TABLE I. TABLE EVALUATION FOR THE *BALL* CLASS

Student	**Evaluation for the *Ball* class**		
	Status program	***Analysis according with the standardization between programs***	***Comparison of behavior trees***
1	Incorrect (with the standard model)	Negative	False-negative
2	Correct	Positive	Positive
3	Correct	Positive	False-negative
4	Incorrect (with the standard model)	Negative	False-negative
5	Correct	Positive	False-negative
6	Correct	Positive	False-negative
7	Correct	Positive	False-negative
8	Correct	Positive	Positive
9	Correct	Positive	False-negative

Student	Evaluation for the *Ball* class		
	Status program	***Analysis according with the standardization between programs***	***Comparison of behavior trees***
10	Incorrect (with the standard model and general behavior of program)	Negative	Negative

TABLE II. TABLE COMPARISON BETWEEN "MOVEX" (STUDENT 1) AND "LEFT" (MODEL PROGRAM) VARIABLES

Comparison between "*movex*" (student 1) and "*left*" (model program) variables			
Type of Behavior	***Number of behaviors in "movex" variable***	***Number of behaviors in "left" variable***	***Difference***
Assignment	5	3	***2***
Conditional	2	2	0
		Total differences	***2***

TABLE III. TABLE COMPARISON BETWEEN "MOVEX" (STUDENT 1) AND "UP" (MODEL PROGRAM) VARIABLES

Comparison between "*movex*" (student 1) and "*up*" (model program)			
Type of Behavior	***Number of behaviors in "movex" variable***	***Number of behaviors in "up" variable***	***Difference***
Assignment	5	3	***2***
Conditional	2	2	0
		Total differences	***2***

The results of the evaluation of the "*Bar*" class at the example is summarized by Table IV. The differences in students 1, 2, 4, 5, 6, 7 and 8 are defined as false-negative because these differences don't modify the program behavior. Most of the differences in this class refer to standardization because the analyzer did not find occurrence of the key words "*Keyboard*" and "*keydown*" in the programs of all the students, according Tab. IV. The standardization model has defined JPlay objects and methods with these names with the purpose of being used to implement the movement of the bar through the keyboard control. Students 1, 2, 4, 5, 6, 7 and 8 implemented the keyboard control in another class program. Some of them even used another JPlay method (called movex()) to accomplish the same behavior, that led to the indication of the errors, although the behavior of the programs were correct. Then the analyzer prints the following comment for the student:

- *"Please, check if you defined the movement of the bar through the keyboard control."*

In the analysis of the behavior trees of students 1, 2, 4, 5, 6, 7 and 8 there are differences in all the behaviors of the variables of the "*Bar*" class comparing with the model program, because the analyzer could not mount pairs of similar variables, thus all the comments associated will be printed as simple suggestions for the student.

In the special case of the student 3, 9 and 10, the program behavior is actually incorrect, because the bar is not moving properly. Thus, it is not possible compare variables, form pair of variables and compare the behaviors trees. Then the following comments are shown as suggestions:

- *"Define the movement of the bar to the right through the use of the keyboard using the KeyDown() method. Check which is, in the game window, the maximum value of the right margin."*
- *"Increase the movement of the bar on the x axis, making the bar moves to the right."*
- *"Define the movement of the bar to the left through the use of the keyboard using the KeyDown() method. Check which is, in the game window, the minimum value of the left margin."*
- *"Decrement the movement of the bar on the x axis, making the bar move to the left."*

TABLE IV. TABLE EVALUATION FOR THE BAR CLASS

Student	Evaluation for *Bar* class		
	Status program	***Analysis according with the standardization between programs***	***Comparison of behavior trees***
1	Incorrect (with the standard model)	Negative	False-negative
2	Incorrect (with the standard model)	Negative	False-negative
3	Incorrect (with the standard model and with the behavior)	Negative	Negative
4	Incorrect (with the standard model)	Negative	False-negative
5	Incorrect (with the standard model)	Negative	False-negative
6	Incorrect (with the standard model)	Negative	False-negative
7	Incorrect (with the standard model)	Negative	False-negative
8	Incorrect (with the standard model)	Negative	False-negative
9	Incorrect	Negative	Negative
10	Incorrect	Negative	Negative

10. Conclusion

This paper presents a novel heuristic strategy based in an analyzer that interpreting semantically a JPlay code, guide a student for a specific game development process. Although our implementations and tests are related to JPlay framework, our proposal can easily be adapted to other program patterns.

