

UNIVERSIDADE FEDERAL FLUMINENSE

**UMA HEURÍSTICA DE COMPARAÇÃO DE COMPORTAMENTOS PARA  
ANÁLISE SEMÂNTICA DE JOGOS APLICADA A APRENDIZAGEM DE  
PROGRAMAÇÃO**

ELANNE CRISTINA OLIVEIRA DOS SANTOS

Niterói

2016

Elanne Cristina Oliveira dos Santos

**UMA HEURÍSTICA DE COMPARAÇÃO DE COMPORTAMENTOS PARA  
ANÁLISE SEMÂNTICA DE JOGOS APLICADA A APRENDIZAGEM DE  
PROGRAMAÇÃO**

Tese apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Doutor em Ciências da Computação. Área de Concentração: Computação Visual.

Orientador: Prof. Dr. ESTEBAN WALTER GONZALEZ CLUA

Niterói

2016

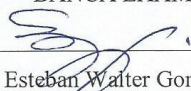
ELANNE CRISTINA OLIVEIRA DOS SANTOS

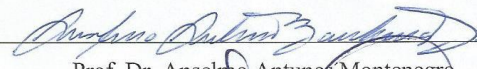
UMA HEURÍSTICA DE COMPARAÇÃO DE COMPORTAMENTOS PARA ANÁLISE  
SEMÂNTICA DE JOGOS APLICADA A APRENDIZAGEM DE PROGRAMAÇÃO

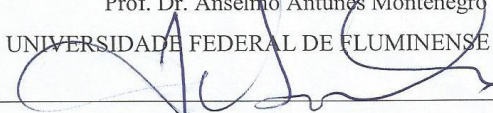
Tese apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Doutor em Ciências da Computação. Área de Concentração: Computação Visual.

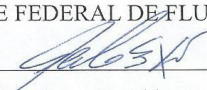
Aprovada por:

BANCA EXAMINADORA

  
Prof. Dr. Esteban Walter Gonzalez Clua – Orientador  
UNIVERSIDADE FEDERAL DE FLUMINENSE (IC-UFF)

  
Prof. Dr. Anselmo Antunes Montenegro  
UNIVERSIDADE FEDERAL DE FLUMINENSE (IC-UFF)

  
Prof. Dr. José Viterbo Filho  
UNIVERSIDADE FEDERAL DE FLUMINENSE (IC-UFF)

  
Prof. Dr. Geraldo Xexéo  
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO (UFRJ)

  
Prof. Dr. Flávio Soares Correa da Silva  
UNIVERSIDADE DE SÃO PAULO (USP)

Niterói, 3 de março de 2016.

A Deus, por ter conferido a mim esta missão em profunda confiança.

## **AGRADECIMENTOS**

Agradeço a Deus, que tendo me conferido esta missão, esteve presente em todos os passos, provendo pequenos milagres em todas as dificuldades que iam surgindo durante o caminho.

A meu ex-aluno e amigo, Gleison Brito Batista, que esteve presente na maior parte da jornada, sempre ao meu lado buscando soluções para os problemas envolvidos no trabalho. Juntos, nossos longos períodos de estudo foram muito gratificantes.

Ao meu orientador, professor Dr. Esteban Walter Gonzalez Clua, pela demonstração de confiança desde o início, sempre acreditando no meu compromisso e dedicação. Por sua orientação atenta e presente, apesar de estarmos geograficamente distantes a maior parte do tempo.

Aos meus alunos do IFPI (Instituto Federal de Educação, Ciência e Tecnologia do Piauí) que, empolgadamente, aceitaram participar dos mini-cursos "Programação de jogos com o framework JPlay" e comprometeram suas manhãs de sábado nesta atividade. Em particular, ao meu ex-aluno Victor Hugo Vieira de Sousa, que ministrou, junto a mim, as aulas dos mini-cursos.

Aos meus companheiros de Doutorado em Computação, nossa amizade e apoio mútuo foi muito importante para tornar a jornada possível.

E ao IFPI (Instituto Federal de Educação, Ciência e Tecnologia do Piauí) que contribuiu com o apoio financeiro necessário à realização do convênio DINTER (Doutorado InterInstitucional) junto à UFF (Universidade Federal de Fluminense).

## RESUMO

Aprendizagem e ensino de algoritmos e programação tem sido um importante desafio, não somente em universidades, mas também em escolas. Enquanto os ambientes de programação possuem atualmente sofisticadas ferramentas para detectar e indicar erros sintáticos, o mesmo não ocorre em relação a erros semânticos, podendo frustrar programadores aprendizes e inexperientes. Com a proposta de fazer este processo mais eficiente e lúdico, o JPlay - ferramenta previamente desenvolvida com o propósito de ensinar a programar com jogos - foi utilizado como um framework que possui estruturas para o desenvolvimento de jogos simples. Nesta tese é apresentado um novo sistema, baseado em uma heurística de comparação de comportamentos, para análise semântica de jogos aplicada ao ensino de programação, baseado no JPlay. O analisador semântico interpreta o código do aluno e retorna a análise detalhada de suas funcionalidades. O processo consiste na comparação do programa do aluno com um programa modelo, previamente definido, procurando por padrões de comportamentos similares e estruturas entre eles. Foi desenvolvido uma heurística baseada na comparação de comportamentos entre pares de classes similares e implementada uma ferramenta que permite esta análise. Nesta tese apresentamos também 3 (três) estudos de caso com base nos resultados obtidos. O primeiro estudo de caso é um programa que possui somente uma classe (classe "*main*") de projeto, enquanto que o segundo e o terceiro são projetos compostos por várias classes, caracterizando uma complexidade maior no uso do paradigma de orientação a objetos.

Palavras-chave: programação; JPlay; classificação; aprendizagem; ensino; jogos.

## ABSTRACT

Teaching and learning algorithms and programming is being an important challenge, not only in universities, but also in schools. Whereas programming environments currently have sophisticated tools to detect and indicate syntax errors, the same does not occur with respect to semantic errors, which can frustrate learners and novice programmers. With the purpose of making this process more efficient and enjoyable, we propose the usage of JPlay – a previous developed library for teaching programming with games - providing structures and architectures for developing simple games. In this thesis we present a new modeling system for semantic analysis of games applies to programming learning within JPlay. The analyzer semantically interprets the student code and return detailed analysis of its functionalities. Our process consists of comparing the student program with a model program, previously defined, searching for similar behaviors and structures between them. We developed a heuristic for pairs of similar classes behaviors comparison and implemented a tool that enables the proposed analysis. In this thesis we also propose 3 (three) case studies based on the results. The first case is a program that has only one class (class "*main*") project, whereas the second and third are projects composed of several classes, featuring greater complexity in the use of object oriented paradigm.

Keywords: programming; JPlay; classification; learning; teaching; games.

## LISTA DE ILUSTRAÇÕES

<b>Figura 2.1:</b> <i>View</i> do plugin JExercise .....	25
<b>Figura 2.2:</b> Os ícones usados na <i>view</i> do JExercise .....	26
<b>Figura 2.3:</b> Janela do ambiente de programação BlueJ .....	27
<b>Figura 2.4:</b> Criação de classes de testes no BlueJ .....	27
<b>Figura 2.5:</b> <i>View</i> de interações do plugin DrJava .....	28
<b>Figura 2.6:</b> Janela de submissão de projeto no Web-CAT .....	29
<b>Figura 2.7:</b> Janela de resultados do Web-Cat .....	30
<b>Figura 2.8:</b> Janela de edição de um exercício no Run.Codes .....	31
<b>Figura 2.9:</b> Exemplo de decomposição de objetivos no sistema Proust .....	32
<b>Figura 3.1:</b> Diagrama de classes JPlay para interação jogador e jogo .....	43
<b>Figura 3.2:</b> Diagrama padrão de sequência para objetos do tipo da classe <i>Keyboard</i> .....	44
<b>Figura 3.3:</b> Diagrama padrão de sequência para objetos do tipo da classe <i>Mouse</i> .....	45
<b>Figura 3.4:</b> Diagrama de classes JPlay para objetos dinâmicos do jogo .....	46
<b>Figura 3.5:</b> Diagrama padrão de sequência para personagens do jogo do tipo <i>Animation</i> .....	47
<b>Figura 3.6:</b> Diagrama padrão de sequência para personagens do jogo do tipo <i>Sprite</i> .....	48
<b>Figura 3.7:</b> Diagrama de classes para saídas do jogo .....	49
<b>Figura 3.8:</b> Diagrama padrão de sequência para saídas do jogo do tipo <i>Window</i> .....	50
<b>Figura 3.9:</b> Diagrama padrão de sequência para saídas do jogo do tipo <i>Time</i> .....	50
<b>Figura 3.10:</b> Diagrama padrão de sequência para saídas do jogo do tipo <i>Sound</i> .....	51
<b>Figura 4.1:</b> Diagrama de classes AST .....	56
<b>Figura 4.2:</b> Sequência utilizada para análise de código-fonte via AST .....	58
<b>Figura 4.3:</b> Representação JavaML de Badros .....	60
<b>Figura 4.4:</b> Trecho de código-fonte da classe "Bar.java" .....	60
<b>Figura 4.5:</b> Trecho de código XML representando a classe "Bar.java" .....	61
<b>Figura 4.6:</b> Trecho de uso <i>id</i> e <i>idref</i> na representação JavaML de Badros .....	65
<b>Figura 4.7:</b> Trecho de código-fonte da classe "Barra.java" .....	66
<b>Figura 4.8:</b> Trecho de código XML (arquivo "Barra.java.XML") representando a classe "Barra.java" .....	67
<b>Figura 5.1:</b> Etapas do processo de análise .....	72
<b>Figura 5.2:</b> Escolhendo o programa modelo no <i>plugin</i> analisador semântico .....	73



<b>Figura 5.3:</b> Etapas do processo de classificação dos pares de classes .....	74
<b>Figura 5.4:</b> Diagrama de um Elemento e de um Par de Classes .....	75
<b>Figura 5.5:</b> Exemplo das classes “A” e “B” que herdam da mesma superclasse <i>Sprite</i> .....	77
<b>Figura 5.6:</b> Exemplo das classes do programa modelo (“Bar” e “Ball”) e das classes do programa do estudante (“MyBar” e “MyBall”) que serão combinadas.....	78
<b>Figura 5.7:</b> Exemplo de comparação entre as variáveis das classes “MyBall” e “Ball” .....	78
<b>Figura 5.8:</b> Exemplo de comparação entre os comportamentos das classes “MyBall” e “Ball” .....	79
<b>Figura 5.9:</b> Etapas do processo de padronização .....	80
<b>Figura 5.10:</b> Exemplo de marcadores aplicados na classe “Ball” do programa modelo .....	83
<b>Figura 5.11:</b> Padrão Sequencial JPlay .....	85
<b>Figura 5.12:</b> Exemplo de comportamento “assignment” da variável “left” da classe “Ball” do programa modelo .....	86
<b>Figura 5.13:</b> Árvore de comportamento da variável “left” da classe “Ball” do programa modelo.....	87
<b>Figura 5.14:</b> Árvore de comportamento da variável “carry2” da classe “MyBall” do programa do estudante .....	88
<b>Figura 5.15:</b> Janela do analisador perguntando ao usuário se deve mostrar as árvores de comportamento da variável “bola” do programa modelo e da variável “aluno_bola” do programa do estudante .....	89
<b>Figura 5.16:</b> Árvore de comportamento da variável “bola” da classe “jogo” do programa do estudante.....	89
<b>Figura 5.17:</b> Árvore de comportamento da variável “aluno_bola” da classe “jogo” do programa modelo .....	90
<b>Figura 6.1:</b> Quando o usuário pressionar o botão “LEFT” do mouse, as bolas devem ser desenhadas na janela do jogo e adicionadas na lista de bolas.....	98
<b>Figura 6.2:</b> Mensagem na janela do jogo informando que já existem bolas na área apontada pelo botão do mouse .....	99
<b>Figura 6.3:</b> Janela do jogo “BrickBreak” com indicação dos personagens (objetos) do jogo .....	114
<b>Figura 6.4:</b> Janela do jogo “Ping Pong” com indicação dos personagens (objetos) do jogo	140

## LISTA DE TABELAS

<b>Tabela 2.1:</b> COMPARATIVO ENTRE OS SISTEMAS QUE USAM TÉCNICAS DE APOIO AO ENSINO DE PROGRAMAÇÃO.....	38
<b>Tabela 6.1:</b> MATRIZ DE CONFUSÃO .....	94
<b>Tabela 6.2:</b> MEDIDAS PARA AFERIR A QUALIDADE DO SISTEMA ANALISADOR	97
<b>Tabela 6.3:</b> MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE PROGRAMAS ANALISADOS PARA O PRIMEIRO ESTUDO DE CASO .....	100
<b>Tabela 6.4:</b> MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O PRIMEIRO ESTUDO DE CASO .....	100
<b>Tabela 6.5:</b> AVALIAÇÃO DOS COMENTÁRIOS DE ACORDO COM A PADRONIZAÇÃO .....	104
<b>Tabela 6.6:</b> AVALIAÇÃO DOS COMENTÁRIOS DE ACORDO COM A COMPARAÇÃO DOS PARES DE VARIÁVEIS .....	105
<b>Tabela 6.7:</b> AVALIAÇÃO DOS COMENTÁRIOS OBTIDOS A PARTIR DAS VARIÁVEIS SEM PAR .....	109
<b>Tabela 6.8:</b> RELAÇÃO DOS PROGRAMAS COM COMPORTAMENTO GERAL INCORRETOS.....	117
<b>Tabela 6.9:</b> MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE CLASSES DE PROGRAMA ANALISADAS PARA O SEGUNDO ESTUDO DE CASO .....	118
<b>Tabela 6.10:</b> MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O SEGUNDO ESTUDO DE CASO .....	119
<b>Tabela 6.11:</b> AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BOLA” DE ACORDO COM A PADRONIZAÇÃO .....	122
<b>Tabela 6.12:</b> AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BARRA” DE ACORDO COM A PADRONIZAÇÃO .....	124
<b>Tabela 6.13:</b> AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “JOGO” DE ACORDO COM A PADRONIZAÇÃO .....	126

<b>Tabela 6.14:</b> AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BOLA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR .....	128
<b>Tabela 6.15:</b> AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BARRA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR .....	131
<b>Tabela 6.16:</b> AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “JOGO” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR .....	133
<b>Tabela 6.17:</b> AVALIAÇÃO DOS RESULTADOS QUANTO A PADRONIZAÇÃO E COMPARAÇÃO DE VARIÁVEIS NAS CLASSES DO JOGO .....	138
<b>Tabela 6.18:</b> AVALIAÇÃO DOS RESULTADOS QUANTO A PADRONIZAÇÃO E COMPARAÇÃO DE VARIÁVEIS NAS CLASSES DO JOGO .....	139
<b>Tabela 6.19:</b> RELAÇÃO DOS PROGRAMAS COM COMPORTAMENTO GERAL INCORRETOS.....	143
<b>Tabela 6.20:</b> MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE CLASSES DE PROGRAMA ANALISADAS PARA O SEGUNDO ESTUDO DE CASO .....	144
<b>Tabela 6.21:</b> MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O SEGUNDO ESTUDO DE CASO .....	145
<b>Tabela 6.22:</b> AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BOLA” DE ACORDO COM A PADRONIZAÇÃO .....	148
<b>Tabela 6.23:</b> AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BARRA” DE ACORDO COM A PADRONIZAÇÃO .....	150
<b>Tabela 6.24:</b> AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “JOGO” DE ACORDO COM A PADRONIZAÇÃO .....	154
<b>Tabela 6.25:</b> AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BOLA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR .....	156
<b>Tabela 6.26:</b> AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BARRA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR .....	158
<b>Tabela 6.27:</b> AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “JOGO” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR .....	160
<b>Tabela 6.28:</b> MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE CLASSES DE PROGRAMA ANALISADAS PARA O TERCEIRO ESTUDO DE CASO ( <i>modificando-se os critérios Negativo-Negativo e Falso-Negativo</i> )..	162

**Tabela 6.29:** MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O TERCEIRO ESTUDO DE CASO (*modificando-se o critério Negativo-Negativo e Falso-Negativo*)..... 163

# SUMÁRIO

<b>Capítulo 1 – Introdução.....</b>	<b>16</b>
1.1 Motivação .....	16
1.2 Definição do problema .....	17
1.3 Objetivos.....	20
1.3.1 Objetivos gerais.....	20
1.3.2 Objetivos específicos.....	21
1.4 Contribuições alcançadas .....	21
1.5 Estrutura da tese.....	21
<b>Capítulo 2 – Trabalhos relacionados.....</b>	<b>23</b>
2.1 Introdução .....	23
2.2 Técnicas de apoio ao ensino de programação .....	24
2.2.1 Desenvolvimento baseado em testes unitários .....	24
2.2.2 Propostas de ambientes de programação.....	26
2.2.3 Avaliação automática .....	28
2.2.4 Sistemas de tutoria usando padrões de programação .....	31
2.2.5 Sistema de depuração automática .....	34
2.3 Representação de código fonte em xml.....	35
2.4 Considerações finais .....	36
<b>Capítulo 3 - JPlay .....</b>	<b>41</b>
3.1 Introdução .....	41
3.2 Arquitetura JPlay: Interação entre jogo e jogador .....	42
3.3 Arquitetura JPlay: Personagens do jogo .....	44
3.4 Arquitetura JPlay: Saídas do jogo .....	48
3.5 Conclusão .....	51

<b>Capítulo 4 – Arquitetura do analisador semântico de programação .....</b>	<b>53</b>
4.1 Introdução .....	53
4.2 AST .....	54
4.3 A representação JavaML de Badros .....	59
4.4 DOM .....	65
4.5 Prolog .....	66
4.6 Considerações finais .....	71
<b>Capítulo 5 O analisador semântico e a heurística baseada em comparação de comportamentos dos programas.....</b>	<b>72</b>
5.1 Introdução .....	72
5.2 Selecionando um programa java para análise.....	73
5.3 Classificando pares de classes .....	74
5.4 A padronização do programa modelo.....	80
5.5 O padrão sequencial JPlay .....	83
5.6 Classificação de pares de variáveis e comparação de árvores de comportamento ...	85
5.7 Considerações finais .....	90
<b>Capítulo 6 Resultados .....</b>	<b>91</b>
6.1 Metodologia.....	92
6.1.1 Matriz de confusão .....	93
6.1.2 Sensibilidade, especificidade, acurácia e precisão .....	95
6.1.3 As três hipóteses propostas .....	96
6.2 Primeiro estudo de caso: prova aplicada na disciplina “tópicos especiais em desenvolvimento de software” .....	97
6.2.1 Resultados obtidos apartir da medição de sensibilidade, especificidade, acurácia e precisão .....	99
6.2.2 Resultados obtidos apartir das três hipóteses propostas.....	101
6.3 Considerações gerais sobre o primeiro estudo de caso.....	112
6.4 Segundo estudo de caso: o jogo “ <i>brickbreak</i> ” .....	113

6.4.1 Resultados obtidos apartir da medição de sensibilidade, especificidade, acurácia e precisão .....	117
6.4.2 Resultados obtidos apartir das três hipóteses propostas.....	120
6.5 Considerações gerais sobre o segundo estudo de caso .....	137
6.6 Terceiro estudo de caso: o jogo “ <i>ping pong</i> ” .....	139
6.6.1 Resultados obtidos apartir da medição de sensibilidade, especificidade, acurácia e precisão .....	144
6.6.2 Resultados obtidos apartir das três hipóteses propostas.....	146
6.7 Considerações gerais sobre o terceiro estudo de caso .....	160
<b>Capítulo 7 CONCLUSÕES .....</b>	<b>165</b>
<b>Referências.....</b>	<b>169</b>
<b>anexos .....</b>	<b>172</b>
<b>ANEXO A - ENUNCIADO DA PROVA (PRIMEIRO ESTUDO DE CASO).....</b>	<b>172</b>
<b>ANEXO B - ENUNCIADO JOGO “<i>BRICKBREAK</i>” (SEGUNDO ESTUDO DE CASO) .....</b>	<b>174</b>
<b>ANEXO C - ENUNCIADO JOGO “<i>PING PONG</i>” (SEGUNDO ESTUDO DE CASO) .....</b>	<b>178</b>

## CAPÍTULO 1 – INTRODUÇÃO

### 1.1 MOTIVAÇÃO

O ensino de programação e algoritmos tem sido um grande desafio, não somente para as universidades mas também em escolas e centros de formação técnica. Estudos apontam para a dificuldade de ensino e aprendizagem de disciplinas relacionadas a algoritmo e programação, resultando em altos índices de reprovação nos cursos de computação (PINHEIRO et al.,2007) (BARBOSA et al., 2011). Um dos principais problemas está na dificuldade em aprender conceitos abstratos de programação (SANTOS, 2007). Muitas propostas para simulação destes processos foram realizadas nos últimos anos (ALLEN et al.,2002)(KOLLING et al.,2003)(ALLOWATT,2005).

A disciplina de algoritmos é considerada a base para o ensino de programação em cursos de ciência da computação. Ela tradicionalmente aborda os princípios da lógica de programação, com o objetivo de desenvolver a capacidade dos alunos de analisarem e solucionarem problemas na forma de algoritmos (RAPKIEWICZ et al.,2006). No entanto, a disciplina é geralmente um dos gargalos existentes nos cursos de graduação, pois existe dificuldade, por parte dos alunos, de compreender os conceitos abstratos da lógica inerente à área. Segundo Galhardo (2004) o ensino de linguagens de programação apresenta conceitos difíceis de serem entendidos, quando vistos em sala de aula, por se tratarem de elementos muito abstratos. Segundo Kasurinen et al. (2008) os alunos perdem o interesse sobre a programação por causa de várias estruturas complexas que devem ser aprendidas antes que qualquer programa visualmente impressionante possa ser criado.

Segundo Kasurinen et al. (2008), o ensino de habilidades fundamentais da programação é um campo que usa extensivamente diferentes tipos de ferramentas para melhorar a experiência de aprendizagem. As ferramentas oferecem uma ampla gama de diferentes abordagens para o ensino introdutório de programação.

A utilização de jogos como metodologia de ensino tem se mostrado uma importante alternativa para a educação de lógica de programação com componentes mais visuais, como ambiente de imersão, o envolvimento entre os participantes e problemas de simulação, dentre outros.

Neste contexto, o framework JPlay foi proposto e desenvolvido para o ensino de programação (FEIJÓ et al.,2010)(JPLAY,2012). JPlay é um framework desenvolvido com o objetivo de facilitar o aprendizado de programação, proporcionando um processo de



aprendizagem algorítmica relacionada com a lógica de desenvolvimento de um simples jogo 2D. O JPlay não interfere com a estrutura de programação básicas necessárias para um aprendizado correto da lógica algorítmica e não apresenta características específicas de padrões de design de jogos no código-fonte. A ferramenta permite aos alunos uma maneira fácil de desenhar e mover imagens na tela do computador e fornece métodos e objetos que ajudam a criar jogos 2D usando a linguagem Java.

Nesta tese foi desenvolvido um processo analisador baseado em uma heurística de comparação entre dois programas que utilizam o código JPlay, de forma que o processo orienta e corrige o aluno para o desenvolvimento de um jogo específico. O analisador tem a função de interpretar semanticamente um programa Java desenvolvido que usa o JPlay e retorna resultados desta análise para o estudante. O processo é baseado em várias etapas: busca e classificação dos pares de classes similares, a padronização (verifica a padronização do programa do estudante de acordo com o programa modelo e o padrão sequencial JPlay), a classificação de variáveis e a comparação das árvores de comportamentos dos pares de variáveis.

A abordagem de comparação ocorre entre o programa do estudante e um programa previamente definido como modelo. O estudante deve selecionar, em sua ferramenta de ambiente de desenvolvimento integrado (IDE), o programa modelo que ele quer usar como referência, previamente disponibilizado em um repositório. Assim, a comparação se baseia numa heurística de busca de comportamentos de correspondência entre pares de classes, e posteriormente, na correspondência de pares de variáveis entre esses programas em tempo de programação.

## 1.2 DEFINIÇÃO DO PROBLEMA

Em uma linguagem de programação destaca-se duas características principais: sintaxe e semântica. A sintaxe descreve a forma como as instruções de uma linguagem são escritas, mas sem atender ao seu significado. Na sintaxe de uma linguagem de programação existem descrições formais, que são chamadas de lexemas. Os lexemas de uma linguagem de programação incluem seus identificadores, palavras reservadas, literais e operadores. Um símbolo de uma linguagem é uma categoria de lexemas e é denominado *"token"*.

A semântica é complementar a sintaxe, ela descreve os significados das estruturas do programa expressos na sua sintaxe. Por exemplo, a sintaxe de uma instrução *"if"* da linguagem C++ é: *"if ( ) {}"* e sua semântica é: “se o valor da expressão for verdadeiro, as

instruções incorporadas serão executadas pelo programa”. É através da semântica que conseguimos utilizar melhor e validar uma linguagem. Ao contrário da sintaxe, não existe ainda um formalismo aceito globalmente para descrever a semântica da linguagem.

A semântica pode ser classificada em semântica estática e semântica dinâmica. A semântica estática descreve as características de um programa válido, e a semântica dinâmica descreve os resultados da execução do programa.

Ao contrário da análise sintática de um programa, que é específica da linguagem e pode ser obtida de forma exata segunda as regras de um compilador, a análise semântica do código-fonte é uma análise quase intuitiva, inexata e profundamente dependente do objetivo do programa. Por ter essas características, obter a análise semântica depende da interpretação do código-fonte e também do conhecimento prévio do problema que se quer resolver.

Erros semânticos dinâmicos em programas são mais difíceis de serem detectados por um programador, pois o programa é executado, apesar de não apresentar os resultados esperados. Estudantes pouco experientes, durante o processo de construção de um programa, têm dificuldade de entender a causa de erros semânticos dinâmicos em seus programas. Baseado neste problema, este trabalho propõe o desenvolvimento de uma heurística que, através da comparação entre dois programas, consiga sugerir possíveis erros semânticos de um programa e orientar o estudante durante o processo de construção de um programa.

A comparação ou similaridade entre programas pode ser utilizada com diferentes objetivos, um dos objetivos mais comuns, por exemplo, é a detecção de plágio. Neste trabalho, a comparação será utilizada com o objetivo de detectar possíveis erros semânticos. Segundo Koschke (2007), não existe um consenso sobre o significado exato de similaridade, nem de clone e redundância de código.

Walenstein et al. 2007 afirma que antes de classificar similaridade o desafio é responder o que é, como medir e como escolher um instrumento de medição da similaridade. Baseado nessas questões, os autores classificam dois tipos de similaridade.

A primeira classificação de similaridade trata da semelhança representacional (sintática), esta refere-se ao programa como sendo uma sequência de caracteres formando uma estrutura de texto complexa, nela a similaridade pode ser definida em termos de suas formas, propriedades ou características desta representação. Pode-se detectar a existência de similaridade em diferentes níveis de abstração (similaridades entre sentenças de programas, em blocos de programas, no nível de classes, no nível de unidades de programas etc).

A segunda classificação de similaridade trata da semelhança comportamental (semântica). Neste caso, os programas podem ser considerados similares quando a principal semelhança está em sua semântica, isto é, em seu comportamento, que pode ser definido pelas funções que estes programas implementam. Neste caso a dificuldade é definir medidas de similaridade semântica. Por exemplo, ao avaliar o contexto de clonagem, questiona-se bastante, se as semelhanças existentes em partes dos códigos (blocos) podem ser consideradas clones ou se é necessário que a semelhança esteja representada em blocos bem definidos (Mota, 2016).

Na semelhança comportamental (semântica), ainda é possível apontar dois tipos de representação que podem ser utilizados neste tipo de comparação:

- semelhança funcional: dois programas podem ser considerados análogos se implementarem uma função similar. Ou seja, a similaridade funcional pode ser descrita como sendo a semelhança entre as entradas relacionadas com suas saídas.
- semelhança de execução: será observada a sequência de execução das instruções do programa, por exemplo em *bytecode* Java ou código *Assembly*. Assim, será necessário encontrar uma correspondência entre as instruções dos programas executáveis.

Segundo Mota (2016), avaliando o comportamento, a semântica do programa pode ser de um dos tipos abaixo:

- denotacional, onde o efeito gerado pelo programa interessa mais que a forma como foi produzido;
- operacional, onde se detalha os passos de execução de um programa;
- lógica, onde se estabelece o significado de cada sentença, suas variáveis e sua lógica;

Neste trabalho, a comparação entre os dois programas (programa do estudante e programa modelo) procura por uma semelhança comportamental (semântica) entre eles. A heurística de comparação proposta define medidas de similaridade próprias para realizar esta comparação, as medidas de similaridade definidas se baseiam nos comportamentos das variáveis no decorrer do código-fonte dos programas. A heurística define que um comportamento de uma variável acontece toda vez que ela está envolvida em um comando de atribuição (comportamento "*assignment*"), um comando de laço (comportamento "*loop*") ou um comando condicional (comportamento "*conditional*").

Assim, a proposta dessa abordagem é obter uma análise semântica mais próxima possível do objetivo do programa sem comparar resultados de saída do programa, uma vez que o analisador deve fornecer correções em tempo de programação.

O processo é baseado em várias etapas: classificação dos pares de classes similares, a padronização (verifica a padronização do programa do estudante de acordo com o programa modelo e o padrão sequencial JPlay), a classificação de variáveis e a comparação das árvores de comportamentos dos pares de variáveis. Após a classificação dos pares de classes similares é possível verificar se o programa do aluno está corretamente padronizado de acordo com o programa modelo e de acordo com o padrão sequencial JPlay. A padronização do programa modelo se dá através da definição de marcadores, que são inseridos pelo próprio professor na forma de comentários no início de cada classe do programa. Uma sequência de código padrão do JPlay é uma sequência de código no programa baseado no JPlay que sempre deve acontecer quando o programa está correto. Uma padronização adequada do programa do aluno pode garantir resultados mais apurados durante a fase seguinte, que consiste na classificação de variáveis e conseqüente comparação das árvores de comportamentos destas variáveis.

Através da comparação entre as árvores de comportamentos dos pares de variáveis é possível verificar na classe do programa modelo os comportamentos existentes e procurar por eles na classe do programa do aluno. A obtenção dos comportamentos na classe do programa modelo funciona como a obtenção dos requisitos para a classe do aluno, assim é possível ter uma análise semântica mais específica ao objetivo daquela classe. Comparações que apontem diferenças mostrarão para o aluno mensagens informando sobre o(s) possível (eis) problema(s) encontrado(s). As mensagens são incluídas pelo próprio professor, no programa modelo, antes de cada comportamento, na forma de comentários.

## **1.3 OBJETIVOS**

### **1.3.1 OBJETIVOS GERAIS**

O objetivo principal da pesquisa consiste em propor uma heurística de comparação de comportamentos entre dois programas (programa do estudante e programa modelo) que utilizam o framework JPlay. Através desta comparação, a heurística possibilita a análise de um código JPlay construído pelo aluno e o orienta para o desenvolvimento de um jogo específico. A análise tem a função de interpretar semanticamente um programa Java

desenvolvido que usa o JPlay e retorna resultados desta análise para o estudante. O processo é baseado em classificar classes similares, verificar a padronização entre os pares de classes e, depois, classificar os pares de variáveis similares de cada par de classes para logo em seguida comparar as árvores de comportamentos dos pares de variáveis selecionados. Assim, a comparação consiste na busca de comportamentos de correspondência entre pares de variáveis resultantes dos pares de classes similares.

### **1.3.2 OBJETIVOS ESPECÍFICOS**

- Classificar os pares de classes similares;
- Verificar a padronização do programa do estudante de acordo com o programa modelo utilizado pelo professor;
- Identificar os padrões sequenciais de código JPlay que acontecem no programa do aluno.
- Identificar os pares de variáveis similares resultados dos pares de classes classificados anteriormente.
- Elaborar a comparação entre comportamentos de pares de variáveis similares.
- Retornar mensagens para o estudante, resultados de comparações entre árvores de comportamentos de variáveis que apontem diferenças, informando sobre o(s) possível (eis) problemas encontrados.

### **1.4 CONTRIBUIÇÕES ALCANÇADAS**

A principal contribuição do trabalho consiste numa heurística baseada na comparação de comportamentos entre programas (programa do aluno e programa modelo). A pesquisa contribui também no sentido de que propõe uma ferramenta capaz de interpretar o código semanticamente construído por programadores, retornando resultados, apontando problemas e sugerindo soluções.

Para trabalhos futuros torna-se necessário a construção de uma interface de tutoria capaz de interpretar os resultados obtidos e, então, a partir disso, retornar as respostas para o estudante.

### **1.5 ESTRUTURA DA TESE**

O documento está dividido em 7 (sete) capítulos. O capítulo 1 apresenta a introdução. Neste é apresentada a motivação para a realização do trabalho, problema, o objetivo geral e objetivos específicos.

No capítulo 2 são apresentados os trabalhos relacionados, a modelagem de código-fonte em código XML e técnicas de apoio ao ensino de programação.

No capítulo 3 é apresentado o framework JPlay e sua arquitetura dividida em três partes: interação entre jogo e jogador, personagens do jogo e saídas do jogo.

No capítulo 4 é apresentada a modelagem formal do analisador semântico de programação. Nele são apresentadas as técnicas envolvidas para realizar a leitura do código-fonte dos programas (*Abstract Syntax Tree*, AST), representar o código-fonte em código XML (javaML de Badros) e realizar pesquisas de informações no código XML e PROLOG gerados.

No capítulo 5 é apresentado o analisador semântico e a heurística baseada em comparação de comportamentos dos programas. O capítulo está dividido em subitens que descrevem a escolha do programa modelo que será comparado com o programa do estudante, a classificação dos pares de classes, a análise de padronização, a classificação dos pares de variáveis e a comparação de suas árvores de comportamentos.

No capítulo 6 são apresentados os resultados através de três estudos de caso. O primeiro é uma prova, realizada pelos alunos da disciplina de “Tópicos Especiais em Desenvolvimento de Software” do Curso Médio Técnico Integrado em Informática do Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI), o segundo é um jogo (“*BrickBreak*”), aplicado aos alunos durante o minicurso “Desenvolvimento de Jogo usando Java e framework JPlay”. Participaram do minicurso alunos dos cursos Médio Técnico Integrado em Informática, Técnico subsequente em Informática e Técnico em Análise e Desenvolvimento de Sistemas do IFPI. O terceiro estudo de caso é um jogo (“*Ping Pong*”), aplicado também aos alunos da disciplina de “Tópicos Especiais em Desenvolvimento de Software” do Curso Médio Técnico Integrado em Informática do Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI). O capítulo está dividido em subitens que descrevem a metodologia utilizada, o primeiro estudo de caso (prova) e as suas considerações gerais, o segundo estudo de caso (“*BrickBreak*”) e as suas considerações gerais e o terceiro estudo de caso (“*Ping Pong*”) e as suas considerações gerais.

No capítulo 7 são apresentadas as conclusões e apontam-se trabalhos futuros.

## CAPÍTULO 2 – TRABALHOS RELACIONADOS

Neste capítulo serão apresentados trabalhos relacionados referentes às abordagens para técnicas de apoio para o ensino de programação e ao uso de estratégias semelhantes para representação de código-fonte em linguagens descritivas.

### 2.1 INTRODUÇÃO

Inicialmente este capítulo trata sobre as técnicas relacionadas a aprendizagem de programação e suas classificações. Essas técnicas são desenvolvidas com o objetivo de melhorar e aumentar o engajamento dos alunos durante o processo de aprendizagem. Baseado nas classificações de trabalhos anteriores (DELGADO, 2005) (PINHEIRO et al., 2007), definiu-se uma classificação possível das técnicas relacionadas a aprendizagem de programação como: programação baseada em teste unitário, propostas de ambiente de programação, avaliador automático, análise de padrões de programação e sistemas de depuração automática (sistemas de tutoria inteligente e sistemas de diagnóstico de programas). Nas próximas sessões, serão apresentadas cada uma das técnicas citadas.

Em seguida, outra abordagem deste capítulo trata do uso de estratégias para a representação do código-fonte. Diferentes trabalhos apresentam a modelagem e representação de código-fonte usando XML. A representação do código-fonte no formato de texto é amplamente utilizada na codificação de algoritmos. No entanto, para um sistema baseado no conhecimento, a representação de programas precisa de uma representação mais abstrata e converter esta representação em meta-estruturas torna-se uma questão importante (SANTOS, 2009). Este trabalho apresenta várias fases de desenvolvimento, cada fase envolvendo técnicas e tecnologias específicas. Inicialmente, para obter as informações do código-fonte dos programas, foi desenvolvido um *parser* que transforma o código-fonte dos dois programas que estão sendo comparados para um código XML. Assim, cada arquivo de classe Java de um projeto, após ser interpretado, tem um arquivo respectivo XML, com o mesmo nome da classe e extensão XML. A transformação é necessária porque a representação em formato texto do código-fonte não oferece versatilidade quando se deseja obter informações do software a partir deste código (SANTOS, 2009) e consequentemente, uma representação de código-fonte em formato XML é capaz de evidenciar os elementos estruturais de um código-fonte e viabilizar a extração de informações sobre o software. Dada a gama de ferramentas existentes para a realização de consultas, manipulação e transformação de informações em formato XML, a

realização de análises sobre estas informações se torna uma tarefa mais simples do que o processamento do código-fonte original (SANTOS, 2009).

Assim, neste capítulo após a seção introdutória, na seção 2.2 são apresentadas algumas abordagens para técnicas de apoio para o ensino de programação, de acordo com a classificação definida, logo em seguida, a seção 2.3 apresenta a representação de código-fonte em representação XML. Por fim, na seção 2.4, são relatadas as considerações finais deste capítulo.

## **2.2 TÉCNICAS DE APOIO AO ENSINO DE PROGRAMAÇÃO**

Durante o processo de aprendizagem de programação diferentes técnicas podem ser utilizadas para os alunos para melhorar e aumentar o engajamento. Baseado nas classificações de trabalhos anteriores (DELGADO, 2005) (PINHEIRO et al., 2007), estabeleceu-se uma classificação possível das técnicas relacionadas a aprendizagem de programação como: programação baseada em teste unitário, propostas de ambiente de programação, avaliador automático, análise de padrões de programação e sistemas de depuração automática (sistemas de tutoria inteligente e sistemas de diagnóstico de programas). A seguir, serão detalhadas cada um dos itens da classificação.

### **2.2.1 DESENVOLVIMENTO BASEADO EM TESTES UNITÁRIOS**

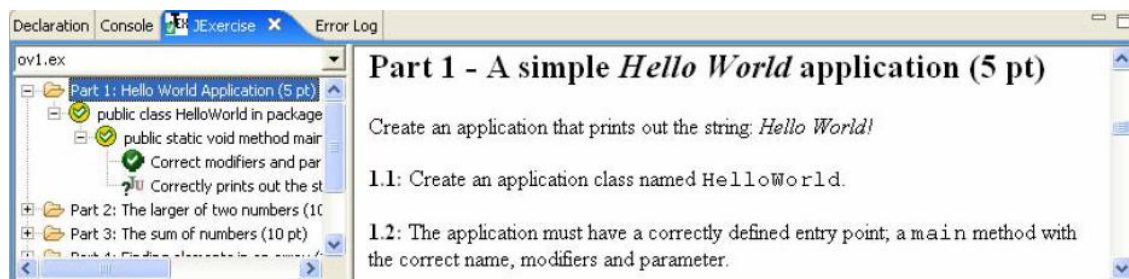
Na programação baseada em teste unitário o professor elabora um conjunto de testes específicos para resolver um problema particular e o estudante deve construir um programa que permite a obter os resultados esperados durante a execução de todos os testes (PINHEIRO et al., 2007) (TRAETTEBERG, 2006).

Chama-se de teste unitário um teste individual de unidades do programa. No caso de um programa orientado a objeto, as unidades de teste podem ser classes e métodos (BLUEJ, 2013).

Como exemplo de desenvolvimento baseado em testes unitários citamos a ferramenta JExercise. O JExercise (TRAETTEBERG, 2006) é um plugin do eclipse que é mostrado como uma *view* pertencente a janela principal da IDE. Na visão do estudante, um exercício é um conjunto de requisitos organizados hierarquicamente. À direita uma janela do navegador mostra o texto de exercício, à esquerda são apresentados os requisitos de programação necessários, quando um requisito é selecionado na árvore, o navegador indica o texto correspondente do lado direito. Um botão é fornecido abaixo da árvore para a execução de











todos os testes JUnit de uma vez. O botão também indica o número atual de pontos alcançados pelo estudante no exercício. A *view* então contém 2 (dois) elementos principais: do lado direito são mostrados os exercícios e as requisições associadas a cada exercício, e do lado esquerdo o texto explicativo referente a requisição selecionada, conforme mostra o exemplo da Figura 2.1.



**Figura 2.1: View do plugin JExercise**  
**Fonte: (TRAETTEBERG,2006)**

No JExercise um exercício é estruturado com um conjunto de partes que contém requisitos para elementos sintáticos específicos que podem ser testados usando o framework JUnit (JUNIT, 2013). O JUnit é um framework que permite pequenos testes organizados através da escrita de métodos. Ele fornece funcionalidades para facilmente executar um conjunto de testes, expressar afirmações sobre os resultados e ser notificado sobre falhas de caso de testes (BLUEJ, 2013).

Uma vantagem do JExercise é que nele os requisitos são continuamente testados, de forma automática quando uma notificação de atualização chega no Eclipse, enquanto que os testes de requisição usando somente o JUnit devem ser ativados manualmente (TRAETTEBERG,2006). Quando o teste é finalizado, o ícone da requisição é atualizado, indicando sucesso ou falha do processo, conforme mostra a Figura 2.2.

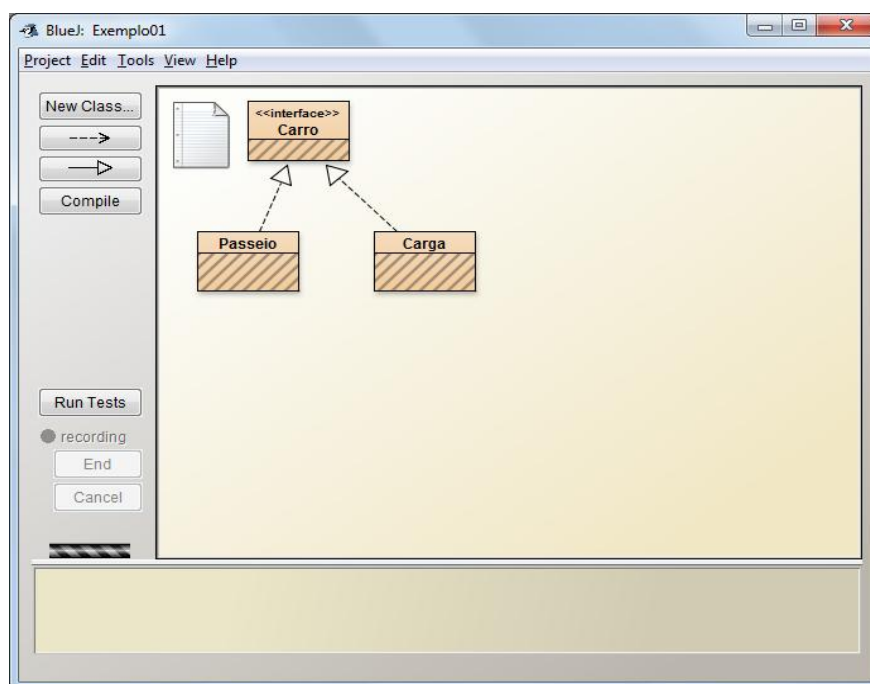
Icon	Meaning
	The requirement's state is currently <i>undecided</i> , presumable because a pre-condition is unsatisfied.
	The requirement is completely <i>satisfied</i> , as indicated by the green color.
	The requirement itself is <i>satisfied</i> , but there are <i>undecided</i> sub-requirements
	The requirement itself is <i>satisfied</i> , but some sub-requirements are <i>violated</i> , as indicated by the red color.
	The requirement is <i>violated</i>
	The requirement is a JUnit test that has not been run.
	The requirement is a JUnit test that has <i>succeeded</i>
	The requirement is a JUnit test that has <i>failed</i>

**Figura 2.2: Os ícones usados na view do JExercise**  
**Fonte: (TRAETTEBERG,2006)**

### 2.2.2 PROPOSTAS DE AMBIENTES DE PROGRAMAÇÃO

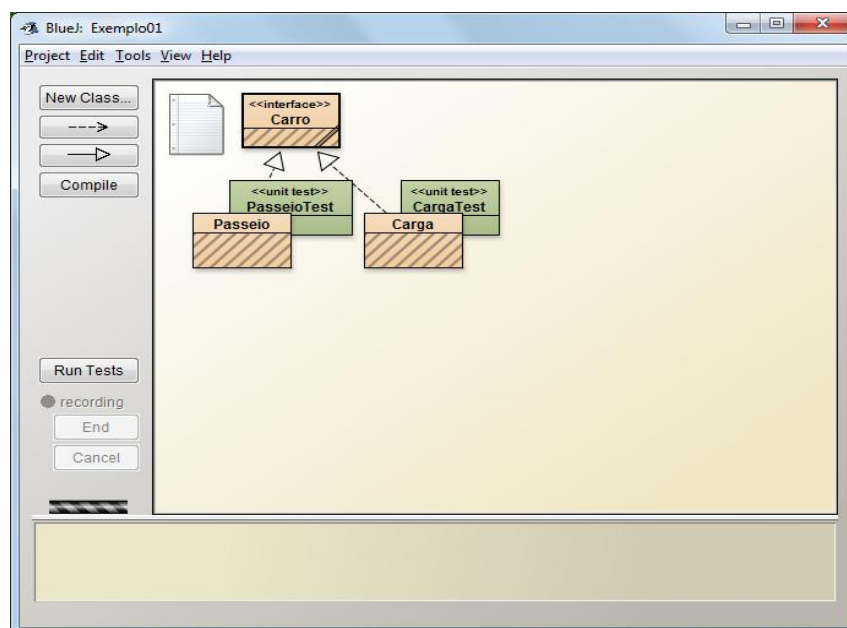
A proposta de ambientes de programação consiste no fato de que algumas ferramentas de desenvolvimento foram criadas com o objetivo de ajudar estudantes a aprender introdução a programação, tais como BlueJ e DrJava (KOLLING et al.,2003) (ALLEN et al.,2002).

O BlueJ é um ambiente integrado de desenvolvimento específico para o ensino introdutório de programação (KOLLING et al.,2003). O objetivo do BlueJ é ajudar em uma primeira abordagem de introdução a programação e facilitar a fixação do padrão de orientação a objeto. Quando um projeto Java é aberto no BlueJ, a janela principal mostra um diagrama de classes em linguagem de modelagem unificada (UML) e os usuários podem interagir diretamente com classes e objetos, conforme Figura 2.3.



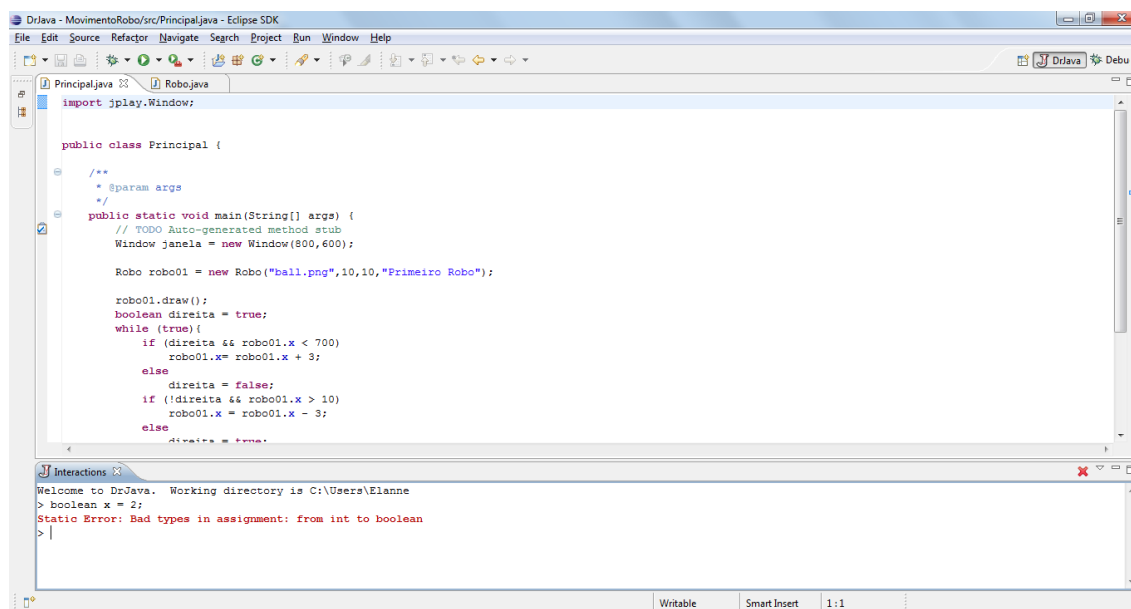
**Figura 2.3: Janela do ambiente de programação BlueJ**

Segundo Kolling et al. (2003) a principal vantagem do ambiente é a simplicidade, considerando que um dos grandes problemas de muitos ambientes de programação é a complexidade, já que a maioria dos ambientes foram projetados principalmente focados para programadores profissionais. O BlueJ também disponibiliza a funcionalidade para realização de testes unitários através de ferramentas para testes unitários sistemáticos. As ferramentas para testes unitários disponibilizadas pelo BlueJ são baseadas no JUnit (BLUEJ, 2013). Ver o exemplo na Figura 2.4.



**Figura 2.4: Criação de classes de testes no BlueJ**

O DrJava (ALLEN et al., 2002) é um plugin para o ambiente Eclipse que tem o objetivo de dar feedback para os alunos através de um painel de interações, conforme Figura 2.5.



