



Universidade Federal de Juiz de Fora

DCC168 - Teste de Software

Daniel Rezende Varoto

Julio Cesar Rosa Trindade

# Sumário

1- Introdução.....	3
2 - Ferramentas .....	3
3 - Teste Funcional .....	3
4 - Teste Estrutural .....	6
5 - Teste de mutação.....	12
6 - Referências.....	13

# 1- Introdução

Dentro do processo de desenvolvimento de software, a qualidade e funcionamento do software são de suma importância para o projeto como um todo. Tendo em vista os diversos aspectos qualitativos de um software, o processo de testes de software prevê métodos de testes para cada uma das partes do software, indo desde testes funcionais, em que a interface com o usuário é explorada até testes estruturais, que avaliam a construção e funcionamento de componentes internos não vistos diretamente pelo usuário final.

A fim de exercitar estas metodologias, apresentamos o trabalho final da disciplina DCC168 – Testes de Softwares.

Para este trabalho, escolhemos o jogo PongGame[1] para aplicação de testes funcionais, estruturais e baseados em defeito, conforme proposto no documento de especificação do trabalho

## 2 - Ferramentas

Para análises e implementações, foram utilizados os seguintes softwares:

1. Eclipse IDE [2]
2. JUnit [3]
3. PIT Mutations [4]

## 3 - Teste Funcional

Para aplicação dos testes funcionais, foi necessário que fizéssemos uma análise exploratória no jogo, já que o mesmo não possuía documentação.

Dentro dessa análise, identificamos as seguintes funcionalidades:

1. Start: Onde o jogo acontece
2. Help: Tela com instruções do jogo
3. Quit: Sair do jogo
4. Tela de FPS: Tela que mostra quantidade de FPS (Frame per second) que o jogo exibe

De posse dessa informação, efetuamos os testes funcionais, conforme previsto.

Primeiramente, executamos a técnica de particionamento por classe de equivalência.

Dessa forma, conseguimos gerar todas as possíveis entradas separando os testes em casos que são válidos (V) e inválidos (I), conforme pode ser visto nas tabelas abaixo.

Menu Principal		
Variáveis de entrada	Saída esperada	Classe de Eq. Cobertas
Seta para baixo e ENTER	Entrada na opção HELP	V1, V2
Pressionar F	Entrada na tela de FPS	V3
Clicar sobre a opção HELP	---	I5
Pressionar 0	---	I1
Pressionar !	---	I2
Pressionar A	---	I3
Pressionar z	---	I4

Tabela 1- Particionamento por classes de equivalência para tela Menu Principal

Help		
Variáveis de entrada	Saída esperada	Classe de Eq. Cobertas
Pressionar enter	Voltar à tela de menu inicial	V5
Clicar sobre a tela	---	I6

Tabela 2 - Particionamento por classes de equivalência para tela Help

Start		
Variáveis de entrada	Saída esperada	Classe de Eq. Cobertas
Pressionar a tecla W, pressionar a tecla seta pra cima	Movimentos de bloco esqueda e direito pra cima	V6, V8
Pressionar a tecla S, pressionar a tecla seta pra baixo	Movimentos de bloco esqueda e direito pra baixo	V7, V9
Pressionar a tecla END	Velocidade volta a nível inicial do jogo	V10
Pressionar a tecla ESC	Voltar à tela de menu inicial	V11
Clicar sobre o bloco esquerdo	---	I7, I9, I11, I13
Pressionar teclas para esquerda e direita	---	I8, I10, I12, I14

Tabela 3- Particionamento por classes de equivalência para tela Start

Com as tabelas montadas, foi possível visualizar quais seriam as classes de equivalência válidas e inválidas, e com isso derivar casos de testes para cada uma delas.

A implementação dos casos de teste funcional foi feita, dentro do projeto base do jogo PongGame, no pacote test, na classe FunctionalTest.java.

Durante a construção dos testes, para simulação de um usuário final, utilizamos a classe java.awt.Robot.

Cada nome de método do teste unitário para testar funcionalidades possui quais classes de equivalência ele atende, para que, dessa forma, fique fácil saber qual o objetivo do método.

Prosseguindo com o desenvolvimento dos testes, aplicamos a técnica de grafo de causa-efeito para tentar identificar condições que as técnicas anteriores não contemplavam.

Foi identificado que, ao pressionar duas teclas juntamente, poderíamos ter um comportamento inesperado, como demonstrado no grafo abaixo

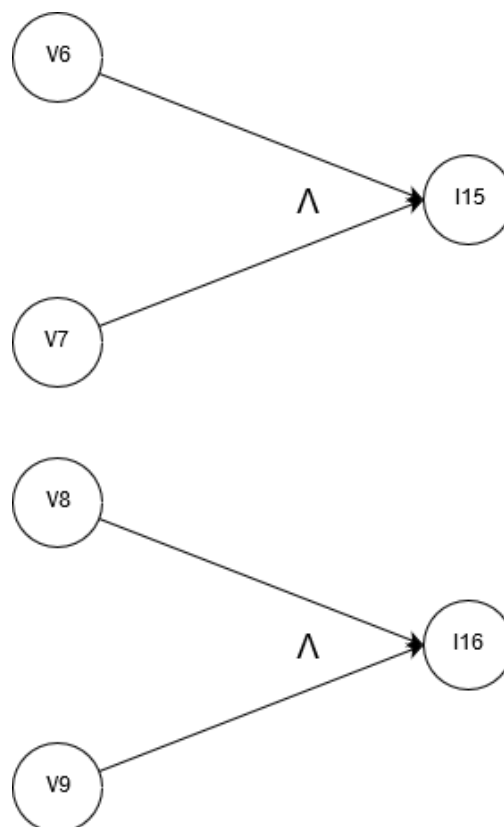


Figura 1- Grafo de causa-efeito para teclas pressionadas juntas no jogo

Com isso, nossa tabela de testes para o jogo (Start) foi atualizada com mais duas linhas ao final