The goals of the analyzer are to interpret semantically a Java program that uses JPlay and return results of this analysis to the student. Our proposal brings significant contributions to researchers working in the field of programming education and software engineering, having as main contributions the architecture for classification of similar classes and the definition of the data structure (behavior tree) starting from the behaviors of variables. Our paper also contributes in the sense that introduces a tool able to semantically interpret code built by students, returning results, pointing out problems and suggesting solutions.

As future work we intend to develop a tutoring interface in order to manage the results received by the analyzer and the communication with the student. Also, as future work, we intend to improve the efficiency of the algorithm using classification approaches.

References

ADAM, A., LAURENT, J., "*LAURA, a system to debug student programs"*. artificial intelligence, v.15, n.1, pp. 75-122, 1980.

ALEXIS, V. de A., DELLER, J. F. *"Aplicando Padrões de Seleção no Ensino de Programação de Computadores para Estudantes do Primeiro Ano do Ensino Médio Integrado"*. In X Encontro Anual de Computação – EnAComp, 2013.

ALLEN, E., CARTWRIGHT, R. AND STOLER, B. *"Drjava: a lightweight pedagogic environment for java"*. SIGCSE Bull., 34(1):137-141, 2002.

ALLOWATT, A. AND EDWARDS, S. *"Ide support for test-driven development and automated grading in both java and c++"*. In eclipse "05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, pages 100-104, New York, NY, USA. ACM Press. 2005.

BARBOSA, L. S., FERNANDES, T.C.B., CAMPOS, A. M. C. *"Takkou: Uma Ferramenta Proposta ao Ensino de Algoritmos"*. In: XXXI CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO - WEI XIX WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO, 2011, Natal. WEI XIX WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO, 2011.

BOTELHO, C. A. *"Sistemas Tutores no domínio da programação"*. Revista de Informática Aplicada/Journal of Applied Computing, v.4, n. 1, 2010.

DELGADO, K. V. *"Diagnóstico baseado em modelos num sistema inteligente para programação com padrões pedagógicos"*. Master's dissertation, Institute of Mathematics and Statistics. 2005.

DOM, available in http://www.w3.org/DOM/. Accessed in November 2012.

FEIJÓ, B., CLUA, E., DA SILVA, F.S.C. Introdução à Ciência da Computação com Jogos: Aprendendo a Programar com Entretenimento. Campos Elsevier.1º ed. 2010.

JOHNSON, W. L., SOLOWAY E.. "Proust: Knowledge-based program understanding". In ICSE 84: Proceedings of the 7th international conference on Software engineering, pp. 369-380, Piscataway, NJ, USA, 1984. IEEE Press.

JPLAY, available in http://www.ic.uff.br/jplay/. Accessed in April 2012.

KOLLING, M., QUIG, B., PATTERN, A., AND ROSENBERG, J. "The BlueJ system and its pedagogy". Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, 13(4):249-268,2003.

PINHEIRO, W.R., BARROS, L.N., Kon, F.. "AAAP: Ambiente de Apoio ao Aprendizado de Programação". In Workshop de Ambientes de Apoio à Aprendizagem de Algoritmos e Programação, São Paulo, 2007.

RAPKIEWICZ, C. E. ET AL."Estratégicas pedagógicas no ensino de algoritmos e programação associadas ao uso de jogos educacionais". RENOTE, v.4, n.2, 2006.

SANTOS, E.C.O., BATISTA, G.B., CLUA, E.W.G. *"A Knowledge Modeling System for Semantic Analysis of Games Applied to Programming Education"*. In SEKE 2013: Proceedings of the twenty-fifth International Conference on Software Engineering & Knoledge Engineering, pp-668-673, Boston, June 27-29, 2013.

SANTOS, N.S.R.S., RAPKIEWICZ, C.E.. "Ensinando princípios básicos de programação utilizando jogos educativos em um programa de inclusão digital". In: SBGAMES - VI Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital, 2007, São Leopoldo - RS.

TRAETTEBERG, H., AALBERG. T. "Jexercise: a specification-based and test-driven exercise support plugin for eclipse". In eclipse "06:Proeedings of 2006 OOPSLA workshop on eclipse technology eXchange, pages 70-74, New York, NY, USA. ACM Press. 2006.