**Figura 2.5: View de interações do plugin DrJava**

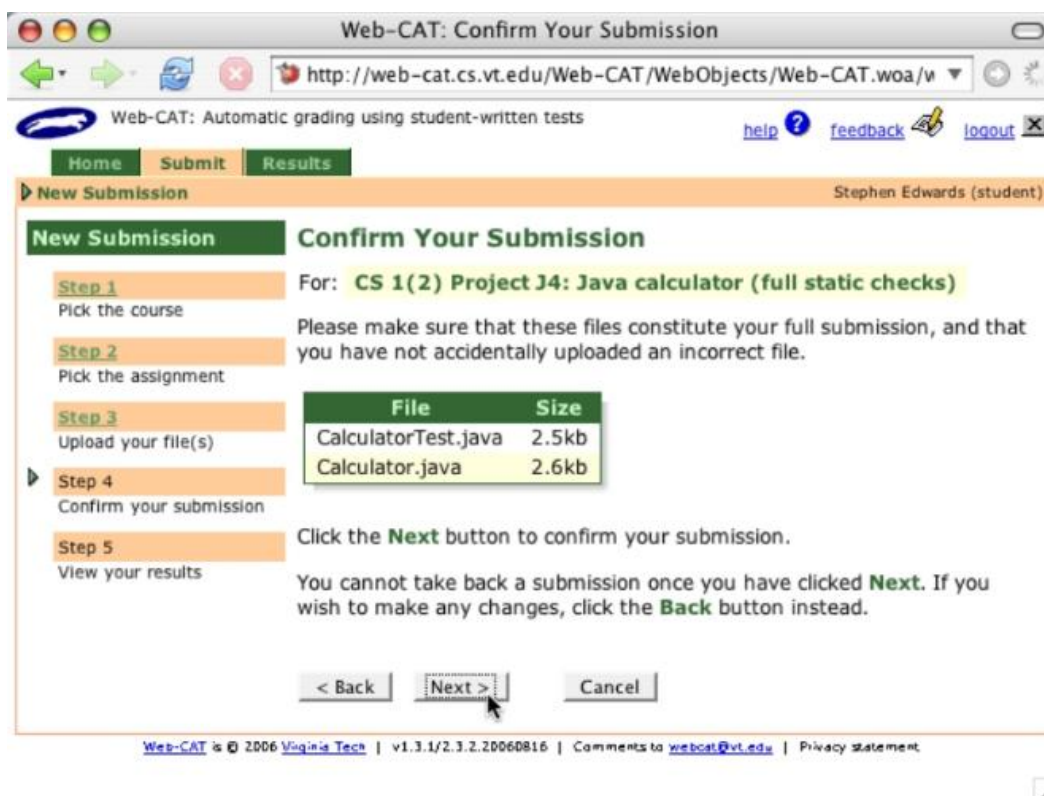
O painel de interações do DrJava permite avaliar blocos de declarações em código Java. Esta funcionalidade é útil para alunos iniciantes de programação, pois a partir da linha de comando do painel de interação é possível obter resultados de como uma classe ou método se comporta.

Um limitação do DrJava consiste no fato de que a interpretação dos códigos na *view* de interação só suporta códigos de bibliotecas nativas da linguagem Java, pacotes importados não são reconhecidos.

### 2.2.3 AVALIAÇÃO AUTOMÁTICA

O avaliador automático é usado para ajudar os professores com tarefas de correções de atividades. O professor pode definir testes para serem executados automaticamente depois que os estudantes enviarem suas atividades de programação e resultados dos testes podem ser usados para definir a nota do estudante (PINHEIRO et al., 2007).

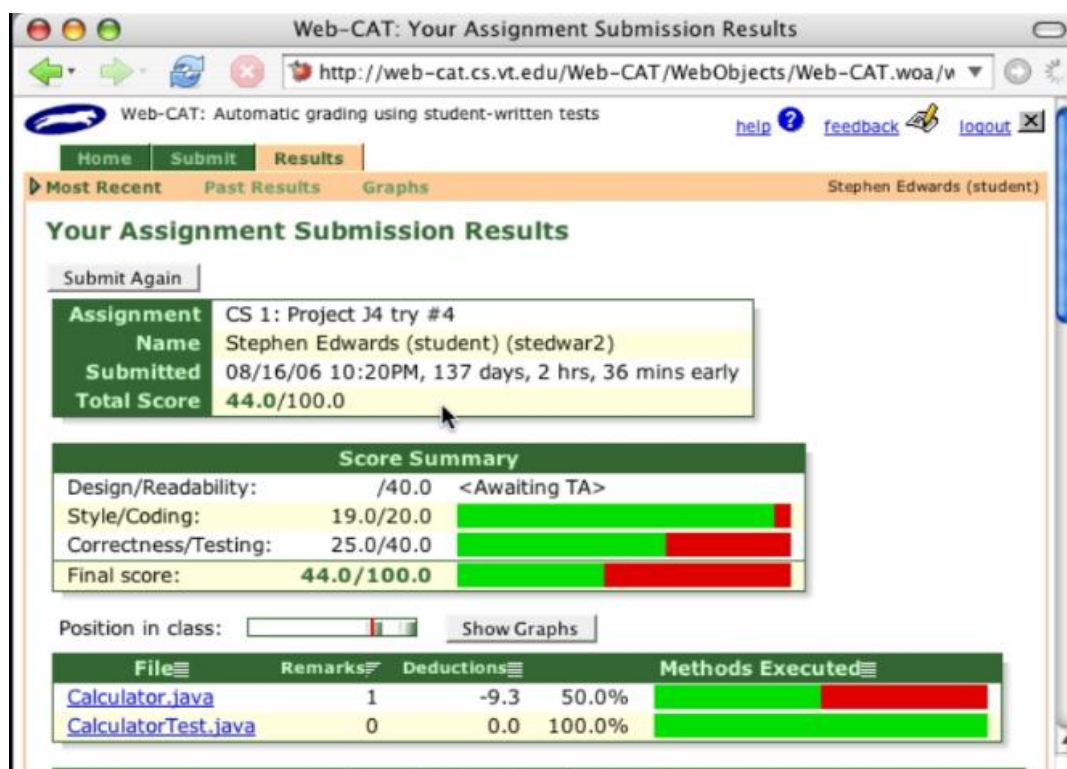
Pode-se citar o Web-Cat como uma ferramenta de avaliação automática. O Web-Cat (ALLOWATT, 2005) é uma aplicação web com uma arquitetura de *plugins* que pode fornecer uma variedade de serviços para o aluno. O Web-Cat possibilita ao aluno fazer um login na aplicação web e submeter sua tarefa para avaliação, conforme mostrado na Figura 2.6.



**Figura 2.6: Janela de submissão de projeto no Web-CAT**

Fonte: (WEB-CAT,2013)

O sistema realiza testes de aceitação que verificam se a atividade do aluno está em conformidade com os requisitos do problema e retorna informações sobre os testes que obtiveram resultados incorretos. Além disso, Web-Cat também avalia a integridade de testes do aluno, estilo e qualidade do código, dando uma indicação de como melhorar. Os resultados de cada um destes aspectos são apresentados tanto separadamente como também combinados em uma única pontuação para os alunos, conforme mostrado na Figura 2.7.



**Figura 2.7: Janela de resultados do Web-Cat**

Fonte: (WEB-CAT,2013)

O sistema mantém uma base de dados de todas as tarefas de alunos em um determinado curso. A submissão de tarefas pode ser realizada através de *plugins* para os ambientes BlueJ e Eclipse (WEB-CAT,2013).

É possível citar também o sistema Run.Codes (RUN.CODES,2016). O sistema cadastra professor, alunos e turmas de uma determinada disciplina. Os alunos podem submeter uma atividade através de *upload* no sistema, a atividade é analisada e posteriormente informações sobre a análise são disponibilizadas para professor e aluno. Ao cadastrar uma atividade o professor pode definir que o exercício é um código do tipo "compilável ou interpretável" e que tipo de arquivo será aceito como resposta (\*.cpp, \*.c, \*.pas, \*.java etc), uma grande quantidade de código compilável e interpretável é reconhecido pelo Run.Codes, assim, quando o aluno submete sua atividade, o sistema verifica se aquele código foi compilado corretamente ou se existe(m) erro(s) de compilação. Uma limitação consiste no fato de que as análises só suportam bibliotecas nativas (pacotes importados Java, por exemplo, não são reconhecidos).

O Run.Codes apresenta uma opção para inclusão de casos de testes, assim o professor pode cadastrar quantos casos de testes quiser, aonde ele determina valor(es) de entrada e

valor(es) de saída correspondente com o objetivo de atestar se o programa do aluno está ou não correto. A Figura 2.8 mostra a janela de edição de um exercício no Run.Codes.

**[run.codes]** Menu Professor elannecristina.santos@ifpi.edu.br Hora do Servidor: 04/01/2016 19:11:18

Home > MIG.3708 > Calculo do valor de uma base x elevada a um expoente y (usando recursão)

**Calculo do valor de uma base x elevada a um expoente y (usando recursão)**

Disciplina: MIG.3708 - Estrutura de Dados  
Data de Abertura: 23/12/2015 15:17:33  
Prazo de Entrega: 24/12/2015 16:00:00 **Fechado**

Escreva uma função recursiva para calcular o valor de uma base x elevada a um expoente y.

OBS: Faça usando recursão

**Esconder Descrição**

Este exercício aceita os seguintes tipos de arquivos: C++ C

**Casos de Teste** Baixar Casos de Teste Remover Todos os Casos de Teste

Caso	Mostrar Entrada	Mostrar Saída Esperada	Mostrar Saída do Aluno	Ações
Caso 1	Sim	Sim	Sim	Editar Caso Deletar

**Novo Envio**

Você pode submeter um arquivo até **24/12/2015 16:00:00**

Selecionar Arquivo

**Figura 2.8: Janela de edição de um exercício no Run.Codes**

## 2.2.4 SISTEMAS DE TUTORIA USANDO PADRÕES DE PROGRAMAÇÃO

A análise de padrões de programação é baseada na pesquisa de sugestões de aprendizagem anteriores, devido a crença de que os programadores experientes procuram resolver problemas baseados em soluções anteriores, relacionadas com o problema novo, que podem ser adaptadas para a situação ideal (DELGADO, 2005). Assim, o conceito de padrões é baseado no fato de que programadores experientes são capazes de resolver novos problemas através da análise de problemas resolvidos anteriormente. Eles podem identificar qual estrutura usar, quais tipos de dados estão envolvidos, bem como outras formas para resolver o mesmo problema, através de experiências anteriores que identificam soluções (ALEXIS, 2013).

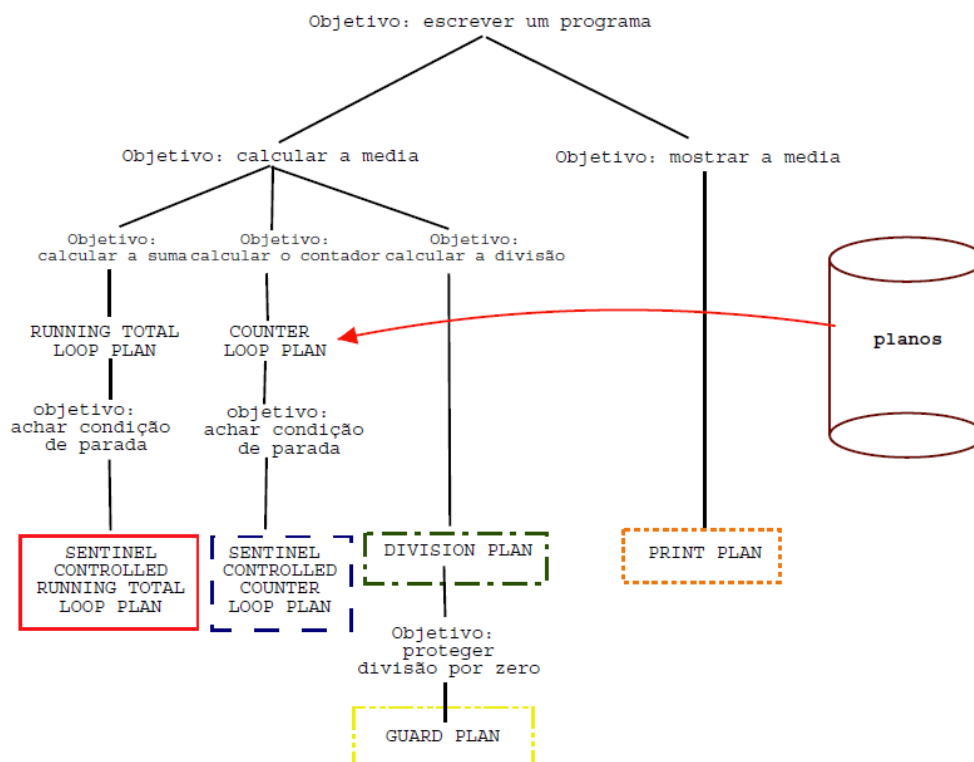
Experiências anteriores são a base para padrões de programação, elas são soluções que aparecem muitas vezes na resolução de problemas computacionais (ALEXIS, 2013). Assim, os padrões traduzem estratégias de programação criadas por especialistas que podem levar a boas práticas de programação. Podemos citar os sistemas Proust (JOHNSON, 1984) e PROPAT (DELGADO, 2005) como sistemas que utilizam a estratégia de análise de padrões de programação.

O conceito de padrões elementares define uma base inicial de micro soluções que podem ser adaptadas e combinadas pelos alunos durante a elaboração de seus programas,



assim os padrões elementares contém o esqueleto no qual o problema será resolvido pelo estudante (PORTER, 2004). Um padrão elementar relata uma solução para um problema e fornece informações sobre o contexto em que ele deve ser aplicado (ALEXIS, 2013).

O sistema Proust, proposto por (JOHNSON, 1984), aplicado ao ensino da linguagem de programação Pascal, foi baseado na resolução de novos problemas utilizando experiências prévias, adaptando as soluções para resolvê-las através de experiências anteriores de estudantes. O tutor Proust analisa o programa do estudante a partir de uma descrição declarativa do problema, ou seja, a partir de uma lista dos requisitos que devem ser satisfeitos, sendo estes especificados previamente pelo professor. Através dos requisitos são definidos um conjunto de objetivos para resolver um determinado problema e os objetivos podem ser expressos no sistema através de uma biblioteca de planos. Proust decompõe objetivos de problemas de forma hierárquica através de planos. Os objetivos podem ser decompostos em sub-objetivos ou em outros planos (pedaços de programa). Como programas podem ser resolvidos de maneiras diferentes, essa decomposição pode não ser única e os programas implementados pelo aluno podem ser associados a mais de uma decomposição. Assim, programas com erros podem ser derivados de uma decomposição de objetivos incorreta ou de implementações incorretas de decomposições de objetivos corretas. A Figura 2.9 ilustra um exemplo de decomposição de planos.



**Figura 2.9: Exemplo de decomposição de objetivos no sistema Proust**  
 Fonte: (DELGADO, 2005)



Segundo Delgado (2005), a proposta do sistema Proust de decompor um programa em planos tem correlação com os padrões de programação elementares, com a diferença de que não existia a preocupação da descrição didática dos planos ao aluno.

Sobre as limitações da abordagem utilizada pelo sistema Proust, podemos citar que para a obtenção de um bom diagnóstico do programa, é necessária uma ampla biblioteca de planos, uma vez que devem haver planos correspondentes a várias maneiras que os estudantes resolvem problemas.

Outra abordagem de sistema usando padrões é o ProPAT<sup>1</sup>, um sub-projeto IBM Eclipse em parceria com o Instituto de Matemática e Estatística da Universidade de São Paulo para a construção de um Ambiente Integrado de Desenvolvimento (IDE) para cursos de introdução à programação (DELGADO, 2005). A ideia básica do ProPAT é a disponibilização de um *plugin* Eclipse para diagnóstico automático de programas, que inclui um ambiente para ensino/aprendizado de programa utilizando padrões elementares (WALLINGFORD, 2013).

As limitações do ProPAT dizem respeito a representação dos padrões na solução de um determinado problema, tais como: a consulta dos padrões pelos alunos é dependente de um determinado domínio do problema, ou seja, para alguns problemas, é possível não haver um conjunto completo de padrões que os solucione; para que o diagnóstico seja correto é necessário que exista uma boa biblioteca de padrões especificados por educadores experientes de programação, portanto os educadores devem estar habilitados a cadastrar vários padrões possíveis para resolução de um determinado problema, considerando que eles devem corresponder às diferentes maneiras que estudantes usam para construir seus programas.

Uma limitação do uso de padrões de programação para promover o aprendizado do aluno se refere ao fato de que em algumas abordagens o aluno deve escolher o padrão a ser utilizado num dado trecho do programa, como é o caso do sistema PROPAT. A dificuldade do estudante em reconhecer qual o plano mais adequado para utilizar num dado momento pode dificultar a construção do programa. Muller (2005) afirma que os padrões de programação se referem à classificação de problemas computacionais de acordo com seu objetivo. Assim, o aluno pode, através de uma classificação, identificar qual padrão pode ser utilizado para a elaboração do algoritmo que soluciona o problema a partir do problema a ser selecionado.

---

<sup>1</sup> <http://www.ime.usp.br/~articuno/eclipse/>

Uma das dificuldades citadas por Muller (2005) na abordagem do uso de padrões no ensino de programação é capacidade do estudante de reconhecer similaridades entre problemas, particularmente entre o problema em questão e problemas encontrados e resolvidos previamente.

Daí surge a necessidade de categorizar, catalogar e descrever os padrões de forma que, quando o estudante se deparar com um determinado problema, possa inferir qual padrão ou qual combinação de padrões de programação pode ser utilizado, bem como definir algumas categorias básicas que podem ser citadas de acordo com a classificação dos tipos de padrões, tais como padrões de seleção e padrões de repetição (MULLER, 2005).

### **2.2.5 SISTEMA DE DEPURAÇÃO AUTOMÁTICA**

Um sistema de depuração automática é um sistema que utiliza técnicas com o objetivo de encontrar e classificar os componentes de um programa. Com base no tipo de técnica utilizada, ele pode ser classificado em sistemas de tutores inteligentes de programação e sistema de diagnóstico de programa (DELGADO, 2005).

Neste trabalho foi considerado um sistema de tutores inteligentes de programação um sistema que usa técnicas específicas de Inteligência Artificial (AI) para ajudar na aprendizagem do estudante. Um sistema de diagnóstico de programas pode usar várias outras técnicas com o objetivo de entender o código-fonte de um programa e pode ser usado com objetivos específicos de aprendizagem ou não.

LAURA é uma das primeiras tentativas de construir um sistema de tutoria para o ensino de programação. A ferramenta foi desenvolvida na Universidade de Caen, na França, e é um sistema de diagnóstico de programa escrito em Fortran (BOTELHO, 2010). Em sua estratégia LAURA se utiliza de uma solução de referência (um programa) para diagnosticar o programa do aluno (SANTOS, 2003), ou seja, o sistema realiza a comparação entre dois programas, o modelo e o candidato. A comparação é possível através da representação do programa modelo e do programa candidato através de grafos, e a sua estratégia heurística é identificar passo a passo os elementos semelhantes dos grafos (ADAM, 1980). Santos (2003) também afirma que LAURA foi o único em seu ramo de atuação a utilizar esta estratégia.

Para representar um programa FORTRAN em um grafo, o LAURA inicialmente realiza um processo de padronização dos grafos do programa modelo e do programa do aluno de forma que eles se tornem o mais semelhantes possível (ADAM, 1980). Depois de realizada a padronização dos grafos, o sistema passa a compará-los. Para comparar os grafos, ele

procura relacionar as variáveis, os nós e os arcos de cada um dos grafos, de forma que cada um dos componentes do grafo modelo se relacionem com um componente do grafo do aluno.

Assim, numa primeira etapa, o objetivo do sistema LAURA é identificar os dois grafos inteiramente, a fim de concluir que o programa do estudante está correto. Se essa total identificação não é possível, o sistema se esforça para chegar a outro objetivo: expressar o diagnóstico de erros (ADAM, 1980).

O sistema LAURA foi testado com programas escritos por estudantes com o objetivo de resolver oito problemas nos campos de (ADAM, 1980): gestão (cálculo de impostos, faturas de companhias elétricas); aritmética (números perfeitos, o triângulo de Pascal); métodos de análise numérica (integração, raízes da equação); ordenação de um vetor A qualquer com N números (utilizando um algoritmo de busca para máximo e permutação).

Também classificado como uma ferramenta usada para diagnóstico automático de programas, podemos citar o sistema desenvolvido neste trabalho, o sistema para análise semântica de jogos aplicada a programação na educação. O sistema é capaz de realizar diagnóstico com objetivos educacionais, tendo a função de interpretar semanticamente e arquiteturalmente um programa Java desenvolvido usando o JPlay e retornar resultados desta análise para o aluno. O processo consiste na análise comportamentos do programa. O comportamento do programa pode ser obtido pelo analisador através da comparação com outro programa que será definido como modelo. Para isto, o aluno deve selecionar, em sua ferramenta de ambiente de desenvolvimento integrado (IDE), o programa modelo que ele quer usar como referência, sendo que o programa modelo deve ser disponibilizado em um repositório. Resumidamente, o analisador é capaz de interpretar semanticamente o programa construído pelo aluno, apontar problemas e sugerir possíveis soluções (SANTOS et al., 2013).

## 2.3 REPRESENTAÇÃO DE CÓDIGO FONTE EM XML

Diferentes trabalhos apresentam a modelagem e representação de código-fonte usando XML para diversos fins. A representação do código-fonte no formato de texto é amplamente utilizada na codificação de algoritmos. No entanto, para um sistema baseado no conhecimento, a representação de programas precisa de uma representação mais abstrata e converter esta representação em meta-estruturas torna-se uma questão importante (SANTOS, 2009).

O CodeMI (SANTOS, 2009) propõe um código-fonte representado no formato XML. O objetivo do CodeMI é realizar estudos sobre a evolução de software comercial, através do

uso de um repositório de dados, onde a representação do código-fonte tem um papel central. Em sua representação XML do código-fonte, ele preserva as características necessárias para realizar os estudos da evolução do software, mas ao mesmo tempo esconde os detalhes da sua implementação. O objetivo do CodeMI ao esconder os detalhes da implementação do código-fonte é assegurar a proteção de autoria do programa.

Outra representação do código para o formato XML é o JavaML (MAMAS, 2000). Neste trabalho são gerados arquivos XML, um para cada classe do sistema, armazenando informações de elementos chave como classes, seus métodos e atributos, porém não armazenando informações sobre a estratégia de implementação dos métodos.

Badros et al. (2000) propõe o JavaML de Badros (ou JavaML 1.0). Nesta proposta também é gerado um arquivo XML para cada classe do sistema. Em comparação com a representação de Mamas e Kontogiannis, neste trabalho é possível obter informações sobre a implementação dos métodos, existindo um identificador (*id*) para cada método e variável, com a referência para o identificador (*idref*) para cada relacionamento que usa esta entidade.

JavaML 2.0 de Ademar Aguiar (AGUIAR et al., 2004) é baseado no JavaML de Badros e apresenta melhorias que se baseiam principalmente na preservação do código-fonte, o que aumentou consideravelmente o número de tags, tornando a representação mais robusta.

Considerando que a abordagem de Badros (BADROS, 2000) permite a preservação do código-fonte e que no presente trabalho torna-se necessário uma subsequente análise semântica, inicialmente optamos por desenvolver um *parser* que converte todas as classes do programa em desenvolvimento em uma representação XML, tomando para isso como base a representação JavaML de Badros. Devido o aumento da representação de tags no JavaML 2.0, foi ignorada esta atualização para a presente proposta. A abordagem CodeMI não atende os objetivos deste trabalho, porque sua representação não preserva totalmente as estruturas do código-fonte, e conseqüentemente não é capaz de atender as necessidades da análise.

## 2.4 CONSIDERAÇÕES FINAIS

Entre os vários sistemas estudados, verificou-se o LAURA como o único sistema que utiliza uma solução de referência, ou seja, compara uma solução usada como referência com o programa do estudante. Segundo Santos (2003), uma das grandes vantagens abordadas por esta estratégia é a capacidade de aceitar programas, mesmo que eles não sejam idênticos ao programa de referência. Santos (2003) afirma que esta capacidade é importante, uma vez que permite que o aluno exerça em parte sua criatividade e não seja coagido a seguir um caminho

específico. Por outro lado, a grande desvantagem é a capacidade de resposta do sistema. LAURA trabalha de forma genérica e só é capaz de detectar erros de baixo nível de abstração, ou seja, erros pequenos que não levam em conta o contexto principal da solução do problema abordado.

Os sistemas Proust e ProPAT são baseados em padrão de programação. O sistema Proust se baseia na descrição declarativa prévia de um problema pelo professor, de forma que os objetivos para resolver um determinado problema podem ser decompostos em planos. Os planos são trechos de código de programa. Esta estratégia apresenta o problema de induzir o aluno a seguir uma sequencia de passos, interferindo na capacidade do aluno de resolver o problema de forma independente. O ProPAT, de forma semelhante, define padrões elementares recomendados para a solução de um dado problema, portanto apresentando também a característica de induzir o aluno a seguir determinados passos.

Na Tabela 2.1 apresentamos um comparativo entre os sistemas abordados neste capítulo. No comparativo podemos observar que o sistema LAURA é o que mais apresenta técnicas semelhantes ao analisador semântico proposto neste trabalho. Adam (1980) afirma que para diferenças superficiais (tais como nomes de variáveis, expressões aritméticas etc), a maioria dos resultados geram equivalência pelo sistema LAURA, mas em casos de diferenças mais profundas entre programas (estruturais, algoritmos, etc) é possível que o sistema LAURA não identifique o grafo do programa do estudante como um todo com o grafo do programa modelo. Nestes casos, o sistema somente marca que encontrou uma diferença entre os programas, envia uma aviso para o estudante sobre a localização da diferença, mas não consegue interpretá-la.

A estratégia do sistema LAURA se concentra em transformar cada um dos programas comparados em um grafo padronizado, ou seja, os grafos dos programas são transformados de maneira que fiquem o mais semelhante possível, e são logo em seguida comparados. Ou seja, cada programa é representado completamente por um único grafo padronizado. No caso do analisador semântico proposto neste trabalho, as estruturas de dados utilizadas para posterior comparação serão criadas baseadas em pares previamente classificados, ou seja, inicialmente, são verificados pares de classes similares, logo em seguida, são verificados os pares de variáveis similares no par de classes selecionadas. Desta forma, serão comparadas somente a estrutura de comportamento de uma variável do programa modelo com a estrutura de comportamento de uma variável similar do programa do estudante e não a estrutura de programa como um todo, como é feito no LAURA. Acreditamos que o algoritmo de

classificação de pares similares é capaz de tornar a análise de comparação das estruturas de comportamento mais simples e mais eficiente do que se compararmos uma estrutura completa do programa modelo com uma estrutura completa do programa do estudante.

**Tabela 2.1: COMPARATIVO ENTRE OS SISTEMAS QUE USAM TÉCNICAS DE APOIO AO ENSINO DE PROGRAMAÇÃO**

SISTEMA	CLASSIFICAÇÃO	TÉCNICA UTILIZADA	LINGUAGEM APLICADA	REPRESENTAÇÃO DO CÓDIGO-FONTE
LAURA (ADAM, 1980)	Diagnóstico automático de programas	<ul style="list-style-type: none"> <li>Solução de referência usando programa modelo.</li> </ul>	FORTTRAN	Representa o código-fonte através de um grafo de fluxo de controle
Analizador Semântico (SANTOS et al., 2013)	Diagnóstico automático de programas	<ul style="list-style-type: none"> <li>Solução de referência usando programa modelo;</li> <li>Baseado em Padrão Sequencial de jogos;</li> <li>Plugin Eclipse.</li> </ul>	JAVA	Representa o código-fonte através de uma estrutura XML
JExercise (TRAETTEBER G, 2006)	Testes unitários	<ul style="list-style-type: none"> <li>Submissão de testes previamente elaborados.</li> </ul>	JAVA	-
WEB-Cat (ALLOWATT, 2005)	Avaliação automática	<ul style="list-style-type: none"> <li>Submissão de testes previamente elaborados;</li> <li>Retorno da nota.</li> </ul>	JAVA	-

SISTEMA	CLASSIFICAÇÃO	TÉCNICA UTILIZADA	LINGUAGEM APLICADA	REPRESENTAÇÃO DO CÓDIGO-FONTE
Run.Codes (RUN.CODES, 2016)	Avaliação automática	<ul style="list-style-type: none"> <li>• Submissão de testes previamente elaborados;</li> <li>• Retorno da nota do aluno.</li> </ul>	PYTHON C/C++ FORTRAN JAVA PASCAL	
Proust (JOHNSON, 1984)	Sistema de tutoria	<ul style="list-style-type: none"> <li>• Baseado em padrão de programação, uso de planos (trechos de código de programa).</li> </ul>	PASCAL	-
ProPAT (DELGADO, 2005)	Sistema de tutoria	<ul style="list-style-type: none"> <li>• Plugin Eclipse CDT<sup>2</sup>;</li> <li>• Baseado em padrões de programação, uso de padrões elementares (padrão que define trecho de código de programa para solucionar determinado problema).</li> </ul>	C	-

Nenhuma dessas representações foi especificamente desenvolvida para analisar semanticamente um código-fonte representado em XML e gerar resultados desta análise. Nosso trabalho se diferencia dos outros propondo uma representação XML que possibilita a análise semântica de um código-fonte, aplicando uma heurística baseada na comparação de comportamentos entre programas (programa do aluno e programa modelo). Outra diferença relevante é que a nossa proposta é baseada em um padrão de design orientado para um jogo

---

<sup>2</sup> plataforma Eclipse para linguagem C e C++

2D simples, de acordo com o propósito original de JPlay (FEIJÓ et al., 2010) (JPLAY,2012), concentrando-se na análise de comportamentos entre os pares de classes dos programas.



## CAPÍTULO 3 - JPLAY

A proposta deste trabalho é baseada em um padrão de design orientado para desenvolvimento de jogos 2D, de acordo com o propósito original de JPlay (FEIJÓ et al., 2010) (JPLAY,2012).

O JPlay foi desenvolvido com o propósito de ensinar ciência da computação e algoritmos baseado no desenvolvimento de jogos. Este trabalho é baseado no framework JPlay e propõe a definição de um padrão sequencial de código baseado nesse framework. Um padrão sequencial de código do JPlay é uma sequência de código no programa que deve sempre acontecer quando o programa está correto.

Com o objetivo de identificar padrões sequenciais na arquitetura JPlay, foi realizado o mapeamento da arquitetura JPlay, sendo este mapeamento dividido em: interação entre jogo e jogador, personagens do jogo e saída do jogo.

Neste capítulo é apresentado na seção 3.1 a introdução sobre a arquitetura JPlay, em seguida, são apresentadas as suas sub-divisões (interação entre jogo e jogador, personagens do jogo e saída do jogo) e por fim, na seção 3.5, são relatadas as considerações finais deste capítulo.

### 3.1 INTRODUÇÃO

Muitos estudos confirmam a capacidade dos jogos influenciarem na motivação dos alunos. Segundo Nickel (2010), os estudantes podem ser mais motivados através de jogos e os jogos podem promover processos de pensamento criativo tais como habilidades gerais para resolver problemas, resultando em alunos automotivados fora da sala de aula. Nickel (2010) também afirma que a interação social e colaboração com o uso de jogos podem apoiar a aprendizagem.

Um motor de jogo (*game engine* ou apenas *engine*) é um programa de computador e/ou conjunto de bibliotecas para simplificar e abstrair o desenvolvimento de jogos. Algumas funcionalidades como motor gráfico para renderizar gráficos 2D e/ou 3D, um motor de física para simular a física, uma linguagem de script, suporte a animação, sons, inteligência artificial e outros podem ser fornecidas num motor de jogos. Muitos motores de jogos estão disponíveis no mercado, sendo a sua escolha dependente de diversos fatores a serem considerados tais como: orçamento disponível, tipo de jogo a ser desenvolvido, o tempo que

se possui para produzir o jogo, plataforma para a qual o jogo está sendo desenvolvida, documentação oferecida e ferramentas disponíveis (FEIJÓ et al., 2010).

Em Feijó et al. (2010) é apresentado o ensino de programação através do uso de um motor de jogo 2D chamado de javaPlay. O javaPlay apresenta-se conciso, didático, mas com uma estrutura de padrão de projeto para jogos. Segundo o autor, um motor de game (*Game Engine*) tem a responsabilidade de lidar com o hardware gráfico, irá mandar os modelos para serem visualizados, cuidará da entrada de dados do jogador e outras tarefas que o desenvolvedor de jogos normalmente não deseja fazer ou não tem tempo para se preocupar.

O JPlay (JPLAY, 2012) não pode ser considerado um *engine*, porém é um framework baseado no JavaPlay. Isto é assim porque o JPlay apenas oferece algumas interfaces para o desenvolvimento, cabendo no entanto ao programador a implementação do padrão de projeto. Embora o poder de desenvolvimento seja menor quando comparado com o JavaPlay, o JPlay permite que o ensino de programação seja mais desacoplado com relação ao ensino do funcionamento de jogos.

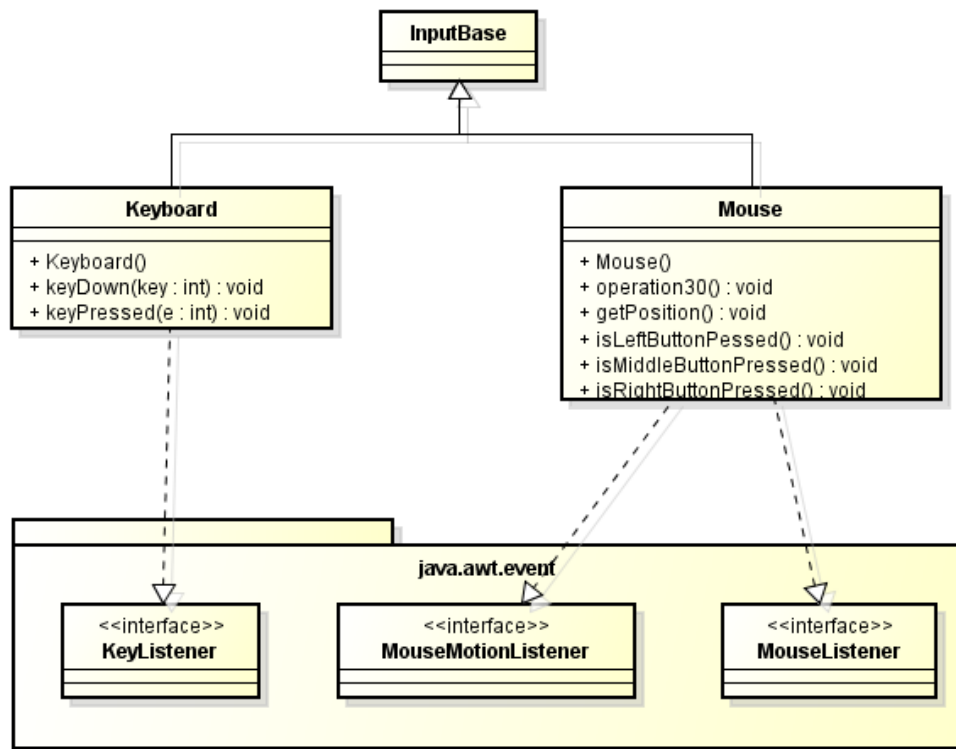
Neste sentido, o framework JPlay pode ser aplicado ao ensino de programação. O JPlay foi desenvolvido com o objetivo de facilitar o ensino de programação, proporcionando o ensino de lógica algorítmica através do desenvolvimento de jogos. Ele procura não interferir na estrutura de programação básica necessária para um correto aprendizado da lógica algorítmica e assim não introduz no código-fonte características específicas de padrões de projeto de jogos. A ferramenta possibilita ao aluno um modo fácil de desenhar e movimentar imagens na tela do computador e dispõe de métodos e objetos auxiliares que ajudam a criar jogos 2D usando a linguagem Java (JPLAY, 2012).

Neste trabalho, com o objetivo de identificar padrões sequenciais na arquitetura JPlay, inicialmente foi realizado o mapeamento da arquitetura JPlay. Este mapeamento foi dividido em três partes: interação entre jogo e jogador, personagens do jogo e saída do jogo. Seguem abaixo o detalhamento de cada uma das subdivisões da arquitetura.

### 3.2 ARQUITETURA JPLAY: INTERAÇÃO ENTRE JOGO E JOGADOR

Conforme Figura 3.1, as classes responsáveis por interação entre jogo e jogador são:

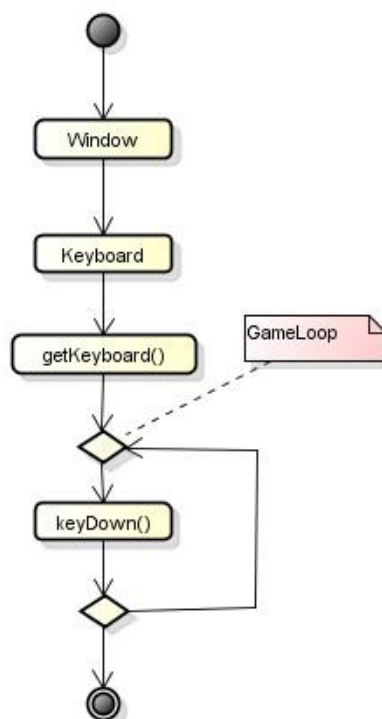
- *Keyboard*: Define entrada de dados pelo teclado do computador;
- *Mouse*: Define entrada de dados pelo mouse do computador;



**Figura 3.1: Diagrama de classes JPlay para interação jogador e jogo**

A partir do diagrama da Figura 3.1, podemos definir os seguintes padrões sequenciais:

- Para o uso da classe *Keyboard*: Ao criar um objeto do tipo *Keyboard*, se a tecla for padrão do JPlay, o programador deve verificar se a tecla foi pressionada através do método *KeyPressed*, senão o programador inicialmente deve indicar qual a tecla utilizada através do método *Keydown* e depois verificar se a tecla foi pressionada, conforme mostra Figura 3.2.



**Figura 3.2: Diagrama padrão de sequência para objetos do tipo da classe *Keyboard***

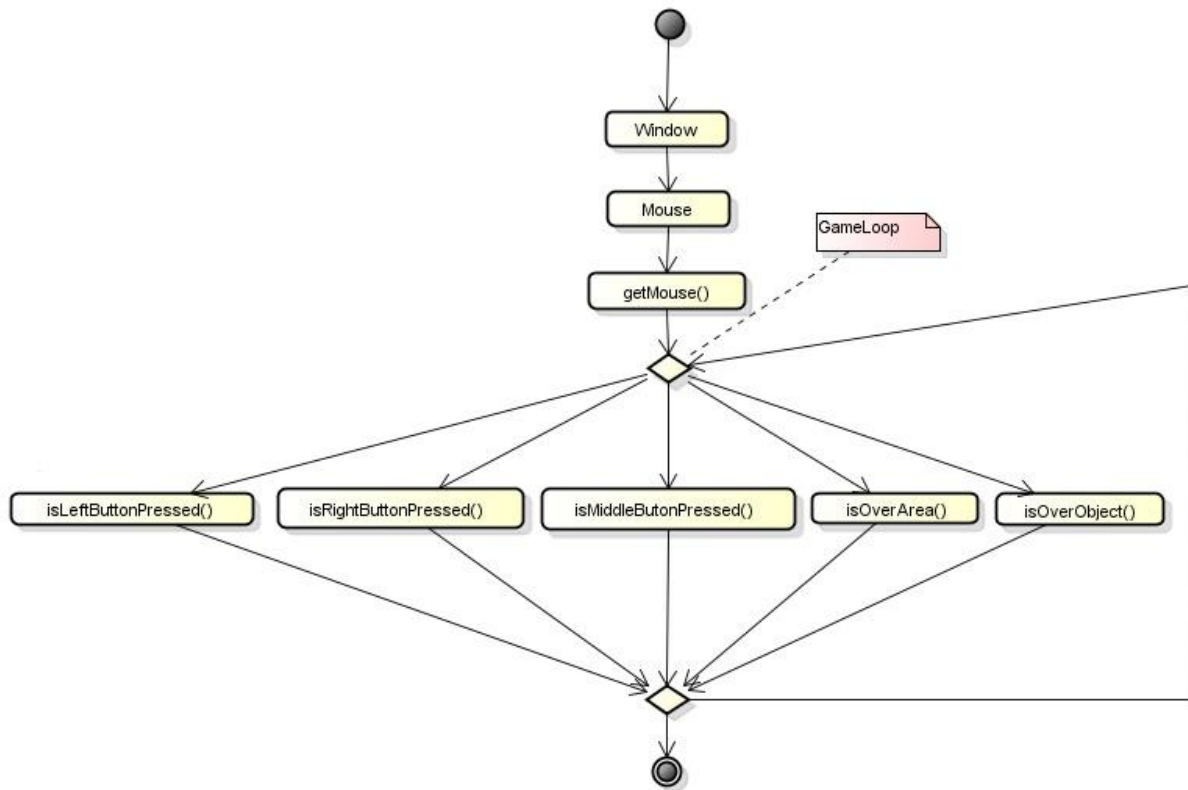
- Para o uso da classe *Mouse*: Ao criar um objeto do tipo *Mouse*, o programador deve verificar qual a posição atual do mouse através do método *getPosition* e depois, verificar qual botão do mouse foi pressionado através dos métodos *isLeftButtonPressed*, *isRightButtonPressed*, *isMiddleButtonPressed*, conforme mostra Figura 3.3.

### 3.3 ARQUITETURA JPLAY: PERSONAGENS DO JOGO

Conforme Figura 3.4, as classes responsáveis pela criação dos personagens do jogo são:

- *Animation*: Define uma animação e deve conter uma tira de imagens, que representam os respectivos frames. Cada frame é um trecho da imagem, sendo toda a sequência responsável pelo movimento de animação.
- *Sprite*: A classe *Sprite* estende a classe *Animation*. A classe *Sprite* possui métodos que podem fazer a imagem se mover pela tela mediante a alternância dos frames.

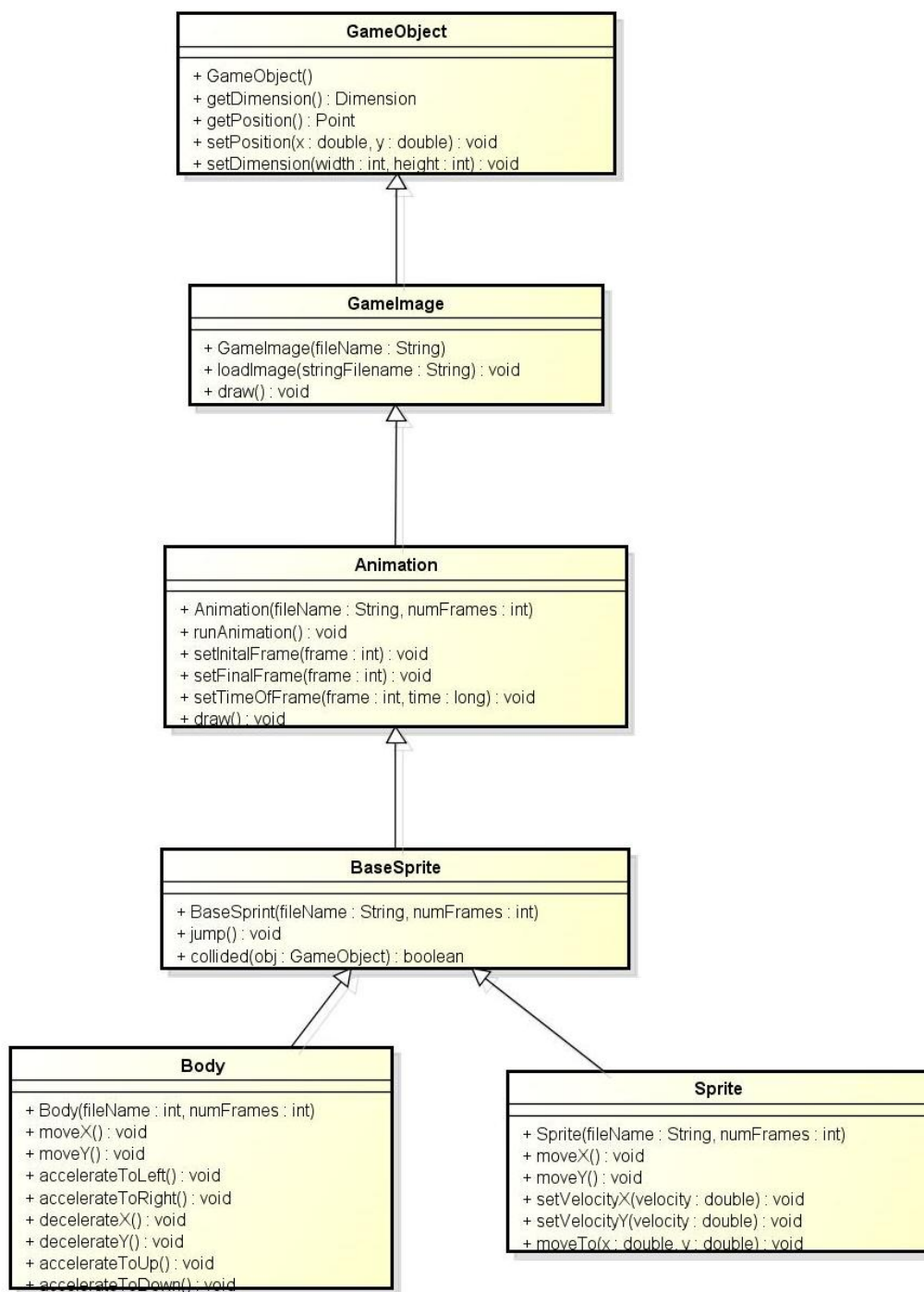
- *Body*: A classe *Body* estende a classe *Animation*. Assim como *Sprite*, a classe *Body* também possui métodos que podem fazer a imagem se mover, mas além destes métodos ela acrescenta métodos para acelerar e desacelerar a imagem pela tela.



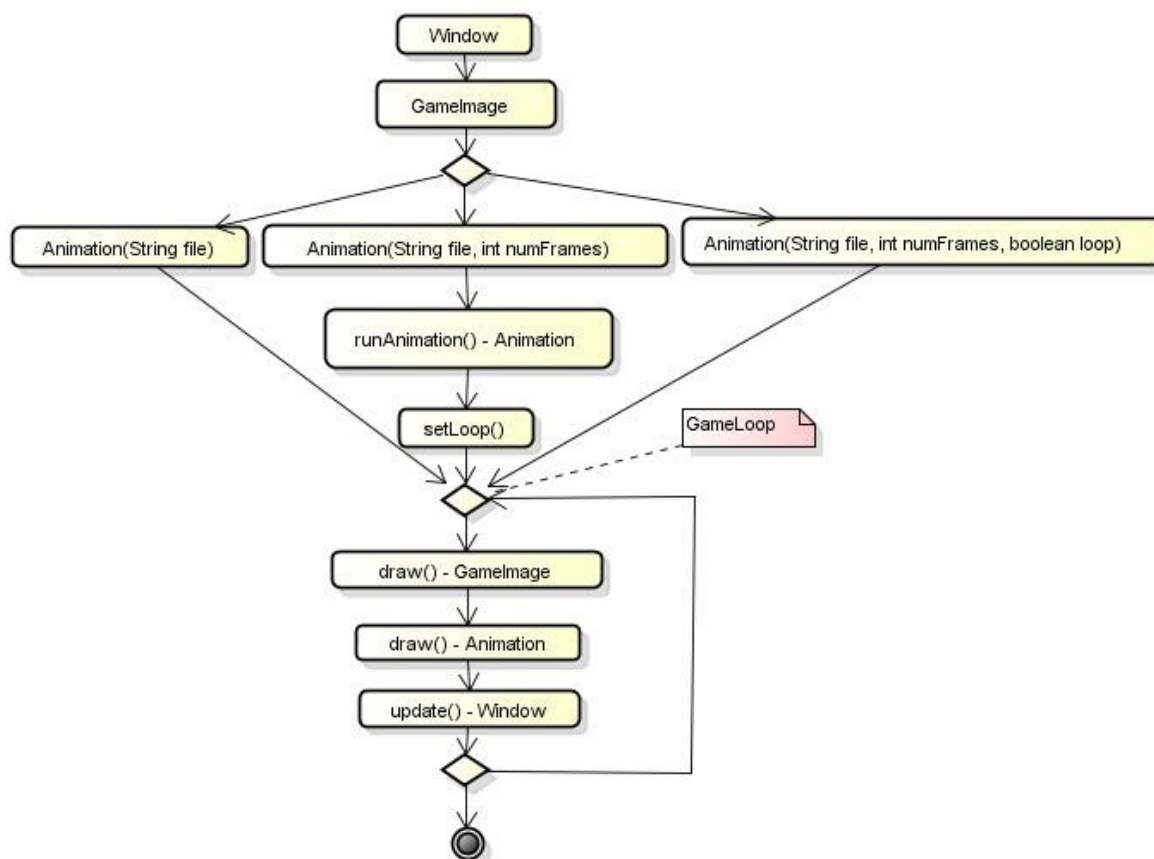
**Figura 3.3: Diagrama padrão de sequência para objetos do tipo da classe *Mouse***

A partir do diagrama da Figura 3.4, podemos definir os seguintes padrões sequenciais:

- Para uso da classe *Animation*: Ao criar um objeto do tipo *Animation*, o programador deve ativar o objeto através do método *runAnimation*, em seguida o objeto deve ser desenhado através do método *draw*, as atualizações do objeto (mudança de posição na janela e mudança de frames) devem ser atualizadas dentro do *gameloop* do jogo através do método *draw*, conforme mostra Figura 3.5.

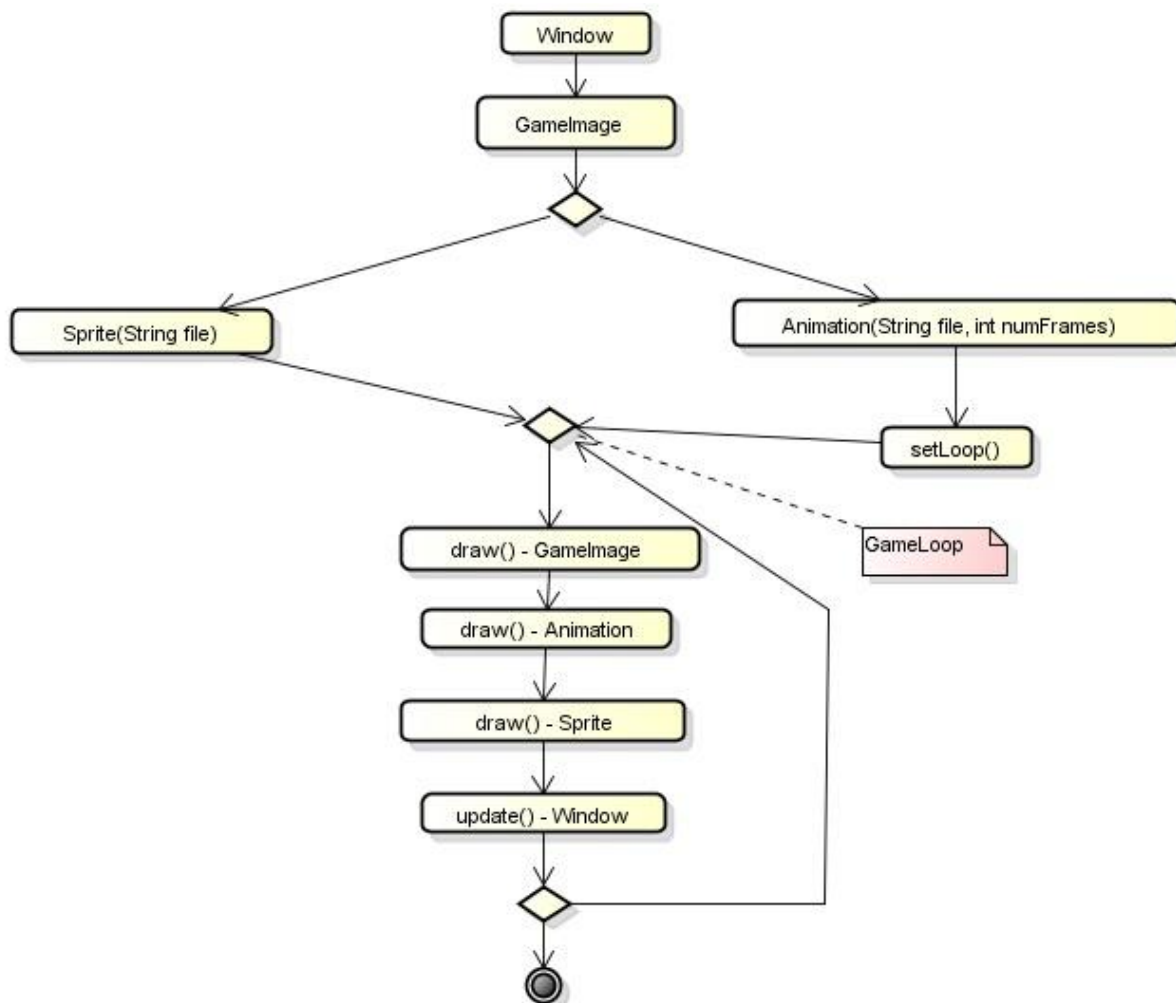


**Figura 3.4: Diagrama de classes JPlay para objetos dinâmicos do jogo**



**Figura 3.5: Diagrama padrão de sequência para personagens do jogo do tipo *Animation***

- Para uso da classe *Sprite*: Ao criar um objeto do tipo *Sprite*, o programador deve indicar qual a posição inicial do objeto através dos atributos *x* e *y*, em seguida o objeto deve ser desenhado através do método *draw*, as atualizações do objeto (mudança de posição na janela e mudança de frames) devem ser atualizadas dentro do *gameloop* do jogo através do método *draw*, conforme mostra Figura 3.6.



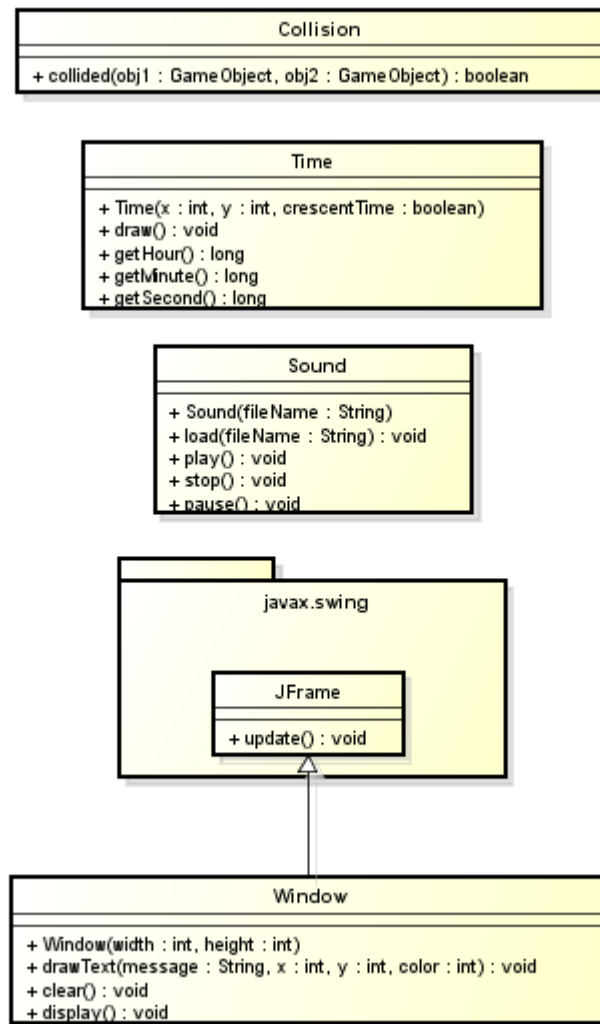
**Figura 3.6: Diagrama padrão de sequência para personagens do jogo do tipo *Sprite***

### 3.4 ARQUITETURA JPLAY: SAÍDAS DO JOGO

Conforme Figura 3.7, as classes responsáveis por saídas no jogo são:

- *Window*: Define uma janela, onde todos os elementos do jogo serão desenhados.
- *Time*: Define um contador de tempo.
- *Sound*: Define a execução de som.
- *Collision*: Classe estática, usada para verificar se houve uma colisão entre dois objetos. A ocorrência de uma colisão pode ser verificada usando o método *collided* presente em todas as classes, ou através da classe estática *Collision*.



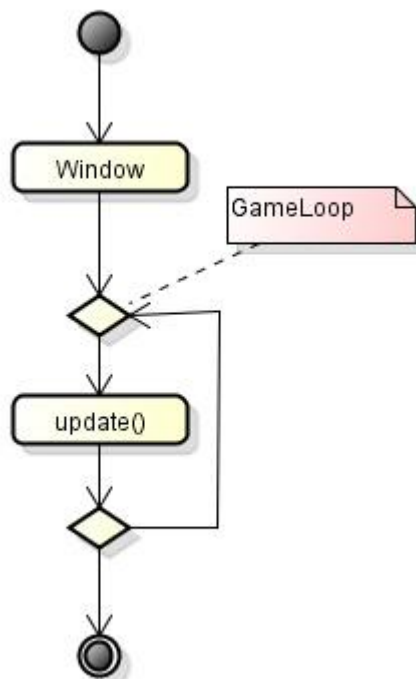


**Figura 3.7: Diagrama de classes para saídas do jogo**

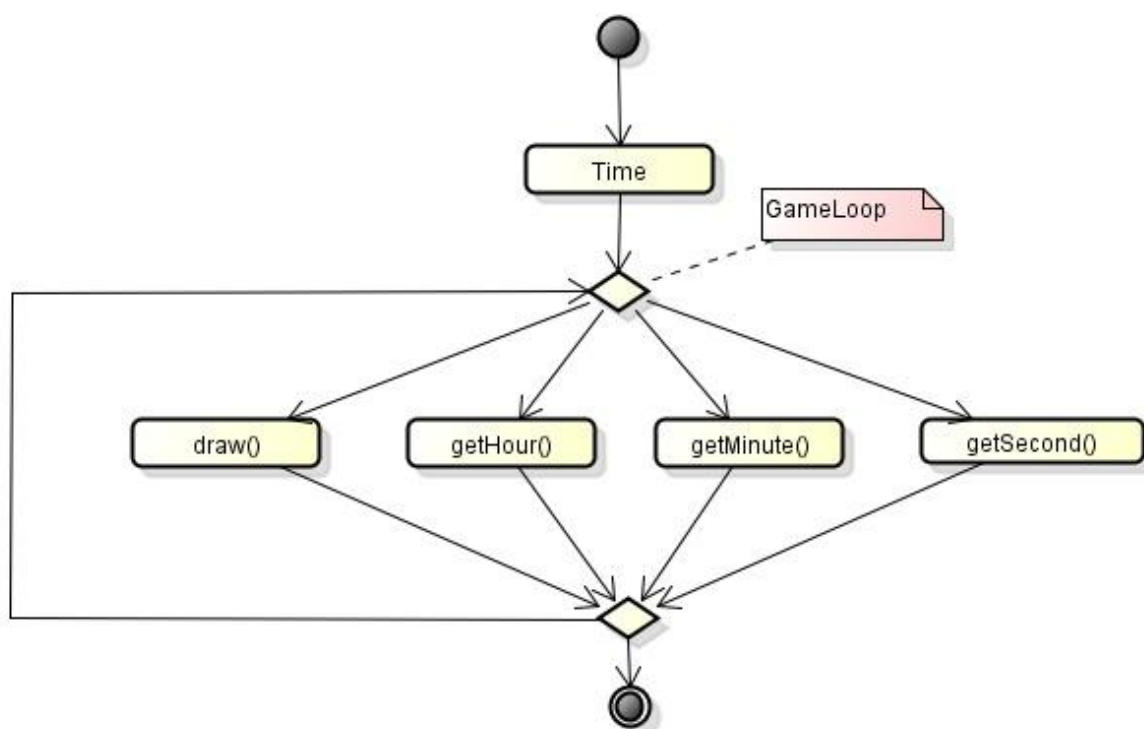
A partir do diagrama da Figura 3.7, podemos definir os seguintes padrões sequenciais:

- Para uso da classe *Window*: Ao criar um objeto do tipo *Window*, é necessário em seguida atualizar todas as modificações feitas dentro da janela durante o jogo. Estas atualizações devem ser feitas dentro do *gameloop* do jogo através do método *update*, conforme mostra Figura 3.8.
- Para uso da classe *Time*: Ao criar um objeto do tipo *Time*, é necessário em seguida desenhar o objeto através do método *draw*, no caso do programador escolher mostrar hora, minuto e segundo. Para mostrar somente a hora, deve ser usado o método *getHour*. Para mostrar somente minutos, deve ser usado o método *getMinute*. Para mostrar somente segundos, deve ser usado o método *getSecond*. É necessário atualizar todas as modificações feitas dentro da janela durante o jogo, estas

atualizações devem ser feitas dentro do *gameloop* do jogo através do método *update*, conforme mostra Figura 3.9.

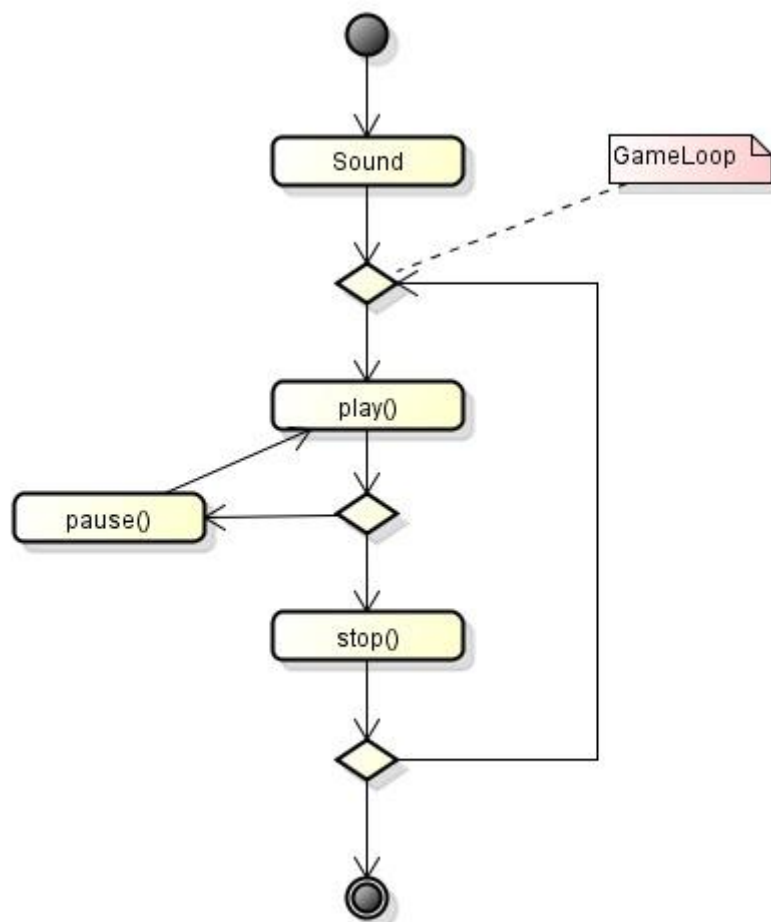


**Figura 3.8:** Diagrama padrão de sequência para saídas do jogo do tipo *Window*



**Figura 3.9:** Diagrama padrão de sequência para saídas do jogo do tipo *Time*

- *Sound*: Ao criar um objeto do tipo *Sound*, é necessário em seguida carregar o som através do método *load*. Na sequência, é necessário ativar o som através do método *play*. Após um som estar ativo ele pode ser desativado, através do método *stop*, ou pode ser pausado, através do método *pause*. Neste último caso é necessário que o som volte a ser ativado depois, através de outra chamada ao método *play*, conforme mostra Figura 3.10.



**Figura 3.10: Diagrama padrão de sequência para saídas do jogo do tipo *Sound***

### 3.5 CONCLUSÃO

O mapeamento da arquitetura JPlay permitiu identificar pontos críticos do desenvolvimento do programa onde uma sequência de código pré-definida (padrões sequenciais) normalmente devem acontecer.

Uma vez identificadas estas sequências de código, o analisador é capaz de realizar uma busca no código-fonte do aluno por uma determinada sequência padrão. Caso a sequência não aconteça, isto pode significar um erro semântico no código-fonte do aluno. Exemplos simples de sequências padrões no JPlay são: após instanciar um objeto do tipo *"Sprite"*, este objeto deve ser desenhado na janela do jogo, para isso o método *"draw"* deve ser chamado, um outro exemplo é após instanciar um objeto do tipo *"window"*, este objeto deve ser atualizado durante a execução do jogo, para realizar a atualização da janela do jogo, o método *"update"* deve ser chamado.

Após a realização do mapeamento da arquitetura foi possível definir quais os padrões sequenciais mais importantes durante o desenvolvimento de um jogo usando o framework JPlay. Para tanto, estes padrões sequenciais foram pré-definidos no analisador. Assim, quando o analisador inicia uma análise no código-fonte do aluno, ele inicialmente verifica se determinadas sequências de código do programa do aluno estão de acordo com o padrão sequencial JPlay.

Este trabalho não pré-definiu, no analisador, todas as buscas de padrões sequenciais possíveis da arquitetura JPlay, restringindo-se aos padrões sequenciais mais importantes para o funcionamento de um jogo básico 2D. No entanto, em trabalhos futuros é possível realizar a adição de mais sequências padrões.

## CAPÍTULO 4 – ARQUITETURA DO ANALISADOR SEMÂNTICO DE PROGRAMAÇÃO

Neste capítulo será apresentada a arquitetura do analisador semântico, englobando o processo para realizar a leitura do código-fonte, representação do mesmo em código XML e posterior pesquisa de informações no código XML gerado.

### 4.1 INTRODUÇÃO

Neste trabalho, a leitura do código-fonte de cada classe Java de um programa é realizada através do uso do Eclipse ASTParser (ECLIPSE ASTParser, 2013). O Eclipse ASTParser faz a leitura e posterior transformação do código-fonte Java em uma árvore de dados, onde os elementos do código-fonte são representados através de nós da árvore AST (*Abstract Syntax Tree*). Cada nó AST armazena informações sobre um elemento do código-fonte.

Representações em XML foram analisadas com o objetivo de definir qual a forma mais adequada para os objetivos deste trabalho. De acordo com Santos (2009), podemos citar algumas representações, tais como JavaML de Mamas e Kontogiannis (MAMAS, 200), JavaML de Badros (BADROS, 2000), JavaML 2.0 de Ademar Aguiar (AGUIAR, 2004).

Das representações citadas, a JavaML de Badros (BADROS, 2000) foi a que mais se aproximou do objetivo do nosso trabalho, pois ao contrário de outras representações, a representação de Badros preserva totalmente o código-fonte original, possibilitando assim análises detalhadas do mesmo. Seguem algumas de suas características:

- Mantém as informações da AST do código-fonte no formato XML;
- As informações são obtidas por meio da utilização de um *parser*;
- Descreve cada elemento do código através de marcadores;
- A representação atribui um identificador (*id*) para cada método e variável, e uma referência ao identificador (*idref*) para todo relacionamento que utilize estas entidades, como invocação de métodos e atribuição de variáveis.
- Embora neste trabalho desenvolvemos um *parser* baseado na representação JavaML de Badros (BADROS, 2000), foi necessário realizar modificações na representação original JavaML com o objetivo de adaptar o modelo de acordo com a proposta. A mais importante

modificação foi a criação de um novo marcador com o nome de <behavior>. O marcador <behavior> torna possível consultar os comportamentos de cada classe do programa.

Após a transformação do código-fonte em XML (de acordo com a representação JavaML de Badros), a pesquisa de informações no código XML gerado é feita através do uso da interface DOM. A análise do código XML consiste em pesquisar um conjunto de regras, através do uso da interface DOM, e verificar os resultados. O objetivo da análise é montar os pares de classes similares entre os programas e identificar as diferenças de comportamento entre eles.

Considerando as fases de leitura e transformação do código-fonte apresentadas, além desta seção introdutória, este capítulo está dividido em 4 seções. Na seção 4.2, é apresentado o AST, responsável pela leitura do código-fonte das classes Java de um programa. Na seção 4.3 é apresentado com detalhes o modelo JavaML de Badros (Badros, 2000), representação escolhida como padrão neste trabalho. Na seção 4.4 é apresentado o DOM, responsável pela leitura e pesquisas no código XML gerado. Por fim, na seção 4.5, são relatadas as considerações finais deste capítulo.

## 4.2 AST

Um código-fonte escrito em Java, através de uma IDE Eclipse, pode ser representado como uma árvore de nós AST através do uso do Eclipse ASTParser (ECLIPSE ASTParser, 2013). Cada nó AST armazena informações sobre um elemento do código-fonte, como por exemplo um nó AST, podendo representar uma classe do código-fonte e guardar informações sobre os atributos e métodos desta classe.

Inicialmente, os arquivos de extensão Java de um projeto são capturados e tratados como um objeto do tipo da classe *CompilationUnit*. A classe *CompilationUnit* é a classe raiz da AST. A partir de uma *CompilationUnit* os seguintes tipos podem ser encontrados (ECLIPSE PLATFORM, 2012):

- *PackageDeclaration* : Um objeto é do tipo da classe *PackageDeclaration* quando representa o pacote ao qual o arquivo Java (objeto do tipo *CompilationUnit*) pertence. É retornado pelo método *getPackage()*.
- *ImportDeclaration*: Um objeto é do tipo da classe *ImportDeclaration* quando representa as importações de pacotes ou classes realizadas dentro do arquivo java. É retornado pelo método *imports()*. A classe *ImportDeclaration* contém o método *getName()* que retorna o nome da importação.

- *TypeDeclaration*: Um objeto é do tipo da classe *TypeDeclaration* quando representa uma declaração de classes ou uma declaração de interface. É retornado pelo método *types()*. Na classe *TypeDeclaration* estão declarados os métodos *getMethods()* e *getFields()*, a partir dos quais é possível capturar métodos e atributos, respectivamente, da classe ou interface a ser tratada.

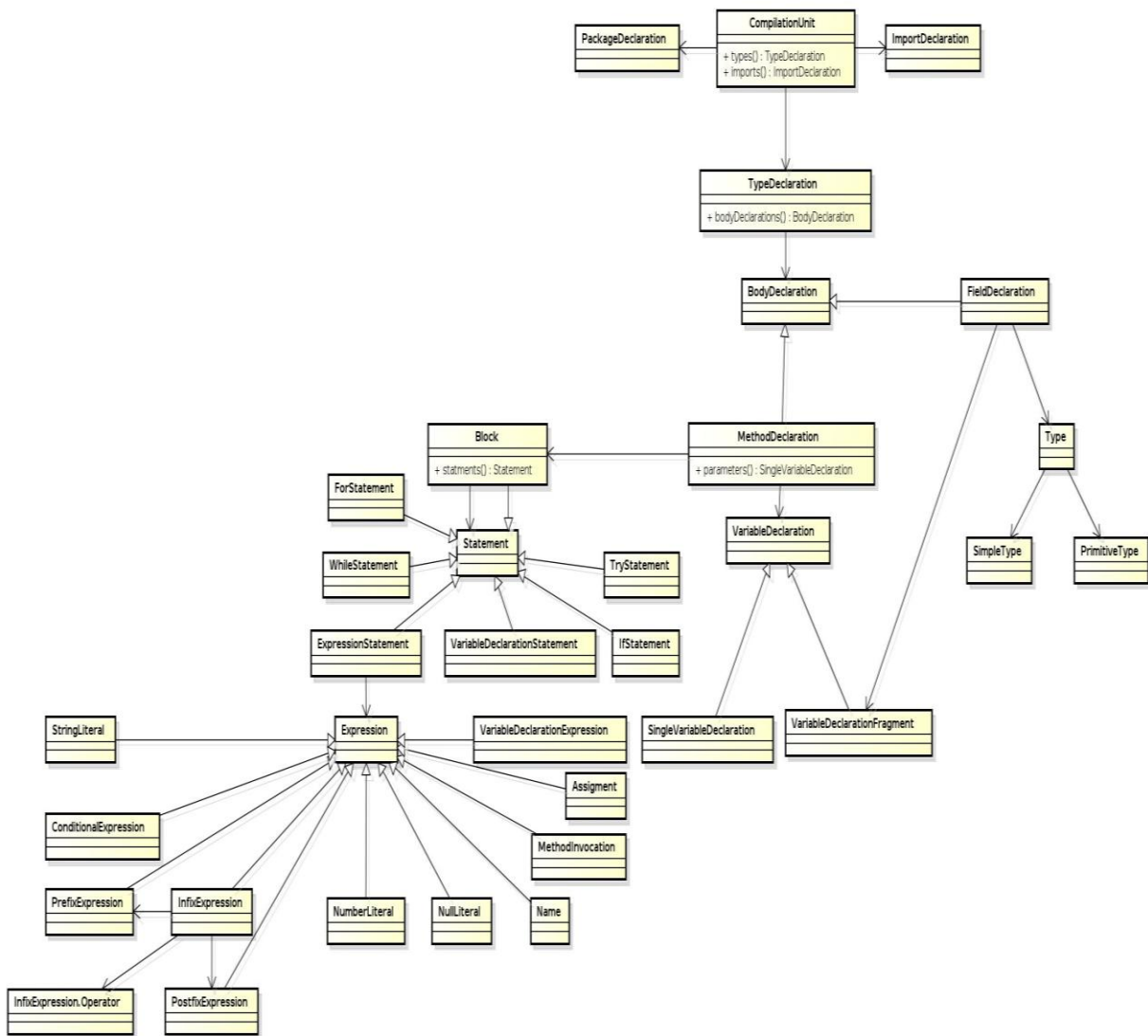
Especificamente neste trabalho a análise e captura dos elementos a partir de uma *TypeDeclaration* é o que vai possibilitar a captura do código fonte escrito pelo usuário.

Na Figura 4.1, podemos observar no diagrama da AST que a partir da classe *TypeDeclaration* é possível obter acesso a classe abstrata *BodyDeclaration*. A classe *BodyDeclaration* é implementada pelas classes *FieldDeclaration* e *MethodDeclaration*, que representam, respectivamente, as declarações de atributos e métodos.

Os parâmetros de um método são acessados através do método *parameters()* da classe *MethodDeclaration*. Este método retorna uma lista de objetos do tipo da classe *SingleVariableDeclaration*.

Na Figura 4.1 podemos observar a superclasse *Statement*. A superclasse *Statement* representa todas as estruturas que se encontram dentro de um método. Cada estrutura é representada por uma classe herdeira de *Statement*. Abaixo segue a descrição de algumas dessas estruturas e suas características mais importantes para a realização deste trabalho:

- *Block*: A classe *Block* representa o trecho do código-fonte referente ao corpo do método e, através de sucessivas análises, representa também trechos menores de código-fonte que estão delimitados por alguma estrutura definida em níveis inferiores a declaração de método, tais como uma estrutura condicional *if*, estruturas de repetição *while*, *for*, etc.



**Figura 4.1: Diagrama de classes AST**

Através da classe *Block* é possível obter acesso às estruturas que compõem o corpo do método que está sendo tratado. Essas estruturas, por sua vez, podem ser declarações de variáveis ou estruturas de controle tais como *if*, *for*, *while*, *do-while*, *return*, *break* ou *continue*.

- *For*: A estrutura de repetição *for* é representada pela classe *ForStatement*. A classe *ForStatement* contém os métodos:
  - (a) Método *initializers()*, que retorna uma lista de expressões (do tipo *Expressions*) representando a inicialização das variáveis a serem incrementadas no *loop for*;



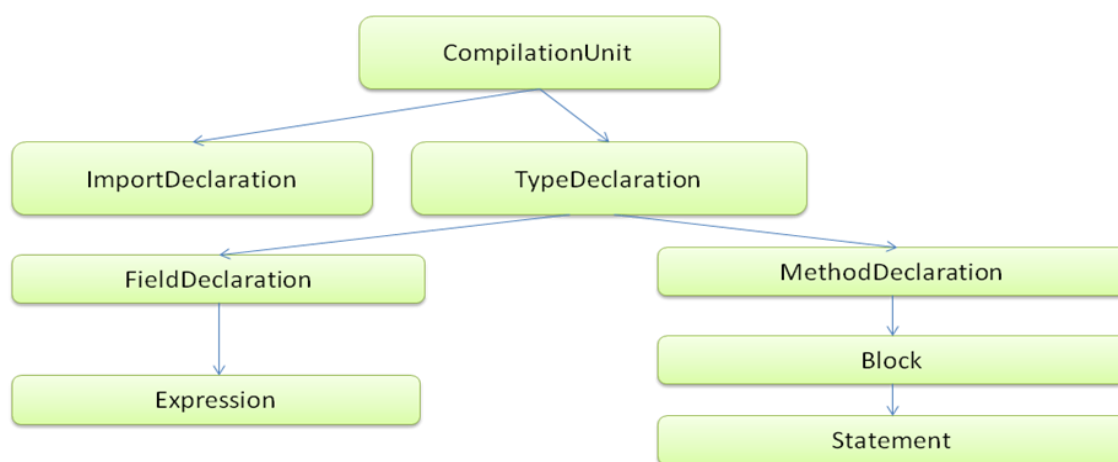
- (b) Método *updaters()*, que retorna uma lista de expressões (do tipo *Expressions*) de incremento, aonde ocorre o incremento da variável usada como contador do *loop for*;
  - (c) Método *getExpression()*, que retorna a expressão (do tipo *Expression*) de parada do *loop for*;
  - (d) Método *getBody()*, que retorna o código-fonte presente no corpo do *loop for*.
- *While*: A estrutura de repetição *while* é representada pela classe *WhileStatement*. A classe *WhileStatement* contém os métodos:
  - (a) Método *getExpression()*, que retorna a expressão (do tipo *Expression*) a ser testada pelo *while*;
  - (b) Método *getBody()*, que retorna o código-fonte presente no corpo do *while*.
- *Do-while*: A estrutura de repetição *do-while* é representada pela classe *DoStatement*. A classe *DoStatement* contém os métodos:
  - (a) Método *getExpression()*, que retorna a expressão (do tipo *Expression*) a ser testada pelo *do-while*;
  - (b) Método *getBody()*, que retorna o código-fonte presente no corpo do *do-while*.
- *If*: A estrutura condicional *if* é representada pela classe *IfStatement*. A classe *IfStatement* contém os métodos:
  - (a) Método *getExpression()*, que retorna a condição a ser testada pelo *if*;
  - (b) Método *getThenStatement()*, que retorna um objeto do tipo *Block*. Este objeto representa o código a ser executado caso a condição testada pelo *if* seja verdadeira;
  - (c) Método *getElseStatement()*, que retorna um objeto do tipo *Block*. Este objeto representa o código a ser executado caso a condição a ser testada pelo *if* seja falsa.
- *Switch*: A estrutura condicional *Switch* é representada pela classe *SwitchStatement*. A classe *SwitchStatement* contém os métodos:
  - (a) Método *getExpression()*, que retorna a expressão (do tipo *Expression*) a ser testada pelo *switch*;
  - (b) Método *statements()*, que retorna uma lista de objetos da classe *Statements*, aonde cada um dos elementos da lista representa uma expressão (do tipo

*Expression*) presente em uma das opções do *switch*. Nesta lista cada uma das opções do *switch* é representado por uma classe *SwitchCase*. A classe *SwitchCase* contém os métodos:

- b.1) Método *getExpression()*, que retorna a lista de expressões (do tipo *Expression*) presentes em cada uma das opções do *case*;
- b.2) Método *isDefault()*, que retorna *true* caso seja default ou *Falso* se contrário.

O método *getBody()* está declarado na classe *MethodDeclaration*. Sua função é retornar o corpo do método que está tratado. Ele retorna uma classe do tipo *Block*. A classe *Block* representa o trecho do código-fonte referente ao corpo do método. Através de sucessivas análises, ela representa também trechos menores de código-fonte que estão delimitados por alguma estrutura definida em níveis inferiores à declaração de método, tais como uma estrutura condicional *if*, estruturas de repetição *while*, *for*, etc.

Nesse trabalho o código-fonte é capturado de acordo com a estrutura AST e transformado em código XML. Cada classe de um projeto corresponde a um arquivo XML gerado com o mesmo nome. A análise do código-fonte através do AST e a sua posterior transformação em código XML é feita segundo a sequencia apresentada na Figura 4.2.



**Figura 4.2:** Sequencia utilizada para análise de código-fonte via AST

### 4.3 A REPRESENTAÇÃO JAVAML DE BADROS

A Figura 4.3 ilustra a representação JavaML de Badros. O elemento raiz desta representação é o *javasource-program*, seguido logo depois pelo sub-elemento *java-class-file*. O *java-class-file* representa uma classe Java após ter sido transformado pelo *parser*. Nessa representação cada classe Java corresponde a um arquivo XML, cujo nome é identificado pelo *java-class-file*.

O *java-classe-file* possui três sub-elementos:

- *package-decl*: representa a declaração do pacote na qual a classe está inserida e é único para cada arquivo XML, uma vez que cada classe Java só pode pertencer a um único pacote.
- *import*: que representa uma declaração de importação, repete-se pelo número de vezes das declarações *import* no código-fonte Java.
- *class*: representa toda declaração de uma classe. Neste elemento está o centro da análise deste trabalho. Nele estão contidos todos os outros sub-elementos que vão representar o código-fonte original. Podemos destacar alguns principais sub-elementos dessa divisão:

a) *field*: representa um atributo de classe.

b) *constructor*: representa um construtor da classe.

c) *superclass*: representa a super classe através da qual a classe foi estendida.

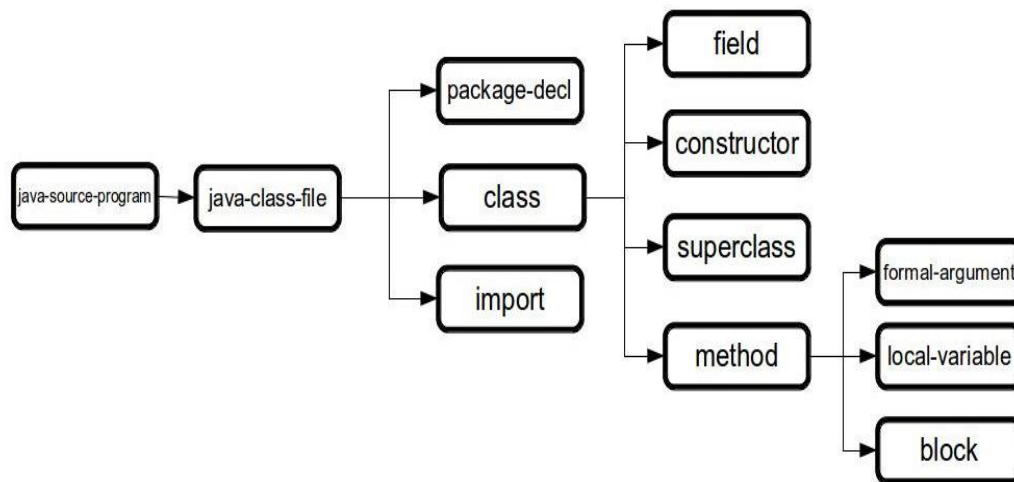
d) *method*: representa a declaração de um método da classe.

O elemento *method* possui os seguintes sub-elementos:

d.1) *formal-arguments*: descreve os parâmetros do método.

d.2) *localvariable*: representa a declaração de variável local, pode ser observado como sub-elemento direto de *method* ou em qualquer nível hierárquico abaixo deste.

d.3) *block*: é utilizado para representar: (a) o corpo do método; (b) sub-blocos de estruturas de decisão; ou (c) sub-blocos de estruturas de repetição. Assim, os sub-elementos de um bloco de código-fonte podem variar sua disposição de acordo com a lógica do programa.



**Figura 4.3: Representação JavaML de Badros**

Pode-se observar na Figura 4.4 o exemplo de um trecho de código-fonte pertencente a uma classe definida com o nome de “*Bar.java*”.

```

public class Bar extends Sprite{
    public Bar(String imagem){
        super(imagem);
    }

    boolean up=false;
    boolean upPc=false;

    public void moveY(int upkey,int downkey,double vel, Keyboard
keyboard{

        if(keyboard.keyDown(upkey) && this.y>0){
            this.y-=vel;
            up=true;
        }
        if(keyboard.keyDown(downkey) && this.y+this.height<600){
            this.y+=vel;
            up=false;
        }
    }
}
  
```

**Figura 4.4: Trecho de código-fonte da classe "Bar.java"**

E, na Figura 4.5, vemos o trecho de código XML correspondente a classe “*Bar.java*” apresentada anteriormente.

```

- <java-source-program>
  - <java-class-file name="Bar.java">
    <import module="jplay.Keyboard"/>
    <import module="jplay.Sprite"/>
    - <class name="Bar" id="54">
      <superclass name="Sprite"/>
      - <constructor id="55">
        - <formal-argument name="imagem" id="56">
          <type name="String" id="56"/>
        </formal-argument>
        - <block>
          - <super-call>
            <var-ref name="imagem" idref="56"/>
          </super-call>
        </block>
      </constructor>
      - <field name="up" id="57">
        <type name="boolean" primitive="true"/>
        <literal-boolean value="false"/>
      </field>
      - <field name="upPc" id="58">
        <type name="boolean" primitive="true"/>
        <literal-boolean value="false"/>
      </field>
      - <method name="moveY" id="59">

```

**Figura 4.5: Trecho de código XML representando a classe "Bar.java"**

Neste trabalho, a modificação consiste no fato de que cada ação determinada por um comando de atribuição, comando condicional ou comando de repetição no código-fonte é definida como um comportamento e, portanto, é descrita como uma tag `<behavior>` no documento XML.

Mais especificamente, um marcador adicional foi criado com o nome de `<behavior>`. Um marcador `<behavior>` possui um tipo, que é identificado através do atributo `<type>`. No caso de estar identificando um comando de atribuição, o atributo `<type>` recebe o valor `<assignment>`; quando identifica um comando condicional, o atributo `<type>` recebe o valor de `<conditional>`; quando identificar um comando de repetição, o atributo `<type>` recebe o valor de `<loop>`.

É importante observar que a inclusão do marcador `<behavior>` não suprimiu ou alterou diretamente quase nenhum dos marcadores já existentes na representação javaML de Badros, sendo que a única exceção diz respeito ao marcador `<do-while>`, que foi definido pela representação de Badros de maneira diferente do restante dos marcadores `<while>` e `<for>`, também representantes de laços de repetição. Optamos, no entanto, por manter um padrão para todos os marcadores `<while>`, `<for>` e `<do-while>`, indicando no atributo `<kind>` do marcador `<loop>` os três tipos de laços de repetição.

O nosso objetivo foi minimizar as modificações e manter a representação original, apenas acrescentando uma marcação a mais no XML gerado. Seguem abaixo as marcações originais da representação e logo em seguida a maneira como a marcação <behavior> foi acrescida no código:

- A marcação <if> é acompanhada de uma sub-marcação <test> que identifica a expressão de teste necessária para que o comando condicional se realize e uma sub-marcação <true-case>, que identifica o que deve ser executado caso a expressão seja verdadeira:

```
<if>
```

```
<test> </test>
```

```
<true-case> </true-case>
```

```
</if>
```

Acrescentando-se a marcação <behavior>:

```
<behavior id= ' ' type= 'conditional'>
```

```
<if>
```

```
<test> </test>
```

```
<true-case> </true-case>
```

```
</behavior>
```

- A marcação <while> é acompanhada de: uma sub-marcação <test> que identifica a expressão de teste necessária para o comando de repetição se realize e uma sub-marcação <block> que identifica o que deve ser executado enquanto a expressão for verdadeira:

```
<loop kind= 'while'>
```

```
<test> </test>
```

```
<block> </block>
```

Acrescentando-se <behavior>:

```
<behavior id= ' ' type= 'loop'>
```

```
<loop kind= 'while'>
```

<test> </test>

<block> </block>

</behavior>

- A marcação <for> vem acompanhada de: uma sub-marcação <init> que identifica a inicialização da variável utilizada no laço, uma sub-marcação <test> que identifica a expressão de teste necessária para o comando de repetição se realize, uma sub-marcação <update> que identifica a atualização da variável a cada execução do laço e uma sub-marcação <block> que identifica o que deve ser executado enquanto a expressão for verdadeira:

<loop kind='for'>

<init> </init>

<test> </test>

<update> </update>

<block> </block>

Acrescentando-se <behavior>:

<behavior id=' ' type='loop'>

<loop kind='for'>

<init> </init>

<test> </test>

<update> </update>

<block> </block>

</behavior>

- A marcação <do-while> vem acompanhada de uma sub-marcação <test> que identifica a expressão de teste necessária para o comando de repetição se realize e uma sub-marcação <block>, que identifica o que deve ser executado enquanto a expressão for verdadeira:

<do-while>

<test> </test>

```

<block> </block>

</do-while>

Acrescenta-se <behavior>:

<behavior id=‘ ’ type=‘loop’>

<loop kind=‘do-while’>

<test> </test>

<block> </block>

</behavior>

```

A inserção da marcação <behavior> no código XML gerado consiste em possibilitar a pesquisa pelo nome deste marcador comum e, conseqüentemente, obter os diversos tipos de comportamentos realizados dentro da classe de um programa. Estes comportamentos, por sua vez, serão usados como parâmetros de comparação com os comportamentos das classes de outro programa, possibilitando assim o processo de análise de pares de classes similares.

Outra modificação realizada na representação javaML original de Badros foi a inclusão do atributo *id* e *idref*. O *id* é um atributo que representa um identificador único. Inicialmente o *id* é representado por uma sequência de caracteres que identifica qual o tipo do identificador, tais como “meth” para representar um *method*, “frmarg” para representar um *formal-argument*, etc. Logo após a identificação de tipo, segue um número inteiro, gerado automaticamente, que deve ser único e responsável por identificar aquele componente no código XML, conforme mostra a Figura 4.6.

O *idref*, por sua vez, identifica a invocação de um atributo ou método já declarado anteriormente no documento XML. O valor do *idref* deve ser o valor do *id* referente ao atributo ou método invocado. Esta ligação é padrão no XML, sendo portanto as ferramentas XML capazes de rastrear a origem de uma variável e utilizar a sua definição para, por exemplo, obter o tipo da variável (BADROS, 2000). Veja na Figura 4.6.

Na representação original de Brados os valores *id* e *idref* são gerados aleatoriamente, neste trabalho os valores são incrementados a medida que surge a necessidade de identificar novos elementos.



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE java-source-program SYSTEM "java-ml.dtd">
3
4  <java-source-program name="FirstApplet.java">
5    <import module="java.applet.*"/>
6    <import module="java.awt.*"/>
7    <class name="FirstApplet" visibility="public">
8      <superclass class="Applet"/>
9      <method name="paint" visibility="public" id="meth-15">
10        <type name="void" primitive="true"/>
11        <formal-arguments>
12          <formal-argument name="g" id="frmarg-13">
13            <type name="Graphics"/></formal-argument>
14          </formal-arguments>
15          <block>
16            <send message="drawString">
17              <target><var-ref name="g" idref="frmarg-13"/></target>

```

**Figura 4.6: Trecho de uso *id* e *idref* na representação JavaML de Badros**

**Fonte: (BADROS, 2000)**

#### 4.4 DOM

DOM é um padrão W3C (*World Wide Web Consortium*) e define padrão de acessos para documentos XML e HTML. O DOM é separado em 3 (três) diferentes partes: Core DOM (padrão de modelo para qualquer estrutura de documento); XML DOM (padrão de modelo para documentos XML) e HTML DOM (padrão de modelo para documentos HTML). A interface DOM define objetos e propriedades para todos os elementos de um documento de um desses tipos citados anteriormente, e métodos capazes de acessar estes elementos (DOM, 2012). No caso da XML DOM, a interface representa um documento XML na forma de uma árvore de objetos, de forma que após a leitura do documento XML, a memória deverá conter a árvore de objetos DOM. O objeto básico do DOM é o nó (*Node*). O elemento mais alto de uma árvore DOM é representado pelo objeto *Document*, que também herda do objeto *Node*. O elemento *Document* é a raiz da árvore e a partir dele é possível acessar a estrutura completa do documento XML.

Neste trabalho cada arquivo de classe Java do programa é convertido em XML (segundo a representação JavaML de Badros), gerando assim outro arquivo com o mesmo nome e extensão XML. Cada um dos arquivos XML gerados serão analisados através do uso da interface DOM, com o objetivo de montar os pares de classes similares.

## 4.5 PROLOG

Prolog é uma linguagem de programação baseada nos conceitos da Programação em Lógica, é uma linguagem declarativa e foi criada no início da década de 70. Prolog é baseada na linguagem das cláusulas de Horn, que é um sub-conjunto da linguagem de cálculos de predicados (ABE, 2002).

Em uma segunda versão deste trabalho, todas as regras anteriormente pesquisadas nos documentos XML através do uso da interface DOM foram transformadas em regras PROLOG. O objetivo da nova versão foi melhorar a eficiência do algoritmo.

Após as classes serem transformadas em XML, uma nova representação destas classes é feita em PROLOG. Para cada arquivo XML, portanto, é gerado um arquivo PL respectivo e todas as consultas necessárias são realizadas a partir do motor de inferência tuProlog (interpretador Java para PROLOG).

Pode-se observar na Figura 4.7 o exemplo de um trecho de código-fonte pertencente a uma classe definida com o nome de "Barra.java". Respectivamente, nas Figuras 4.8 e 4.9, observa-se os trechos de código XML e PL correspondentes.

```

import jplay.Keyboard;

/**
 * @comment the bar class must inherit from Sprite, must have two methods:
 *           the constructor may have initialized the bar picture,
 *           and other method may control the movement of the x-axis using the keyboard
 *           (using the keydown method of the keyboard class of the JPlay)
 *
 * @method 2
 * @comment The class may have 2 methods, the constructor may have initialized the bar picture,
 *           and other method may control the movement of the x-axis using the keyboard
 *           (using the keydown method of the keyboard class of the JPlay)
 *
 * @inheritance Sprite
 * @comment the Bar class may inherit from Sprite
 *
 * @movex true
 * @comment Do the movement in the x-axis to left and to right
 *
 * @keyboard true
 * @comment The movement of the bar class in the x-axis may to be controlled using to keyboard
 *           (in order to do that use the keydown method of the JPlay)
 */
public class Barra extends Sprite{
    public Barra(String Imagem){
        super(Imagem);
    }
}

```

**Figura 4.7: Trecho de código-fonte da classe "Barra.java"**

```

-<java-source-program>
- <java-class-file name="Barra.java">
  <import module="jplay.Keyboard"/>
  <import module="jplay.Sprite"/>
- <class name="Barra" id="116">
  - <header comment=" the bar class must inherit from Sprite, must have two methods:">
    <method value=" 2" comment=" The class may have 2 methods, the constructor may have initialized the bar picture, "/>
    <inheritance value=" Sprite" comment=" the Bar class may inherit from Sprite"/>
    <movex value=" true" comment=" Do the movement in the x-axis to left and to right"/>
    <keyboard value=" true" comment=" The movement of the bar class in the x-axis may to be controlled using to keyboard "/>
  </header>
  <superclass name="Sprite"/>
- <constructor id="118" type="changed">
  - <formal-argument name="Imagem" nameType="simple" id="120">
    <type name="String" primitive="false"/>
  </formal-argument>
- <block>
  - <super-call>
    <var-ref name="Imagem" idref="120"/>
  </super-call>
- </block>
</constructor>

```

**Figura 4.8: Trecho de código XML (arquivo "Barra.java.XML") representando a classe "Barra.java"**

```

javasourceprogram(id0).
javaclassfile(id0,id1).
tag(id0,id1) :- javaclassfile(id0,id1).
name(id1,"Barra.java").
import(id1,id2).
tag(id1,id2) :- import(id1,id2).
module(id2,"jplay.Keyboard").
import(id1,id3).
tag(id1,id3) :- import(id1,id3).
module(id3,"jplay.Sprite").
class(id1,id4).
tag(id1,id4) :- class(id1,id4).
id(id4,"116").
name(id4,"Barra").
header(id4,id5).
tag(id4,id5) :- header(id4,id5).
comment(id5," the bar class must inherit from Sprite, must have two methods:").
method(id5,id6).
tag(id5,id6) :- method(id5,id6).
comment(id6," The class may have 2 methods, the constructor may have initialized the bar picture, ").
value(id6," 2").
inheritance(id5,id7).
tag(id5,id7) :- inheritance(id5,id7).
comment(id7," the Bar class may inherit from Sprite").
value(id7," Sprite").
movex(id5,id8).
tag(id5,id8) :- movex(id5,id8).
comment(id8," Do the movement in the x-axis to left and to right").
value(id8," true").
keyboard(id5,id9).
tag(id5,id9) :- keyboard(id5,id9).
comment(id9," The movement of the bar class in the x-axis may to be controlled using to keyboard ").
value(id9," true").
superclass(id4,id10).
tag(id4,id10) :- superclass(id4,id10).
name(id10,"Sprite").
constructor(id4,id11).
tag(id4,id11) :- constructor(id4,id11).
id(id11,"118").

```

**Figura.: Trecho de código PROLOG (arquivo "Barra.java.PL") representando a classe "Barra.java"**

Todas as tags definidas no XML são transformadas em regras no PROLOG, para que sejam feitas, através delas, posteriores consultas PROLOG.

Seguem exemplos de trechos de código PROLOG (arquivo *"Barra.java.PL"*) descrevendo os comportamentos *"assignment"*, *"conditional"* e *"loop"*:

- Comportamento *<assignment>*:

```
block(id73,id74).
tag(id73,id74) :- block(id73,id74).
behavior(id74,id75).
tag(id74,id75) :- behavior(id74,id75).
id(id75,"131").
type(id75,"assignment").
assignmentexpr(id75,id76).
tag(id75,id76) :- assignmentexpr(id75,id76).
op(id76,"-=").
comment(id76,id77).
tag(id76,id77) :- comment(id76,id77).
help(id77,"comment04:Decrementar o movimento do Barra no eixo x,
fazendo a mesma se mover para a esquerda").
lvalue(id76,id78).
tag(id76,id78) :- lvalue(id76,id78).
this(id78,id79).
tag(id78,id79) :- this(id78,id79).
varref(id79,id80).
tag(id79,id80) :- varref(id79,id80).
idref(id80,"-1").
name(id80,"x").
.....
```

- Comportamento *<conditional>*:

```
block(id17,id25).
tag(id17,id25) :- block(id17,id25).
behavior(id25,id26).
tag(id25,id26) :- behavior(id25,id26).
id(id26,"128").
type(id26,"conditional").
if(id26,id27).
tag(id26,id27) :- if(id26,id27).
comment(id27,id28).
tag(id27,id28) :- comment(id27,id28).
help(id28,"comment01:Definir movimento da Barra para a direita
atraves do uso do teclado usando o metodo KeyDown. Verifique que,
na janela do jogo, a margem direita maxima de x e 800.").
test(id27,id29).
tag(id27,id29) :- test(id27,id29).
.....
```

- Comportamento *<loop>*:

```
behavior(id41,id99).
tag(id41,id99) :- behavior(id41,id99).
id(id99,"47").
type(id99,"loop").
loop(id99,id100).
tag(id99,id100) :- loop(id99,id100).
kind(id100,"for").
comment(id100,id101).
tag(id100,id101) :- comment(id100,id101).
help(id101,"comment07:No metodo carrega() defina um laco de
repeticao para inicializar os 70 blocos definidos.Devem ser
definidos 10 linhas, cada uma com 7 blocos. Utilize um vetor de
duas dimensoes para definir a estrutura (Ex.: bloco[10][7]).").
.....
```

As primeiras consultas realizadas em PROLOG têm o objetivo de montar os pares de classes similares, inicialmente as classes são comparadas através de consultas que tem o objetivo de montar o primeiro e o segundo par de classes entre o programa modelo e o programa do aluno:

- *Primeiro par*: procura pelas classes que contenham o método "main" nos dois programas.

A consulta em tuProlog é realizada da seguinte maneira:

```
return engine.solve("method(X,Y), name(Y,Z), Z = \"main\".").isSuccess();
```

- *Segundo par*: procura pelas classes que contenham um *gameloop* ou loop infinito nos dois programas (representado através da definição de um "while(true)" no código-fonte do programa):

```
return engine.solve("loop(X,Y), kind(Y,Z), Z = \"while\", test( Y, W),
literalboolean(W, J), value(J, K), K = \"true\".").isSuccess();
```

Após os resultados dos dois primeiros pares, é montada uma estrutura de matriz de pares de classes com o restante das classes dos dois programas. O objetivo é definir os pares de classes mais similares para o restante das classes existentes.

O primeiro critério para adicionar um par na matriz de pares é que as classes herdem da mesma superclasse, logo é necessária a definição de uma consulta que busca a origem da superclasse:

```
return engine.solve("superclass(X,Y),
name(Y,Z).").getVarValue("Z").toString().replaceAll("'", "");
```

O segundo critério para adicionar um par na matriz de pares é verificar se objetos do tipo desta classe foram definidos na classe do *gameloop*, se foram definidos, a origem de cada tipo é comparada, se a origem dos tipos for diferente o par não vai ser selecionado na matriz.

A origem de um tipo de um objeto é classificada em:

- *Simple*: Quando o objeto é definido diretamente pelo tipo de uma classe. Por exemplo:

```
Bloco aluno_bloco;
```

- *Array*: Quando o objeto é definido um array ou matriz de classe. Por exemplo:

```
Bloco[][] aluno_bloco;
```

- *List*: Quando o objeto é definido um tipo List do Java. Por exemplo:

```
ArrayList<Bloco> aluno_bloco;
```

- *Map*: Quando o objeto é definido um tipo Map do Java. Por exemplo:

```
HashMap<int, Bloco> aluno_bloco;
```

A definição da consulta que busca a origem do tipo de um objeto de uma determinada classe é:

```
ArrayList<String> lista = new ArrayList<>();

SolveInfo info =
engine.solve("field(X,Y),name(Y,N),nameType(Y,W),type(Y,M),name(M,\""+nomeclasse+"
\").");

while (info.isSuccess()){
    lista.add(info.getVarValue("W").toString().replaceAll("'", ""));

    if (info.hasOpenAlternatives()) {
        info = engine.solveNext();
    } else {
        break;
    }
}

return lista;
```

Neste caso a consulta gera uma lista com todas as origens de tipo encontradas para aquele objeto. Por exemplo:

```
Bloco[][] aluno_bloco;
ArrayList<Bloco> aluno_bloco2;
```

O nome da classe procurada é "*Bloco*". A pesquisa verifica que existem dois objetos do tipo "*Bloco*" ("*aluno\_bloco*", "*aluno\_bloco2*") e retorna uma lista com a origem desses tipos, sendo este retorno a lista ("*array*", "*list*").

A lista encontrada no programa do aluno é comparada com o a lista encontrada no programa modelo se apenas uma das origens de tipo nas duas listas forem constatadas como iguais. Neste caso, o par é adicionado na matriz de pares similares.

Por exemplo, no programa do aluno:

```
AlunoBloco[][] aluno_bloco;
ArrayList<AlunoBloco> aluno_bloco2;
```

E no programa modelo:

```
Bloco[][] bloco;
```

As listas são comparadas:

Aluno: ("array", "list")

Modelo: ("array")

Neste caso o par ["bloco", "AlunoBloco"] será incluído na matriz de pares similares pois as origens iguais ("array", "array") foram encontradas.

Desta forma, é possível selecionar os pares de classes que apresentem mais similaridade. Estes pares são selecionados através de uma heurística baseada na comparação de comportamentos. Mais detalhes sobre a heurística de comportamentos serão descritos no próximo capítulo. Com o objetivo de realizar a comparação de comportamentos entre os pares de classes selecionados também são utilizadas consultas PROLOG.

## 4.6 CONSIDERAÇÕES FINAIS

Neste trabalho é realizada a leitura do código-fonte Java através do AST, bem como o desenvolvimento do *parser* de acordo com a representação JavaML de Badros.

Após a transformação do código-fonte em XML, a pesquisa de informações no código XML gerado foi feita inicialmente através do uso da interface DOM. A análise do código XML consiste em pesquisar um conjunto de regras, através do uso desta interface, e verificar os resultados. O objetivo da análise é montar os pares de classes similares entre os programas e identificar as diferenças de comportamento entre eles.

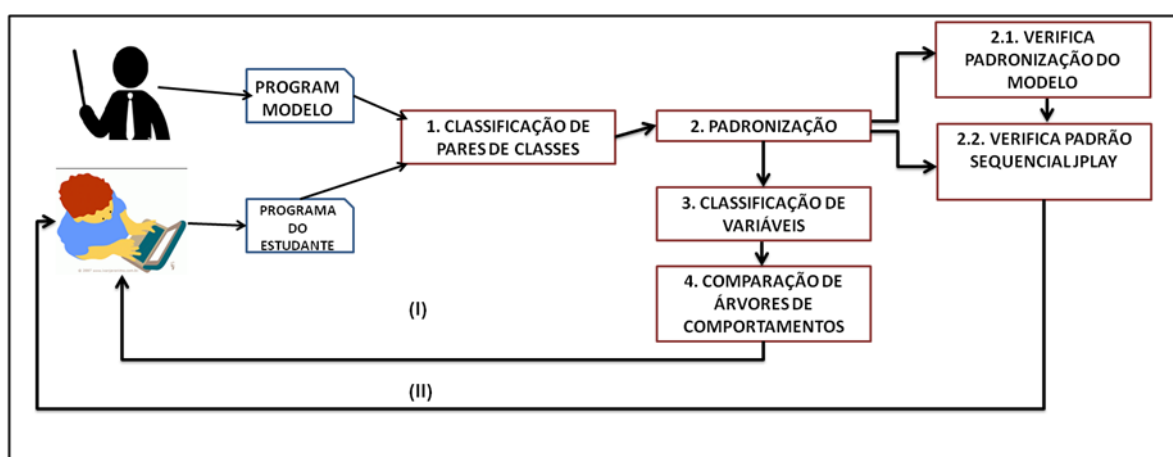
Posteriormente, em uma segunda versão deste trabalho, todas as regras anteriormente pesquisadas nos documentos XML através do uso da interface DOM foram transformadas em regras PROLOG. O uso da linguagem PROLOG possibilitou a construção de buscas mais eficazes e concisas.

## CAPÍTULO 5 O ANALISADOR SEMÂNTICO E A HEURÍSTICA BASEADA EM COMPARAÇÃO DE COMPORTAMENTOS DOS PROGRAMAS

Neste capítulo são apresentados o analisador semântico e a heurística proposta, baseada na comparação de comportamentos de programas. A heurística é composta por várias etapas e módulos. Neste capítulo também serão detalhadas cada uma das etapas envolvidas.

### 5.1 INTRODUÇÃO

O objetivo da comparação entre programas é analisar o código que está sendo gerado pelo aluno e realizar feedback (s) para o mesmo, caso o resultado da análise encontre diferença (s) na comparação. O sistema é composto por várias etapas e módulos, que estão ilustrados na Figura 5.1.



**Figura 5.1: Etapas do processo de análise**

De acordo com a Figura 5.1, o sistema inicia classificando pares de classes similares (1) entre os programas (o par é formado por uma classe do programa modelo, que é a classe do programa escrito pelo professor, e uma classe do programa do estudante). O analisador verifica se a classe do aluno está de acordo com a padronização necessária (2). A padronização é verificada de acordo com: a padronização do modelo (2.1), que é definido pelo próprio professor através de marcadores no início de cada classe no programa modelo, e o padrão sequencial do JPlay (2.2), que é o mapeamento de sequências de código que sempre acontecem no JPlay. É importante que o programa do aluno esteja padronizado de acordo com programa modelo. Quanto mais padronizado, mais precisos serão os resultados obtidos

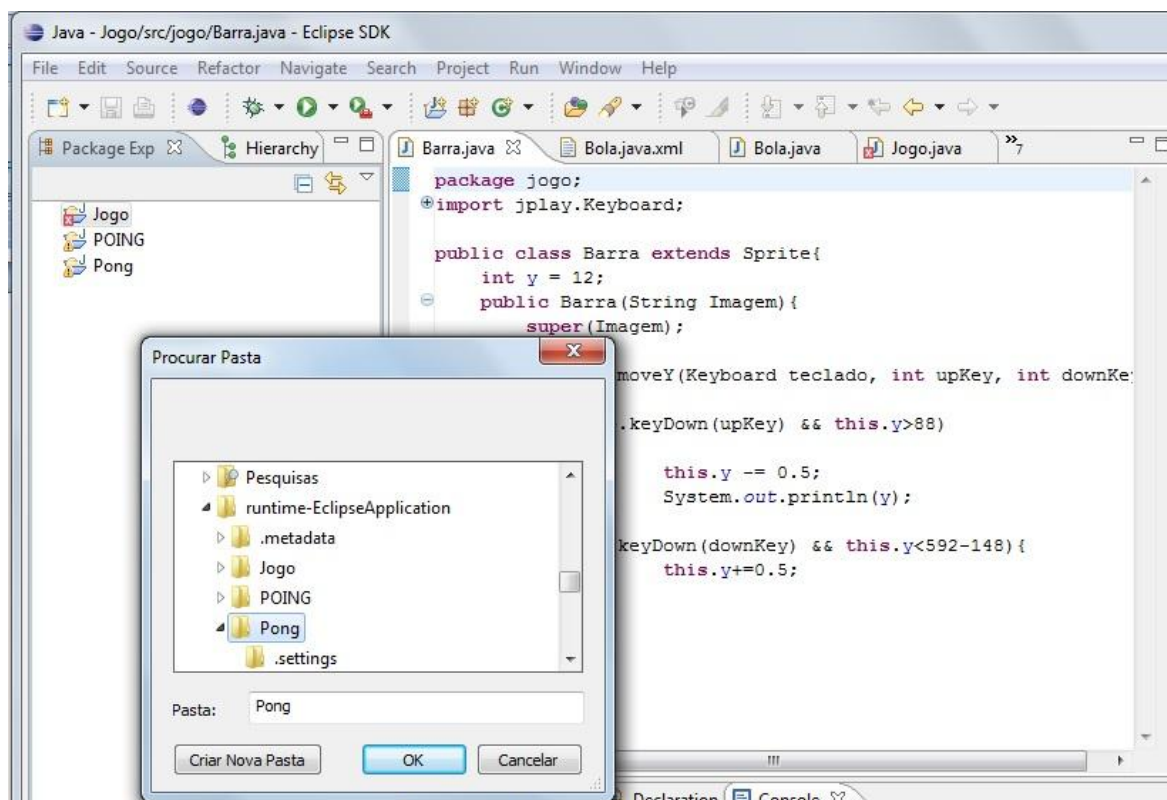


através da análise. Na fase (3) o analisador classifica pares de variáveis similares entre os pares de classes estabelecidos em (1). O analisador mapeia o comportamento de cada variável no programa e define a árvore de comportamento de cada uma, logo, após a classificação dos pares de variáveis, as árvores de comportamentos (4) de ambas podem ser comparadas e as diferenças de comportamento entre elas são identificadas.

## 5.2 SELECIONANDO UM PROGRAMA JAVA PARA ANÁLISE

Para analisar semanticamente um programa, o aluno deve selecionar no seu repositório IDE um programa considerado como programa modelo, conforme mostra a Figura 5.1. A análise consiste em comparar pares de classes. Cada classe “java” do programa do aluno e do programa modelo serão inicialmente transformadas em arquivos XML pelo *parser*. Cada arquivo XML deve ser lido e interpretado usando o DOM [DOM 2012].

No exemplo da Figura 5.2, comparamos o programa do estudante chamado *Jogo* com o programa modelo, selecionado no repositório, chamado *Pong*. Os pares de classes são comparados e os resultados são armazenados na matriz.

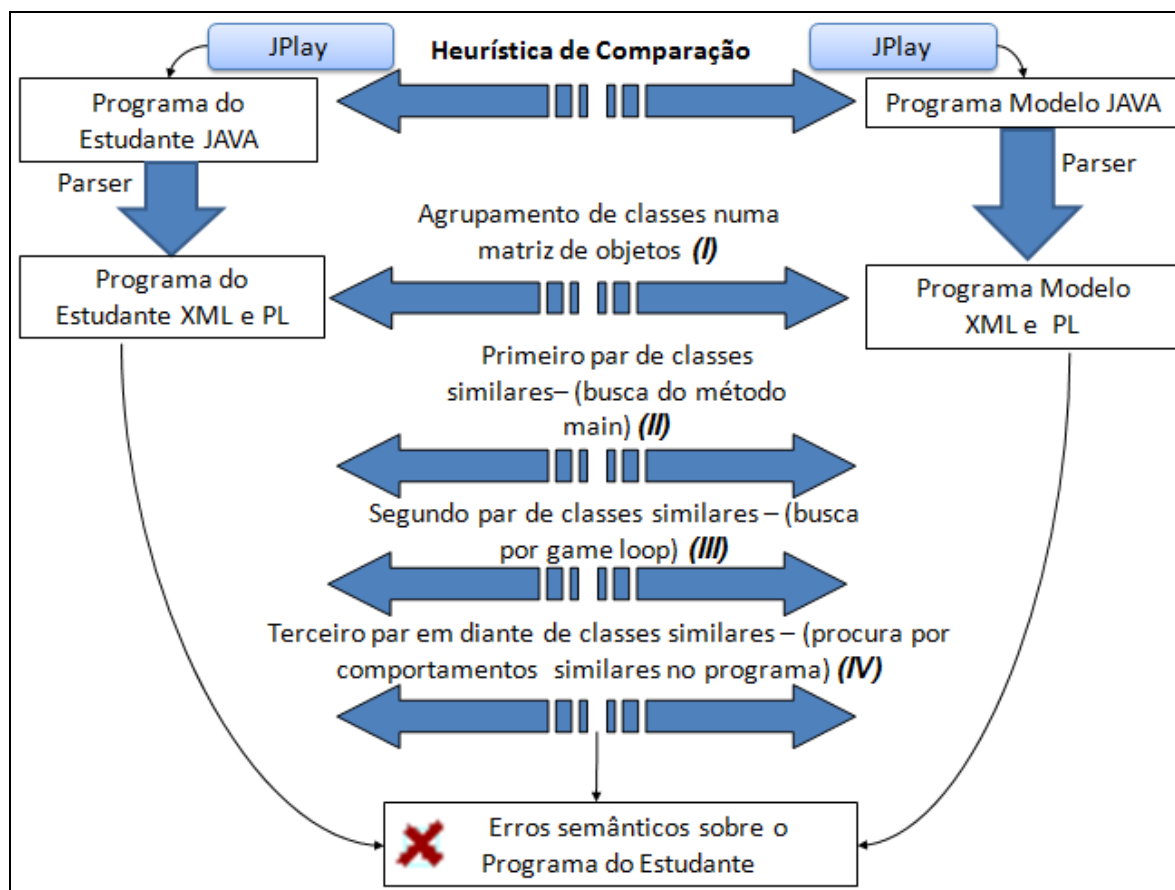


**Figura 5.2: Escolhendo o programa modelo no *plugin* analisador semântico**

O repositório tem disponíveis apenas arquivos XML e PL que representam o programa modelo.

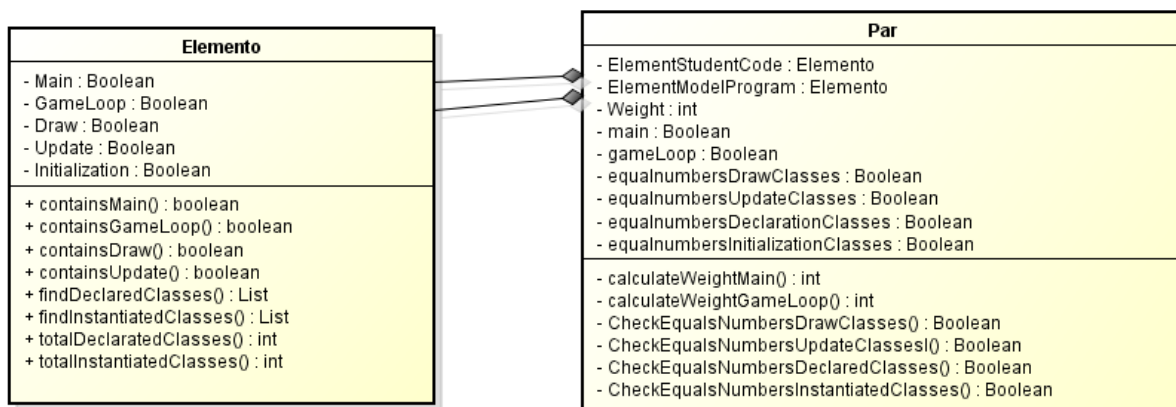
### 5.3 CLASSIFICANDO PARES DE CLASSES

A classificação dos pares de classes é formada por três etapas: classificação do primeiro par, classificação do segundo par de classes e classificação do restante das classes do projeto, como mostra a Figura 5.3.



**Figura 5.3:** Etapas do processo de classificação dos pares de classes

Conforme mostra a Figura 5.3, na fase (I), todas as classes do programa modelo e todas as classes do programa do estudante são agrupadas em uma matriz de objetos. Nesta fase todos os pares *<classe do programa do estudante, classe do programa modelo>* possíveis são inseridos na matriz. Cada classe do programa representa um elemento do par na matriz. Cada elemento e cada par da matriz contêm seus próprios atributos, conforme mostra o diagrama da Figura 5.4.



**Figura 5.4: Diagrama de um Elemento e de um Par de Classes**

Os atributos da classe Elemento e da classe Par são armazenados com o objetivo de serem usados depois como parâmetros para a análise do programa.

Descreve-se a seguir cada um dos atributos da classe Elemento:

- **Main:** parâmetro que é Verdadeiro se a classe contém o método *main* e Falso se a classe não contém o método *main*.
- **GameLoop:** Verdadeiro se a classe contém um *game loop* (ou *loop* infinito) e Falso se a classe não contém um *game loop*.
- **Draw:** Verdadeiro se a classe chama um método *Draw* e Falso se a classe não chama um método *Draw*.
- **Update:** Verdadeiro se a classe chama um método *Update* e Falso se a classe não chama um método *Update*.
- **Initialization:** Verdadeiro se a quantidade de classes instanciadas é igual a quantidade de classes declaradas e Falso se as quantidades são diferentes.

A seguir a descrição de cada um dos atributos da classe Par:

- **ElementStudentCode:** classe do programa do estudante.
- **ElementModelProgram:** classe do programa modelo.
- **Weight:** Peso de similaridade de cada par.
- **Main:** Verdadeiro somente se as duas classes do par contém o método *main*.
- **Gameloop:** Verdadeiro somente se as duas classes do par contém o método *game loop* (ou *loop* infinito).

- `equalNumberDrawClasses`: Verdadeiro somente se as duas classes do par contém o mesmo número de chamadas ao método *draw*.
- `equalNumberUpdateClasses`: Verdadeiro somente se as duas classes do par contém o mesmo número de chamadas ao método *update*.
- `equalNumberDeclarationClasses`: Verdadeiro somente se as duas classes do par contém o mesmo número de classes declaradas.
- `equalNumberInitializationClasses`: Verdadeiro somente se as duas classes do par contém o mesmo número de chamadas ao método *draw*.

Os atributos de cada classe são utilizados pelo analisador com o objetivo de realizar comparações entre pares de classes. Assim, após o agrupamento dos pares de classes, as comparações entre os pares são realizadas. O objetivo neste ponto consiste em classificar os pares de classes com maior grau de similaridade.

A heurística define regras para a comparação dos pares de classes. Cada classe no programa do estudante é comparada com todas as classes do programa modelo, e, de acordo com as regras, os pesos são atribuídos a cada um dos pares.

Inicialmente, a matriz é inicializada com os pesos de similaridade, representados pelo atributo *weight*, iguais a 0 (zero) em todos os pares. Os outros atributos de tipo *boolean* são inicializados como falso.

Existem regras específicas para encontrar o 1º. (primeiro) e o 2º. (segundo) par de classes similares.

Conforme mostra a Figura 5.3, na fase (II), com o objetivo de classificar o 1º. (primeiro) par de classes similares foi estabelecida uma única regra: a classe deve conter o método *main*. Quando é encontrado um par onde os dois elementos contém o método *main*, o peso é igual a 2 (dois) e o atributo *main* é igual a *true*.

Em seguida, conforme Figura 5.3, na fase (III), com o objetivo de classificar o 2º. (segundo) par de classes similares, estabelecemos uma única regra: a classe deve conter um *gameloop* (ou *loop* infinito). Quando os dois elementos do par contém um *game loop*, o peso é igual a 2 e o atributo *gameloop* corresponde a *true*.

Para classificar o 1º (primeiro) par de classes similares, aplica-se a regra específica e o par de classes com maior peso é selecionado. Para classificar o 2º (segundo) par de classes similares, aplica-se a regra específica e o par de classes com maior peso é selecionado.

Após o primeiro e o segundo par de classes similares serem classificados, o analisador deve classificar o resto das classes dos programas (ver fase (III) da Figura 5.3). Esta classificação é realizada em níveis.

Primeiro, o analisador busca pares de classes que estendem da mesma superclasse.

Segundo, o analisador busca objetos criados do tipo das classes do projeto na classe *gameloop*, assim, se objetos de tipos iguais foram encontrados no programa modelo e no programa do aluno, o par é selecionado e a verificação continua, se os tipos dos objetos forem diferentes, o par é desconsiderado.

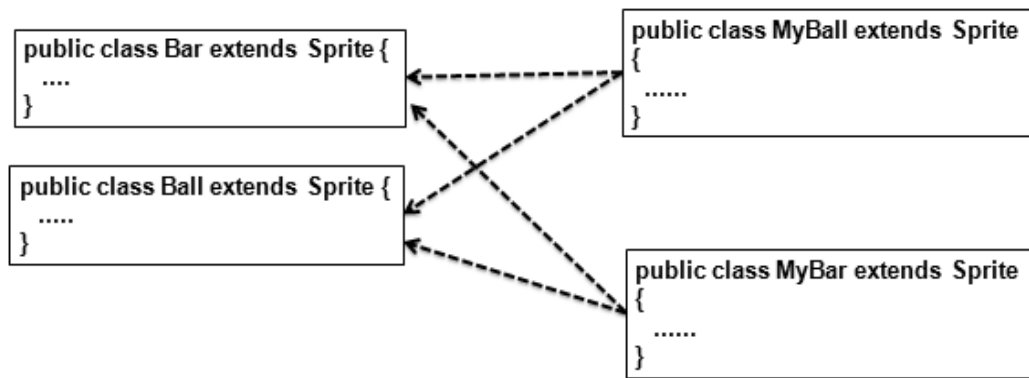
Terceiro, para os pares considerados, é realizada uma classificação baseado na diferença entre as quantidades de variáveis do mesmo tipo dos pares de classes. Por fim, realiza-se uma classificação baseada na diferença entre os comportamentos do mesmo tipo dos pares de classes.

No exemplo da Figura 5.4, a classe “A” pertence ao programa do estudante e a classe “B” pertence ao programa modelo. Durante a classificação o analisador verifica que a classe “A” e a classe “B” herdam da mesma superclasse “*Sprite*”.



**Figura 5.5: Exemplo das classes “A” e “B” que herdam da mesma superclasse *Sprite***

No exemplo da Figura 5.5, as classes “*MyBall*” e “*MyBar*” pertencem ao programa do estudante e as classes “*Ball*” e “*Bar*” pertencem ao programa modelo. Todas as classes herdam da mesma superclasse “*Sprite*”. Então, todas as combinações entre as classes do dois programas que herdam da mesma superclasse serão analisadas com o objetivo de encontrar pares de classes mais similares.



**Figura 5.6:** Exemplo das classes do programa modelo (“*Bar*” e “*Ball*”) e das classes do programa do estudante (“*MyBar*” e “*MyBall*”) que serão combinadas

Na segunda fase, o analisador busca os objetos criados nas classes *gameloop* do programa modelo e do programa do aluno. Por exemplo, no programa do aluno:

```

MyBall[][] aluno_bola;
ArrayList<Bar> aluno_barra;

```

E no programa modelo:

```

Ball[][] bola;

```

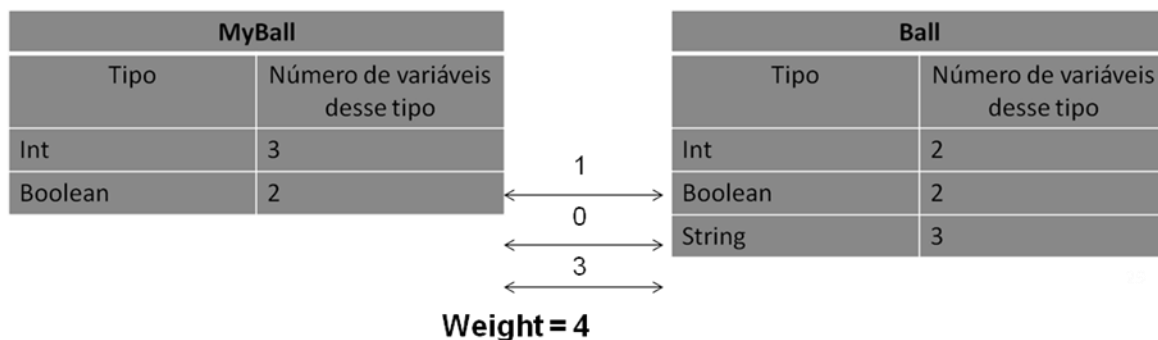
As listas são comparadas:

Aluno: (“array”, “list”)

Modelo: (“array”)

Neste caso o par [“*Ball*”, “*MyBall*”] será incluído como um possível par de classes similares pois as origens iguais (“array”, “array”) foram encontradas.

Em seguida, para cada uma das combinações de pares de classes possíveis são realizadas comparações entre suas listas de tipos, como mostra o exemplo entre as classes “*Ball*” e “*MyBall*” da Figura 5.6.

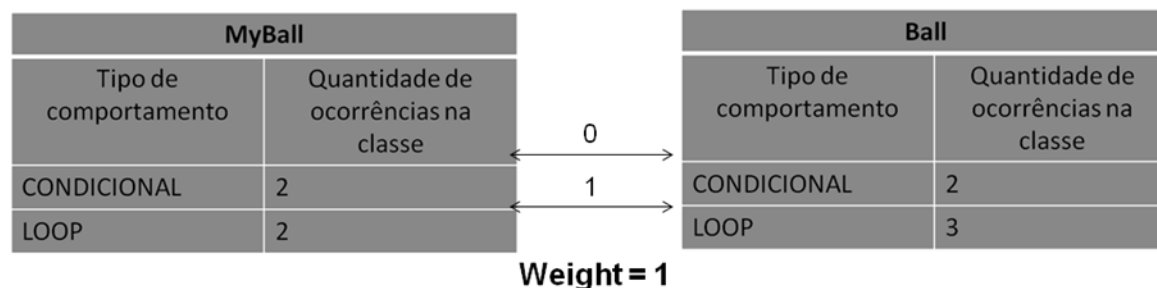


**Figura 5.7:** Exemplo de comparação entre as variáveis das classes “*MyBall*” e “*Ball*”

Cada elemento da lista contém dois valores, um tipo e o número de vezes que foram declaradas variáveis daquele tipo na classe. Por exemplo, na Figura 5.6, a lista da classe *"MyBall"* indica que existem 3 (três) variáveis do tipo *"int"* e 2 (duas) variáveis do tipo *"boolean"*. O algoritmo compara cada par do mesmo tipo e calcula a diferença entre esses valores. Em seguida, o peso de similaridade (atributo *"weight"*) recebe a soma dos resultados. O par com o menor peso de similaridade é escolhido como o par de classes mais similar.

A comparação entre a lista de tipos não é totalmente precisa. Os resultados são próximos da realidade, mas não inteiramente corretos. Então, com o objetivo de obter um resultado mais apurado, na terceira fase do algoritmo é realizada outra comparação, que consiste em comparar as listas de comportamentos de cada par de classes.

Para comparar a lista de comportamentos de cada par de classes, o algoritmo gera a lista de comportamentos contendo o tipo do comportamento (do tipo *"assignment"*, *"conditional"* ou *"loop"*) e o valor de cada tipo de comportamento (o valor de cada tipo diz respeito à quantidade de vezes que aquele comportamento ocorre na classe do programa). O algoritmo compara cada par do mesmo tipo de comportamento e em seguida o par contendo o menor peso de similaridade é classificado como o mais similar. O exemplo da comparação de lista de comportamentos pode ser visto na Figura 5.7.



**Figura 5.8: Exemplo de comparação entre os comportamentos das classes *"MyBall"* e *"Ball"***

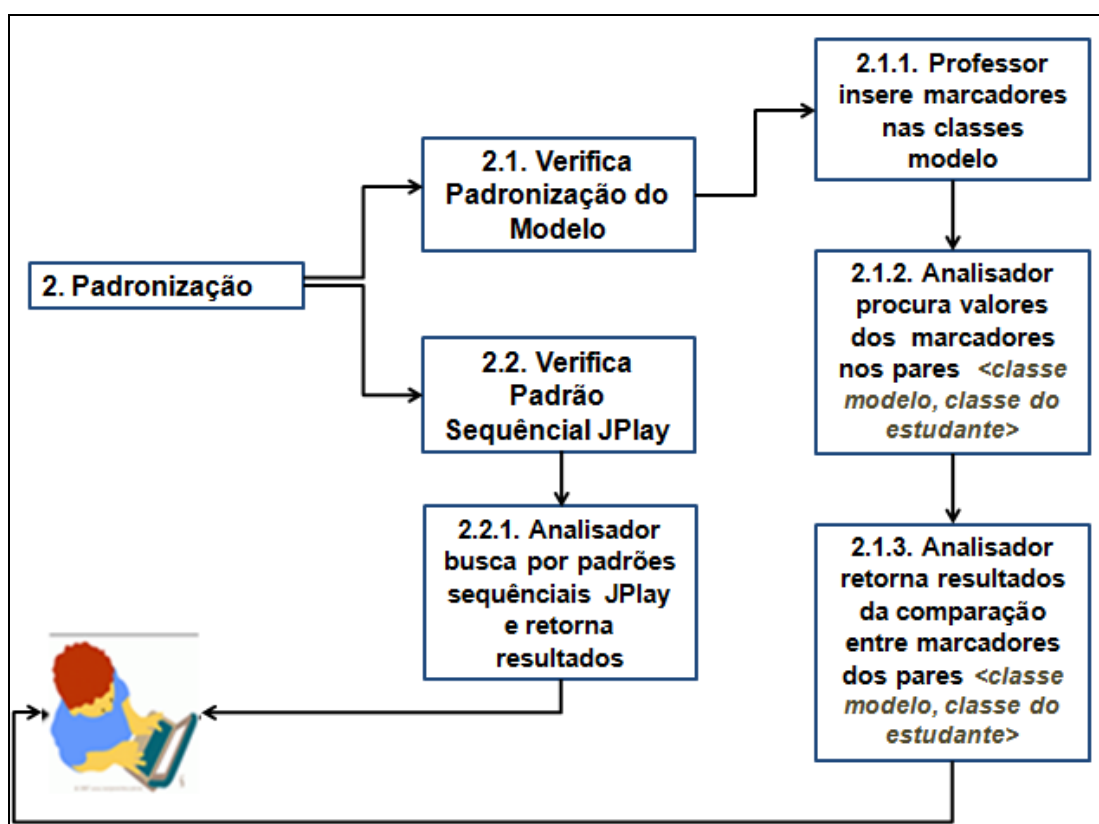
Nesse ponto, todos os pares de classes similares foram definidos e os pares podem ser comparados entre si.

Com o objetivo de comparar cada par de classes, a próxima etapa do algoritmo, conforme ilustrado na Figura 5.1, é verificar se as classes do programa do aluno estão padronizadas de acordo com as classes do programa modelo e, logo em seguida, classificar os pares de variáveis de cada par de classes.

## 5.4 A PADRONIZAÇÃO DO PROGRAMA MODELO

Com o objetivo de padronizar o programa do estudante de acordo com o programa modelo (programa do professor), foi proposto o uso de marcadores específicos em cada classe do programa modelo. Os marcadores devem ser declarados pelo próprio professor no início de cada arquivo "java", antes da declaração da classe do programa. Cada marcador representa um valor de um comportamento que a classe do aluno deve seguir.

O analisador mapeia cada uma das classes do programa do aluno, calculando o valor de cada um dos marcadores. Os pares de classes selecionados *<classe do programa do aluno, classe do programa modelo>* são comparados quanto aos valores dos marcadores, conforme mostra a Figura 5.9, na fase 2.1.



**Figura 5.9: Etapas do processo de padronização**

Cada marcador tem associado a ele um comentário, o comentário também é preenchido pelo próprio professor e corresponde a uma explicação ao aluno sobre a padronização daquela classe do programa, conforme mostra a fase 2.1.1, na Figura 5.9. Se existirem diferenças entre os valores dos marcadores no par, o comentário será emitido para o aluno (fases 2.1.2 e 2.1.3 na Figura 5.9).



Quanto mais padronizado de acordo com o programa modelo o programa do aluno estiver, mais preciso será o resultado do analisador. Programas fora da padronização podem gerar resultados incorretos. Alguns dos mais importantes marcadores são:

- *Inheritance*: Identifica uma superclasse se a classe estende de outra classe.
- *Constructor*: Indica que o construtor da classe deve ser declarado.
- *Movex*: Indica que deve ser declarado um método para mover o objeto no eixo “x”.
- *Movey*: Indica que deve ser declarado um método para mover o objeto no eixo “y”.
- *Keyboard*: Indica que o controle do teclado deve ser utilizado na classe.

- *Method*: Indica que a quantidade de métodos da classe do aluno deve ser igual a da classe do professor. O analisador calcula a quantidade de métodos da classe do professor, a quantidade de métodos da classe do aluno e depois realiza a comparação.

- *Mouse*: Indica que o controle do mouse deve ser utilizado na classe.
- *ObjectGame*: Identifica instancias de objetos do jogo na classe.
- *Main*: Identifica um método “*main*” na classe.
- *Gameloop*: Identifica um gameloop na classe.

O analisador encontra os marcadores no programa modelo e procura pelo valor de cada um dos marcadores correspondentes no programa do aluno. A busca dos valores dos marcadores, no programa do aluno, acontece por meio de consultas de palavras-chaves no código. Por exemplo, para procurar o valor do marcador “*movex*”, o analisador realiza uma busca pela palavra-chave “*this.x*” (atributo usado pelo JPlay) ou “*movex*” (método usado no JPlay), se estas palavras-chaves não forem encontradas no código significa que a implementação do movimento no eixo “x” não foi realizada. No caso do marcador “*Keyboard*”, o analisador procura, no código-fonte do programa do aluno, a palavra-chave “*keydown*” (método usado pelo JPlay). Para procurar o valor do marcador “*Mouse*”, o analisador procura a palavra-chave “*getMouse*” e assim por diante.

O marcador “*comment*” é usado de duas maneiras diferentes. Ele pode guardar uma mensagem sobre o comportamento geral da classe ou pode guardar uma mensagem sobre um comportamento específico.

O marcador “*comment*” usado para descrever o comportamento geral da classe é o primeiro marcador a ser declarado. Ele guarda uma mensagem que explica o comportamento geral da classe. O código abaixo ilustra como funciona o marcador:

```
/**
 * @comment 1.Uma bola a mais deve ser inserida na lista de bolas toda vez q o
 botao LEFT do mouse for pressionado. 2. Modifique o codigo de maneira que, so seja
 incluida uma nova bola na lista de bolas, se a posicao selecionada for uma posicao
 vazia. Senao a inclusao nao deve ser permitida e a mensagem AREA JA PREENCHIDA
 deve ser apresentada.
```

Os marcadores seguintes encontrados do tipo *"comment"* estão sempre associados a um outro marcador e guardam uma mensagem sobre um comportamento específico relacionado a este marcador. Segue exemplo no trecho de código abaixo:

```
/*
....
* @keyboard true
* @comment Você deve checar se a tecla "ESC" foi pressionada. Se a tecla "ESC" foi
pressionada, encerre o programa (DICA: você deve usar o metodo
"keyboard1.keydown()" do JPlay)
* @mouse true
* @comment Você deve checar se o botão LEFT do mouse foi pressionado. (DICA: Você
deve usar o metodo "mouse1.isLeftButtonPressed()" do JPlay)
```

No trecho de código acima, o primeiro marcador *"comment"* está associado ao marcador *"keyboard"* e o segundo marcador *"comment"* está associado ao marcador *"mouse"*.

O exemplo ilustrado na Figura 5.10 mostra os marcadores de uma classe chamada *"Ball"* pertencente ao programa modelo do professor. Os marcadores são declarados no início do arquivo, antes da declaração da classe *"Ball"*. De acordo com a Figura 5.10, o comportamento geral da classe *"Ball"* é descrito no primeiro marcador *"comment"* declarado, a classe *"Ball"* deve herdar da classe *"Sprite"* (marcador *"inheritance"*), deve possuir 3 métodos (marcador *"method"*), deve possuir um método construtor (marcador *"constructor"*), deve mover o objeto bola no eixo "x" (marcador *"movex"*) e deve mover o objeto bola no eixo "y" (marcador *"movey"*).

```

/**
 *
 * @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{
    /**

```

**Figura 5.10: Exemplo de marcadores aplicados na classe "Ball" do programa modelo**

## 5.5 O PADRÃO SEQUÊNCIAL JPLAY

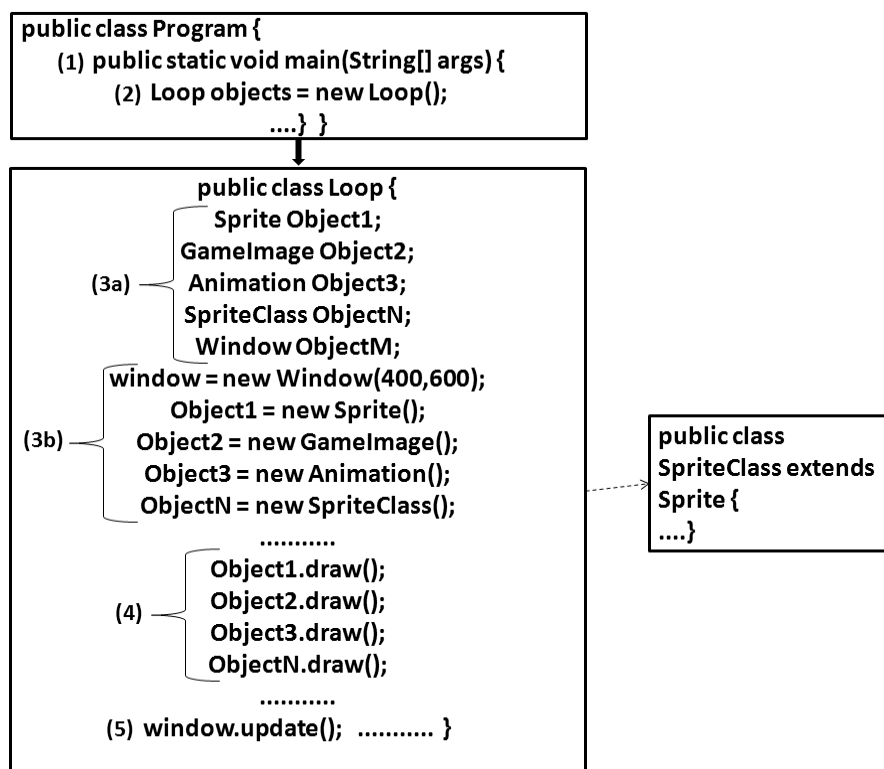
O padrão sequencial JPlay é uma sequência de código no programa que sempre acontece quando o programa está correto.

Baseado no padrão sequencial JPlay, o analisador mapeia as sequências de código padrão possíveis e procura por elas na fase de padronização, conforme mostra Figura 5.9, na fase 2.2.

O JPlay mostra um padrão de projeto de jogos típico: objetos de jogo, também chamados de *game objects*, que são inicialmente definidos; Um loop é iniciado (também chamado de *game loop*) e cada iteração corresponde a um frame do jogo sendo produzido. Dentro do loop, todos os *game objects* são atualizados com sua correspondente lógica (tais como algoritmos de inteligência artificial (IA), algoritmos de física ou mesmo de uma sequência de interface de usuário).

Finalmente, todos os elementos do jogo são desenhados na janela. Através do mapeamento detalhado do padrão de sequência do jogo é possível definir a ordem semântica em que devem acontecer algumas atividades, independente do objetivo particular do jogo. Assim, neste trabalho foi realizado o mapeamento da sequência típica de atividades do JPlay com objetivo de identificar a ordem sequencial de atividades que devem acontecer no programa e, caso esta ordem não aconteça, isto indica a possibilidade de uma ocorrência de erro semântico. Uma sequência típica de atividades do framework JPlay é ilustrada na Figura 5.11:

- Um método *main* deve ser definido na inicialização do programa (1);
- No corpo do método *main* objetos são instanciados, sendo que um desses objetos deve conter um *gameloop* (loop infinito). Encontrar a classe que contém o loop infinito significa encontrar o loop de execução do programa (2).
- Na classe que contém o *gameloop* (loop infinito) objetos são declarados (3a). Neste ponto é possível checar se todos os objetos que foram declarados foram também instanciados (3b).
- Objetos declarados como *Sprites*, *Animations* e *GameImages* devem ser desenhados na janela do jogo; neste ponto é necessário verificar se todos os objetos deste tipo foram desenhados (4). Para desenhar objetos na janela do jogo, eles devem chamar o método *draw* usado no JPlay.
- Um objeto declarado do tipo *Window* deve ser atualizado na janela do jogo; neste ponto é necessário verificar se o objeto *Window* foi atualizado (5). Para atualizar os objetos da janela, eles devem chamar o método *update* usado no JPlay.



**Figura 5.11: Padrão Sequencial JPlay**

Este trabalho realiza o mapeamento dos códigos sequenciais padrões existentes na arquitetura do JPlay, conforme visto no Capítulo 3. Baseado neste mapeamento, o analisador realiza a busca por estes padrões sequenciais no código do estudante (ver Figura 5.9, fase 2.2.1). Caso ele encontre, no código, sequências diferentes das mapeadas, mensagens serão enviadas indicando possíveis erros do código relacionados ao padrão sequencial JPlay. Por exemplo, se um objeto tipo *Sprite* for definido, o analisador vai procurar pela chamada do método *draw()* deste objeto, indicando que o objeto foi desenhado na janela do jogo. Se a chamada não for encontrada, o analisador informa que aquele objeto deveria ter sido desenhado na janela do jogo, no entanto isto não aconteceu.

## 5.6 CLASSIFICAÇÃO DE PARES DE VARIÁVEIS E COMPARAÇÃO DE ÁRVORES DE COMPORTAMENTO

Depois de definir os pares de classes similares, variáveis do mesmo tipo em cada uma das classes do par devem ser comparadas de acordo com seus comportamentos e com o objetivo de classificar os pares de variáveis mais similares. O algoritmo compara as variáveis do mesmo tipo usando uma lista de comportamentos contendo o comportamento e a

quantidade de vezes que este comportamento ocorreu relacionado àquela variável. O algoritmo então compara cada par de variáveis de cada tipo e calcula a diferença entre esses valores. Em seguida, o peso de similaridade (atributo “*weight*”) recebe a soma dos resultados. O par com o menor peso de similaridade é escolhido como o par mais similar de variáveis.

Depois disso, os pares de variáveis similares serão analisados. Para realizar a comparação entre o par de variáveis e apontar as diferenças encontradas, foi proposta uma estrutura de árvore chamada árvore de comportamento. Esta estrutura contém todos os comportamentos de uma variável. Os comportamentos são definidos de acordo com 3 (três) tipos:

- *Assignment*: É identificado quando a variável está envolvida em um comando de atribuição na classe. No exemplo, o trecho de código mostra um comando de atribuição associado à variável “*cont*”: *cont := 0*;
- *Conditional*: É identificado quando a variável está envolvida em um comando condicional na classe. No exemplo, o trecho de código mostra um comando condicional associado a variável “*cont*”: *if (cont > -1)*.
- *Loop*: É identificado quando a variável está envolvida em um comando de repetição na classe. No exemplo, o trecho de código mostra um comando de repetição associado à variável “*cont*”: *while (cont > -1)*.

O exemplo da Figura 5.12 mostra um comportamento do tipo “*assignment*” da variável “*left*” no construtor da classe “*Ball*”.

```
public Ball(String Imagem){
    super(Imagem);
    /*
    Inicialize o atributo que controla o movimento inicial da bola para
    esquerda ou para a direita no construtor da classe
    */
    left=true;
```

**Figura 5.12: Exemplo de comportamento “*assignment*” da variável “*left*” da classe “*Ball*” do programa modelo**

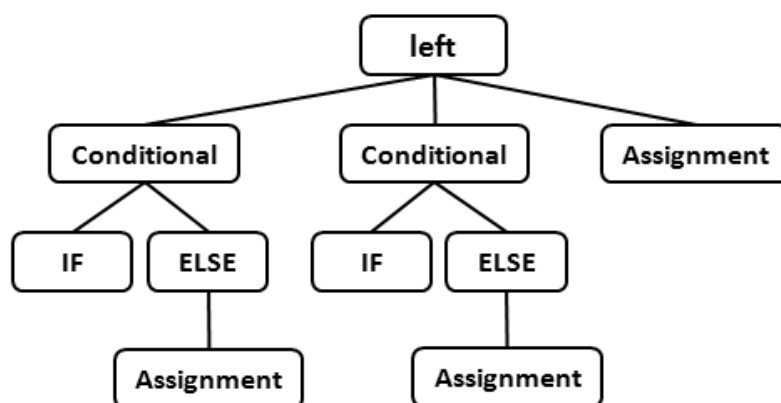
O analisador compara os comportamentos dos pares de variáveis similares <variável da classe do estudante, variável da classe do aluno> e caso diferenças entre as variáveis dos pares sejam encontradas é possível que seja pelo fato de haver um erro no programa do

estudante. Assim, através das diferenças encontradas, o estudante pode localizar a causa de possíveis erros em seu programa.

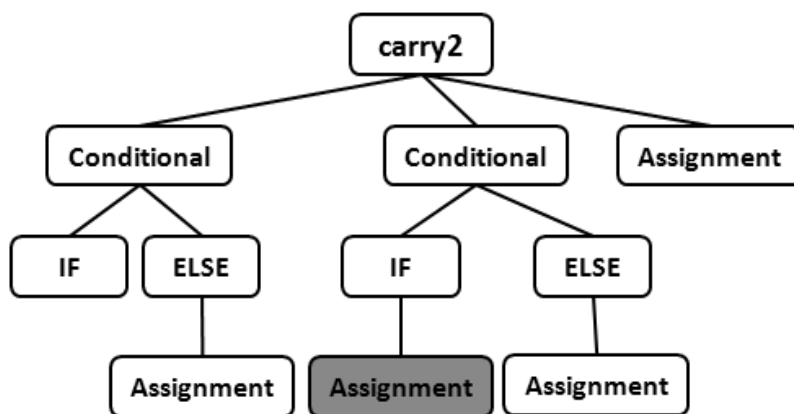
Quando diferenças são encontradas na comparação das variáveis, o analisador faz sugestões sobre o(s) comportamento(s) diferente(s) encontrado(s) no programa do estudante, indicando um possível erro semântico. As sugestões são definidas como comentários no código-fonte do programa modelo. Cada comentário deve ser editado pelo professor imediatamente antes do seu comportamento associado, como mostra o exemplo da Figura 5.12. Na Figura 5.12, se o comportamento do tipo *"assignment"* (associado a variável *"left"*) do programa do estudante não encontrar um comportamento similar na árvore de comportamento de sua variável par na classe do programa modelo, o seguinte comentário associado é capturado e sugerido ao estudante:

- *"Inicialize o atributo que controla o movimento inicial da bola para esquerda ou para a direita no construtor da classe."*

Os exemplos da Figura 5.13 e da Figura 5.14 mostram, respectivamente, a árvore de comportamento da variável *"left"* da classe *"Ball"* do programa modelo e a árvore de comportamento da sua variável similar *"carry2"* no programa do estudante. É possível verificar neste caso uma diferença encontrada, que corresponde a um comportamento *"assignment"* a mais.



**Figura 5.13:** Árvore de comportamento da variável *"left"* da classe *"Ball"* do programa modelo



**Figura 5.14: Árvore de comportamento da variável “carry2” da classe “MyBall” do programa do estudante**

Depois que o analisador encontra a diferença entre os comportamentos “assignment” das variáveis “left” (do programa modelo) e “carry2” (do programa do estudante), todos os comentários associados aos comportamentos “assignment” da variável “left” do programa modelo serão apresentados ao estudante:

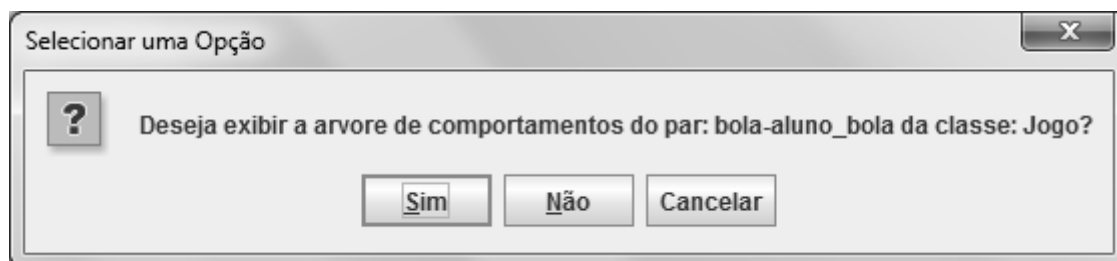
- “Inicialize o atributo que controla o movimento inicial da bola para a esquerda ou direita no construtor da classe.”
- “Se a posição da bola no eixo do x é menor que o limite mínimo da janela do jogo e ela está indo para a esquerda, modifique o sentido do movimento da bola para a direita.”
- “Se a posição da bola no eixo do x é maior do que o limite máximo da janela do jogo e ela está indo para a direita, modifique o sentido do movimento da bola para a esquerda. Considere neste caso a largura da bola. Exemplo: limite máximo – largura da bola.”

Neste exemplo, um comportamento a mais no programa do estudante não muda necessariamente o comportamento da classe “MyBall”. Neste caso ele poderia ou não influenciar no funcionamento correto do programa de acordo com o valor atribuído à variável “carry2”. É necessário observar que algumas variações de programação (formas diferentes de escrever um programa a fim solucionar o mesmo desafio) podem acarretar em diferenças que não necessariamente indicam erros, portanto o analisador não tem como apontar um resultado totalmente preciso, mas é capaz de dar sugestões próximas da solução correta baseado nas diferenças encontradas resultantes da comparação dos comportamentos.

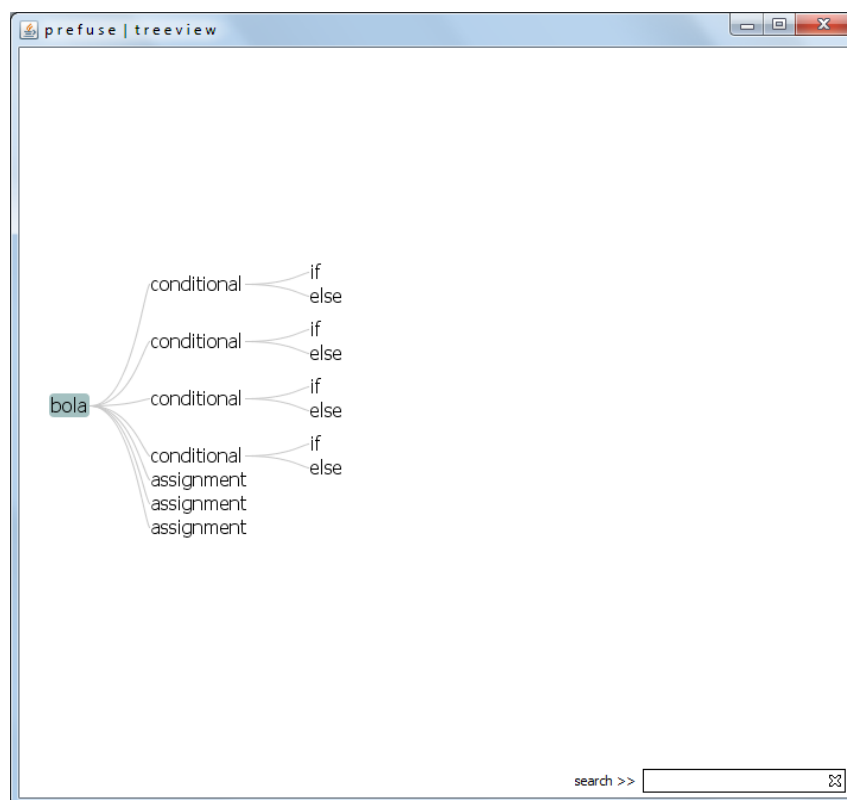
Além das sugestões apontadas, ao encontrar diferença(s) de comportamento entre os programas, o analisador pergunta se ele quer visualizar as árvores de comportamentos, conforme mostra o exemplo da Figura 5.15. Caso o usuário responda afirmativamente, o



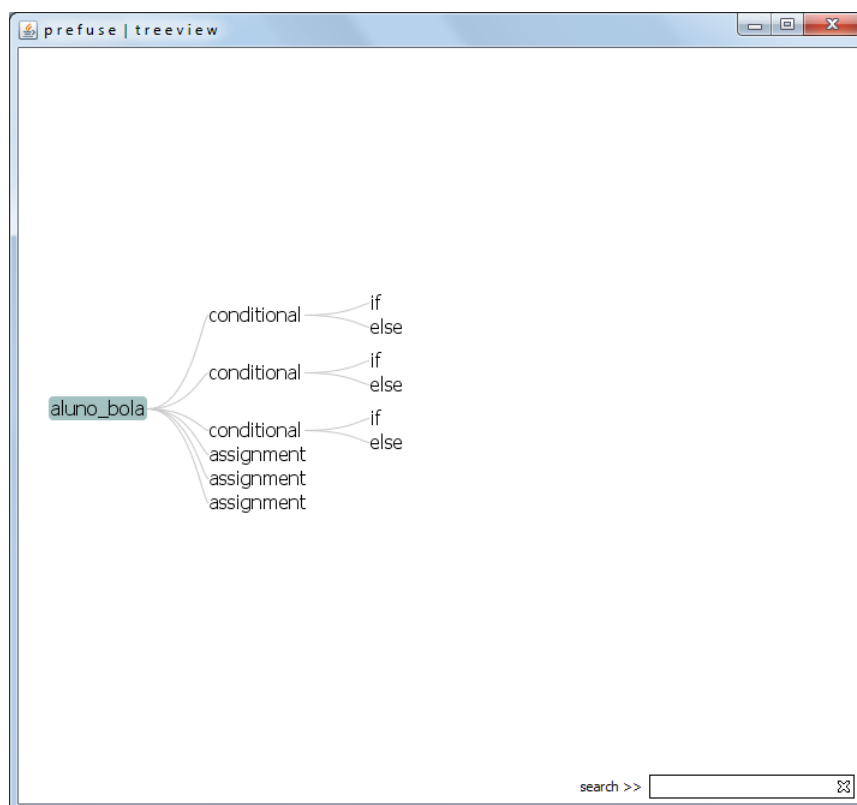
analisador constrói as árvores de comportamento do par de variáveis, sendo possível visualizar, através da comparação das árvores, a(s) diferença(s) encontradas, conforme mostra os exemplos da Figura 5.16 e Figura 5.17.



**Figura 5.15:** Janela do analisador perguntando ao usuário se deve mostrar as árvores de comportamento da variável “bola” do programa modelo e da variável “aluno\_bola” do programa do estudante



**Figura 5.16:** Árvore de comportamento da variável “bola” da classe “jogo” do programa do estudante



**Figura 5.17: Árvore de comportamento da variável “aluno\_bola” da classe “jogo” do programa modelo**

## 5.7 CONSIDERAÇÕES FINAIS

Observou-se que as variações de programação (formas diferentes de escrever um programa a fim de solucionar o mesmo desafio) podem acarretar em diferenças que não necessariamente indicam erros. Com o objetivo de minimizar a ocorrência de diferenças que não influenciam no funcionamento correto do programa, são repassadas aos alunos instruções sobre a padronização do programa modelo. Para tanto foi necessário propor uma padronização com o objetivo de conseguir mais precisão na comparação entre os programas. A padronização é definida através de marcadores que são incluídos pelo próprio professor no início de cada classe do programa modelo.

Desta forma, apesar do analisador não poder garantir um resultado totalmente correto, ele pode dar uma sugestão mais próxima da realidade baseada nas diferenças de comportamentos encontradas.

## CAPÍTULO 6 RESULTADOS

Neste capítulo são apresentados estudos de caso relacionados a 3 (três) programas desenvolvidos usando o framework JPlay. O primeiro programa foi desenvolvido por 8 (oito) estudantes do curso Médio Integrado em Informática do Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI) e foi aplicado como prova na disciplina "Tópicos Especiais em Desenvolvimento de Software". Esta primeira análise mostra os resultados do analisador ao verificar um programa que foi construído com uma única classe (classe "*main*") de projeto.

O segundo programa foi desenvolvido por 10 (dez) estudantes, entre eles, alunos do Médio Técnico Integrado em Informática, Técnico subsequente em Informática e Tecnólogo em Análise e Desenvolvimento de Sistemas. O jogo "*BrickBreak*" foi aplicado como atividade durante o minicurso "Desenvolvimento de jogos usando Java e framework JPlay". A segunda análise mostra os resultados do analisador ao verificar um programa constituído de 5 (cinco) classes de projeto ("*Barra*", "*Bloco*", "*Bola*", "*Jogo*" e "*Principal*").

O terceiro programa foi desenvolvido por 10 (dez) estudantes, alunos do Médio Técnico Integrado em Informática. O jogo "*Ping Pong*" foi aplicado como atividade durante a disciplina de "Tópicos Especiais em Desenvolvimento de Software". A terceira análise mostra os resultados do analisador ao verificar um programa constituído de 4 (quatro) classes de projeto ("*Barra*", "*Bola*", "*Jogo*" e "*Principal*").

Neste capítulo é apresentado na seção 6.1 a metodologia utilizada, logo depois, na seção 6.2 é apresentado o primeiro estudo de caso, que foi uma prova aplicada na disciplina de "Tópicos Especiais em Desenvolvimento de Software", na seção 6.3 são apresentadas as conclusões sobre o primeiro estudo de caso, na seção 6.4 é apresentado o segundo estudo de caso, que é o jogo "*BrickBreak*", mostrando as análises das classes "*Barra*", "*Bola*" e "*Jogo*" contidas no projeto do jogo, na seção 6.5 são apresentadas as conclusões sobre o segundo estudo de caso, na seção 6.6 é apresentado o terceiro estudo de caso, que é o jogo "*Ping Pong*", mostrando as análises das classes "*Barra*", "*Bola*" e "*Jogo*" contidas no projeto do jogo e, por fim, na seção 6.7 são apresentadas as conclusões sobre o terceiro estudo de caso.

## 6.1 METODOLOGIA

Baseado em uma abordagem metodológica do tipo qualitativa, a análise é caracterizada como um estudo de caso e tem o objetivo de executar a validação do sistema analisador semântico.

São realizados 3 (três) estudos de caso. Procura-se com esta escolha validar dois tipos de programas desenvolvidos usando Java e o framework JPlay:

O primeiro programa foi aplicado em uma prova, e se caracteriza por ser um desafio mais simples (não é um jogo) no sentido de que as funcionalidades necessárias a serem implementadas estão divididas em 2 (duas) questões de uma prova e elas podem ser desenvolvidas em uma única classe (classe “*main*”) do projeto Java. A prova foi aplicada como 1ª avaliação aos alunos da turma de "Tópicos Especiais em Desenvolvimento de Software" do Curso Médio Técnico Integrado em Informática. Participaram da análise um total de 8 (oito) provas.

O segundo programa proposto foi o jogo “*BrickBreak*”, sendo o mesmo aplicado durante o minicurso “Desenvolvimento de jogos usando Java e framework JPlay”. Participaram do minicurso alunos dos cursos Médio Técnico Integrado em Informática, Técnico Subsequente em Informática e Tecnólogo em Análise e Desenvolvimento de Sistemas do IFPI. O jogo envolve a necessidade de mais conhecimentos sobre o uso do paradigma de orientação a objetos, visto que é proposto a divisão do projeto em classes (“*Barra*”, “*Bloco*”, “*Bola*”, “*Jogo*” e “*Principal*”), e em cada uma das classes o jogo envolve a implementação de funcionalidades específicas. Participaram da análise um total de 10 (dez) programas.

O terceiro programa proposto foi o jogo “*Ping Pong*”, sendo o mesmo aplicado como atividade da disciplina, aos alunos da turma de "Tópicos Especiais em Desenvolvimento de Software" do Curso Médio Técnico Integrado em Informática. O jogo, assim como o “*BrickBreak*”, envolve a necessidade de mais conhecimentos sobre o uso do paradigma de orientação a objetos, visto que é proposto a divisão do projeto em classes (“*Barra*”, “*Bola*”, “*Jogo*” e “*Principal*”), e em cada uma das classes o jogo envolve a implementação de funcionalidades específicas. Participaram da análise um total de 10 (dez) programas.

Desta forma procura-se generalizar os resultados obtidos do analisador semântico aplicado a programas básicos usando Java e o framework JPlay. Um programa básico caracteriza-se, neste caso, por envolver poucas funcionalidades (o programa não é necessariamente um jogo), sendo desenvolvido somente com o uso da classe principal

“*main*” (caso do primeiro programa) e para programas mais complexos (como no caso do segundo programa).

Segundo Wainer (2007), a validação de um sistema consiste no processo de julgar quão bem um sistema resolve o problema para o qual ele foi concebido. Considerando que o algoritmo utilizado no analisador semântico consiste em uma heurística, ou seja, baseia-se em aproximações de um resultado exato, o algoritmo vai procurar encontrar as melhores soluções possíveis, mas não é possível garantir, em todas as ocorrências de sua aplicação, soluções ótimas ou exatas. Assim, através dos estudos de caso procuramos responder a seguinte questão: O analisador é eficaz em apontar problemas oriundos das diferenças entre a comparação de dois programas (o programa do aluno e o programa modelo do professor)?

Para responder essa questão são verificados 2 (dois) tipos de resultados obtidos pelo analisador:

- Análise quanto à padronização, ela é estabelecida através de configuração de marcadores, definidos pelo professor no início de cada classe do programa.

- Análise quanto à detecção de diferenças de comportamentos entre as variáveis dos programas. A detecção de diferença de comportamentos pode gerar 2 (dois) tipos de resultados: diferenças nos comportamentos dos pares formados pelo analisador (formação das árvores de comportamentos) e comportamentos das variáveis que não conseguiram formar pares, sendo que isto acontece no caso do analisador não encontrar nenhuma variável no programa do aluno que seja do mesmo tipo de uma variável que se encontra no programa do professor.

Os resultados da análise são apresentados em duas etapas: inicialmente, uma análise geral, onde são verificados os dados quantitativos dos resultados. Logo em seguida, uma análise mais detalhada, onde é realizado um estudo mais específico sobre os comentários sugeridos pelo analisador.

### **6.1.1 MATRIZ DE CONFUSÃO**

A matriz de confusão é uma tabela específica para avaliar o desempenho de um algoritmo ou método de classificação. As colunas da matriz representam classes previstas, enquanto as linhas representam as classes reais dos elementos analisados (HASTIE et al., 2009).

As células da matriz de confusão que correspondem ao cruzamento das informações das classes previstas e das classes reais representam a quantidade de positivo-positivos (PP), negativo-negativos (NN), falso-negativos (FN) e falso-positivos (FP).

Na Tabela 6.1, a matriz de confusão deste trabalho é construída a partir da observação dos comentários emitidos pelo analisador sobre uma determinada classe de programa. Os comentários são sugestões que tratam dos possíveis problemas encontrados no programa, são associados aos comportamentos e inseridos pelo próprio professor no programa modelo. De acordo com a matriz define-se:

PP: Quando a classe de programa não apresenta comentários e o comportamento da classe está realmente correto.

NN: Quando um comentário detecta um erro de comportamento da classe e realmente existe um problema real no comportamento.

FP: Quando a classe de programa não apresenta comentários, mas o comportamento da classe está incorreto.

FN: Quando um comentário detecta um erro de comportamento da classe, no entanto não existe um problema real no comportamento.

**Tabela 6.1: MATRIZ DE CONFUSÃO**

		CLASSES PREVISTAS	
		<b>Correto</b> <i>(não gera impressão de comentários)</i>	<b>Incorreto</b> <i>(gera impressão de comentários)</i>
CLASSES REAIS	<b>Correto</b>	PP	FN
	<b>Incorreto</b>	FP	NN

Um programa dado como "correto" não apresenta diferenças na comparação com o programa modelo, conseqüentemente não gera nenhum comentário ao aluno. Este resultado caracteriza-se como PP (positivo-positivo) ou FP (falso-positivo).

Um programa dado como do tipo "incorreto" apresenta diferenças na comparação com o programa modelo, conseqüentemente gera comentário (s) resultante (s) dessa (s) diferença (s). Este tipo de resultado pode ser classificado como NN(negativo-negativo) ou FN (falso-negativo).

A classificação de uma classe do tipo positivo-positivo versus falso-positivo, negativo-negativo versus falso-negativo será realizada através da observação do pesquisador (o próprio professor da disciplina/minicurso) baseado no real funcionamento do programa.

A seguir veremos os tipos de resultados obtidos a partir das análises.

### 6.1.2 SENSIBILIDADE, ESPECIFICIDADE, ACURÁCIA E PRECISÃO

Com o objetivo de validar o diagnóstico do analisador para um programa ou classe, é possível verificar os valores da sensibilidade, especificidade, eficiência, acurácia e precisão, baseado na classificação dos comentários impressos. Essa validação é utilizada para aferir o poder discriminativo do sistema analisador como bom ou não para uma determinada análise.

Sensibilidade e especificidade são medidas estatísticas de desempenho de técnicas de classificação calculadas utilizando informações presentes na matriz de confusão. São medidas muito utilizadas em problemas de classificação médicas. Neste contexto, a **sensibilidade** mede a proporção de casos positivos (diagnóstico de doença positivo) classificados corretamente como positivos e a **especificidade** mede a proporção de casos negativos (diagnóstico de doença negativo) classificados corretamente como negativos (HASTIE et al., 2009, p. 314). Neste trabalho, associando estas definições ao diagnóstico apresentado pelo analisador para uma determinada classe de programa, tem-se a sensibilidade como a capacidade de classificar como correta uma classe de programa realmente correta, e a especificidade como a capacidade de classificar como incorreta uma classe de programa realmente incorreta.

Acurácia e precisão são utilizados em contexto de medição. A **acurácia** refere-se ao grau de conformidade e correção de algo quando comparado com um valor verdadeiro ou absoluto, enquanto a **precisão** refere-se a um estado de exatidão estrita, ou seja, quanto algo é estritamente exato. Assim, a precisão de um experimento, objeto ou valor é uma medida da confiabilidade e consistência. A acurácia de um experimento, objeto ou valor é uma medida de quão próximos os resultados estão em relação ao valor verdadeiro ou valor aceito (DIFFEN, 2015).

Assim, a acurácia consiste no grau de proximidade de um valor medido ou calculado em relação a sua definição real ou sua referência padrão e a precisão representa o grau de variação dos resultados obtidos.

Neste trabalho, a acurácia significa o quanto o analisador se aproximou, em seus diagnósticos, dos resultados realmente reais.

A precisão, por sua vez, significa o quanto o analisador, em seus diagnósticos, identificou resultados realmente reais.

Podem ser calculados dois índices de precisão. Estes índices são conhecidos como preditividade positiva e preditividade negativa. A **preditividade positiva** indica a proporção de resultados positivos que realmente são positivos. A **preditividade negativa** indica a proporção de resultados negativos que realmente são negativos (WIKIHOW, 2015). Neste trabalho, são verificados os valores da preditividade positiva e negativa.

A Tabela 6.2 descreve algumas das medidas utilizadas para a discriminação da qualidade dos resultados apresentados pelo sistema analisador.

### 6.1.3 AS TRÊS HIPÓTESES PROPOSTAS

Nos casos onde são detectadas diferenças entre a comparação de programas e comentários são impressos como sugestões ao aluno, para medir a precisão do analisador verificamos os seguintes percentuais sobre os resultados obtidos:

Percentual de negativo-negativo (PNN): total de comentários negativo-negativo/total de comentários.

Percentual de falso-negativo (PFN): total de comentários falso-negativo/total de comentários.

Baseado nesses valores, uma avaliação eficaz do analisador vai indicar alguma das seguintes hipóteses:

*Hipótese 1: um percentual de negativo-negativo sempre maior que um percentual de falso-negativo ( $PNN > PFN$ ).*

Para os casos do tipo positivo-positivo e negativo-positivo, quando diferenças não são detectadas pelo analisador e, portanto, não existem comentários, é considerada, como resultado, a observação do pesquisador sobre o funcionamento/execução do programa (aonde o funcionamento/execução do programa pode ser considerado pelo observador como correto ou incorreto). Assim, para medir a precisão do programa, realizamos as seguintes medidas:

Percentual de positivo-positivo (PPP): total de programas positivo-positivo/total de programas de comportamento correto.

Percentual de falso-positivo (PFP): total de programas falso-positivo/total de programas de comportamento correto.

Baseado nesses valores uma avaliação eficaz do analisador vai indicar a seguinte hipótese, para o caso de valores PPP e PFP diferentes de 0 (zero):



*Hipótese 2: um percentual de positivo-positivo sempre maior que um percentual de falso-positivo ( $PPP > PFP$ ).*

Para os casos onde não existirem ocorrências de positivo-positivo ( $PPP=0$ ), uma avaliação eficaz do analisador indicará a seguinte hipótese:

*Hipótese 3: percentual positivo-positivo igual a percentual falso-positivo ( $PPP=PFP$ ).*

**Tabela 6.2: MEDIDAS PARA AFERIR A QUALIDADE DO SISTEMA ANALISADOR**

	Significado	Fórmula
<b>Sensibilidade</b>	capacidade de classificar como correta uma classe de programa realmente correta.	$\text{Sensibilidade} = \frac{PP}{PP + FN}$
<b>Especificidade</b>	capacidade de classificar como incorreta uma classe de programa realmente incorreta.	$\text{Especificidade} = \frac{NN}{NN + FP}$
<b>Acurácia</b>	quanto o analisador se aproximou, em seus diagnósticos, de resultados realmente reais.	$\text{Acurácia} = \frac{PP + NN}{PP + FP + NN + FN}$
<b>Preditividade Positiva</b>	indica a proporção de resultados positivos que realmente são positivos.	$\text{Preditividade Positiva} = \frac{PP}{PP + FP}$
<b>Preditividade Negativa</b>	indica a proporção de resultados negativos que realmente são negativos.	$\text{Preditividade Negativa} = \frac{NN}{NN + FN}$

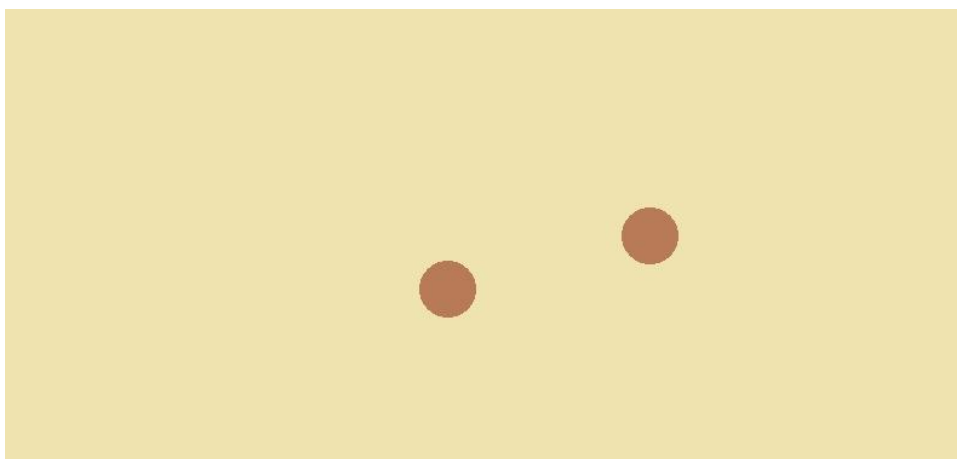
## 6.2 PRIMEIRO ESTUDO DE CASO: PROVA APLICADA NA DISCIPLINA “TÓPICOS ESPECIAIS EM DESENVOLVIMENTO DE SOFTWARE”

O primeiro estudo de caso se concentrou em códigos de programas que eram formados por uma única classe (todos os códigos continham somente a classe “main” do projeto).

Este cenário foi escolhido porque ele corresponde ao paradigma de aprendizagem aplicado aos estudantes daquela turma.

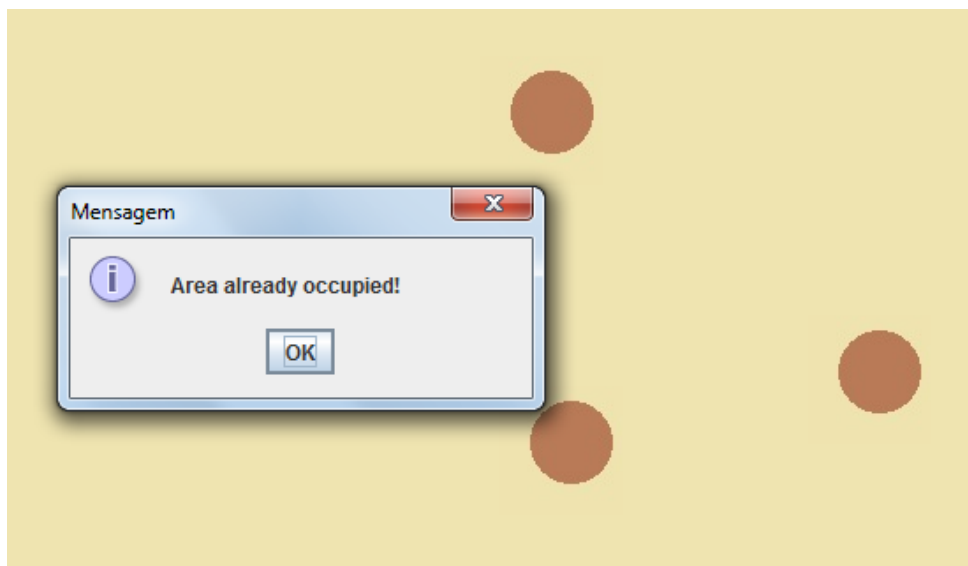
O programa foi aplicado como prova prática para os estudantes da disciplina de "Tópicos Especiais em Desenvolvimento de Software". Os alunos tiveram 2 (duas) horas e 50 (cinquenta) minutos (referentes a 3 aulas de 50 minutos) para responder a prova. A prova foi realizada em laboratório de informática com 24 (vinte e quatro) computadores. O enunciado da prova (descrito no ANEXO A), foi dividido em 2 (duas) questões. A primeira questão foi dividida em 2 (dois) itens:

- Questão 1.1: Quando o usuário pressionar o botão “*LEFT*” do mouse a bola deve ser desenhada na área da janela apontada pelo mouse e adicionada na lista de bolas, conforme mostra a Figura 6.1.
- Questão 1.2: Quando o usuário pressionar a tecla “*ESC*”, o programa deve ser encerrado.



**Figura 6.1:** Quando o usuário pressionar o botão “*LEFT*” do mouse, as bolas devem ser desenhadas na janela do jogo e adicionadas na lista de bolas.

Na segunda questão do teste o desafio foi verificar se a área onde o usuário pressionou o botão “*LEFT*” do mouse já está ocupada por uma bola ou não. Se a área já estiver ocupada, a mensagem com a informação “*Área já ocupada!*” deve aparecer, como mostra a Figura 6.2. Se a área não estiver ocupada, a nova bola deve ser adicionada na lista de bolas e desenhada na janela do jogo na posição corrente do mouse.



**Figura 6.2: Mensagem na janela do jogo informando que já existem bolas na área apontada pelo botão do mouse**

### **6.2.1 RESULTADOS OBTIDOS APARTIR DA MEDIÇÃO DE SENSIBILIDADE, ESPECIFICIDADE, ACURÁCIA E PRECISÃO**

Foram verificados dois tipos de resultados: o primeiro baseado por unidade de programa analisado, assim são verificadas todos os comentários que ocorreram durante a análise e seus respectivos tipos (Negativo-Negativo, Falso-Negativo) ou, no caso do programa não apresentar comentários, é verificada a análise do observador sobre o programa (Falso-Positivo, Positivo-Positivo). Então o programa é classificado de acordo com o tipo que possui o maior peso de classificação. O maior peso de classificação é de um comentário do tipo Negativo-Negativo, seguido pelos Falso-Negativo, Falso-Positivo e Positivo-Positivo. Assim, mesmo que o programa possua vários comentários de tipos diferentes, ele será classificado com o tipo de maior peso. O objetivo é considerar que, se o programa apresenta pelo menos um comentário incorreto, pode-se concluir que todo o programa será classificado como de um tipo incorreto.

O segundo tipo de resultado é baseado na quantidade de comentários apresentados por cada programa, onde a classificação de tipo de cada um dos comentários é quantificada. Neste caso, é importante observar que para calcular os resultados da análise serão consideradas as quantidades de comentários Negativo-Negativo e Falso-Negativo, mas como os resultados do tipo Positivo-Positivo e Falso-Positivo não resultam em comentários, nestes dois casos, serão considerados uma ocorrência do tipo Positivo-Positivo ou uma ocorrência do tipo Falso-

Positivo. As Tabelas 6.3 e 6.4 apresentam os resultados das medidas de eficiência obtidas através dos dois resultados.

**Tabela 6.3: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE PROGRAMAS ANALISADOS PARA O PRIMEIRO ESTUDO DE CASO**

<i>Medidas de eficácia do sistema analisador de acordo com o total de programas analisados</i>			
<i>Classe "main" do projeto</i>	<i>Quantidades e Medidas Obtidas</i>	<i>De acordo com a Padronização do Modelo</i>	<i>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</i>
	<i>NN</i>	2	8
	<i>FN</i>	0	0
	<i>PP</i>	6	0
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	-
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	100%
	<i>Preditividade Positiva</i>	100%	-
	<i>Preditividade Negativa</i>	100%	100%

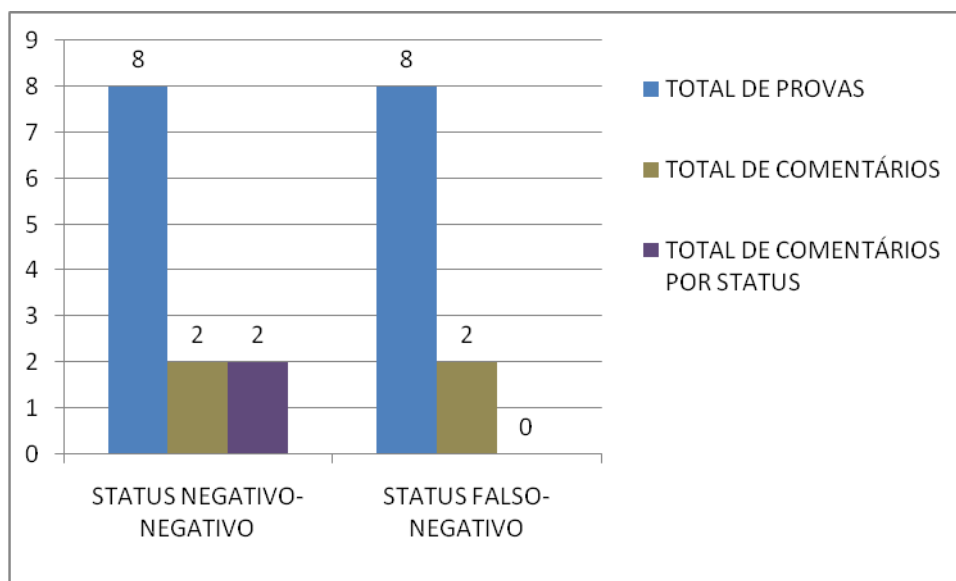
**Tabela 6.4: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O PRIMEIRO ESTUDO DE CASO**

<i>Medidas de eficácia do sistema analisador de acordo com o total de comentários analisados</i>			
<i>Classe "main" do projeto</i>	<i>Quantidades e Medidas Obtidas</i>	<i>De acordo com a Padronização do Modelo</i>	<i>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</i>
	<i>NN</i>	2	29
	<i>FN</i>	0	9
	<i>PP</i>	6	0
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	0%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	76%
	<i>Preditividade Positiva</i>	100%	-
	<i>Preditividade Negativa</i>	100%	76%

## 6.2.2 RESULTADOS OBTIDOS APARTIR DAS TRÊS HIPÓTESES PROPOSTAS

### 6.2.2.1 ANÁLISE GERAL QUANTO A PADRONIZAÇÃO DO MODELO

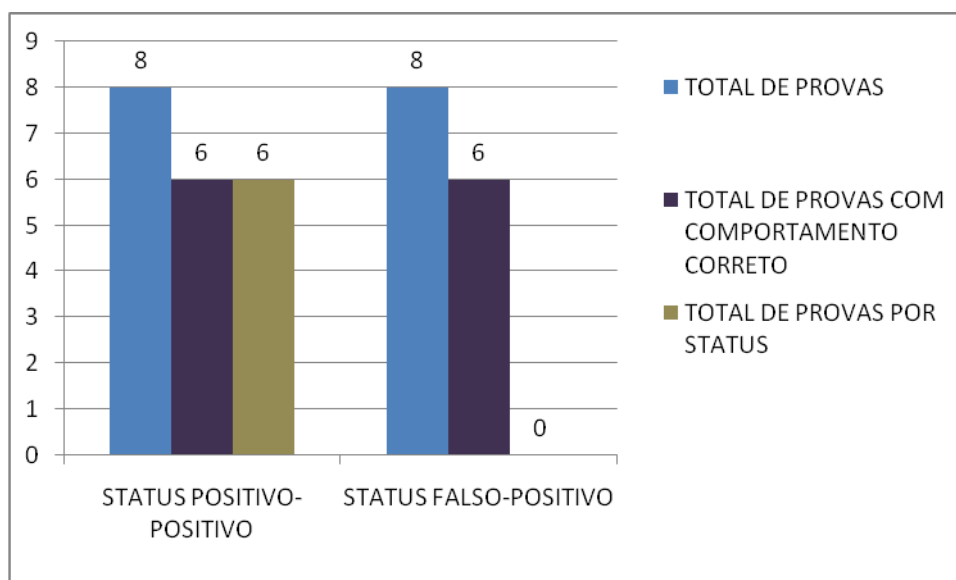
A análise constatou que, de acordo com a padronização do modelo, o percentual de comentários do tipo negativo-negativo (PNN) foi de 100% e o percentual de comentários do tipo falso-negativo (PFN) foi de 0%. Portanto, conclui-se que  $PNN > PFN$ , sendo possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.1.



**Gráfico 6.1: Distribuição de mensagens do tipo negativo-negativo e falso-negativo nas provas da turma de "Tópicos Especiais em Desenvolvimento de Software", de acordo com padronização do modelo**

Fonte: Pesquisa direta, 2015.

A análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo positivo-positivo (PPP) foi de 100% e o percentual de precisão do tipo falso-positivo (PFN) foi de 0%. Portanto, concluindo  $PPP > PFN$ , podemos aceitar a hipótese 2 como verdadeira. Ver Gráfico 6.2.



**Gráfico 6.2: Distribuição de mensagens do tipo positivo-positivo e falso-positivo nas provas da turma de "Tópicos Especiais em Desenvolvimento de Software", de acordo com padronização do modelo**

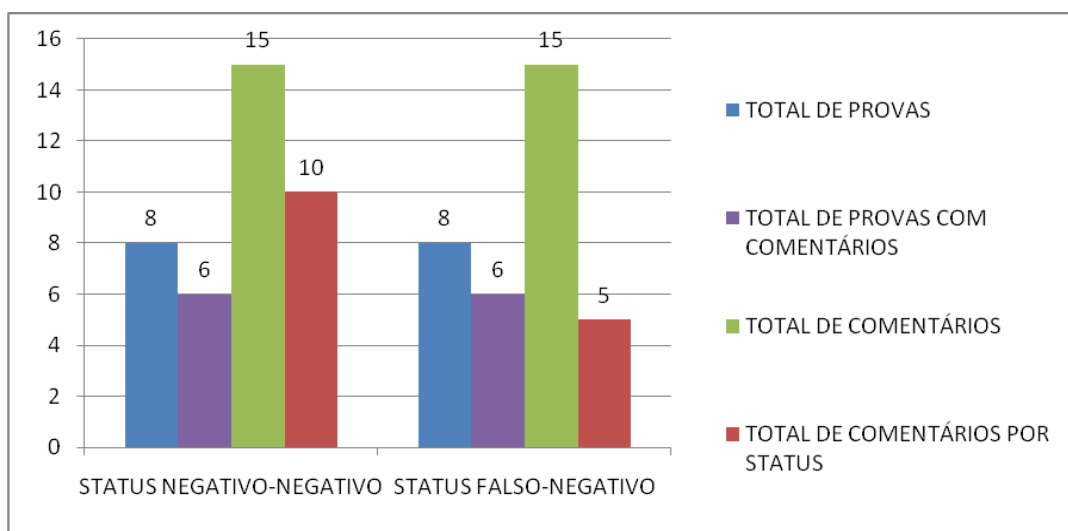
Fonte: Pesquisa direta, 2015.

Sendo PNN igual a 100% e PPP igual a 100%, pode-se concluir que o percentual de precisão neste caso possui 100% de eficácia.

#### 6.2.2.2 ANÁLISE GERAL QUANTO A DIFERENÇAS DE COMPORTAMENTOS

Na heurística proposta neste trabalho a diferença de comportamentos entre programas é resultado de dois tipos de análises: análise de acordo com a comparação de par de variáveis (árvores de comportamento) e análise de variáveis sem par. A seguir, seguem as duas análises:

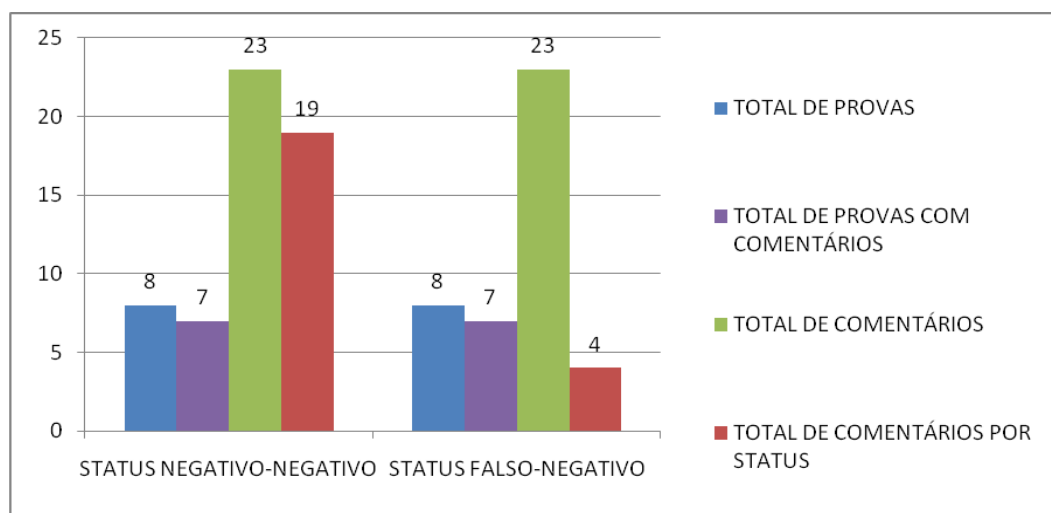
De acordo com a comparação de par de variáveis (árvores de comportamento), o percentual de comentários do tipo negativo-negativo (PNN) foi de 67% e o percentual de comentários do tipo falso-negativo (PFN) foi de 33%. Portanto, conclui-se que  $PNN > PFN$ , sendo possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.3.



**Gráfico 6.3: Distribuição de mensagens do tipo negativo-negativo e falso-negativo nas provas da turma de "Tópicos Especiais em Desenvolvimento de Software", de acordo com a comparação de par de variáveis**

Fonte: Pesquisa direta, 2015.

De acordo com a comparação de variáveis sem par, a análise constatou que, o percentual dos comentários do tipo negativo-negativo (PNN) foi de 83% e o percentual dos comentários do tipo falso-negativo (PFN) foi de 17%. Portanto, conclui-se que  $PNN > PFN$ , sendo possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.4.



**Gráfico 6.4: Distribuição de mensagens do tipo negativo-negativo e falso-negativo nas provas da turma de "Tópicos Especiais em Desenvolvimento de Software", de acordo com a comparação de variáveis sem par**

Fonte: Pesquisa direta, 2015.

De acordo com a diferença de comportamentos, reunindo os dois tipos de resultados (comparação de par de variáveis e variáveis sem par), todas as provas apresentaram diferenças de comportamento, portanto nenhuma apresentou resultado positivo-positivo ou falso-positivo. Constatamos, por meio de observação das provas, que realmente todas as provas apresentavam comportamento incorreto. Logo o percentual do tipo positivo-positivo (PPP) foi de 0% nesse caso. Em nenhuma das provas foram constatados comportamentos do tipo falso-positivo, portanto o percentual do tipo falso-positivo (PFP) foi de 0%. Portanto, conclui-se que  $PPP=PFP$ , podendo aceitar a hipótese 3 como verdadeira.

### 6.2.2.3 ANÁLISE DETALHADA QUANTO A PADRONIZAÇÃO DO MODELO

Conforme a Tabela 6.5, de acordo com a padronização do modelo, o analisador detectou que 25% dos alunos (estudantes 6 e 7) apresentaram uma diferença de padronização relacionada a Questão 1.2 (quando o usuário pressionar a tecla “ESC”, o programa deve ser encerrado). Ou seja, estes alunos não realizam este comportamento nos seus programas.

Todos os resultados desta análise são do tipo negativo-negativo, ou seja, apontam para um problema real de implementação do programa do aluno.

**Tabela 6.5: AVALIAÇÃO DOS COMENTÁRIOS DE ACORDO COM A PADRONIZAÇÃO**

Estudante	Avaliação de comentários de acordo com a Padronização do Modelo		
	Status do programa	Marcador relacionado ao comentário	Status do comentário
6	Incorreto	“Keyboard”	Tipo Negativo-Negativo
7	Incorreto	“Keyboard”	Tipo Negativo-Negativo

Os estudantes 6 e 7 não realizam o comportamento relacionado a Questão 1.2 da prova. A padronização foi estabelecida no programa modelo através do marcador “Keyboard” e seu comentário associado, conforme trecho de código abaixo:

```
.....
* @keyboard true
* @comment Você deve verificar se a tecla “ESC” foi pressionada. Se ela
foi pressionada, encerre o programa. (DICA: Você deve usar o método do
JPlay: keyboard1.keydown())
```



Os estudantes 6 e 7 não realizam a instanciação do teclado (palavra-chave “Keyboard”) e também não realizam o comportamento da Questão 1.2 (palavra-chave “Keydown”). Desta forma, o analisador encontra a diferença no marcador “Keyboard” e mostra a mensagem associada definida no marcador “Comment”:

- “Voce deve verificar se a tecla “ESC” foi pressionada. Se ela foi pressionada, encerre o programa. (DICA: Você deve usar o método do JPlay: `keyboard1.keydown()`)”

#### 6.2.2.4 ANÁLISE DETALHADA QUANTO A COMPARAÇÃO DE PAR DE VARIÁVEIS

Conforme a Tabela 6.6, o analisador detectou que 75% (alunos 1, 4, 5, 6, 7 e 8) obtiveram diferenças de comportamentos relacionados a pares de variáveis formados.

Entre os estudantes que apresentaram diferenças na comparação, aproximadamente 37% deles (total de 3 estudantes) apresentaram resultados do tipo falso-negativo: estudante 6 (comentários 6 e 8), estudante 7 (comentários 5 e 7) e estudante 8 (comentário 6).

**Tabela 6.6: AVALIAÇÃO DOS COMENTÁRIOS DE ACORDO COM A COMPARAÇÃO DOS PARES DE VARIÁVEIS**

Estudante	Avaliação dos comentários de acordo com a Comparação dos Pares de Variáveis		
	Status do programa	Identificador do comentário	Status do comentário
1	Incorreto	6	Tipo Negativo-Negativo
4 e 5	Incorreto	6	Tipo Negativo-Negativo
		8	Tipo Negativo-Negativo
		12	Tipo Negativo-Negativo
6	Incorreto	6	Tipo Falso-Negativo
		7	Tipo Negativo-Negativo
		8	Tipo Falso-Negativo
		9	Tipo Negativo-Negativo
		10	Tipo Negativo-Negativo

Estudante	Avaliação dos comentários de acordo com a Comparação dos Pares de Variáveis		
	Status do programa	Identificador do comentário	Status do comentário
		11	Tipo Negativo-Negativo
7	Incorreto	5	Tipo Falso-Negativo
		6	Tipo Negativo-Negativo
		7	Tipo Falso-Negativo
		11	Tipo Negativo-Negativo
8	Incorreto	6	Tipo Falso-Negativo

Considerando os comentários negativo-negativos como resultados que apontam para um problema real na implementação do código do estudante, as análises detalhadas que seguem vão se concentrar nos comentários falso-negativos por apontarem os problemas na precisão dos resultados:

- **Estudante 6 (Comentários falso-negativo 6):**

O comentário 6, no programa modelo, descreve um comportamento “*CONDITIONAL*” relacionado a uma variável do tipo “*mouse*”, conforme trecho de código do programa modelo abaixo:

```
/*
comment06: In the infinite loop check if the “LEFT” type button of the
mouse was pressed, then if it was pressed verify if it is possible to
create a new ball in the mouse current position.
*/
if (mouse.isLeftButtonPressed()){
.....
```

Como a diferença encontrada está no comportamento “*CONDITIONAL*” entre a variável do tipo “*Mouse*” do programa do aluno e seu par do tipo “*Mouse*” do programa modelo (par de variáveis <tipo “*Mouse*”, tipo “*Mouse*”>, o seguinte comentário relacionado a este comportamento, no programa modelo, será apresentado ao aluno:

- “comment06: No loop infinito verifique se o botão do tipo “LEFT” do mouse foi pressionado, então, se ele foi pressionado, adicione uma nova bola na posição corrente do mouse somente se a posição estiver vazia (a posição corrente não pode estar ocupada com outra bola).”

Para o caso do aluno 6, o comentário 6 é considerado falso-negativo, pois é causado por variações de programação (um comportamento “*CONDITIONAL*” a mais relacionado a variável do tipo “*Mouse*” no programa do estudante) que não influenciam no comportamento geral do programa.

- **Estudante 6 (Comentário falso-negativo 8):**

Para o caso do aluno 6, o comentário 8 é falso-negativo. Este comentário está relacionado a um comportamento “*LOOP*”, o aluno implementa dois comportamentos “*LOOP*” e o objetivo era, conforme solicitado na Questão 1.1, desenhar as bolas de uma lista de bolas. No entanto, no programa do professor, somente um comportamento “*LOOP*” é utilizado para solucionar o problema. Acontece, portanto, neste caso, uma variação na forma de programação, que não influencia no comportamento geral do programa. O falso-negativo poderia ter sido evitado caso o professor tivesse orientado de maneira mais específica que as bolas da lista de bolas deveriam ser desenhadas durante cada atualização da janela do jogo dentro do “*gameloop*”.

- **Estudante 7 (comentário falso-negativo 5):**

Somente o estudante 7 apresentou o comentário 5. Isto acontece porque a diferença é encontrada no par de variáveis do tipo “*Point*” (par de variáveis <tipo “*Point*”, tipo “*Point*”>. No programa do estudante, o nome da variável do tipo “*Point*” é “*pos*”. O estudante inicializa o valor da variável “*pos*” 2 (duas) vezes (gerando 2 comportamentos do tipo “*ASSIGNMENT*”), gerando uma ação desnecessária, pois a variável pode ser inicializada apenas uma vez. Abaixo está o trecho de código relacionado ao estudante 7:

```
Point pos;
pos = mouse.getPosition();
.....
if (mouse.isLeftButtonPressed()) {
pos = mouse.getPosition();
.....
```

No programa modelo, a variável par do tipo “*Point*” (também chamada de “*pos*”), é inicializada somente uma vez, conforme mostra o trecho de código abaixo:

```
/*
comment05:
No loop infinito atualize a atual posicao do mouse usando o metodo
“getPosition()” do JPlay
*/
pos = mouse.getPosition();
```

Logo, o par de variáveis <“pos”, “pos”> geram uma diferença no comportamento do tipo “ASSIGNMENT”. Portanto, o comentário relacionado é apresentado ao aluno:

- “comment05: No loop infinito atualize a atual posição do mouse usando o método “getPosition()” do JPlay.”

Assim, no caso do estudante 7, o comentário 5 é falso-negativo, pois é considerado uma variação de programação que não influencia no comportamento geral do programa. O problema poderia ser evitado se o aluno verificasse que existem atribuições de valores desnecessárias.

- **Estudante 7 (comentário falso-negativo 7):**

Ainda no caso do aluno 7, o comentário 7 foi causado pela formação de par da variável “jaexiste” do programa do professor com a variável “isOver” do programa do aluno (par de variáveis < “jaexiste”, “isOver”>). O aluno criou uma variável chamada “isOver” do tipo “boolean” e a utilizou no programa com o objetivo de verificar se a área apontada pelo mouse já está ocupada por uma bola ou não, conforme solicitado nas orientações do professor. No entanto a variável par “isOver” gera diferenças no comportamento “ASSIGNMENT”, indicando que ela recebeu mais atribuições que o necessário. Abaixo, as diferenças encontradas pelo analisador:

- *Lista Pesos Par<modelo,aluno>: <jaexiste , isOver> - Assignment:2 - Conditional:0 - Loop:0 - TOTAL: 2*

No código do professor, o comportamento “ASSIGNMENT” da variável “jaexiste” acontece nos seguintes trechos de código:

```
1  /*
2  comment07:
3  Se o botao LEFT do mouse foi pressionado, verifique se ja existe uma
   bola nessa posicao, no inicio considere que nao existe nenhuma bola
   nessa posicao (DICA: jaexiste = false)
4  */
5  jaexiste = false; // comportamento “assignment”
```

No caso específico do aluno 7, o comentário 7 é um falso-negativo, pois é considerado uma variação na forma de programação e não influencia no comportamento geral do programa. O falso-negativo poderia ter sido evitado caso o aluno não tivesse realizado

atribuições de valor desnecessárias a variável e considerasse, conforme a orientação da professor, o valor inicial igual a falso.

- **Estudante 8 (comentário falso-negativo 6):**

Para o caso do aluno 8 aconteceu o mesmo que no aluno 6: o comentário 6 é considerado falso-negativo, pois é causado por variações de programação (um comportamento “*CONDITIONAL*” a mais relacionado a variável do tipo “*Mouse*” no programa do estudante) que não influenciam no comportamento geral do programa.

#### 6.2.2.5 ANÁLISE DETALHADA DE VARIÁVEIS SEM PAR

Conforme a Tabela 6.7, o analisador detectou que aproximadamente 87% dos alunos (alunos 1, 2, 3, 4, 5, 6 e 8) obtiveram diferenças de comportamentos relacionados às variáveis sem pares (variáveis que não formaram pares).

Nessa análise, a maioria dos estudantes apresentaram os mesmos comentários (comentários 7, 9 e 10) do tipo negativo-negativo. Além dos comentários 7, 9 e 10, somente o estudante 1 apresentou o comentário 2.

**Tabela 6.7: AVALIAÇÃO DOS COMENTÁRIOS OBTIDOS A PARTIR DAS VARIÁVEIS SEM PAR**

Estudante	Avaliação dos comentários obtidos a partir das variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
1	Incorreto	2	Tipo Negativo-Negativo
		7	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		10	Tipo Negativo-Negativo
2	Incorreto	7	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		10	Tipo Negativo-Negativo
3, 4 and 5	Incorreto	5	Tipo Falso-Negativo
		7	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		10	Tipo Falso-Negativo
6	Incorreto	5	Tipo Falso-Negativo
8	Incorreto	7	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		10	Tipo Negativo-Negativo

Seguem análises de comentários do tipo negativo-negativos encontrados:

- **Estudantes 1, 2, 3, 4, 5, 8 (comentários negativo-negativos 7, 9 e 10):**

Conforme a Tabela 6.7, quase todos os alunos (com exceção do aluno 6), apresentaram os comentários 7, 9 e 10.

O comentário 7 está associado a um comportamento “*ASSIGNMENT*” da variável de nome “*jaexiste*” (do tipo “*boolean*”) do programa do professor. Da mesma forma, os comentários 9 e 10 estão associados a um comportamento “*CONDITIONAL*” da variável “*jaexiste*”.

Os comentários são apresentados porque os estudantes não definiram nenhuma variável do tipo “*boolean*” nos seus códigos de programa, logo a variável “*jaexiste*” (associada aos comentários no programa do professor) não formou par com nenhuma variável do programa do estudante. Os comentários relacionados apresentados ao estudante são:

- “*comment07: Se o botao LEFT do mouse foi pressionado, verifique se ja existe uma bola nessa posicao, no inicio considere que nao existe nenhuma bola nessa posicao (DICA: jaexiste = false)*”
- “*comment09: Se o botao LEFT do mouse foi pressionado e se nao existe nenhuma bola na posicao atual do mouse, crie uma nova bola nessa posicao e adicione a nova bola na lista. (DICA: if (jaexiste == false))*”
- “*comment10: Se o botao LEFT do mouse foi pressionado e se ja existe uma bola na posicao atual do mouse (nao eh possivel criar uma bola nova), mostre uma mensagem AREA JA PREENCHIDA!! (DICA: if (jaexiste == true))*”

Os comentários 7, 9 e 10 são considerados do tipo negativo-negativo, pois realmente apontam para um problema de implementação no programa dos estudantes (os estudantes não verificaram corretamente se a área apontada pelo mouse já estava ocupada por uma bola).

- **Estudantes 1 (comentário negativo-negativo 2):**

No caso do aluno 1, além dos comentários 7, 9 e 10, ele apresenta também o comentário 2. O comentário 2 está relacionado ao comportamento “*ASSIGNMENT*” da variável “*fundo*” (do tipo “*GameImage*”) do programa do professor. O comentário 2 é apresentado porque o aluno não criou a variável do tipo “*GameImage*”, por isso o analisador não conseguiu formar par (es) com nenhuma variável deste tipo no programa do aluno. Neste

caso, o programa do aluno não desenhou a imagem de fundo da janela. Logo, o comentário relacionado é apresentado ao aluno:

- *“comment02: No metodo main defina a imagem de fundo do jogo com a classe GameImage.”*

No entanto, a orientação para que fosse definida a imagem de fundo da janela foi repassada pelo professor através de trechos de código na prova. Portanto, o comentário 2 realmente aponta para um problema na implementação do aluno e é considerado do tipo negativo-negativo. O problema poderia ser resolvido se o aluno seguisse as orientações repassadas pelo professor na prova.

Sobre os comentários do tipo falso-negativos, somente o comentário 5 foi considerado falso-negativo, tendo ele sido resultado na análise dos estudantes 3, 4, 5 e 6. Segue análise do comentário 5 do tipo falso-negativo encontrado:

- **Estudantes 3, 4, 5 e 6 (comentário falso-negativo 5):**

Os alunos 3, 4, 5 e 6 apresentaram o comentário do tipo falso-negativo 5. O comentário está relacionado ao comportamento *“ASSIGNMENT”* da variável de nome *“pos”* (do tipo *“Point”*) do programa do professor. Neste caso, os alunos não criaram nenhuma variável do tipo *“Point”* e por isso o analisador não consegue formar par (es) com nenhuma variável deste tipo no programa do aluno. Logo, o comentário relacionado é apresentado ao aluno:

- *“comment05: No loop infinito atualize a atual posicao do mouse usando o metodo getPosition() do JPlay”*

O comportamento do programa, no entanto, não se modifica, pois os alunos usam a expressam *“mouse.getPosition().x”* e *“mouse.getPosition().y”* para efetuar operações semelhantes. Logo o comentário 5 é considerado falso-negativo.

Novamente o problema poderia ser resolvido se o aluno seguisse as orientações repassadas pelo professor na prova. Outra forma de solucionar o problema seria o professor verificar o resultado recorrente falso-negativo deste comentário e então optar por não comentar este comportamento, já que existem muitas variações de programação possíveis para o mesmo.

### 6.3 CONSIDERAÇÕES GERAIS SOBRE O PRIMEIRO ESTUDO DE CASO

A análise mediante a padronização do modelo não apresentou resultados do tipo falso-negativos. Todos os resultados foram do tipo negativo-negativo, significando que eles apontam para um problema real de implementação do programa do estudante. Logo, o percentual de acerto da análise de acordo com a padronização foi de 100%.

A análise que mais apresentou resultados falso-negativos (37 % dos estudantes) foi a análise de comparação de pares de variáveis (comparação de árvores de comportamentos). A análise mostra um total de 15 (quinze) comentários, sendo 5 do tipo falso-negativos. Portanto, o percentual de erro, de acordo com a comparação dos pares de variáveis, foi de aproximadamente 33% e o percentual de acerto foi de aproximadamente 66%. Os erros de análise, neste caso, foram decorrentes da falta de padronização dos programas dos estudantes de acordo com o programa modelo do professor. É possível concluir que a comparação de variáveis se mostra mais suscetível a erros de análise. Isto acontece porque os erros de análise decorrentes da comparação de variáveis são consequência principalmente das variações de programação e da falta de padronização do programa do aluno de acordo com o programa modelo do professor. Quanto mais padronizado o código do aluno estiver, menos erros de análise decorrentes de variações de programação serão apresentados. Portanto, a precisão de acerto da comparação de variáveis depende do estudante seguir corretamente as orientações de padronização repassadas pelo professor na atividade. A precisão de acerto, neste caso, também depende da forma como o professor padroniza o programa modelo, sendo importante que o professor identifique quais comportamentos precisam de padronização ou não. O professor deve verificar quais são os pontos do programa que são mais sujeitos a variações de programação e se estes pontos são comportamentos relevantes para a execução correta do programa do estudante. Em caso de serem comportamentos relevantes, o professor deve estabelecer orientações de padronização bem definidas para o aluno e comentar os comportamentos no código do programa modelo. Se não forem comportamentos relevantes, o professor pode optar por não definir padronização, bem como não comentar estes pontos do programa evitando impressão de comentários falso-negativos desnecessários.

Na análise de variáveis sem pares, 50% dos alunos apresentaram resultados falso-negativos, no entanto, todos os alunos apresentaram o mesmo comentário falso-negativo, o comentário 5. Em consequência do resultado recorrente falso-negativo deste comentário, podemos concluir que o professor poderia optar por não comentar este comportamento, posto que existem variações de programação possíveis para o mesmo. Esta análise demonstra um



nível mais elevado de falta de padronização (orientações repassadas pelo professor) do programa do aluno.

#### 6.4 SEGUNDO ESTUDO DE CASO: O JOGO “BRICKBREAK”

O segundo estudo de caso foi um jogo 2D chamado “*BrickBreak*”, e se concentrou em códigos de programa que são formados por 5 (cinco) classes (“*Bola*”, “*Barra*”, “*Bloco*”, “*Jogo*”, “*Principal*”). Cada uma das classes envolve a implementação de funcionalidades específicas dos personagens (objetos) do jogo. Neste caso, a construção do jogo envolve a necessidade de mais conhecimentos sobre o uso do paradigma de orientação a objetos do que no primeiro estudo de caso. Desta forma procuramos mostrar os resultados de análises para um caso mais complexo que o apresentado no primeiro estudo de caso. Este cenário foi escolhido porque corresponde ao paradigma de aprendizagem aplicado as disciplinas de “Programação Orientada a Objetos” e “Tópicos Especiais em Desenvolvimento de Software” do curso Técnico em Informática e Tecnólogo em Análise e Desenvolvimento de Sistemas do IFPI.

O jogo “*BrickBreak*” foi desenvolvido como atividade do minicurso “Desenvolvimento de Jogos usando Java e o framework JPlay”. Os alunos desenvolveram o jogo durante 2 (dois) sábados de aula. Foi utilizado laboratório de programação com 24 (vinte e quatro) computadores. Inicialmente os comportamentos de cada uma das classes envolvidas no projeto foram apresentados aos alunos, o enunciado (ver em anexo B) divide as orientações por classe de projeto, de forma que para cada classe cada um dos comportamentos foi discutido em sala de aula, bem como as maneiras de definir aquele comportamento em termos de implementação (que métodos java e JPlay estariam envolvidos, por exemplo). As classes “*Bola*”, “*Barra*” e “*Bloco*” foram desenvolvidas inicialmente, as classes “*Jogo*” e “*Principal*” foram as últimas classes desenvolvidas.

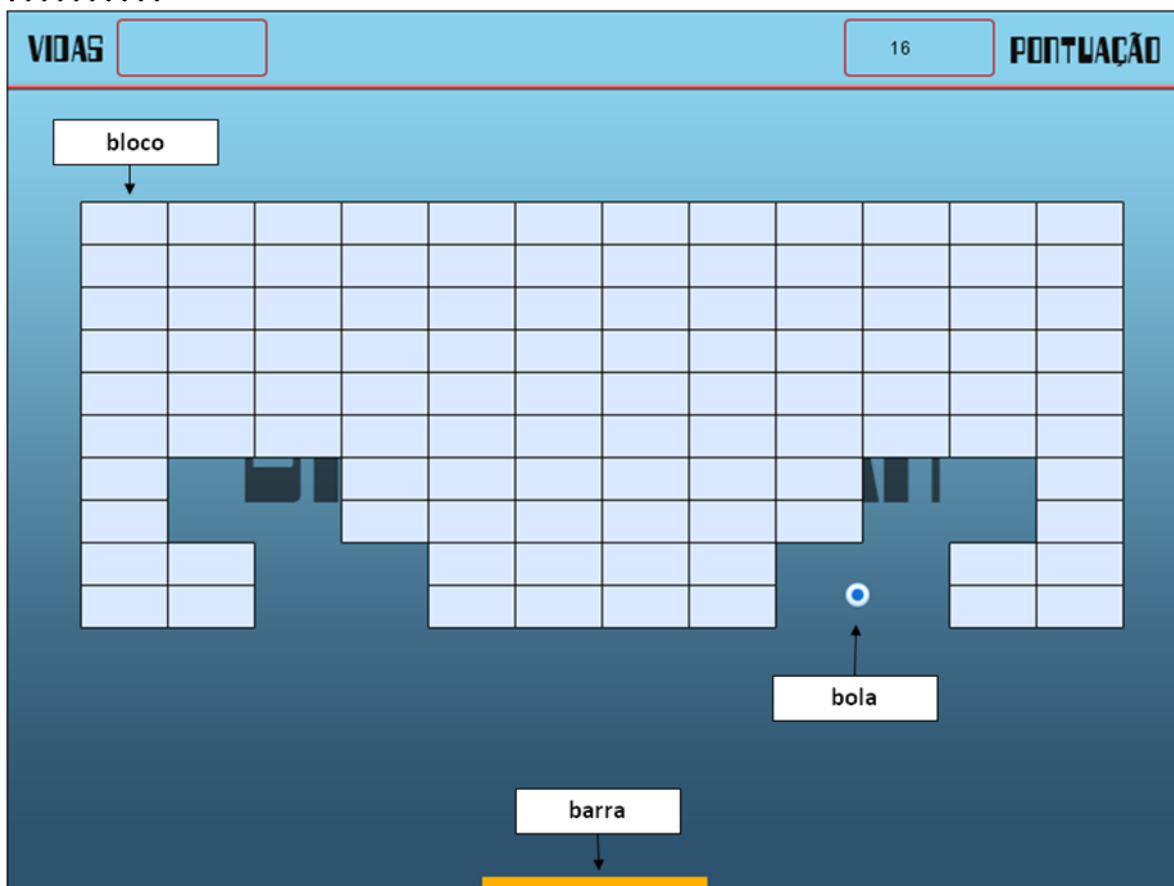
As classes “*Bola*”, “*Barra*” e “*Bloco*” definem os respectivos comportamentos dos personagens (objetos) do jogo: bola, barra e bloco. Ver na Figura 6.3 os objetos do jogo.

De acordo com o enunciado apresentado aos alunos (ver em anexo B), a classe “*Bola*” possui 2 (duas) especificações básicas: a bola deve se mover no eixo “x” e “y”. Assim, na definição do modelo do professor, a classe “*Bola*” possui 3 (três) métodos, o construtor da classe, um método para realizar o movimento no eixo “x” e um método para realizar o movimento no eixo “y”. No trecho de código abaixo seguem os marcadores de padronização da classe:

```

/**
 *
 * @comment A classe Bola possui 2 especificacoes básicas: deve se mover
no eixo x e y.
 * @method 3
 * @comment A classe Bola deve possuir 3 metodos: o construtor da classe,
um método para realizar o movimento no eixo x e um método para realizar o
movimento no eixo y.
 * @inheritance Sprite
 * @comment A classe deve herdar de Sprite
 * @movex true
 * @comment realize o movimento no eixo do x
 * @movey true
 * @comment realize o movimento no eixo do y
 */
public class Bola extends Sprite{
.....

```



**Figura 6.3: Janela do jogo “BrickBreak” com indicação dos personagens (objetos) do jogo**

De acordo com o enunciado apresentado aos alunos (ver em anexo B), a classe “Barra” também possui 2 (duas) especificações: a barra deve se mover, através do controle do teclado, para os lados esquerdo e direito. Assim, na definição do modelo do professor, a classe

“Barra” possui 2 (dois) métodos, o construtor da classe e um método para realizar o movimento para os lados esquerdo e direito através do teclado. No trecho de código abaixo seguem os marcadores de padronização da classe “Barra”:

```
/**
 *
 * @comment A classe deve herdar de Sprite e possuir 2 metodos: o
 construtor deve inicializar a figura da barra, e um método para controlar
 o movimento no eixo x usando o teclado (DICA: use o metodo Keydown() da
 classe Keyboard do JPlay)
 * @method 2
 * @comment A classe deve conter 2 metodos The class may have 2 methods, o
 construtor deve inicializar a figura da barra, e um método para controlar
 o movimento no eixo x usando o teclado (DICA: use o metodo Keydown() da
 classe Keyboard do JPlay)
 * @inheritance Sprite
 * @comment A classe Barra deve herdar de Sprite
 * @movex true
 * @comment A classe deve realizar o movimento no eixo x para a
 * esquerda e para a direita
 * @keyboard true
 * @comment O movimento da classe Barra no eixo x deve ser controlado
 * pelo teclado (DICA: use o metodo Keydown() da classe Keyboard
 * do JPlay).
 */
```

De acordo com o enunciado apresentado aos alunos (ver em anexo B), a classe “Bloco” não possui especificações, considerando que o objeto bloco permanece estático durante o decorrer de todo o jogo e não possui nenhuma ação específica. Apenas a colisão da bola com este objeto, faz com que ele seja destruído, mas esta ação de colisão é tratada pela classe “Jogo”. Portanto, na definição do modelo do professor, a classe “Bloco” possui somente o método construtor da classe. Por este motivo esta classe não foi inserida na análise de resultados. No trecho de código abaixo seguem os marcadores de padronização da classe “Bloco”:

```
**
/* @comment O objeto bloco não possui nenhuma acao especifica e permanece
estatico durante o decorrer de todo o jogo. Apenas a colisao da bola com
este objeto, faz com que ele seja destruido, mas esta ação de colisao e
tratada pela classe “Jogo”.
 * @method 1
 * @comment A classe Bloco possui somente o método construtor da classe
 * @inheritance Sprite
 * @comment A classe Bloco herda de Sprite
```

\*/

A classe “*Jogo*” define o comportamento do “*gameloop*” do jogo (enunciado em anexo B). Ela é a classe que possui a maior quantidade de comportamentos a serem implementados, pois o “*gameloop*” define o padrão de projeto central do jogo, ou seja, nesta classe os objetos devem ser instanciados e interagir entre si de acordo com as regras estabelecidas pelo jogo (as colisões entre os objetos bola e barra, bola e bloco são tratadas nesta classe). No trecho de código abaixo seguem os marcadores de padronização da classe “*Jogo*”:

```
/**
 *
 * @comment Esta classe deve contar o loop infinito do jogo (coracao da
 execucao do jogo), nela vao ser criados os gameobjects envolvidos, A
 classe deve conter 5 metodos, um metodo construtor, um metodo para
 carregar os objetos envolvidos, um metodo para inicializar esses objetos,
 um metodo para desenhar os objetos e outro metodo que representa o loop
 infinito do jogo. O metodo construtor deve ficar responsavel por chamar os
 outros metodos.
 * @method 4
 * @comment A classe deve conter 5 metodos, um metodo construtor, um
 metodo para carregar os objetos envolvidos, um metodo para inicializar
 esses objetos, um metodo para desenhar e outro metodo que representa o
 loop infinito do jogo (coracao da execucao do jogo). O metodo construtor
 deve ficar responsavel por chamar os outros metodos.
 * @gameobject 3
 * @comment A classe deve definir 3 gameobjects: Barra, Bola e Bloco.
 * @window
 * @comment Voce deve atualizar a janela do jogo atraves do metodo update
 */
```

A classe “*Principal*” define somente a instanciação de um objeto do tipo da classe “*Jogo*”, iniciando assim a execução do programa. Portanto, por ser uma classe de um único comportamento, a classe “*Principal*” também não foi inserida nesta análise de resultados.

Na análise deste capítulo do jogo “*BrickBreak*” são verificados os resultados das classes “*Bola*”, “*Barra*” e “*Jogo*”. Os resultados são quanto às análises gerais e detalhadas destas classes. Para cada tipo de análise (geral e detalhada), as classes são avaliadas quanto a padronização do modelo, comparação de pares de variáveis (árvores de comportamento) e resultados dos comentários das variáveis sem par.

Entre os 10 (dez) programas analisados, os estudantes 3, 5, 8, 9 e 10 estão com o comportamento geral do programa incorreto na primeira tentativa de desenvolver o exercício, conforme descrito na Tabela 6.8 abaixo.

**Tabela 6.8: RELAÇÃO DOS PROGRAMAS COM COMPORTAMENTO GERAL INCORRETOS**

ESTUDANTE	COMPORTAMENTO DO JOGO INCORRETO
3	O movimento da barra não é realizado através do controle do teclado, ao invés disso a barra segue a direção do objeto bola, sem que o usuário tenha qualquer tipo de controle através do teclado.
5	Não realiza o movimento da bola corretamente, ela fica presa no lado superior da janela.
10	Todos os objetos estão sem movimento.
8	O objeto bola não se movimenta para os lados corretamente. O movimento do objeto barra, apesar de ser controlado pelo teclado, também segue para o centro da janela do jogo sempre que a bola alcança uma determinada posição central na janela.
9	O movimento está incorreto, a barra é controlada através do uso do teclado somente quando o comando do teclado segue a mesma direção da bola, e está se movimentando no eixo “y”, sendo que foi solicitado que a barra se movimentasse somente no eixo “x”.

#### **6.4.1 RESULTADOS OBTIDOS APARTIR DA MEDIÇÃO DE SENSIBILIDADE, ESPECIFICIDADE, ACURÁCIA E PRECISÃO**

Da mesma forma que o estudo de caso 1, foram verificados os dois tipos de resultados: o primeiro baseado por unidade de programa analisado e o segundo baseado na quantidade de comentários apresentados por programa. As Tabelas 6.9 e 6.10 apresentam os resultados das medidas de eficiência obtidas através dos dois resultados.

**Tabela 6.9: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE CLASSES DE PROGRAMA ANALISADAS PARA O SEGUNDO ESTUDO DE CASO**

<i>Projeto</i> <i>"BrickBreak"</i>	<i>Medidas de eficácia do sistema analisador de acordo com o total de programas analisados</i>		
<i>Classe "Bola"</i>	<i>Quantidades e Medidas Obtidas</i>	<i>De acordo com a Padronização do Modelo</i>	<i>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</i>
	<i>NN</i>	1	1
	<i>FN</i>	2	1
	<i>PP</i>	7	8
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	78%	89%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	80%	90%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	33%	50%
<i>Classe "Barra"</i>	<i>NN</i>	3	3
	<i>FN</i>	7	7
	<i>PP</i>	0	0
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	0%	0%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	30%	30%
	<i>Preditividade Positiva</i>	-	-
	<i>Preditividade Negativa</i>	30%	30%
<i>Classe "Jogo"</i>	<i>NN</i>	5	9
	<i>FN</i>	4	0
	<i>PP</i>	1	1
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	20%	100%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	60%	100%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	56%	100%

**Tabela 6.10: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O SEGUNDO ESTUDO DE CASO**

<i>Projeto</i> <i>"BrickBreak"</i>	<i>Medidas de eficácia do sistema analisador de acordo com o total de comentários analisados</i>		
<i>Classe "Bola"</i>	<i>Quantidades e Medidas Obtidas</i>	<i>De acordo com a Padronização do Modelo</i>	<i>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</i>
	<i>NN</i>	3	12
	<i>FN</i>	2	6
	<i>PP</i>	7	8
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	78%	57%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	83%	77%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	60%	67%
<i>Classe "Barra"</i>	<i>NN</i>	4	8
	<i>FN</i>	14	28
	<i>PP</i>	0	0
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	0%	0%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	22%	22%
	<i>Preditividade Positiva</i>	-	-
	<i>Preditividade Negativa</i>	22%	22%
<i>Classe "Jogo"</i>	<i>NN</i>	6	37
	<i>FN</i>	6	33
	<i>PP</i>	1	1
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	14%	3%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	54%	54%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	50%	53%

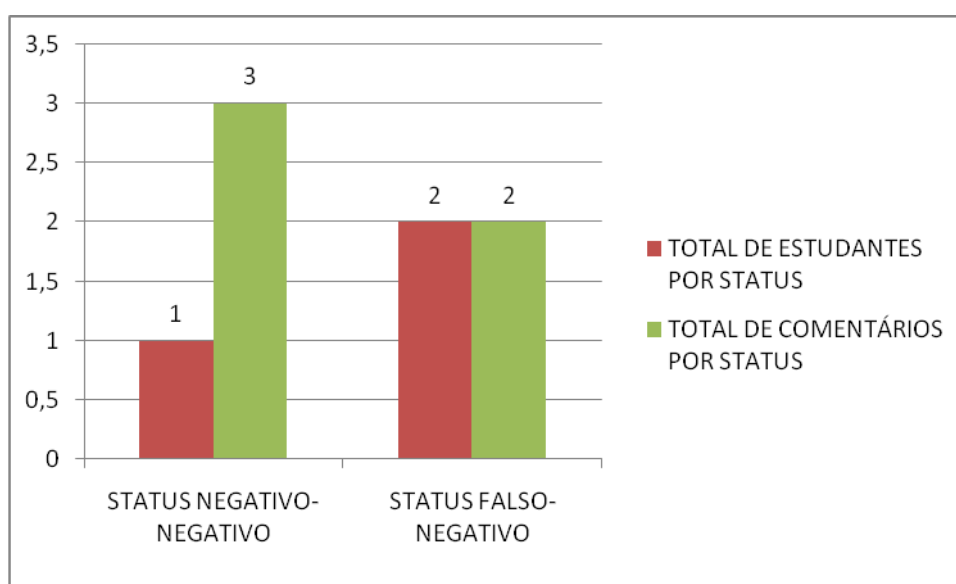
## 6.4.2 RESULTADOS OBTIDOS APARTIR DAS TRÊS HIPÓTESES PROPOSTAS

### 6.4.2.1 ANÁLISE GERAL QUANTO A PADRONIZAÇÃO DO MODELO

As classes serão analisadas individualmente, assim considera-se, para efeito de avaliação de uma classe, que o comportamento individual da classe está correto ou incorreto, de forma que problemas que forem localizados em outras classes não serão considerados na avaliação individual.

#### 6.4.2.1.1 PADRONIZAÇÃO DO MODELO: CLASSE “BOLA”

A análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo negativo-negativo (PNN) foi de 60% e o percentual de precisão do tipo falso-negativo (PFN) foi de 40%. Portanto, concluindo  $PNN > PFN$ , e para o caso da classe “Bola” é possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.5.



**Gráfico 6.5: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Bola” do jogo “BrickBreak”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.

De acordo com o Gráfico 6.6, a análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo positivo-positivo (PPP) foi de 100% e o percentual de precisão do tipo falso-positivo (PFP) foi de 0%. Verificamos, neste caso, que  $PPP > PFP$ . Portanto, é possível aceitar a hipótese 2 como verdadeira.



Apesar dos estudantes 3, 5, 8 e 9 apresentarem o comportamento do programa incorreto, eles apresentaram resultados do tipo positivo-positivo. Através deste resultado, foi considerado que o erro de comportamento não está na implementação da classe “Bola”, e sim em outras classes do programa. Para os estudantes 3 e 9, o comportamento do programa está incorreto devido a problemas com outro objeto (o objeto “Barra”). Para os estudantes 5 e 8, o movimento da bola está incorreto, mas não por motivo de problemas na implementação da classe “Bola”, ao invés disso, os problemas de comportamento são gerados por implementação da classe “Jogo” (classe que contém o “gameloop” e é responsável pela iteração dos objetos), o objeto do tipo “Bola” neste classe é manipulado de maneira incorreta.



**Gráfico 6.6: Distribuição de mensagens tipo positivo-positivo e falso-positivo nos códigos da classe “Bola” do jogo “BrickBreak”, de acordo com a padronização**

Fonte: Pesquisa direta, 2015.

A Tabela 6.11 mostra a distribuição detalhada das mensagens de padronização na classe “Bola”. Os estudantes 1 e 4, apesar de apresentarem o comportamento da classe “Bola” e o comportamento do programa corretos, não efetuaram a padronização da classe “Bola” conforme foi solicitado. O estudante 1 implementou um método a mais no código e o estudante 4 deixou de implementar um método. Na classe “Bola” do modelo do professor o marcador “@method” é igual a 3, indicando que a classe deve possuir 3 (três) métodos definidos. O analisador calcula valores diferentes para o estudante 1 e 4, gerando diferenças na comparação, e fazendo com que seja impressa a mensagem armazenada no marcador “@comment” associado. Ver trecho de código abaixo correspondente:

\* @method 3

\* @comment A classe Bola deve possuir 3 metodos: o construtor da classe, um método para realizar o movimento no eixo x e um método para realizar o movimento no eixo y.

O estudante 10 não implementou a classe “Bola” completamente, sendo que o seu código incompleto gerou diferenças em 3 (três) marcadores diferentes: marcador “@method”, pois não implementou a quantidade de métodos solicitados na padronização e marcadores “@movex” e “@movey”, pois não implementou a movimentação do objeto tipo “Bola” no eixo x e no eixo y. Neste caso, as mensagens armazenadas nos “@comment” associados a estes marcadores são apresentadas.

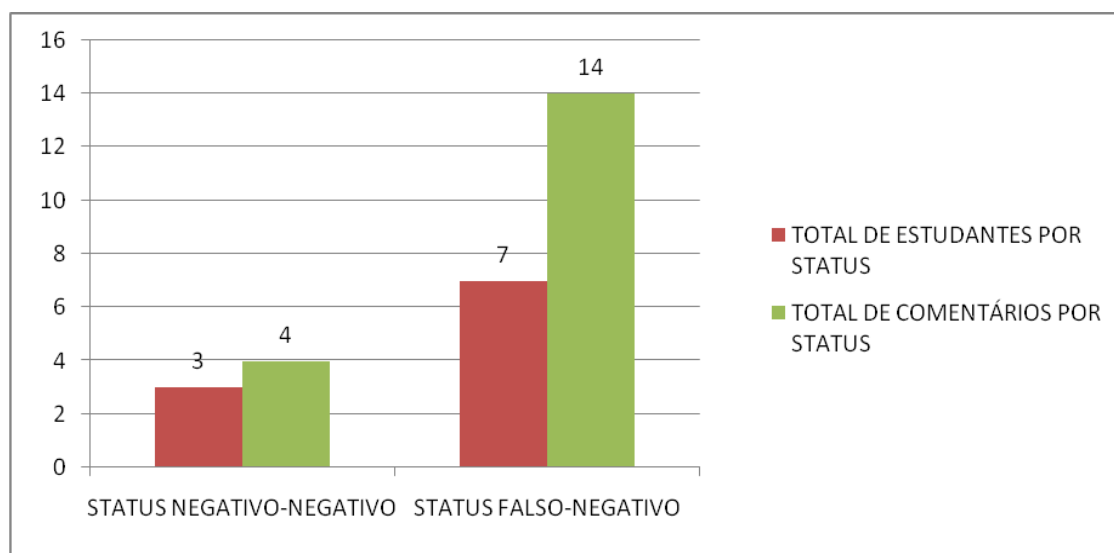
**Tabela 6.11: AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BOLA” DE ACORDO COM A PADRONIZAÇÃO**

Estudante	Avaliação de comentários da classe “Bola” de acordo com a Padronização do Modelo		
	Status do programa	Marcador relacionado ao comentário	Status do comentário
1	Correto	“method”	Tipo Falso-Negativo
4	Correto	“method”	Tipo Falso-Negativo
10	Incorreto	“movex”	Tipo Negativo-Negativo
		“movey”	Tipo Negativo-Negativo
		“method”	Tipo Negativo-Negativo

De acordo com a Tabela 6.11 é possível verificar que os estudantes 1 e 4, apesar de possuírem o comportamento geral do programa correto, não seguiram a padronização estabelecida pelo modelo gerando comentários falso-negativos.

#### 6.4.2.1.2 PADRONIZAÇÃO DO MODELO: CLASSE “BARRA”

A análise constatou que, de acordo com a padronização do modelo, o percentual de comentários do tipo negativo-negativo (PNN) foi de 22% e o percentual de comentários do tipo falso-negativo (PFN) foi de 78%. Conclui-se portanto que  $PNN < PFN$ , não sendo possível aceitar a hipótese 1 como verdadeira (ver Gráfico 6.7).



**Gráfico 6.7: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Barra” do jogo “BrickBreak”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.

A análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo positivo-positivo (PPP) foi de 0% e o percentual de precisão do tipo falso-positivo (PFP) foi de 0%. Ou seja, os códigos de todos os alunos apresentaram erro na padronização da classe “Barra”. Isto acontece porque a padronização do modelo solicitou a implementação do movimento do objeto barra através do teclado. Isto é feito através da configuração do marcador “@keyboard” no código do programa modelo, conforme o trecho de código:

```
* @keyboard true
* @comment O movimento da classe Barra no eixo x deve ser controlado
pelo teclado (DICA: use o metodo Keydown() da classe Keyboard do Play).
```

No código de todos os estudantes, como a implementação não foi realizada, o analisador calcula o marcador “@keyboard” com valor igual a “false”. Ele verifica uma diferença no valor solicitado do marcador e identifica um erro. Com isto, a mensagem guardada no marcador “@comment” associado é então apresentada para todos os alunos. A Tabela 6.12 mostra a distribuição detalhada das mensagens de padronização na classe “Barra”. Os estudantes 1, 2, 4 e 7 realizaram a movimentação do objeto barra, mas de maneira diferente da solicitada pela padronização do modelo, eles realizaram o comportamento através do método “movex” (método do JPlay) no “gameloop” da classe “Jogo”. Por esta razão o status do comentário para o marcador “@keyboard” foi classificado

como Falso-Negativo. A padronização também solicitou 2 (dois) métodos definidos na classe. Isto é feito através da configuração do marcador “*@method*” no código do programa modelo. A maioria dos alunos (estudantes 1, 2, 4, 5, 6, 7, 8 e 10) só definiu 1 (um) método, o construtor da classe. Segue abaixo o trecho de código relacionado:

\* *@method* 2

\* *@comment* A classe deve conter 2 metodos The class may have 2 methods, o construtor deve inicializar a figura da barra, e um método para controlar o movimento no eixo x usando o teclado (DICA: use o metodo Keydown() da classe Keyboard do JPlay)

**Tabela 6.12: AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BARRA” DE ACORDO COM A PADRONIZAÇÃO**

Estudante	Avaliação de comentários da classe “Bola” de acordo com a Padronização do Modelo		
	Status do programa	Marcador relacionado ao comentário	Status do comentário
1	Correto	“keyboard”	Tipo Falso-Negativo
		“method”	Tipo Falso-Negativo
2	Correto	“keyboard”	Tipo Falso-Negativo
		“method”	Tipo Falso-Negativo
3	Incorreto	“keyboard”	Tipo Negativo-Negativo
4	Correto	“keyboard”	Tipo Falso-Negativo
		“method”	Tipo Falso-Negativo
5	Incorreto	“keyboard”	Tipo Falso-Negativo
		“method”	Tipo Falso-Negativo
6	Correto	“keyboard”	Tipo Falso-Negativo
		“method”	Tipo Falso-Negativo
7	Correto	“keyboard”	Tipo Falso-Negativo
		“method”	Tipo Falso-Negativo
8	Incorreto	“keyboard”	Tipo Falso-Negativo
		“method”	Tipo Falso-Negativo
9	Incorreto	“keyboard”	Tipo Negativo-Negativo
10	Incorreto	“keyboard”	Tipo Negativo-Negativo
		“method”	Tipo Negativo-Negativo

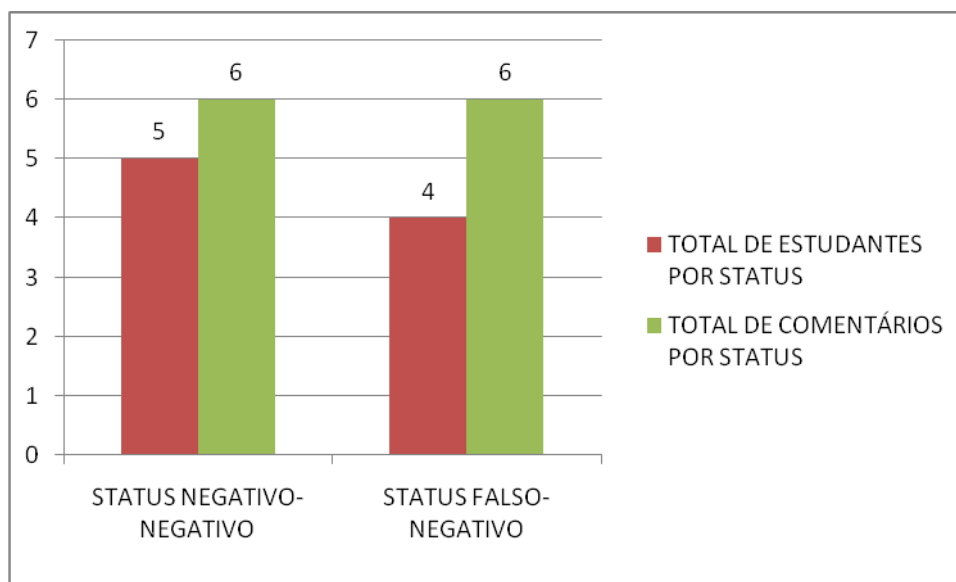
De acordo com a Tabela 6.12 é possível verificar que todos os estudantes envolvidos apresentaram problemas na padronização do modelo. Os estudantes 1, 2, 4, 6 e 7, apesar de possuírem o comportamento geral do programa correto, não seguiram a padronização estabelecida pelo modelo, gerando comentários falso-negativos.

No caso da classe “Barra”, ao verificar a ocorrência da diferença no marcador “@keyboard” em todos os códigos, sendo que a maioria dos alunos realizou o comportamento de outra maneira na classe “Jogo”, o professor poderia optar por documentar mais claramente esta padronização com a intenção de esclarecer o objetivo da atividade, ou poderia optar por não mais usar o marcador “@keyboard”, aceitando assim as variações de programação relacionadas a esta atividade.

A alta incidência de falso-negativos na padronização da classe compromete a eficiência da próxima fase de comparação do analisador, a comparação de pares de variáveis, gerando muitos resultados falso-negativos, também, na próxima fase.

#### 6.4.2.1.3 PADRONIZAÇÃO DO MODELO: CLASSE “JOGO”

A análise constatou que, de acordo com a padronização do modelo, o percentual de comentários do tipo negativo-negativo (PNN) foi de 50% e o percentual de comentários do tipo falso-negativo (PFN) foi de 50%. Portanto, concluindo  $PNN=PFN$ , não é possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.7.



**Gráfico 6.8: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Jogo” do jogo “BrickBreak”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.

A análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo positivo-positivo (PPP) foi de 100% e o percentual de precisão do tipo falso-positivo (PFP) foi de 0%. Verificamos, neste caso, o  $PPP > PFP$ . Portanto, é possível aceitar a hipótese 2 como verdadeira. No caso da classe “Jogo”, somente 1(um) código de estudante não apresentou mensagem de erro, sendo que o comportamento do programa estava correto e foi implementado de acordo com a padronização do modelo.

A Tabela 6.13 mostra a distribuição detalhada das mensagens de padronização na classe “Jogo”.

**Tabela 6.13: AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “JOGO” DE ACORDO COM A PADRONIZAÇÃO**

Estudante	Avaliação de comentários da classe “Jogo” de acordo com a Padronização do Modelo		
	Status do programa	Marcador relacionado ao comentário	Status do comentário
2	Correto	“method”	Tipo Falso-Negativo
		“gameobject”	Tipo Falso-Negativo
3	Incorreto	“method”	Tipo Negativo-Negativo
4	Correto	“method”	Tipo Falso-Negativo
5	Incorreto	“method”	Tipo Negativo-Negativo
		“gameobject”	Tipo Negativo-Negativo
6	Correto	“method”	Tipo Falso-Negativo
		“gameobject”	Tipo Falso-Negativo
7	Correto	“method”	Tipo Falso-Negativo
8	Incorreto	“method”	Tipo Negativo-Negativo
9	Incorreto	“method”	Tipo Negativo-Negativo
10	Incorreto	“method”	Tipo Negativo-Negativo

De acordo com a Tabela 6.13 é possível verificar que os estudantes 2, 4, 6 e 7, apesar de apresentarem o comportamento do programa correto, não implementaram a classe “Jogo” de acordo com o solicitado pela padronização do modelo, definindo uma quantidade de métodos diferente da que foi orientada (marcador “@method”). No caso do aluno 5, além de definir uma quantidade de métodos diferente da solicitada pelo modelo, também definiu um objeto a mais dos 3 (três) objetos solicitados pelo modelo (o marcador “@gameobject” recebe o valor igual a 3 indicando que devem ser criados os objetos bola, barra e bloco). Um objeto a

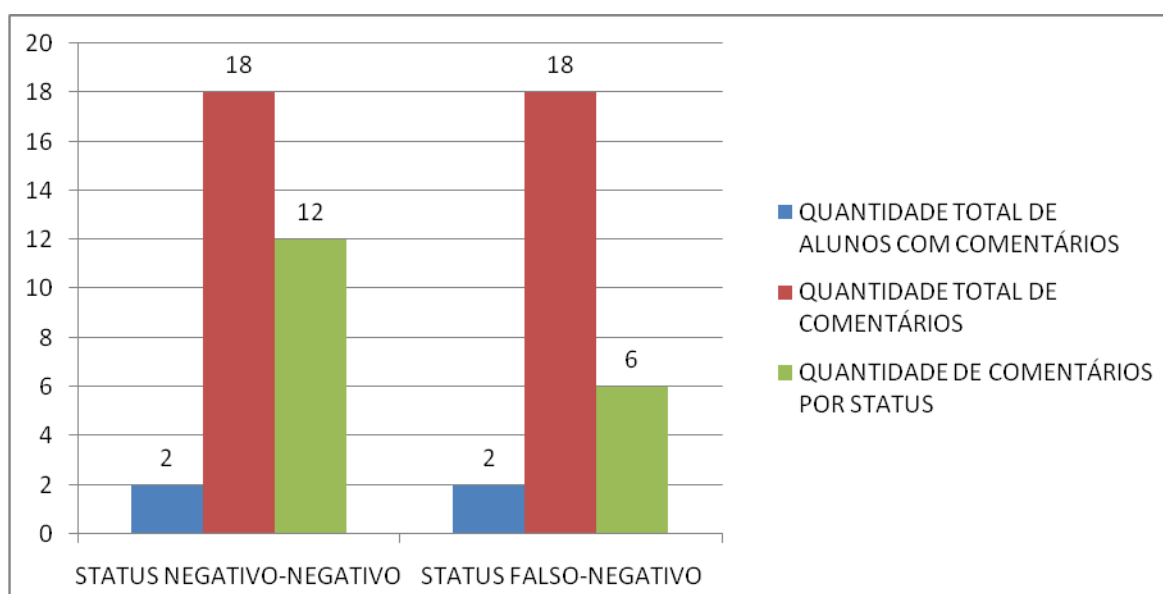
mais foi criado, porém não foi utilizado na classe, indicando um má prática de programação por parte do estudante.

#### 6.4.2.2 ANÁLISE QUANTO A DIFERENÇAS DE COMPORTAMENTOS

Neste item são verificadas as análises geral e detalhada das 3 (três) classes do jogo: “Bola”, “Barra” e “Jogo”. Os 2 (dois) tipos de análise quanto a diferença de comportamentos (análise de acordo com a comparação de par de variáveis e análise de variáveis sem par) são reunidos em um único gráfico de resultados. A seguir, para cada uma das classes, seguem as análises:

##### 6.4.2.2.1 ANÁLISE QUANTO A DIFERENÇAS DE COMPORTAMENTO: CLASSE “BOLA”

De acordo com o Gráfico 6.9, com a comparação de par de variáveis (árvores de comportamento), o percentual de comentários do tipo negativo-negativo (PNN) foi de 67% e o percentual de comentários do tipo falso-negativo (PFN) foi de 33%.



**Gráfico 6.9: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Bola” do jogo “BrickBreak”, de acordo com a comparação de par de variáveis e variáveis sem par**

Fonte: Pesquisa direta, 2015.

Portanto, no caso da classe “Bola” conclui-se que  $PNN > PFN$ , sendo portanto possível aceitar a hipótese 1 como verdadeira.

De acordo com a Tabela 6.14, podemos verificar como a falta de padronização do programa do aluno prejudica o resultado da comparação e aumenta a quantidade de resultados falso-negativos. No caso da classe “Bola”, os falso-negativos foram gerados na implementação de 1 (um) aluno: o estudante 1 não seguiu as orientações de padronização indicadas pelo modelo, incluindo um método a mais do que o solicitado e causando diferenças falso-negativas na comparação dos pares de variáveis.

**Tabela 6.14: AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BOLA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR**

Estudante	Avaliação dos comentários na classe "Bola" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
1	Correto	3	Tipo Falso-Negativo
		7	Tipo Falso-Negativo
		9	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
		12	Tipo Falso-Negativo
		14	Tipo Falso-Negativo
2	Correto	<Sem comentários>	Tipo Positivo-Positivo
3	Incorreto	<Sem comentários>	Tipo Positivo-Positivo
4	Correto	<Sem comentários>	Tipo Positivo-Positivo
5	Incorreto	<Sem comentários>	Tipo Positivo-Positivo
6	Correto	<Sem comentários>	Tipo Positivo-Positivo
7	Correto	<Sem comentários>	Tipo Positivo-Positivo
8	Correto	<Sem comentários>	Tipo Positivo-Positivo
9	Incorreto	<Sem comentários>	Tipo Positivo-Positivo
10	Incorreto	6	Tipo Negativo-Negativo
		8	Tipo Negativo-Negativo
		3	Tipo Negativo-Negativo
		7	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		11	Tipo Negativo-Negativo
		13	Tipo Negativo-Negativo
		4	Tipo Negativo-Negativo
		12	Tipo Negativo-Negativo
		14	Tipo Negativo-Negativo
		16	Tipo Negativo-Negativo



Estudante	Avaliação dos comentários na classe "Bola" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	<i>Status do programa</i>	<i>Identificador do comentário</i>	<i>Status do comentário</i>
		15	Tipo Negativo-Negativo

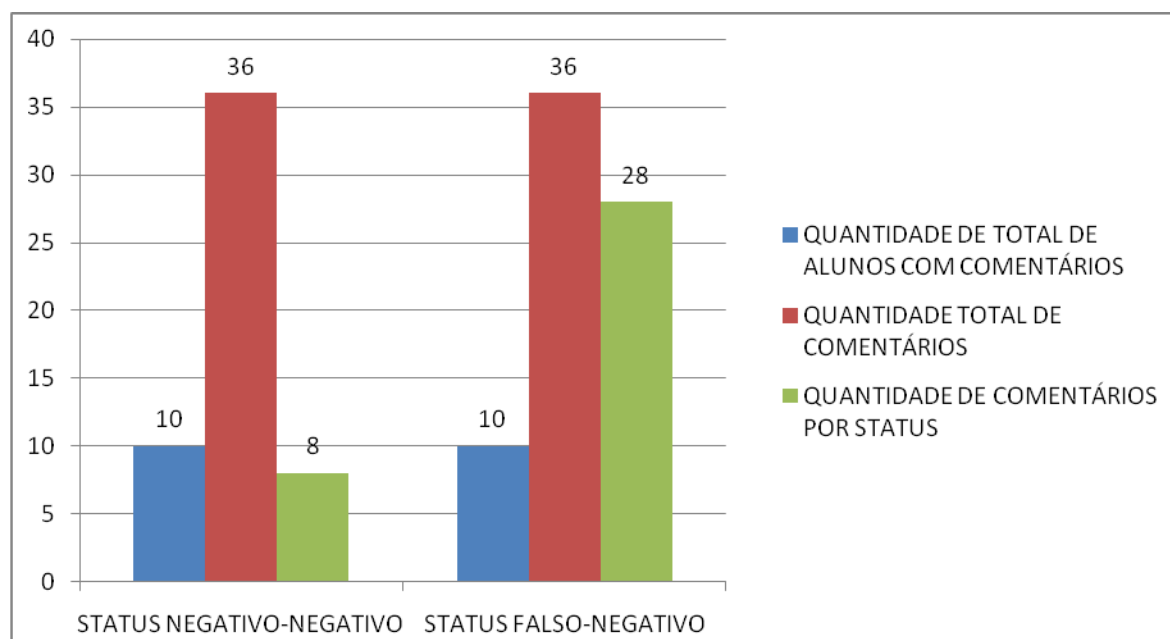
Apesar dos estudantes 3, 5 e 9 apresentarem problemas na execução do jogo, o mal funcionamento do objeto bola está ligado a sua má utilização na classe “Jogo” (classe que contém o *gameloop* e é responsável por realizar as interações dos objetos entre si, respeitando as regras do jogo) e não à implementação da classe “Bola”. Desta forma, os resultados desses códigos para a classe “Bola” são considerados do tipo positivo-positivo.

O estudante 10 foi o único que recebeu comentários resultantes de variáveis sem pares. Este estudante gerou um código de comportamento incorreto, sendo que ele não implementou a classe “Bola” completamente, causando que as variáveis ficassem sem pares e consequentemente fossem gerados comentários negativo-negativos.

#### 6.4.2.2.2 ANÁLISE QUANTO A DIFERENÇAS DE COMPORTAMENTO: CLASSE “BARRA”

De acordo com o Gráfico 6.10, com a comparação de par de variáveis (árvores de comportamento), o percentual de comentários do tipo negativo-negativo (PNN) foi de 22% e o percentual de comentários do tipo falso-negativo (PFN) foi de 78%.

Portanto, no caso da classe “Barra”, conclui-se que  $PNN < PFN$ , não sendo possível aceitar a hipótese 1 como verdadeira. Podemos concluir que o alto índice de falta de padronização da classe “Barra” (todos os estudantes apresentaram problemas na padronização do modelo resultantes de diferenças no marcador “*keyboard*”, sendo que os estudantes 1, 2, 4, 5, 6, 7, 8 e 10 também apresentaram problemas resultantes de diferenças no marcador “*method*”) comprometeu os resultados da comparação dos pares de variáveis.



**Gráfico 6.10: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Barra” do jogo “BrickBreak”, de acordo com a comparação de par de variáveis e variáveis sem par**

Fonte: Pesquisa direta, 2015.

De acordo com a Tabela 6.15, podemos verificar que para os estudantes 1, 2, 4, 5, 6, 7, 8 e 10 foram impressos os comentários 1, 2, 3 e 4, sendo os mesmos resultantes de comparação de variáveis sem pares. Estes estudantes obtiveram 2 (dois) problemas de padronização (diferenças nos marcadores “*keyboard*” e “*method*”). No código eles definiram somente o método construtor, gerando uma classe sem variáveis (atributos da classe), portanto sem comportamentos. Consequentemente o analisador não é capaz de formar pares de variáveis, logo todos os comentários (1, 2, 3 e 4) da classe “Barra” do programa modelo foram impressos. Segue exemplo de trecho de código de um dos estudantes:

```
public class Barra extends Sprite {

    public Barra(String fileName,int frame) {
        super(fileName,frame);
        // TODO Auto-generated constructor stub
    }
}
```

Os comentários 1 e 2 estão relacionados ao comportamento de uma variável sem par do tipo “*Keyboard*” (responsável pela movimentação através do teclado) e os comentários 3 e 4 estão relacionados ao comportamento de uma variável sem par do tipo “*int*” (responsável

pela movimentação do objeto através do eixo x). Desta forma, são impressos os seguintes comentários:

- *"comment01: Definir movimento da Barra para a direita através do uso do teclado usando o metodo KeyDown. Verifique que, na janela do jogo, a margem direita maxima de x e 800."*
- *comment02: Definir movimento da Barra para a esquerda através do uso do teclado usando o metodo KeyDown. Verifique que, na janela do jogo, a margem esquerda minima de x e 8.*
- *comment03: Incrementar o movimento do Barra no eixo x, fazendo a mesma se mover para a direita.*
- *comment04: Decrementar o movimento do Barra no eixo x, fazendo a mesma se mover para a esquerda.*

**Tabela 6.15: AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BARRA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR**

Estudante	Avaliação dos comentários na classe "Barra" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
1	Correto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
2	Correto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
3	Incorreto	1	Tipo Negativo-Negativo
		2	Tipo Negativo-Negativo
4	Correto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
5	Incorreto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
6	Correto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo

Estudante	Avaliação dos comentários na classe "Barra" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
7	Correto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
8	Correto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
9	Incorreto	1	Tipo Negativo-Negativo
		2	Tipo Negativo-Negativo
10	Incorreto	1	Tipo Negativo-Negativo
		2	Tipo Negativo-Negativo
		3	Tipo Negativo-Negativo
		4	Tipo Negativo-Negativo

De acordo com a Tabela 6.15, os estudantes 3 e 9 apresentaram os comentários negativo-negativos 1 e 2, ambos relacionados a diferenças no comportamento "*conditional*" da variável sem par do tipo "*Keyboard*". Assim, os seguintes comentários são impressos:

- *"comment01: Definir movimento da Barra para a direita atraves do uso do teclado usando o metodo KeyDown. Verifique que, na janela do jogo, a margem direita maxima de x e 800."*
- *comment02: Definir movimento da Barra para a esquerda atraves do uso do teclado usando o metodo KeyDown. Verifique que, na janela do jogo, a margem esquerda minima de x e 8."*

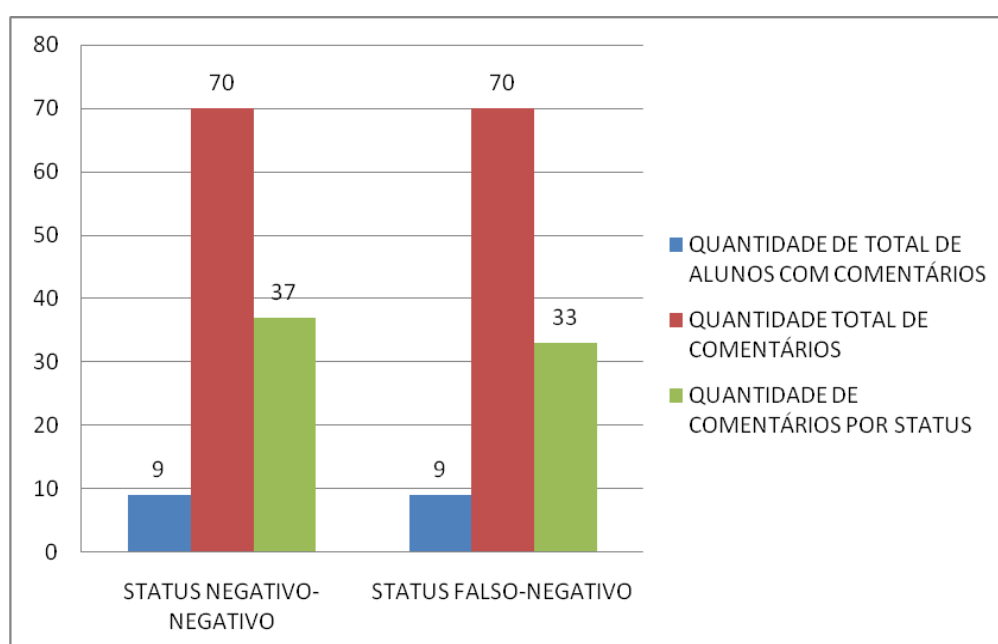
#### 6.4.2.2.3 ANÁLISE QUANTO A DIFERENÇAS DE COMPORTAMENTO: CLASSE "JOGO"

De acordo com o Gráfico 6.11, referente à comparação de par de variáveis (árvores de comportamento), o percentual de comentários do tipo negativo-negativo (PNN) foi de 53% e o percentual de comentários do tipo falso-negativo (PFN) foi de 47%.

Portanto, no caso da classe "*Jogo*" conclui-se que  $PNN > PFN$ , sendo portanto possível aceitar a hipótese 1 como verdadeira.

A Tabela 6.16 mostra a avaliação dos comentários na classe "*Jogo*" obtidos a partir da comparação de par de variáveis e variáveis sem par, podendo-se verificar que os mesmos

estudantes que apresentaram problemas de padronização (ver na Tabela 6.13 a avaliação de acordo com a padronização) também apresentaram comentários negativos indicando a ocorrência de erros no programa do estudante (estudantes 2, 3, 4, 5, 6, 7, 8, 9 e 10). Dentre eles, os estudantes 2, 3, 4, 6 e 7 apresentaram quase todos os comentários (com exceção somente do comentário 40 do tipo negativo-negativo) classificados como do tipo falso-negativo. É possível verificar, portanto, que a falta de padronização prejudica os resultados da comparação dos pares de variáveis, resultando em mais falso-negativos. a Tabela 6.13 mostra que os estudantes 2, 3, 4, 5, 6, 7, 8, 9 e 10 apresentaram problemas na padronização do modelo resultantes de diferenças no marcador “*method*”, sendo que os estudantes 2, 5 e 6 também apresentaram problemas resultantes de diferenças no marcador “*gameobject*”.



**Gráfico 6.11: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Jogo” do jogo “BrickBreak”, de acordo com a comparação de par de variáveis e variáveis sem par**

Fonte: Pesquisa direta, 2015.

**Tabela 6.16: AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “JOGO” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR**

Estudante	Avaliação dos comentários na classe "Jogo" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
1	Correto	<Sem comentários>	Tipo Positivo-Positivo

Estudante	Avaliação dos comentários na classe "Jogo" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
2	Correto	40	Tipo Negativo-Negativo
		8	Tipo Falso-Negativo
		9	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
		27	Tipo Falso-Negativo
		31	Tipo Falso-Negativo
3	Incorreto	40	Tipo Negativo-Negativo
		10	Tipo Falso-Negativo
		11	Tipo Falso-Negativo
		28	Tipo Falso-Negativo
		5	Tipo Falso-Negativo
		27	Tipo Falso-Negativo
4	Correto	31	Tipo Falso-Negativo
		40	Tipo Negativo-Negativo
		10	Tipo Falso-Negativo
		11	Tipo Falso-Negativo
		28	Tipo Falso-Negativo
		5	Tipo Falso-Negativo
5	Incorreto	27	Tipo Falso-Negativo
		31	Tipo Falso-Negativo
		40	Tipo Negativo-Negativo
		10	Tipo Negativo-Negativo
		11	Tipo Negativo-Negativo
		28	Tipo Negativo-Negativo
6	Correto	5	Tipo Negativo-Negativo
		27	Tipo Negativo-Negativo
		40	Tipo Negativo-Negativo
		8	Tipo Falso-Negativo
		9	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
		31	Tipo Falso-Negativo
		10	Tipo Falso-Negativo
		11	Tipo Falso-Negativo
		28	Tipo Falso-Negativo
		5	Tipo Falso-Negativo

Estudante	Avaliação dos comentários na classe "Jogo" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
7	Correto	40	Tipo Negativo-Negativo
		8	Tipo Falso-Negativo
		9	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
		6	Tipo Falso-Negativo
		31	Tipo Falso-Negativo
		10	Tipo Falso-Negativo
		11	Tipo Falso-Negativo
		28	Tipo Falso-Negativo
		5	Tipo Falso-Negativo
8	Incorreto	8	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		4	Tipo Negativo-Negativo
		10	Tipo Negativo-Negativo
		11	Tipo Negativo-Negativo
		28	Tipo Negativo-Negativo
		5	Tipo Negativo-Negativo
		27	Tipo Negativo-Negativo
9	Incorreto	31	Tipo Negativo-Negativo
		40	Tipo Negativo-Negativo
		8	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		4	Tipo Negativo-Negativo
		27	Tipo Negativo-Negativo
10	Incorreto	31	Tipo Negativo-Negativo
		8	Tipo Negativo-Negativo
		9	Tipo Negativo-Negativo
		4	Tipo Negativo-Negativo
		27	Tipo Negativo-Negativo
		10	Tipo Negativo-Negativo
		11	Tipo Negativo-Negativo
		28	Tipo Negativo-Negativo
		5	Tipo Negativo-Negativo
		7	Tipo Negativo-Negativo
		40	Tipo Negativo-Negativo

No caso do estudante 3, apesar do programa estar com o comportamento do programa incorreto, foram classificados comentários como do tipo falso-negativos (10, 11, 28, 5, 27 e 31). Todos os comentários falso-negativos foram impressos porque o analisador encontrou diferenças nos comportamentos do tipo *"assignment"* e do tipo *"conditional"* no objeto do tipo *"Bola"*. No entanto, o problema apresentado está no objeto do tipo *"Barra"*, que não é controlado através do teclado e ao invés disso, seu movimento apenas segue o movimento do objeto tipo *"Bola"* durante a execução do jogo (o problema já havia sido detectado inicialmente durante a padronização da classe *"Barra"* através da diferença encontrada no marcador *"@keyboard"*). A falta de padronização no programa do estudante 3 prejudica os resultados gerados a partir da comparação de variáveis e o analisador não consegue detectar o problema real no comportamento do objeto do tipo *"Barra"*, pois o método responsável pelo movimento da barra é chamado na classe *"Jogo"* de forma correta, no entanto foi implementado de forma errada na classe *"Barra"*.

Sobre os resultados negativo-negativos, o comentário 40 é apresentado para os estudantes 2, 3, 4, 6, 7 e 9. O comentário é relacionado a uma das regras do jogo que define a finalização do jogo caso o jogador conclua o total de 120 pontos. Nenhum dos estudantes citados implementou a finalização do jogo ao total dos 120 pontos, logo, o comentário 40 foi classificado como negativo-negativo. Assim, foi impresso, para os estudantes citados, o seguinte comentário:

- *comment40:No metodo loop(), verifique se a pontuacao final do jogador for igual a 120 entao ele ganhou o jogo,caso contrario ele perdeu o jogo*

Para os estudantes 5, 8, 9 e 10 os programas estão completamente incorretos, logo todos os comentários impressos foram considerados do tipo negativo-negativo.

No caso do estudante 5, atribuições de valores incorretos (classificados como comportamento do tipo *"assignment"*) no objeto tipo *"Bola"* ocasionam problemas de comportamento do objeto. Assim sendo, todos os comentários impressos estão relacionados aos comportamentos do objeto do tipo *"Bola"* e foram considerados do tipo negativo-negativo.

No caso do estudante 8, o objeto do tipo *"Barra"* se movimenta através do controle do teclado, porém também segue o movimento da bola. O movimento do objeto tipo *"Bola"* também está incorreto, pois a bola se movimenta sempre em direção ao centro da janela do



jogo. Assim, todos os comentários impressos estão relacionados os objetos do tipo *"Barra"* e do tipo *"Bola"*, que estão incorretos.

No caso do estudante 9 o movimento do objeto do tipo *"Barra"* está incorreto, pois a barra se move sob o controle do teclado somente quando este segue a mesma direção de movimento da bola, além disso a barra não se movimenta somente no eixo x, executando também o comportamento errado de movimentação no eixo y (somente foi solicitado o movimento da barra no eixo x). Assim, todos os comentários impressos estão relacionados ao objeto do tipo *"Barra"*.

No caso do estudante 10, o comportamento do programa está complementemente incorreto, os objetos do jogo do tipo *"Barra"* e *"Bola"* estão imóveis, sendo que os objetos do tipo *"Bloco"* não foram desenhados na janela do jogo. Assim, todos os comentários impressos relacionados aos objetos do tipo *"Barra"*, *"Bola"* e *"Bloco"*.

## 6.5 CONSIDERAÇÕES GERAIS SOBRE O SEGUNDO ESTUDO DE CASO

Na análise de acordo com a padronização somente a classe *"Bola"* foi aprovada, ou seja, a hipótese h1 ( $PNN > PFN$ ) foi aceita. Quase todas as medições estatísticas, no caso da classe *"Bola"* foram superiores a 50%. Somente a preditividade negativa para o cálculo baseado na quantidade de classes de programas é inferior a 50%. No entanto, a mesma medida baseada na quantidade de comentários é superior a 50%. A classe *"Barra"* e a classe *"Jogo"*, no entanto, foram reprovadas, ou seja, no caso delas, a hipótese h1 não foi aceita. Podemos verificar que o alto índice de falta de padronização nas classes *"Barra"* e *"Jogo"* influenciam os resultados da comparação na próxima fase de análise (comparação de variáveis), conforme se pode observar nas Tabelas 6.17 e 6.18. A classe *"Barra"* indica falha na padronização da classe quanto a hipótese 1 (problemas identificados pelos marcadores *"@Keyboard"* e *"@method"*). Por sua vez, a falha na padronização compromete a comparação de variáveis, gerando altos índices de falso-negativos nesta fase. Por motivo da falta de padronização, a acurácia e a preditividade negativa, na classe *"Barra"*, apresenta um valor inferior a 50%.

**Tabela 6.17: AVALIAÇÃO DOS RESULTADOS QUANTO A PADRONIZAÇÃO E COMPARAÇÃO DE VARIÁVEIS NAS CLASSES DO JOGO**

Classe	Avaliação dos resultados quanto a padronização e comparação de variáveis nas classes do jogo					
	PADRONIZAÇÃO QUANTO Hipótese 1			COMPARAÇÃO DE VARIÁVEIS QUANTO Hipótese 1		
	Status da classe	PNN	PFN	Status da classe	PNN	PFN
"Bola"	Aprovado	60%	40%	Aprovado	67%	33%
"Barra"	Reprovado	22%	78%	Reprovado	22%	78%
"Jogo"	Reprovado	50%	50%	Aprovado	53%	47%

A classe "Jogo" também apresenta falha na padronização da classe quanto a hipótese 1, com um resultado de PNN=PFN. Quanto a comparação de variáveis a classe alcança-se uma situação de aprovação, mas com uma taxa de PFN muito próxima a PNN.

A classe "Jogo" inclui os objetos necessários para a criação do cenário do jogo, controle através do teclado e instanciação dos objetos do tipo "Bola" e "Barra". Todos os comportamentos destes objetos devem ser executados, bem como as regras do jogo devem ser implementadas no *gameloop* (loop infinito) desta classe. Por ser uma classe mais detalhada, que envolve as outras classes do programa, a descrição da padronização e das regras do jogo demanda mais dificuldade de descrição do que as outras classes e deve ser feita de maneira minuciosa pelo professor. O alto índice de falso-negativos faz a medição da sensibilidade, na classe "Jogo", ficar abaixo de 50%, os valores acurácia e preditivo negativo se mantiveram iguais ou um pouco acima de 50%. Na classe "Jogo", mais uma vez, verifica-se que problemas de padronização comprometem os resultados na próxima fase de comparação (comparação de variáveis). Somente a classe "Bola" indica aprovação nas duas fases da análise (padronização da classe e comparação de variáveis). Quanto a hipótese 1, a classe "Bola" também apresenta os melhores resultados quanto as medidas estatísticas.

Como a comparação de variáveis depende diretamente de uma padronização adequada do programa do aluno de acordo com o programa modelo do professor, sugere-se que os problemas de padronização sejam apontados pelo analisador ao aluno e que a segunda fase de comparação de variáveis só seja realizada mediante um resultado de padronização que não apresente reprovações.

**Tabela 6.18: AVALIAÇÃO DOS RESULTADOS QUANTO A PADRONIZAÇÃO E COMPARAÇÃO DE VARIÁVEIS NAS CLASSES DO JOGO**

Classes	Avaliação dos resultados quanto a padronização					
	QUANTO A QUANTIDADE DE CLASSES DE PROGRAMAS			QUANTO A QUANTIDADE DE COMENTÁRIOS		
	Sensibilidade	Acurácia	Preditividade Negativa	Sensibilidade	Acurácia	Preditividade Negativa
“Bola”	78%	80%	33%	78%	83%	60%
“Barra”	- <<não apresentou valores PP e FP>>	30%	30%	- <<não apresentou valores PP e FP>>	22%	22%
“Jogo”	20%	60%	56%	3%	54%	53%
Classes	Avaliação dos resultados quanto a comparação de variáveis					
	QUANTO A QUANTIDADE DE CLASSES DE PROGRAMAS			QUANTO A QUANTIDADE DE COMENTÁRIOS		
	Sensibilidade	Acurácia	Preditividade Negativa	Sensibilidade	Acurácia	Preditividade Negativa
“Bola”	89%	90%	50%	57%	77%	67%
“Barra”	- <<não apresentou valores PP e FP>>	30%	30%	- <<não apresentou valores PP e FP>>	22%	22%
“Jogo”	100%	100%	100%	3%	54%	53%

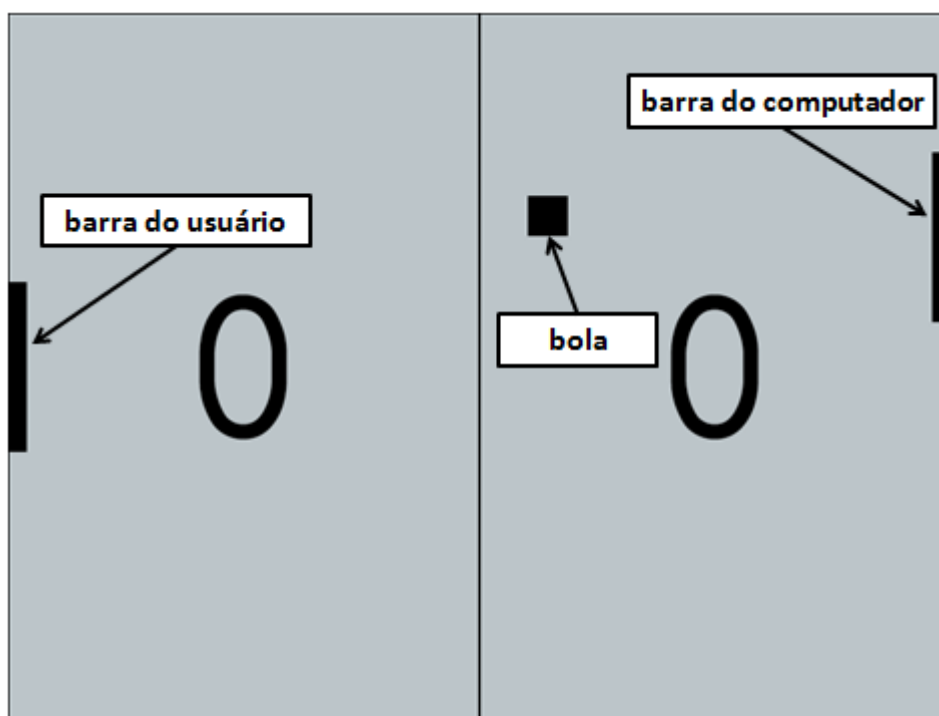
## 6.6 TERCEIRO ESTUDO DE CASO: O JOGO “PING PONG”

O terceiro estudo de caso foi um jogo 2D chamado “*Ping Pong*”, e se concentrou em códigos de programa que são formados por 4 (quatro) classes (“*Bola*”, “*Barra*”, “*Jogo*”, “*Principal*”). Neste caso também, a construção do jogo envolve a necessidade de mais conhecimentos sobre o uso do paradigma de orientação a objetos do que no primeiro caso. Este cenário foi escolhido porque também corresponde ao paradigma de aprendizagem aplicado as disciplinas de “Programação Orientada a Objetos” e “Tópicos Especiais em Desenvolvimento de Software” do curso Técnico em Informática e Tecnólogo em Análise e Desenvolvimento de Sistemas do IFPI.

O jogo “*Ping Pong*” foi desenvolvido como atividade da disciplina “Tópicos Especiais em Desenvolvimento de Software”. O jogo foi desenvolvido durante aproximadamente 3 semanas, onde em cada semana foram realizadas 3 aulas de 50 minutos (9:40 as 12:10). Foi

utilizado laboratório de programação com 24 (vinte e quatro) computadores. Inicialmente os comportamentos de cada uma das classes envolvidas no projeto foram apresentados aos alunos, o enunciado (ver em anexo C) divide as orientações por classe de projeto, assim em cada classe cada um dos comportamentos foi discutido em sala de aula, bem como as maneiras de definir aquele comportamento em termos de implementação (que métodos java e JPlay estariam envolvidos, por exemplo). As classes *"Bola"* e *"Barra"* foram desenvolvidas inicialmente, as classes *"Jogo"* e *"Principal"* foram as últimas classes desenvolvidas.

Segundo o enunciado (ver em anexo C), neste jogo o usuário vai jogar com o computador, ambos devem tentar rebater a bola através de barras, a barra do usuário fica localizada do lado esquerdo da janela e a barra do computador fica localizada ao lado direito da janela do jogo, conforme mostra a Figura 6.4.



**Figura 6.4:** Janela do jogo “Ping Pong” com indicação dos personagens (objetos) do jogo

Assim, as classes *"Bola"* e *"Barra"* definem os respectivos comportamentos dos personagens (objetos) do jogo: a bola, o jogador usuário (representado pela barra do usuário) e o jogador PC (representado pela barra do computador). A barra do jogador usuário será movimentada através do teclado e a barra do jogador PC, será movimentada através do controle do computador.

Segundo o enunciado (ver em anexo C), a classe “*Bola*” possui 3 (três) especificações básicas: deve se mover no eixo “x”, se mover no eixo “y” e verificar se aconteceu colisão entre a bola e as barras. Assim, na definição do modelo do professor, a classe “*Bola*” possui 3 (três) métodos: o primeiro método é o construtor da classe, o segundo método deve realizar o movimento nos eixos “x” e “y” e chamar o terceiro método e o terceiro método que deve verificar se existiu colisão entre a bola e as barras. No trecho de código abaixo seguem os marcadores de padronização da classe:

```
/**
 *
 * @comment A classe Bola deve herdar Sprite, deve ter 3 metodos:o
construtor deve inicializar a figura da bola e inicializar os atributos,
um metodo para mover a bola no eixo x e no eixo y e chamar um metodo que
testa colisao com a barra
 * @method 3
 * @comment 3 metodos devem ser definidos:o construtor deve inicializar a
figura da bola e inicializar os atributos, um metodo para mover a bola no
eixo x e no eixo y e chamar um metodo que testa colisao com a barra
 * @inheritance Sprite
 * @comment A classe Bola deve herdar Sprite
 * @keyboard false
 * @comment Nao existe controle atraves do teclado para objetos do tipo
Bola
 * @comment realize o movimento no eixo x
 * @movey true
 * @comment realize o movimento no eixo y
 *
 */
public class Bola extends Sprite{
.....
}
```

A classe “*Barra*” também possui 2 (duas) especificações: a classe deve controlar o movimento da barra do computador (ela deve se mover no eixo “y”, para os lados esquerdo e direito, seguindo o movimento da bola) e deve controlar a pontuação dos jogadores (atualização da pontuação da barra do usuário e da barra do computador). Assim, na definição do modelo do professor, a classe “*Barra*” possui 3 (três) métodos, o construtor da classe, que deve inicializar a figura da barra e sua pontuação, um método para mover a barra do computador no eixo do y (regra: o método deve mover a barra do computador sempre que a bola ultrapassar a metade da posição da janela do jogo no eixo x, ou seja, quando a posição da bola no eixo do x for igual ou superior a 400) e um método que incrementa a pontuação do

jogador. No trecho de código abaixo seguem os marcadores de padronização da classe “Barra”:

```
/**
 *
 * @comment A classe Barra deve herdar de Sprite, no jogo ping pong dois
 jogadores do tipo da classe Barra serao criados, um sera controlado pelo
 usuario e outro sera controlado pelo computador. A classe Barra entao deve
 possuir 3 metodos: o construtor deve inicializar a imagem do e sua
 pontuacao, um metodo para mover a barra do computador no eixo y e um
 metodo que incrementa a pontuacao do jogador (um dos jogadores)
 * @method 3
 * @comment A classe Barra entao deve possuir 3 metodos: o construtor deve
 inicializar a imagem da barra e sua pontuacao, um metodo para mover a
 barra do computador no eixo y e um metodo que incrementa a pontuacao do
 jogador (um dos jogadores)
 * @inheritance Sprite
 * @comment A classe Barra deve herdar de Sprite
 * @keyboard false
 * @comment Nao existe controle do objeto do tipo da classe Barra atraves
 do teclado, o movimento da barra do usuario deve ser feito na classe Jogo,
 atraves do metodo movey() do JPlay
 */
```

A classe “Jogo” define comportamento do “*gameloop*” do jogo. Como já visto no segundo estudo de caso do jogo “*BrickBreak*”, a classe que contém o “*gameloop*” define o projeto central do jogo, nesta classe os objetos devem ser instanciados e interagir entre si de acordo com as regras estabelecidas pelo jogo. No trecho de código abaixo seguem os marcadores de padronização da classe “Jogo”:

```
/**
 *
 * @comment voce deve criar 7 objetos: 2(dois) do tipo Barra, sendo 1 (um)
 o jogador usuario e o outro jogador computador, 1 (um) do tipo Bola para
 ser a bola, 1 (um) do tipo Window que sera a janela do jogo, 1 (um) do
 tipo GameImage que sera o background do janela, um do tipo Keyboard que
 ira captar o teclado, um do tipo Font que sera a fonte personalizada usada
 no game
 * @method 5
 * @comment O construtor que chama o metodo init e o metodo loop, o metodo
 init() inicializa todos os atributos e objetos,o metodo loop() contem o
 loop infinito do jogo,o metodo desenha() que desenha todas as imagens na
 tela,o metodo pontua() atualizacao a pontuacao dos jogadores
 * @gameobject 3
 * @comment A classe deve definir 3 gameobjects: 2 jogadores do tipo
 Barra, o jogador usuario e o jogador computador e uma bola (tipo Bola)
 * @window
```

```
* @comment Voce deve atualizar a janela do jogo atraves do metodo
update() do jplay
*
*/
```

A classe “*Principal*” define somente a instanciação de um objeto do tipo da classe “*Jogo*”, iniciando assim a execução do programa. Portanto, por ser uma classe de um único comportamento, a classe “*Principal*” não foi inserida nesta análise de resultados.

Na análise deste capítulo sobre o jogo “*Ping Pong*” são verificados os resultados da classe “*Bola*”, “*Barra*” e “*Jogo*”. Como é o segundo jogo apresentado, onde novamente existe a situação de um projeto formado por mais de uma classe, nesta análise foi retirada a análise detalhada do jogo e somente os resultados quanto às análises gerais das classes são apresentados. Cada classe é avaliada quanto a padronização do modelo, comparação de pares de variáveis (árvores de comportamento) e resultados dos comentários das variáveis sem par.

Entre os 10 (dez) programas analisados, os estudantes estão com o comportamento geral do programa incorreto na primeira tentativa de desenvolver o exercício, conforme descrito na Tabela 6.19 abaixo.

**Tabela 6.19: RELAÇÃO DOS PROGRAMAS COM COMPORTAMENTO GERAL INCORRETOS**

ESTUDANTE	COMPORTAMENTO DO JOGO INCORRETO
3	Não finaliza o jogo quando a tecla ESC é pressionada pelo usuário.
4	Não finaliza o jogo quando a tecla ESC é pressionada pelo usuário. Não testa a posição da bola no eixo x para realizar o movimento da barra do computador.
9	Não finaliza o jogo quando a tecla ESC é pressionada pelo usuário. Não testa a posição da bola no eixo x para realizar o movimento da barra do computador.
10	Não testa a posição da bola no eixo x para realizar o movimento da barra do computador.

### 6.6.1 RESULTADOS OBTIDOS APARTIR DA MEDIÇÃO DE SENSIBILIDADE, ESPECIFICIDADE, ACURÁCIA E PRECISÃO

Da mesma forma que o estudo de caso 1, foram verificados os dois tipos de resultados: o primeiro baseado por unidade de programa analisado e o segundo baseado na quantidade de comentários apresentados por programa. As Tabelas 6.20 e 6.21 apresentam os resultados das medidas de eficiência obtidas através dos dois resultados.

**Tabela 6.20: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE CLASSES DE PROGRAMA ANALISADAS PARA O SEGUNDO ESTUDO DE CASO**

<i>Projeto "BrickBreak"</i>	<i>Medidas de eficácia do sistema analisador de acordo com o total de programas analisados</i>		
	<i>Quantidades e Medidas Obtidas</i>	<i>De acordo com a Padronização do Modelo</i>	<i>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</i>
<i>Classe "Bola"</i>	<i>NN</i>	0	0
	<i>FN</i>	1	7
	<i>PP</i>	9	3
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	90%	30%
	<i>Especificidade</i>	-	-
	<i>Acurácia</i>	90%	30%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	-	-
<i>Classe "Barra"</i>	<i>NN</i>	0	3
	<i>FN</i>	6	3
	<i>PP</i>	4	4
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	40%	57%
	<i>Especificidade</i>	-	100%
	<i>Acurácia</i>	40%	70%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	-	50%
<i>Classe "Jogo"</i>	<i>NN</i>	3	3
	<i>FN</i>	6	0
	<i>PP</i>	1	7



	<b>FP</b>	0	0
	<b>Sensibilidade</b>	14%	100%
	<b>Especificidade</b>	100%	100%
	<b>Acurácia</b>	40%	100%
	<b>Preditividade Positiva</b>	100%	100%
	<b>Preditividade Negativa</b>	33%	100%

**Tabela 6.21: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O SEGUNDO ESTUDO DE CASO**

<b>Projeto</b> <b>"Ping Pong"</b>	<b>Medidas de eficácia do sistema analisador de acordo com o total de comentários analisados</b>		
<b>Classe "Bola"</b>	<b>Quantidades e Medidas Obtidas</b>	<b>De acordo com a Padronização do Modelo</b>	<b>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</b>
	<b>NN</b>	0	0
	<b>FN</b>	1	13
	<b>PP</b>	9	3
	<b>FP</b>	0	0
	<b>Sensibilidade</b>	90%	20%
	<b>Especificidade</b>	-	-
	<b>Acurácia</b>	90%	20%
	<b>Preditividade Positiva</b>	100%	100%
	<b>Preditividade Negativa</b>	-	-
<b>Classe "Barra"</b>	<b>NN</b>	0	3
	<b>FN</b>	11	3
	<b>PP</b>	4	4
	<b>FP</b>	0	0
	<b>Sensibilidade</b>	27%	57%
	<b>Especificidade</b>	-	100%
	<b>Acurácia</b>	27%	70%
	<b>Preditividade Positiva</b>	100%	100%
	<b>Preditividade Negativa</b>	-	50%
<b>Classe "Jogo"</b>	<b>NN</b>	3	3
	<b>FN</b>	7	0
	<b>PP</b>	1	7
	<b>FP</b>	0	0

	<i>Sensibilidade</i>	13%	100%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	36%	100%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	30%	100%

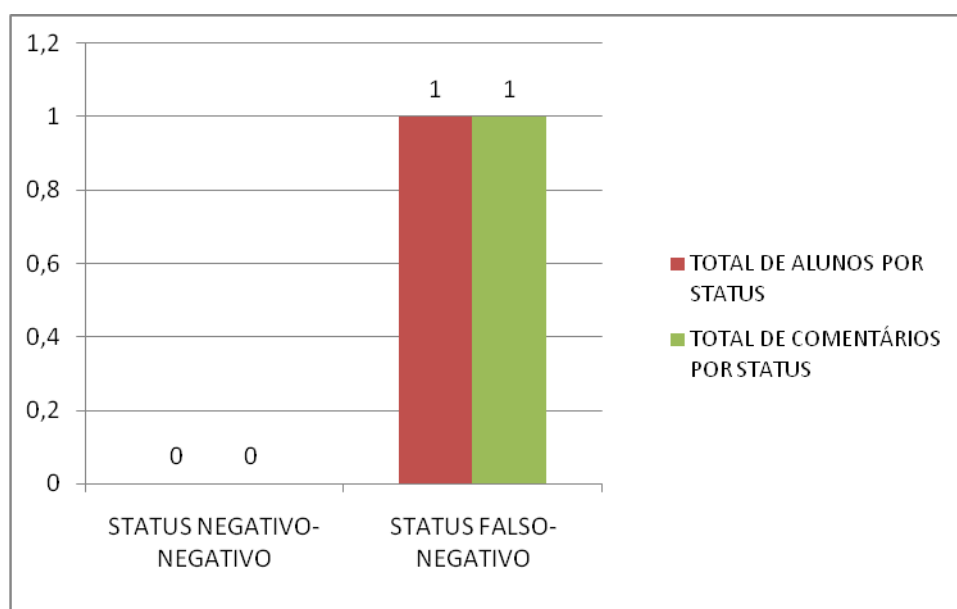
## 6.6.2 RESULTADOS OBTIDOS APARTIR DAS TRÊS HIPÓTESES PROPOSTAS

### 6.6.2.1 ANÁLISE GERAL QUANTO A PADRONIZAÇÃO DO MODELO

As classes serão analisadas individualmente, assim considera-se, para efeito de avaliação de uma classe, que o comportamento individual da classe está correto ou incorreto, de forma que problemas que forem localizados em outras classes não serão considerados na avaliação individual.

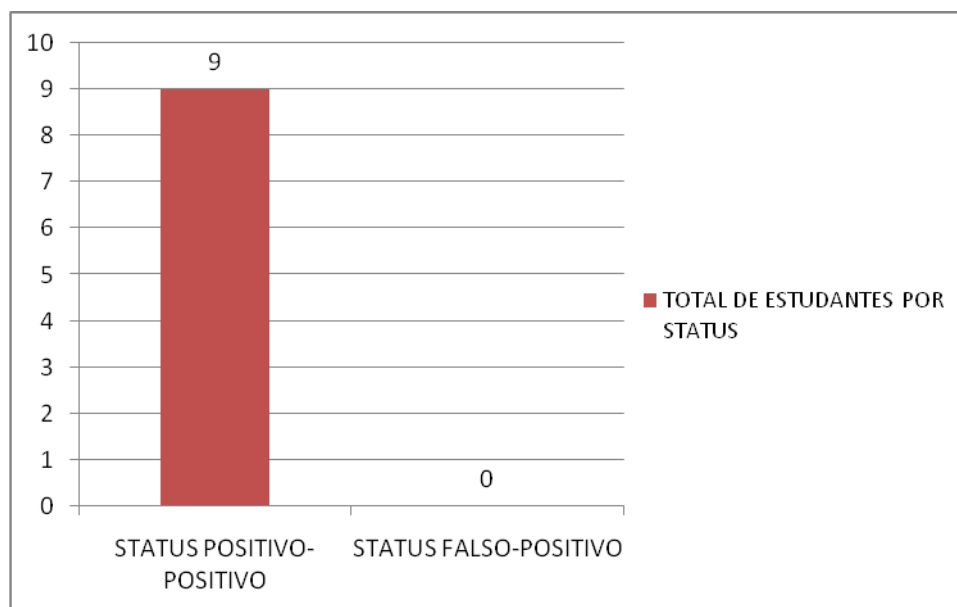
#### 6.6.2.1.1 PADRONIZAÇÃO DO MODELO: CLASSE “BOLA”

A análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo negativo-negativo (PNN) foi de 0% e o percentual de precisão do tipo falso-negativo (PFN) foi de 100%. Portanto, concluindo  $PNN < PFN$ , e para o caso da classe “Barra” não é possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.12.



**Gráfico 6.12: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Bola” do jogo “Ping Pong”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.

De acordo com o Gráfico 6.13, a análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo positivo-positivo (PPP) foi de 100% e o percentual de precisão do tipo falso-positivo (PFP) foi de 0% . Verificamos, neste caso, que  $PPP > PFP$ . Portanto é possível aceitar a hipótese 2 como verdadeira.



**Gráfico 6.13: Distribuição de mensagens tipo positivo-positivo e falso-positivo nos códigos da classe “Bola” do jogo “Ping Pong”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.

A Tabela 6.22 mostra a distribuição detalhada das mensagens de padronização na classe “Bola”. Segundo o resultado da padronização, somente o estudante 1, apesar de apresentar o comportamento da classe “Bola” e o comportamento do programa corretos, não efetuou a padronização da classe “Bola” conforme foi solicitado. O estudante declarou a quantidade de métodos diferentes da solicitada na padronização. Na classe “Jogo” do modelo do professor o marcador “@method” é igual a 3, no entanto o estudante declarou somente 2 métodos. Ver trecho de código abaixo correspondente a padronização no programa modelo:

```
* @method 3
* @comment A classe Bola deve possuir 3 metodos: 1 - o construtor deve
inicializar a figura da bola e inicializar os atributos, 2- um metodo para
mover a bola no eixo x e no eixo y e chamar o metodo que testa colisao da
bola com as barras, 3- um metodo para testar a colisao da bola com as
barras
```

De acordo com o código acima, o terceiro método deveria testar a colisão da bola com as barras do jogo, o estudante 1 não definiu este método, mas realizou o teste de colisão com

as barras na classe “*Jogo*”, o que modificou a padronização, mas não comprometeu o comportamento correto do programa, por isso a mensagem originada da diferença entre os marcadores “*@method*” foi classificada como falso-negativo. Neste caso aconteceu que, para o restante dos alunos, a quantidade de métodos estavam corretas, no entanto, somente os alunos 2 e 3 definiram o terceiro método como foi solicitado pela padronização. Os alunos 4, 6, 7 e 10 definiram um método a mais (um método para mover a bola no eixo x e outro para mover a bola no eixo y, sendo que a padronização pedia em somente um método -segundo método- o controle do eixo x e y) e não fizeram nenhum teste de colisão. Logo a padronização para os estudantes 6, 7 e 10, apesar de não emitir mensagens, está diferente da padronização do modelo. Foram classificados como positivo-positivo porque, apesar de diferente da padronização, eles realizam o teste de colisão na classe “*Jogo*”, o que faz com que o comportamento correto da classe e do jogo não seja comprometido.

O estudante 5 definiu também um método a mais (um método para mover a bola no eixo x e outro para mover a bola no eixo y, sendo que a padronização pedia em somente um método -segundo método- o controle do eixo x e y), mas neste caso, ele realizou o teste de colisão com as barras dentro do método responsável por mover a bola no eixo do x. Logo a padronização para os estudantes 5, apesar de não emitir mensagens, também está diferente da padronização do modelo. Ele foi classificado como positivo-positivo porque, apesar de diferente da padronização, ele realizou o teste de colisão em outro método da classe “*Bola*”, o que faz com que o comportamento correto da classe e do jogo não seja comprometido.

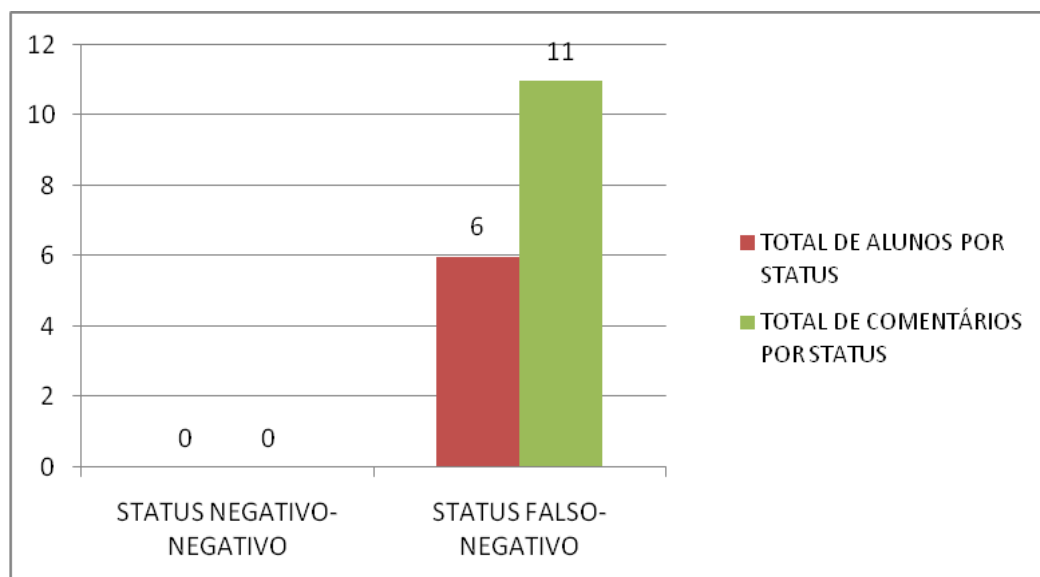
**Tabela 6.22: AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BOLA” DE ACORDO COM A PADRONIZAÇÃO**

Estudante	Avaliação de comentários da classe “Jogo” de acordo com a Padronização do Modelo		
	<i>Status do programa</i>	<i>Marcador relacionado ao comentário</i>	<i>Status do comentário</i>
1	Correto	“ <i>method</i> ”	Tipo Falso-Negativo

#### 6.6.2.1.2 PADRONIZAÇÃO DO MODELO: CLASSE “BARRA”

A análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo negativo-negativo (PNN) foi de 0% e o percentual de precisão do tipo falso-

negativo (PFN) foi de 100%. Portanto, concluindo  $PNN < PFN$ , e para o caso da classe “Barra” não é possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.14.



**Gráfico 6.14:** Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Barra” do jogo “Ping Pong”, de acordo com a padronização  
Fonte: Pesquisa direta, 2015.

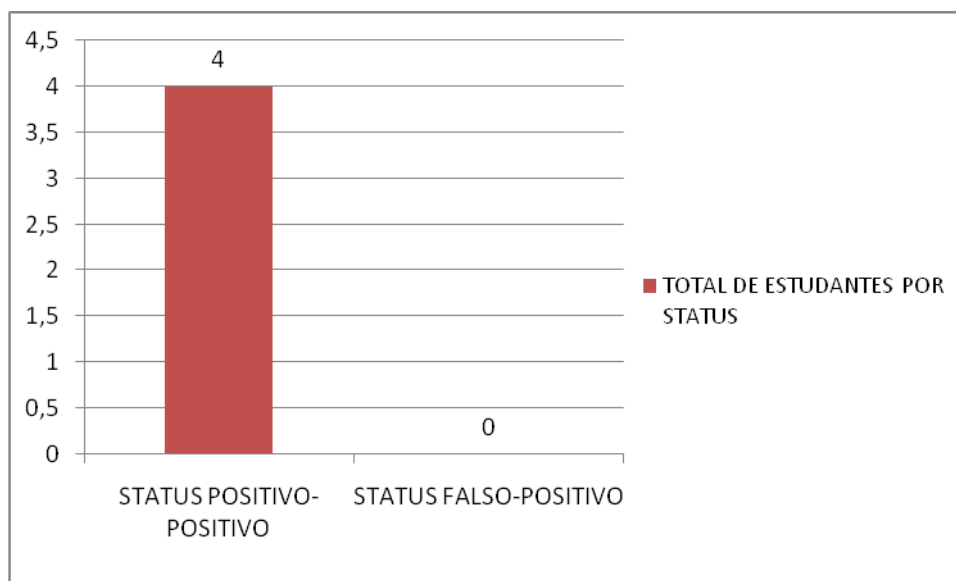
De acordo com o Gráfico 6.15, a análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo positivo-positivo (PPP) foi de 100% e o percentual de precisão do tipo falso-positivo (PFP) foi de 0% . Verificamos, neste caso, que  $PPP > PFP$ . Portanto é possível aceitar a hipótese 2 como verdadeira.

A Tabela 6.23 mostra a distribuição detalhada das mensagens de padronização na classe “Barra”. Os estudantes 1, 5, 7 e 8 e 10 apresentam diferenças nos marcadores “@method” e “@keyboard”. Para o marcador “@method”, a padronização solicita a definição de 3 (três) métodos. Ver no código abaixo:

```
* @method 3
* @comment A classe Barra deve possuir 3 metodos: 1- o construtor deve
inicializar a imagem da barra e sua pontuacao, 2- um metodo para mover a
barra do computador no eixo y e 3- um metodo que incrementa a pontuacao
das barras (um dos jogadores)
```

Para o marcador “@keyboard”, a padronização define que não existirá controle de objetos através do teclado na classe “Barra”. Ver no código abaixo:

```
* @keyboard false
* @comment Nao existe controle do objeto do tipo Barra atraves do teclado,
o movimento da barra do usuario deve ser feito na classe jogo, atraves do
metodo movey() do JPlay
```



**Gráfico 6.15: Distribuição de mensagens tipo positivo-positivo e falso-positivo nos códigos da classe “Barra” do jogo “Ping Pong”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.

**Tabela 6.23: AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “BARRA” DE ACORDO COM A PADRONIZAÇÃO**

Estudante	Avaliação de comentários da classe “Jogo” de acordo com a Padronização do Modelo		
	Status do programa	Marcador relacionado ao comentário	Status do comentário
1	Correto	“method”	Tipo Falso-Negativo
		“keyboard”	Tipo Falso-Negativo
5	Correto	“method”	Tipo Falso-Negativo
	Correto	“keyboard”	Tipo Falso-Negativo
6	Correto	“keyboard”	Tipo Falso-Negativo
7	Correto	“method”	Tipo Falso-Negativo
		“keyboard”	Tipo Falso-Negativo
8	Correto	“method”	Tipo Falso-Negativo
		“keyboard”	Tipo Falso-Negativo
10	Incorreto	“method”	Tipo Falso-Negativo
		“keyboard”	Tipo Falso-Negativo

Os estudantes 1, 5, 7, 8 e 10 definiram um método a mais em suas classes, gerando uma diferença no marcador “@method”(no programa dos estudantes: @method = 4). O método a mais definido por eles implementa o movimento da barra através do teclado usando o método *keydown()* do JPlay, o que causa a diferença no marcador “@keyboard” (no programa dos estudantes: @keyboard = true). Assim, as mensagens armazenadas nos marcadores “@comment” relacionados a “@method” e “@keyboard” são impressas:

- *A classe Barra deve possuir 3 metodos: 1- o construtor deve inicializar a imagem da barra e sua pontuacao, 2- um metodo para mover a barra do computador no eixo y e 3- um metodo que incrementa a pontuacao das barras (um dos jogadores)*
- *Nao existe controle do objeto do tipo Barra atraves do teclado, o movimento da barra do usuario deve ser feito na classe jogo, atraves do metodo movey() do JPlay*

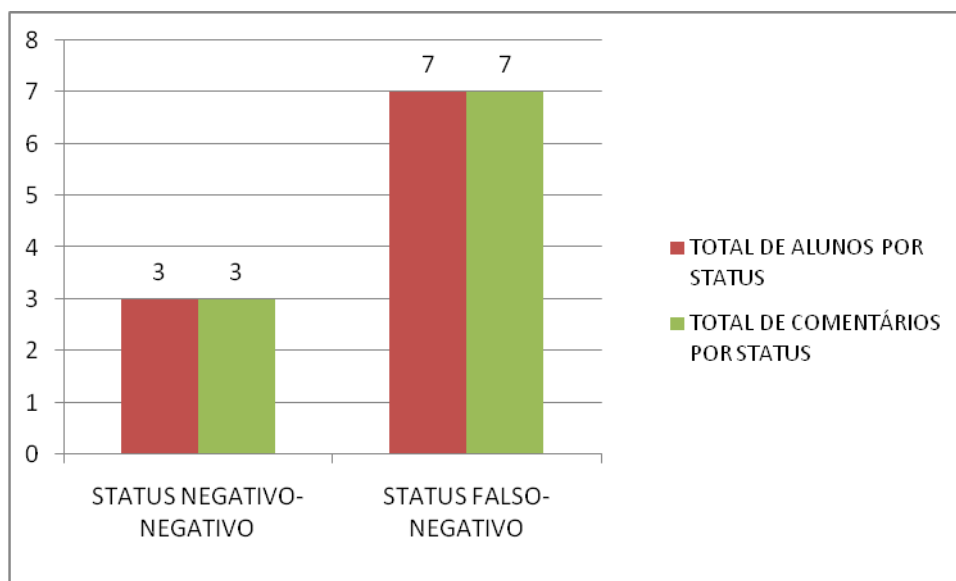
As classes dos estudantes não estão de acordo com a padronização do modelo, mas as diferenças não comprometeram o comportamento correto do programa, por isso as mensagens foram classificadas como falso-negativas.

O estudante 6 apresentou diferença apenas no marcador “@keyboard”. Ele também define um método que implementa o movimento da barra através do teclado usando o método *keydown()* do JPlay, o que causa a diferença no marcador “@keyboard” (no programa do estudante: @keyboard = true). Neste caso a quantidade de métodos no programa do estudante é igual a quantidade de métodos no programa modelo (@method = 3), pois o estudante não definiu o terceiro método solicitado no modelo (o método que incrementa a pontuação das barras).

A classe do estudante 6 também não está de acordo com a padronização do modelo, porém as diferenças encontradas não comprometeram o comportamento correto do programa, por isso a mensagem gerada foi classificada como falso-negativo.

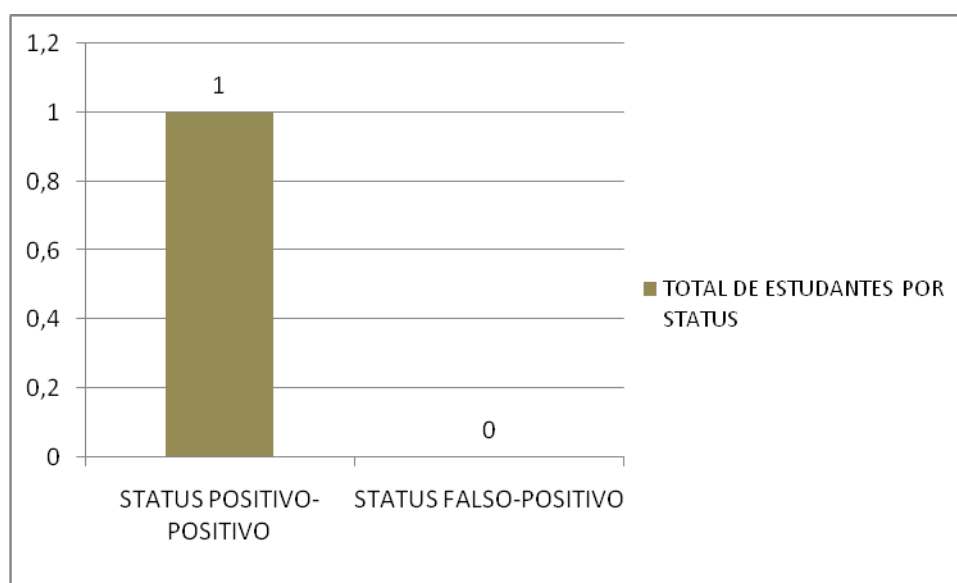
#### **6.6.2.1.3 PADRONIZAÇÃO DO MODELO: CLASSE “JOGO”**

A análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo negativo-negativo (PNN) foi de 30% e o percentual de precisão do tipo falso-negativo (PFN) foi de 70%. Portanto, concluindo  $PNN < PFN$ , e para o caso da classe “Jogo” não é possível aceitar a hipótese 1 como verdadeira. Ver Gráfico 6.16.



**Gráfico 6.16: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe “Jogo” do jogo “Ping Pong”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.

De acordo com o Gráfico 6.17, a análise constatou que, de acordo com a padronização do modelo, o percentual de precisão do tipo positivo-positivo (PPP) foi de 100% e o percentual de precisão do tipo falso-positivo (PFP) foi de 0% . Verificamos, neste caso, que  $PPP > PFP$ . Portanto é possível aceitar a hipótese 2 como verdadeira.



**Gráfico 6.17: Distribuição de mensagens tipo positivo-positivo e falso-positivo nos códigos da classe “Jogo” do jogo “Ping Pong”, de acordo com a padronização**  
 Fonte: Pesquisa direta, 2015.



A Tabela 6.24 mostra a distribuição detalhada das mensagens de padronização na classe “*Jogo*”. Os estudantes 1, 5, 6, 7, 8 e 10 não efetuaram a padronização da classe “*Jogo*” conforme foi solicitado. Os estudantes declararam quantidade de métodos diferentes da solicitada na padronização. Na classe “*Jogo*” do modelo do professor o marcador “*@method*” é igual a 3, indicando que a classe deve possuir 3 (três) métodos definidos. O analisador calcula valores diferentes para os estudantes 1, 5, 6, 7, 8 e 10, e isto gera diferenças na comparação, fazendo com que seja impressa a mensagem armazenada no marcador “*@comment*” associado. Ver trecho de código abaixo correspondente:

```
* @method 5
* @comment A classe Jogo deve possuir 6 metodos: 1- O construtor que
chama o metodo init e o metodo loop, 2- O metodo init que inicializa
todos os atributos e objetos, 3- O metodo que contem o loop infinito
do jogo, 4- O metodo desenha() que desenha todas as imagens na
janela do jogo, 5- O metodo pontua() que atualiza a pontuacao dos
jogadores
```

Os estudante 3, 4 e 9 apresentaram o comportamento do programa incorreto, pois não implementaram a finalização do jogo quando o usuário pressionasse a tecla ESC, esta padronização é indicada no programa modelo através do marcador “*@keyboard*” . Ver trecho de código correspondente:

```
* @keyboard true
* @comment O jogo deve ser encerrado quando o usuario pressionar a
tecla ESC (DICA: use o metodo KeyDown() da classe keyboard do
JPlay).
```

O analisador encontra a diferença entre os valores “*@keyboard*” do programa modelo e do programa do aluno e imprime o marcador a mensagem armazenada no “*@comment*” associado:

- *O jogo deve ser encerrado quando o usuario pressionar a tecla ESC (DICA: use o metodo KeyDown() da classe keyboard do JPlay).*

Como a mensagem está relacionada ao comportamento incorreto do jogo, ela é classificada como Negativo-Negativo.

O estudante 9 também apresenta uma diferença no marcador “@method”, a padronização solicita a definição 5 (cinco) métodos e o estudante define somente 1 (um) método na classe. Como a diferença relacionada a “@method” não modificou o comportamento correto do programa, a mensagem foi classificada como Falso-Negativo.

**Tabela 6.24: AVALIAÇÃO DOS COMENTÁRIOS DA CLASSE “JOGO” DE ACORDO COM A PADRONIZAÇÃO**

Estudante	Avaliação de comentários da classe “Jogo” de acordo com a Padronização do Modelo		
	Status do programa	Marcador relacionado ao comentário	Status do comentário
1	Correto	“method”	Tipo Falso-Negativo
3	Incorreto	“keyboad”	Tipo Negativo-Negativo
4	Incorreto	“keyboad”	Tipo Negativo-Negativo
5	Correto	“method”	Tipo Falso-Negativo
6	Correto	“method”	Tipo Falso-Negativo
7	Correto	“method”	Tipo Falso-Negativo
8	Correto	“method”	Tipo Falso-Negativo
9	Incorreto	“method”	Tipo Falso-Negativo
		“keyboad”	Tipo Negativo-Negativo
10	Incorreto	“method”	Tipo Falso-Negativo

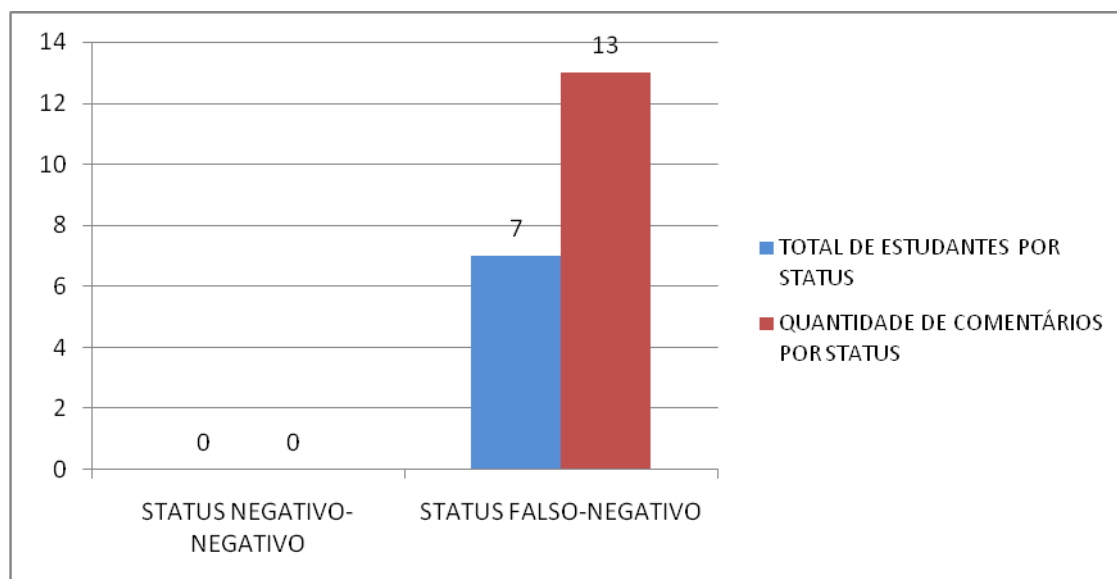
De acordo com a Tabela 6.24 é possível verificar que os estudantes 1, 5, 6, 7, 8 e 10, apesar de possuírem o comportamento geral do programa correto, não seguiram a padronização estabelecida pelo modelo gerando comentários falso-negativos e, consequentemente, comprometendo a execução da próxima fase da análise, a de comparação de variáveis

#### 6.6.2.2 ANÁLISE GERAL QUANTO A DIFERENÇA DE COMPORTAMENTOS

Neste item são verificadas as análises geral e detalhada das 3 (três) classes do jogo: “Bola”, “Barra” e “Jogo”. Os 2 (dois) tipo de análise quanto a diferença de comportamentos (análise de acordo com a comparação de par de variáveis e análise de variáveis sem par) são reunidos em um único gráfico de resultados. A seguir, para cada uma das classes, seguem as análises:

### 6.6.2.2.1 ANÁLISE QUANTO A DIFERENÇAS DE COMPORTAMENTO: CLASSE “BOLA”

De acordo com o Gráfico 6.18, com a comparação de par de variáveis (árvores de comportamento), o percentual de comentários do tipo Negativo-Negativo (PNN) foi de 0% e o percentual de comentários do tipo Falso-Negativo (PFN) foi de 100% .



**Gráfico 6.18: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe "Bola" do jogo "Ping Pong", de acordo com a comparação de par de variáveis e variáveis sem par**

Fonte: Pesquisa direta, 2015.

Portanto, no caso da classe "Bola" conclui-se que  $PNN < PFN$ , portanto não é possível aceitar a hipótese 1 como verdadeira.

A Tabela 6.25 mostra a avaliação detalhada dos comentários obtidos a partir da comparação de par de variáveis e variáveis sem par. Os comentários 1, 2, 3 e 4 foram gerados por variações de programação, as diferenças detectadas neste caso não causaram modificação no comportamento correto do programa, portanto foram classificados como Falso-Negativos.

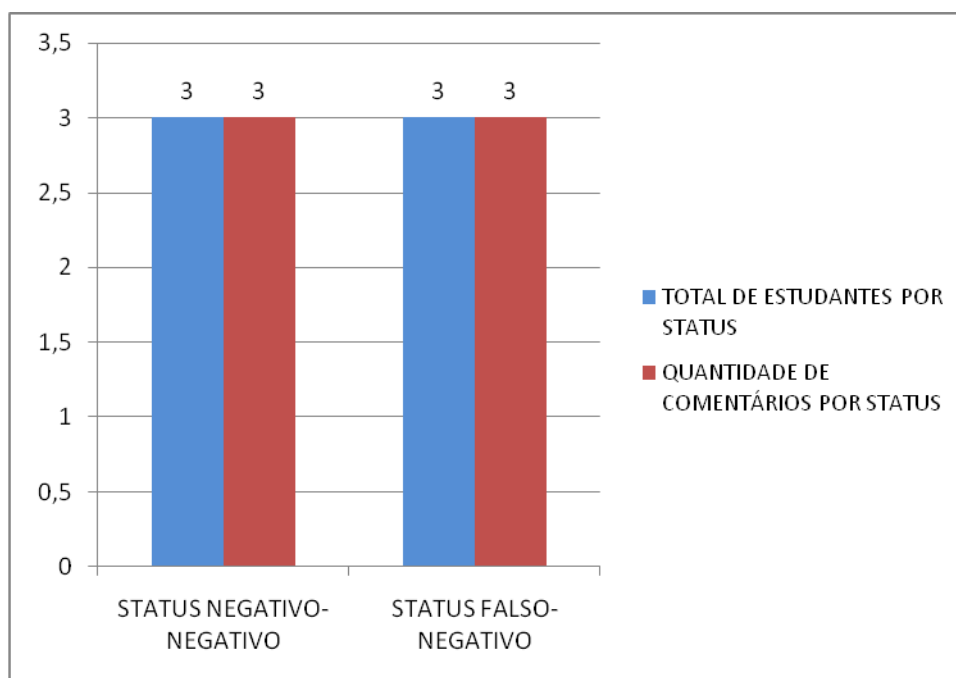
O comentário 6 está relacionado ao teste de colisão da bola com as barras, o teste realmente não foi realizado pelo alunos na classe "Bola". No entanto, eles realizaram o teste de colisão na classe "Jogo", assim, o comportamento geral do jogo não se modificou, portanto as ocorrências do comentário 6 foram classificadas como Falso-Negativas.

**Tabela 6.25: AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BOLA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR**

Estudante	Avaliação dos comentários na classe "Bola" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
1	Correto	<Sem comentários>	Tipo Positivo-Positivo
2	Correto	<Sem comentários>	Tipo Positivo-Positivo
3	Incorreto	<Sem comentários>	Tipo Positivo-Positivo
4	Incorreto	3	Tipo Falso-Negativo
5	Correto	6	Tipo Falso-Negativo
6	Correto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		3	Tipo Falso-Negativo
		4	Tipo Falso-Negativo
		6	Tipo Falso-Negativo
7	Correto	6	Tipo Falso-Negativo
8	Correto	6	Tipo Falso-Negativo
9	Incorreto	1	Tipo Falso-Negativo
		2	Tipo Falso-Negativo
		6	Tipo Falso-Negativo
10	Incorreto	6	Tipo Falso-Negativo

#### 6.6.2.2.2 ANÁLISE QUANTO A DIFERENÇAS DE COMPORTAMENTO: CLASSE “BARRA”

De acordo com o Gráfico 6.19, com a comparação de par de variáveis (árvores de comportamento), o percentual de comentários do tipo Negativo-Negativo (PNN) foi de 50% e o percentual de comentários do tipo Falso-Negativo (PFN) foi de 50%. Portanto, no caso da classe "Barra" conclui-se que  $PNN=PFN$ , portanto não é possível aceitar a hipótese 1 como verdadeira.



**Gráfico 6.19: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe "Barra" do jogo "Ping Pong", de acordo com a comparação de par de variáveis e variáveis sem par**

Fonte: Pesquisa direta, 2015.

A Tabela 6.26 mostra a avaliação detalhada dos comentários obtidos a partir da comparação de par de variáveis e variáveis sem par. Os estudantes 6, 7 e 8 foram gerados por variações de programação, as diferenças detectadas neste caso não causaram modificação no comportamento correto do programa, portanto foram classificados como Falso-Negativos. Os estudantes 4, 9 e 10 não realizam o teste da posição da bola no eixo x ("*bola.x* > 400") para realizar o movimento da barra do computador, isso gera a variável sem par ("*bola.x*") no programa modelo, e o analisador imprime o comentário 2 associado ao comportamento "*conditional*", ver trecho de código no programa modelo:

```
/*
comment02: Criar um metodo responsavel por mover a barra inimiga sempre
que a bola ultrapassar a metade da posicao da janela do jogo no eixo x
(x>400)
*/
if(ball.x > 400){
    if (ball.y > this.y)
        this.y++;
    else this.y--; }
```

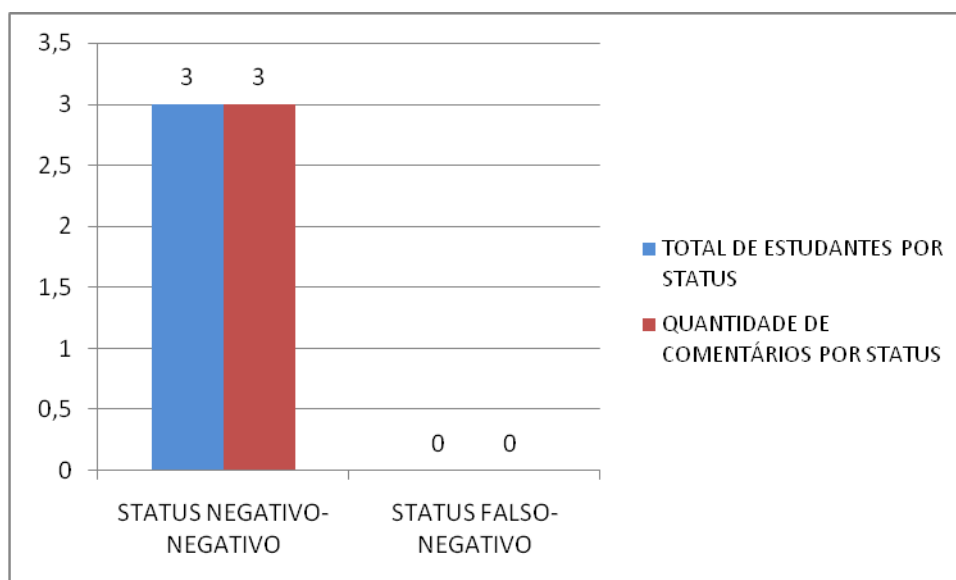
Portanto, para os casos dos estudantes 4, 9 e 10, o comentário 2 está associado a um problema real no comportamento do jogo, por isso ele foi classificado como Negativo-Negativo.

**Tabela 6.26: AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “BARRA” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR**

Estudante	Avaliação dos comentários na classe "Barra" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
1	Correto	<Sem comentários>	Tipo Positivo-Positivo
2	Correto	<Sem comentários>	Tipo Positivo-Positivo
3	Incorreto	<Sem comentários>	Tipo Positivo-Positivo
4	Incorreto	2	Tipo Negativo-Negativo
5	Correto	<Sem comentários>	Tipo Positivo-Positivo
6	Correto	2	Tipo Falso-Negativo
7	Correto	2	Tipo Falso-Negativo
8	Correto	2	Tipo Falso-Negativo
9	Incorreto	2	Tipo Negativo-Negativo
10	Incorreto	2	Tipo Negativo-Negativo

#### 6.6.2.2.3 ANÁLISE QUANTO A DIFERENÇAS DE COMPORTAMENTO: CLASSE “JOGO”

De acordo com o Gráfico 6.20, com a comparação de par de variáveis (árvores de comportamento), o percentual de comentários do tipo Negativo-Negativo (PNN) foi de 100% e o percentual de comentários do tipo Falso-Negativo (PFN) foi de 0% . Portanto, no caso da classe "Jogo" conclui-se que  $PNN > PFN$ , portanto é possível aceitar a hipótese 1 como verdadeira.



**Gráfico 6.20: Distribuição de mensagens tipo negativo-negativo e falso-negativo nos códigos da classe "Jogo" do jogo "Ping Pong", de acordo com a comparação de par de variáveis e variáveis sem par**

A Tabela 6.27 mostra a avaliação detalhada dos comentários obtidos a partir da comparação de par de variáveis e variáveis sem par. Os estudantes 3, 4 e 9 não implementaram a finalização do jogo quando a tecla ESC é pressionada pelo usuário. Isto é implementado no programa modelo através de um comportamento "conditional" de uma variável do tipo "Keyboard". Como os estudantes não realizam este comportamento, o analisador encontra uma diferença no comportamento "conditional" entre a variável do tipo "Keyboard" do programa do aluno e seu par do tipo "Keyboard" do programa modelo (par de variáveis <tipo "Keyboard", tipo "Keyboard">. Ver código relacionado no programa modelo:

```
/*
comment05: se a tecla ESC for pressionada, encerre o jogo
*/
if(keyboard.keyDown(Keyboard.ESCAPE_KEY)){
    window.exit();
}
```

Assim, o seguinte comentário relacionado, é apresentado ao aluno:

- "comment05: se a tecla ESC for pressionada, encerre o jogo"

Portanto, nos casos dos estudantes 3, 4 e 9, o comentário 5 está associado a um problema real no comportamento do jogo, por isso ele foi classificado como Negativo-Negativo.

**Tabela 6.27: AVALIAÇÃO DOS COMENTÁRIOS NA CLASSE “JOGO” OBTIDOS A PARTIR DA COMPARAÇÃO DE PAR DE VARIÁVEIS E VARIÁVEIS SEM PAR**

Estudante	Avaliação dos comentários na classe "Jogo" obtidos a partir da comparação de par de variáveis e variáveis sem par		
	Status do programa	Identificador do comentário	Status do comentário
1	Correto	<Sem comentários>	Tipo Positivo-Positivo
2	Correto	<Sem comentários>	Tipo Positivo-Positivo
3	Incorreto	5	Tipo Negativo-Negativo
4	Incorreto	5	Tipo Negativo-Negativo
5	Correto	<Sem comentários>	Tipo Positivo-Positivo
6	Correto	<Sem comentários>	Tipo Positivo-Positivo
7	Correto	<Sem comentários>	Tipo Positivo-Positivo
8	Correto	<Sem comentários>	Tipo Positivo-Positivo
9	Incorreto	5	Tipo Negativo-Negativo
10	Incorreto	<Sem comentários>	Tipo Positivo-Positivo

## 6.7 CONSIDERAÇÕES GERAIS SOBRE O TERCEIRO ESTUDO DE CASO

Na análise de acordo com a padronização todas as classes foram reprovadas, demonstrando problemas na padronização dos programas, conforme mostra a Tabela 6.28. Todas as diferenças detectadas na padronização descrevem uma situação real no programa do estudante, no entanto todas as mensagens nas classes "Bola" e "Barra" foram classificadas como Falso-Negativas porque a diferença na padronização não modificou o comportamento geral do programa, diferenças que interferiram no comportamento geral do programa só aconteceram somente na classe "Jogo". Se modificarmos os critérios de classificação Falso-Negativo e Negativo-Negativo e considerarmos as mensagens que apontam para problemas de padronização reais e para diferenças reais entre pares de variáveis como Negativo-Negativo, mesmo que isso não modifique o comportamento correto do programa, e considerarmos como Falso-Negativo as mensagens que apontam para diferenças incorretas de padronização e pares de variáveis (a avaliação do analisador não mostra o verdadeiro estado do código do programa) teremos uma situação conforme a Tabela 6.29. Para as classe "Barra" e "Jogo" os resultados continuam os mesmos. No entanto, modificando o critério, o resultado da classe



"Bola" foi alterado. O motivo é que 6 (seis) estudantes apresentam o comentário 6, relacionado ao comportamento *"conditional"* que testa a colisão da bola com uma das barras do jogo, e nenhum dos 6 (seis) estudantes realizou esta verificação realmente em seus códigos da classe *"Bola"*. Ver código do programa modelo relacionado:

```
public void colision(Sprite player1, Sprite player2){
    /*
    comment06: Criar um metodo responsavel por testar a colisao da bola com as
    barras do jogo e caso exista a colisao, a bola deve mudar de direcao no
    eixo x
    */
    if(this.collided(player1)){
        horizontal = true;
    }
    else if(this.collided(player2)) {
        horizontal = false;
    }
}
```

Ainda considerando a modificação do critério de classificação de uma mensagem Negativo-Negativo, temos resultados de sensibilidade, especificidade, acurácia, preditividade negativa e preditividade positiva modificados. De acordo com as Tabelas 6.28 e 6.29 todos os resultados baseado na padronização do modelo obtiveram 100% de precisão, os resultados baseados na comparação de variáveis não foram 100% precisos, mas com a modificação do critério Negativo-Negativo apresentam resultados mais positivos em todos os valores das Tabelas 6.28 e 6.29, quanto comparados aos valores anteriores. Podemos verificar que os critérios de avaliação Falso-Negativo e Negativo-Negativo podem modificar os resultados finais de eficiência apresentados, nesta tese não se definiu qual dos critérios é mais adequado para medir a eficiência do analisador, de forma que o estudo e escolha do critério mais adequado torna-se relevante em trabalhos futuros.

**Tabela 6.28: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE CLASSES DE PROGRAMA ANALISADAS PARA O TERCEIRO ESTUDO DE CASO (modificando-se os critérios Negativo-Negativo e Falso-Negativo)**

<i>Projeto</i> <i>"Ping Pong"</i>	<i>Medidas de eficácia do sistema analisador de acordo com o total de programas analisados</i>		
<i>Classe "Bola"</i>	<i>Quantidades e Medidas Obtidas</i>	<i>De acordo com a Padronização do Modelo</i>	<i>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</i>
	<i>NN</i>	1	6
	<i>FN</i>	0	1
	<i>PP</i>	9	3
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	75%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	90%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	100%	86%
<i>Classe "Barra"</i>	<i>NN</i>	6	3
	<i>FN</i>	0	3
	<i>PP</i>	4	4
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	57%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	70%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	100%	50%
<i>Classe "Jogo"</i>	<i>NN</i>	9	3
	<i>FN</i>	0	0
	<i>PP</i>	1	7
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	100%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	100%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	100%	100%

**Tabela 6.29: MEDIDA DE EFICIÊNCIA DO SISTEMA ANALISADOR DE ACORDO COM O TOTAL DE COMENTÁRIOS ANALISADOS PARA O TERCEIRO ESTUDO DE CASO (modificando-se o critério Negativo-Negativo e Falso-Negativo)**

<i>Projeto "Ping Pong"</i>	<i>Medidas de eficácia do sistema analisador de acordo com o total de comentários analisados</i>		
<i>Classe "Bola"</i>	<i>Quantidades e Medidas Obtidas</i>	<i>De acordo com a Padronização do Modelo</i>	<i>De acordo com a Comparação de Pares de Variáveis e Variáveis sem Par</i>
	<i>NN</i>	1	6
	<i>FN</i>	0	7
	<i>PP</i>	9	3
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	30%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	56%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	100%	46%
<i>Classe "Barra"</i>	<i>NN</i>	11	3
	<i>FN</i>	0	3
	<i>PP</i>	4	4
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	57%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	70%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	100%	50%
<i>Classe "Jogo"</i>	<i>NN</i>	10	3
	<i>FN</i>	0	0
	<i>PP</i>	1	7
	<i>FP</i>	0	0
	<i>Sensibilidade</i>	100%	100%
	<i>Especificidade</i>	100%	100%
	<i>Acurácia</i>	100%	100%
	<i>Preditividade Positiva</i>	100%	100%
	<i>Preditividade Negativa</i>	100%	100%

Verifica-se, novamente, neste estudo de caso, que a ocorrência de programas não padronizados de acordo com o programa modelo prejudica a fase de comparação de pares de variáveis e aumenta a possibilidade de ocorrerem mensagens do tipo Falso-Negativo, prejudicando assim a precisão da análise.

## CAPÍTULO 7 CONCLUSÕES

Neste trabalho, o analisador semântico de programação se baseia em três tipos de análises: análise de acordo com a padronização do modelo, análise do reconhecimento de padrões de código sequencial JPlay e a comparação de pares de variáveis (construção de árvores de comportamento). Nossa abordagem de comparação ocorre entre o programa do estudante e o programa modelo. O estudante deve selecionar, em sua ferramenta de ambiente de desenvolvimento integrado (IDE), o programa modelo que ele quer usar como referência, previamente disponibilizado em um repositório. A comparação consiste na busca de comportamentos de correspondência entre pares de classes e posteriormente, pares de variáveis, destes programas. Assim, os comportamentos dizem respeito ao objetivo do programa e são obtidos através de uma heurística de comparação de pares de classes e pares de variáveis similares.

Inicialmente, para obter as informações do código-fonte dos programas, foi proposto e desenvolvido um *parser* que transforma o código-fonte dos dois programas que estão sendo comparados para um código XML. Assim, cada arquivo de classe Java de um projeto, após ser interpretado, tem um respectivo arquivo XML, com o mesmo nome da classe e extensão XML. Após a transformação do código-fonte em XML, a consulta de informações no código XML gerado foi feita inicialmente através do uso da interface DOM. No entanto, constatamos que a representação das consultas necessárias geraram um código-fonte de alta complexidade. Por este motivo, foram feitas modificações no intuito de realizar as consultas através da representação de fatos e regras da linguagem PROLOG. A representação de fatos PROLOG foi utilizada na construção das consultas de comparação entre as classes, sendo que desta forma o resultado das comparações vão mostrar as diferenças entre as classes analisadas. Estas diferenças podem representar erros semânticos na classe do programa do estudante e serão apresentadas como resultado da comparação.

A primeira etapa da heurística de comparação é a classificação dos pares de classes entre os dois programas. A próxima etapa consiste em analisar cada um dos pares de classes classificados. Para isso, as variáveis de cada uma das classes devem ser analisadas e pares de variáveis do mesmo tipo devem ser classificados. Após a classificação dos pares de variáveis, cada par de variáveis será analisado. Para isso é construído para cada variável uma árvore de comportamentos.

Foram realizados três estudos de caso relacionados a 3 (três) programas desenvolvidos usando o framework JPlay. O primeiro programa foi desenvolvido por 8 (oito) estudantes do curso Médio Integrado em Informática do Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI). O programa foi aplicado como prova na disciplina "Tópicos Especiais em Desenvolvimento de Software". Esta primeira análise mostra os resultados do analisador ao verificar um programa que foi construído com uma única classe (classe "*main*") de projeto.

O segundo programa foi desenvolvido também por 10 (dez) estudantes, entre eles, alunos do Médio Técnico Integrado em Informática, Técnico subsequente em Informática e Tecnólogo em Análise e Desenvolvimento de Sistemas. Os alunos tiveram que desenvolver o jogo "*BrickBreak*" como atividade durante o minicurso "Desenvolvimento de jogos usando Java e framework JPlay". A segunda análise mostra os resultados do analisador ao verificar um programa constituído de 5 (cinco) classes de projeto. O jogo envolve a necessidade de mais conhecimentos sobre o uso do paradigma de orientação a objetos, visto que é proposto a divisão do projeto em classes ("*Barra*", "*Bloco*", "*Bola*", "*Jogo*" e "*Principal*"), e em cada uma das classes o jogo envolve a implementação de funcionalidades específicas.

O terceiro programa proposto foi o jogo "*Ping Pong*", sendo o mesmo aplicado como atividade da disciplina, aos alunos da turma de "Tópicos Especiais em Desenvolvimento de Software" do Curso Médio Técnico Integrado em Informática. O jogo, assim como o "*BrickBreak*", envolve a necessidade de mais conhecimentos sobre o uso do paradigma de orientação a objetos, visto que é proposto a divisão do projeto em classes ("*Barra*", "*Bola*", "*Jogo*" e "*Principal*"), e em cada uma das classes o jogo envolve a implementação de funcionalidades específicas. Participaram da análise um total de 10 (dez) programas.

Desta forma procura-se generalizar os resultados obtidos do analisador semântico aplicado a programas básicos e também programas mais complexos usando Java e o framework JPlay. Um programa básico caracteriza-se, neste caso, por envolver poucas funcionalidades (o programa não é necessariamente um jogo), sendo desenvolvido somente com o uso da classe principal "*main*" (caso do primeiro programa) e programas mais complexos caracterizam-se, neste caso, pela necessidade do paradigma de orientação a objetos e conseqüentemente o uso de várias classes no projeto (como no caso do segundo programa).

É possível concluir de acordo com a análise de resultados realizada nos estudos de casos, que quanto mais padronizado o programa do estudante em relação ao programa modelo, menos imprecisões serão encontradas na comparação (o que significa uma menor

ocorrência de resultados do tipo Falso-Negativos). Ainda no caso do comportamento do programa estar correto, mas não se encontrar de acordo com a padronização, a falta de padronização vai gerar comentários falso-negativos. Uma alta incidência de falso-negativos na padronização da classe compromete a eficiência da próxima fase de comparação do analisador, a comparação de pares de variáveis, gerando muitos resultados falso-negativos, também, na próxima fase.

Problemas de padronização podem ser causados pelo aluno, no caso dele não ter seguido as orientações corretamente, mas também podem ser causados pelo mau detalhamento ou má escolha (quando existem muitas variações de programação) do professor quanto aos comportamentos da classe que devem ser padronizados. Uma vantagem do sistema analisador é que o próprio professor pode identificar pontos críticos que causam problemas na padronização dos programas e pode alterar, no programa modelo, os comportamentos padronizados, de forma a corrigir e/ou minimizar esses problemas.

Resultados do tipo falso-negativos, em geral, são ocasionados pelo fato de existirem muitas variações de programação para resolver o mesmo problema. Neste caso, o professor deve padronizar, no programa modelo, o modo de solucionar o problema, e o aluno deve seguir as orientações de acordo com a padronização do modelo. O professor ainda pode escolher não associar nenhum comentário ao comportamento envolvido (no caso do professor verificar que o comportamento não modifica o comportamento geral do programa e/ou não é relevante para avaliação naquela atividade específica), liberando o aluno para uma implementação mais subjetiva (desta forma as diferenças são computadas, mas não existem comentários associados a serem enviados para o aluno). Mais uma vez, o sistema analisador apresenta a vantagem de o próprio professor, através de sua verificação durante a aplicação da atividade, poder modificar as configurações do programa modelo e assim corrigir e/ou minimizar os problemas apresentados.

Conclui-se também que dois tipos de critérios podem ser utilizados para a classificação de tipos Falso-Negativo e Negativo-Negativo, o primeiro critério diz que uma diferença detectada é do tipo Falso-Negativo quando ela não modifica o comportamento correto do programa e um Negativo-Negativo quando a diferença encontrada modifica o comportamento correto do programa (foi o critério utilizado nos estudos de casos). O segundo critério possível diz que uma diferença detectada é do tipo Falso-Negativo somente quando ela não identifica uma diferença real de padronização entre os dois programas ou quando ela não identifica uma diferença real entre pares de variáveis e um Negativo-Negativo é

identificado toda vez que uma diferença real de padronização ou de pares de variáveis for encontrada, independentemente desta diferença modificar ou não o comportamento correto do programa. O segundo critério, tal como pode ser visto nas Tabelas 6.28 e 6.29, se utilizado, é capaz de causar uma melhora significativa nos resultados de medidas de eficiência do sistema analisador (acurácia, preditividade negativa etc). Nesta tese não se definiu qual dos critérios é mais adequado para medir a eficiência do analisador, de forma que o estudo e escolha do critério mais adequado torna-se relevante em trabalhos futuros.

Para trabalhos futuros torna-se necessário a construção de uma interface de tutoria capaz de interpretar os resultados obtidos e, então, a partir disso, retornar as respostas para o estudante. A interface de tutoria ficaria encarregada, dentre outras funções, por verificar os resultados na fase de padronização do modelo, de maneira que o analisador somente passaria para a próxima fase de análise (fase de comparação de pares de variáveis) se o programa do aluno estivesse corretamente padronizado. Outra alternativa seria a de os índices de aceitação h1 para a análise da próxima fase poderiam ser configurados previamente pelo professor, de forma que ele possa controlar o quão padronizado deve estar o programa do estudante para que seja permitida a análise da próxima fase (comparação de variáveis).

Para trabalhos futuros também se propõe realizar estudos de casos com a participação de mais de um professor, de forma a avaliar o comportamento do professor no uso do analisador e se diferentes professores poderiam interferir nos resultados da análise desenvolvida.

Esta proposta traz contribuições significativas na área de programação e ensino de programação, tendo como principal contribuição, uma heurística baseada na comparação de comportamentos entre programas (programa do aluno e programa modelo). A proposta contribui também com ferramenta capaz de interpretar o código semanticamente construído por programadores, retornando resultados, apontando problemas e sugerindo soluções semânticas.



## REFERÊNCIAS

- ABE, J.M., SCALZITTI, A., FILHO, J.I.S. "Introdução a Lógica para a Ciência da Computação". Editora Arte e Ciência. 2º ed. São Paulo, 2002.
- ADAM, A., LAURENT, J., "LAURA, a system to debug student programs". Artificial Intelligence, v.15, n.1, pp. 75-122, 1980.
- AGUIAR, A., DAVID, G., BADROS, G.J., "JavaML 2.0: Enriching the Markup Language for Java Source Code". In XML: Aplicações e Tecnologias Associadas (XATA'2004), Porto, Portugal, February 2004.
- 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". In ACM SIGCSE Bulletin. ACM, 2002. p. 137-141.
- 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.
- BADROS, G.J., "JavaML: A Markup Language for Java Source Code". In Proceedings of the 9<sup>th</sup> Int. Conf. On the World Wide Web (WWW9), Amsterdam, Netherlands, May 2000.
- BLUEJ, disponível em <http://www.bluej.org/tutorial/testing-tutorial.pdf>. Acessado em agosto de 2013.
- 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.
- DIFFEN, disponível em [http://www.diffen.com/difference/Accuracy\\_vs\\_Precision](http://www.diffen.com/difference/Accuracy_vs_Precision). Acessado em novembro de 2015.
- DOM, disponível em <http://www.w3.org/DOM/>. Acessado em novembro de 2012.
- ECLIPSE ASTParser, disponível em <http://help.eclipse.org/helios/index.jsp>. Acessado em setembro de 2013.
- ECLIPSE PLATAFORM, disponível em <http://help.eclipse.org/indigo/index.jsp>. Acessado em agosto de 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.

- GALHARDO, M. F.; ZAINA, L. A. M. Simulação para ensino de conceitos da orientação a objetos. In: XIII Seminário de Computação - SEMINCO, Blumenau, 2004. Blumenau: Furb, 2004. p. 109-116.
- HASTIE, T., TIBSHIRANI, R., FRIEDMAN, J.H. "The elements of statistical learning data mining, inference, and prediction. New York: Springer, 2009.
- HOLZ, W., PREMRAJ, R., ZIMMERMANN, T., Zeller, A. "Predicting software metrics at design time". In 9<sup>th</sup> International conference on Product-Focused Software Process Improvement. Rome, pp. 34 – 44, June 2008.
- JOHNSON, W. L., SOLOWAY E.. "Proust: Knowledge-based program understanding". In ICSE 84: Proceedings of the 7<sup>th</sup> international conference on Software engineering, pp. 369-380, Piscataway, NJ, USA, 1984. IEEE Press.
- PLAY, disponível em <http://www.ic.uff.br/JPlay/>. Acessado em abril de 2012.
- JUNIT, disponível em <http://junit.org/>. Acessado em abril de 2013.
- KASURINEN, J., PURMONEN, M., NIKULA, U. A Study of Visualization in Introductory Programming. In: 20th annual Meeting of Psychology of Programming Interest Group, Lancaster, UK, 2008.
- KOLLING, M., QUIG, B., PATTERSON, A., and ROSENBERG, J. "The BlueJ system and its pedagogy". Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, v.13, n.4, p.249-268, 2003.
- KOSCHKE, R. (2007). Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software'06*, volume 06301 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. In MOTA, Ana Maria; GOYA, Denise H. Técnicas para Análise de Similaridade de Código de Software em Litígios de Propriedade Intelectual. Disponível em [http://www.ime.usp.br/~dhgoya/forense\\_paper.pdf](http://www.ime.usp.br/~dhgoya/forense_paper.pdf). Acessado em março de 2016.
- MAMAS, E. and KONTOGIANNIS, C., "Towards Portable Source Code Representations Using XML". In Proc. of the 7<sup>th</sup> Working Conf. on Reverse Engineering (WCRE'00), Brisbane, Australia, pp. 172-18, November 2000.
- MOTA, Ana Maria; GOYA, Denise H. Técnicas para Análise de Similaridade de Código de Software em Litígios de Propriedade Intelectual. Disponível em [http://www.ime.usp.br/~dhgoya/forense\\_paper.pdf](http://www.ime.usp.br/~dhgoya/forense_paper.pdf). Acessado em março de 2016.
- MULLER, O. "Pattern Oriented Instruction and the Enhancement of Analogical Reasoning". In: Proceedings of the first international workshop on Computing education research (ICER), ACM Press, p. 57-67, 2005.
- NICKEL, Andrea; BARNES, Tiffany, "Games for CS Education: Computer-Supported Collaborative Learning and Multiplayer Games". In: Fifth International Conference on the Foundations of Digital Games 2010.
- 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.
- PORTER, R., CALDER, P. "Patterns in Learning to Program – An Experiment?". In: Proceedings of the Sixth Australasian Conference on Computing Education, ACM Press, p. 241-246, 2004.

- RAPKIEWICZ, C. E. et al. "Estratégias pedagógicas no ensino de algoritmos e programação associadas ao uso de jogos educacionais". *RENOTE*, v.4, n.2, 2006.
- RUN.CODES, disponível em <https://run.codes/>. Acessado em janeiro de 2016.
- 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 & Knowledge Engineering*, pp-668-673, Boston, June 27-29, 2013.
- SANTOS, G., DIRENE, A. I., GUEDES, A. L. P. "Autoria e Interpretação Tutorial de Soluções Alternativas para Promover o Ensino de Programação de Computadores." In *XIV Simpósio Brasileiro de Informática na Educação – SBIE 2003*. Rio de Janeiro, RJ, Brasil.
- SANTOS, O.J.P., BARROS, O. M., "CodeMI – Source Code as XMI. Uma Representação de Código-fonte para Coleta de Métricas". In *SBSI*. 2009.
- TRAETTEBERG, H., AALBERG, T. "Jexercise: a specification-based and test-driven exercise support plugin for eclipse". In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 70-74, New York, NY, USA. ACM Press. 2006.
- WALLINGFORD, E. The elementary patterns homepage, disponível em <http://www.cs.uni.edu/~wallingf/patterns/elementary/>. Acessado em agosto de 2013.
- WALENSTEIN, A., El-Ramly, M., Cordy, J. R., Evans, W. S., Mahdavi, K., Pizka, M., Ramalingam, G., and von Gudenberg, J. W. (2007). Similarity in programs. In *MOTA, Ana Maria; GOYA, Denise H. Técnicas para Análise de Similaridade de Código de Software em Litígios de Propriedade Intelectual*. Disponível em [http://www.ime.usp.br/~dhgoya/forense\\_paper.pdf](http://www.ime.usp.br/~dhgoya/forense_paper.pdf). Acessado em março de 2016.
- WEB-CAT, disponível em <http://wiki.web-cat.org/WCWiki/>. Acessado em agosto de 2013.
- WEKA, disponível em <http://www.cs.waikato.ac.nz/ml/weka/arff.html>. Acessado em julho de 2013.
- WIKIHOW, disponível em <http://pt.wikihow.com/Calcular-Sensibilidade,-Especificidade,-Valor-Preditivo-Positivo-e-Valor-Preditivo-Negativo>. Acessado em novembro de 2015.
- WITTEN, I.H.; FRANK, E.. "DataMining: Practical Machine Learning Tools and Techniques with Java Implementations". Morgan Kaufmann Publishers, 3<sup>rd</sup> ed., 2011.
- XML, disponível em <http://www.w3.org/XML/>. Acessado em abril de 2012.

## ANEXOS

### ANEXO A - ENUNCIADO DA PROVA (PRIMEIRO ESTUDO DE CASO)

INSTITUTO FEDERAL DO PIAUÍ – IFPI

CURSO: Ensino Médio Técnico em Informática

DISCIPLINA: Tópicos Especiais

PROFESSOR: Elanne O dos Santos

#### Prova No. 1

01. No trecho de código abaixo, uma bola a mais deve ser inserida na lista de bolas toda vez que o botão LEFT do mouse for pressionado: (2.0 pts)

(Modelo do trecho de código:)

```
.....
pos = mouse.getPosition();
List<Sprite> bolas = new ArrayList();
Window janela = new Window(800,600);
GameImage fundo = new GameImage("fundo2.png");
Keyboard teclado = janela.getKeyboard();
Mouse mouse = janela.getMouse();
Point pos;
.....
while (true){
if (mouse.isLeftButtonPressed()){

}
.....
}
```

Complete o código de modo que uma bola seja incluída na lista de bolas sempre que a tecla LEFT do mouse seja pressionada. A bola deve ser desenhada na posição atual do mouse.

02. Modifique o código de maneira que, só seja incluída uma nova bola na lista de bolas, se a posição selecionada for uma posição vazia. Se já existir uma bola naquela posição, a inclusão não deve ser permitida e uma mensagem deve ser apresentada: (4.0 pts)

"Área já preenchida!!!"

**DICA 01:** Compare se a posição na x e y atual já existe algum objeto bola na lista de bolas.

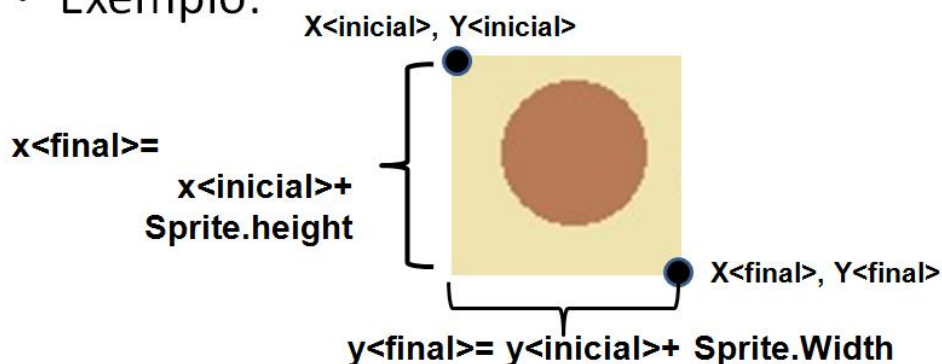
posição inicial do objeto bola no eixo x: bola.x

posição final do objeto bola no eixo x: bola.x+bola.height

posição inicial do objeto bola no eixo y: bola.y

posição final do objeto bola no eixo y: bola.y + bola.width

- Exemplo:



#### DICA 02:

Use o *JOptionPane* para imprimir a mensagem. Exemplo:

```
JOptionPane.showMessageDialog(janela, "Área já preenchida!!!");
```

(Modelo do trecho de código:)

```
.....
pos = mouse.getPosition();
List<Sprite> bolas = new ArrayList();
Window janela = new Window(800,600);
GameImage fundo = new GameImage("fundo2.png");
Keyboard teclado = janela.getKeyboard();
Mouse mouse = janela.getMouse();
Point pos;
.....
while (true){
if (mouse.isLeftButtonPressed()){

}
.....
}
```

## ANEXO B - ENUNCIADO JOGO "BRICKBREAK" (SEGUNDO ESTUDO DE CASO)

### AULA 2:

- ❑ BrickBreak e Comportamento de personagens do jogo
- ❑ Usando herança, this e super na construção do jogo

#### Classe Bola: Comportamento:

- A classe Bola deve herdar de **Sprite**;
- Ela deve possuir **3 métodos**:
  - um construtor, que deve inicializar a figura da bola,
  - um método para mover a bola no eixo do x
  - e um método para mover a bola no eixo do y;
- Controle do teclado: **Não**.
- Código:

```
import jplay.Sprite;

public class Bola extends Sprite {
    public Bola(String imagem){
        super(imagem);
    }
    void movex() {
    }
    void movey() {
    }
}
```

#### Classe Barra: Comportamento:

- Herança: **Sprite**
- A classe deve possuir 2 métodos:
  - O construtor , que deve inicializar a figura da barra.
  - O outro método deve controlar o movimento da barra no eixo do x para a direita e para a esquerda através do controle do teclado (para isso use o método keydown da classe Keyboard do JPlay).

- Controle do teclado: **Sim**
- Movimento: **Eixo x**

#### **Controle do teclado:**

- O movimento do x é controlado pelo teclado, então deve receber como parâmetro:
  - a variável `keyboard` do teclado,
  - uma variável inteira que armazene o valor da tecla da esquerda
  - e uma variável inteira que armazene o valor da tecla da direita.
- Teste para o teclado: `Keyboard.KeyDown`. Ex:

`Keyboard.KeyDown(direita)`

`Keyboard.KeyDown(esquerda)`

- Código:

```
import jplay.Keyboard;
import jplay.Sprite;
public class Barra extends Sprite{
public Barra(String Imagem){
    super(Imagem);
}
void movimentoX(Keyboard teclado, int direita, int esquerda){

}
}
```

#### **Classe Bloco: Comportamento:**

- A classe herda de `Sprite`.
- Só possui 1 método:
  - O construtor, que deve inicializar a figura do bloco.
- Código:

```
public class Bloco extends Sprite {
public Bloco(String Imagem){
    super(Imagem);
}
}
```

### Classe Jogo: Comportamento:

- O cenário do jogo deve ser definido nesta classe.
- Esta classe deve contar o loop infinito do jogo, nela vão ser criados os gameobjects envolvidos.
- Devem ser definidos 3 gameobjects: **barra, bola e barra.**
- A classe deve conter **5 métodos, um método construtor**, um método para **carregar** os objetos envolvidos, um método para **inicializar** esses objetos, um método para **desenhar** os objetos na janela do jogo e outro método que representa o **loop** infinito do jogo.
- O método **construtor** deve ficar responsável por chamar os métodos **carregar** , **inicializar e loop.**
- O método **loop** deve definir o loop infinito do jogo, a cada iteração do loop ele deve chamar o método **desenha, atualizar a janela do jogo e definir as regras do jogo.**
- **PASSOS PARA CRIAR O CENÁRIO:**
- Criar janela Window
- Criar teclado Keyboard
- Criar fundo GameImage
- Código:

```
public class Jogo {
    Window janela; // criar cenario window
    GameImage fundo; // criar cenario GameImage
    Keyboard teclado; // criar cenario Keyboard
    .....
}
```

- **PASSOS PARA CRIAR OS PERSONAGENS:**
- Criar Bola
- Criar Barra
- Criar Bloco
- Código:

```
public class Jogo {
    .....
    Barra aluno_barra; // criar gameobject barra
    Bloco[][] aluno_bloco; // criar gameobject bloco
```



```
Bola aluno_bola; // criar gameobject bola
.....}
```

- **REGRAS DO JOGO:**

- Realizar a cada iteração do loop infinito:

- Verique se o gameobject barra foi movida através das teclas right ou left do teclado.
- No método loop() movimente o gameobject bola no eixo do x para a esquerda ou para a direita.
- No método loop() movimente o gameobject bola no eixo do y para cima ou para a baixo.
- No método loop(), verifique se a bola colidiu com a barra:
  - ✓ Se ocorrer uma colisão entre os dois, a bola deve mudar de direção no eixo do y (ir para cima).
- No método loop(), verifique se o gameobject bola colidiu com algum dos blocos definidos. Essa verificacao deve ser feita nas 7 colunas de cada uma das 10 linhas aonde foram definidas cada um dos blocos.
- No método loop(), se foi detectada uma colisão entre a bola e um dos blocos:
- retire este bloco da janela do jogo (para isso posicione o mesmo na posicao 0 do eixo do x e posição 0 do eixo do y)
- esconda este bloco da janela do jogo (para isso chame o metodo hide do jplay).
- incremente os pontos do jogador.
- No método loop(), verifique a direção atual da bola no eixo do y e inverta esta direção.
- No método loop(), **verifique a finalização do jogo**. O jogo é finalizado quando acontecem uma das situações:
  1. O jogador pede para encerrar pressionando a barra de espaço do teclado.
  2. O jogador alcançou o numero de pontos igual a 120.
  3. A bola não foi rebatida pela barra e atravessou o limite inferior da janela do jogo.
- No método loop(), verifique se a pontuação final do jogador for igual a 120 então ele ganhou o jogo, caso contrário ele perdeu o jogo.

## ANEXO C - ENUNCIADO JOGO "*PING PONG*" (SEGUNDO ESTUDO DE CASO)

### AULA 3:

- ❑ Ping Pong e os comportamentos dos personagens do jogo
- ❑ **CLASSE BALL:**
  - A classe Bola deve herdar de Sprite;
  - Ela deve possuir 3 métodos:
    - um construtor, que deve inicializar a figura da bola e inicializar atributos,
    - um método para mover a bola no eixo do x e no eixo do y e chamar o metodo que testa colisao com as barras
    - um metodo para testar colisao entre bola e as barras
  - Controle do teclado: Não.
- ❑ **CLASSE PLAYER (BARRA):**
  - Serão dois jogadores do tipo da classe Player
  - A classe Barra deve herdar de **Sprite**;
  - Ela deve possuir **3 métodos**:
    - um construtor, que deve inicializar a figura da barra e sua pontuação
    - Um método para mover a barra do PC no eixo do y
    - Um método que incrementa a pontuação do jogador
  - Controle do teclado: **Não. A barra do usuário é controlada através do teclado, mas o controle da barra do usuário é feito na classe Jogo através do método do jplay: moveY(Double velocity);**
- ❑ **CLASSE JOGO:**
  - **Terá 3 objetos do jogo (gameobjects):**
    - 2 do tipo player, sendo 1 o jogador e o outro o computador
    - Um do tipo Ball para ser a bola
  - **Para criar o cenário vamos definir os seguintes objetos:**
    - Um do tipo Window que será a janela
    - Um do tipo GameImage que será o fundo
    - Um do tipo Keyboard que ira captar o teclado

- um do tipo Font que será nossa fonte personalizada usada no game.

**Obs: Para criar uma fonte:** Font fonte = new Font("Courier New", Font.ROMAN\_BASELINE, 200);

- **Terá 6 métodos:**

- O construtor que deve chamar um método que inicializa atributos e objetos e um método que chama o loop infinito do jogo.
- Um método que inicializa todos os atributos e objetos
- Um método que contenha o loop infinito
- Um método que desenha todas as imagens na tela
- Um método que reseta as posições da bola e dos dois players
- Um método que é responsável pela pontuação do jogo

- **Regras básicas do jogo:**

- Quando a bola sair pelas extremidades do eixo X, deve-se pontuar o jogador q fez o ponto e logo em seguida resetar as posições dos dois players , da bola e a velocidade da bola.
- A velocidade da bola vai aumentando a cada iteração dos jogadores, aonde não exista pontuação de nenhum dos dois (jogo não resetado).