Start		
Variáveis de entrada	Saída esperada	Classe de Eq. Cobertas
Pressionar a tecla W, pressionar a tecla seta pra cima	Movimentos de bloco esqueda e direito pra cima	V6, V8
Pressionar a tecla S, pressionar a tecla seta pra baixo	Movimentos de bloco esqueda e direito pra baixo	V7, V9
Pressionar a tecla END	Velocidade volta a nível inicial do jogo	V10
Pressionar a tecla ESC	Voltar à tela de menu inicial	V11
Clicar sobre o bloco esquerdo	---	I7, I9, I11, I13
Pressionar teclas para esquerda e direita	---	I8, I10, I12, I14
Pressionar teclas W e S juntas	---	I15
Pressionar seta para cima e seta para baixo juntas	---	I16

Tabela 4 - Tabela de particionamento de classes de equivalência após grafo de causa-efeito

Implementados os testes baseados nesses particionamentos, obtivemos uma cobertura de 95,20% com 11 métodos, métrica essa que foi obtida com a execução do EclEmma.

## 4 - Teste Estrutural

Para o teste estrutural, nos baseamos no relatório gerado pelo EclEmma para decidir em quais métodos do sistema atuar, com o propósito de aumentar a cobertura total do jogo.

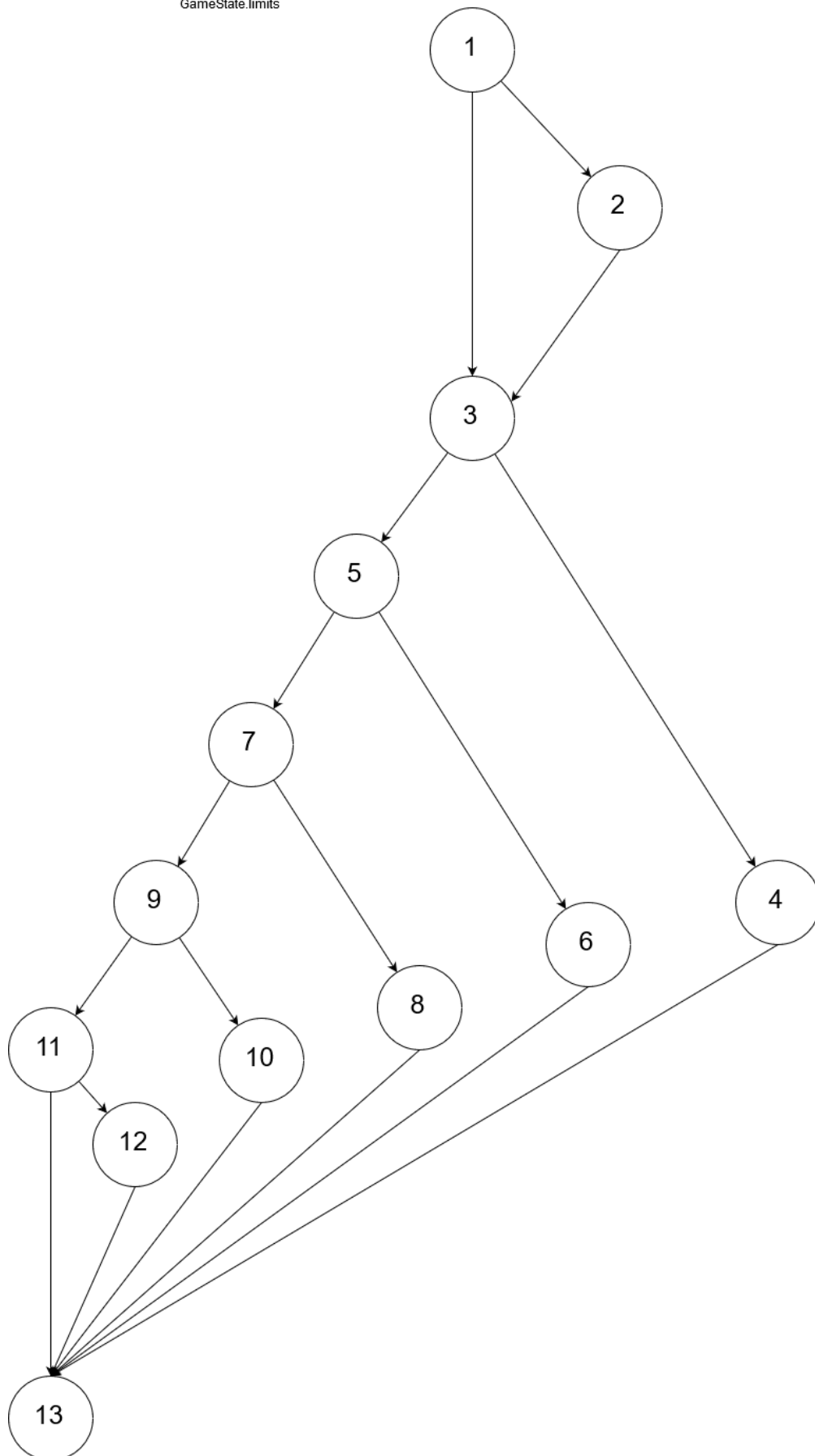
Os GFCs (Grafos de fluxo de controle) foram gerados com base na análise do algoritmo de cada um dos métodos.

- GameState.limits()

```
1  public class GameState implements States {
2      // [...]
3
4      /*1*/  private void limits() {
5          /*1*/      if(ball.x < 0) {
6              /*2*/          start();
7              /*2*/          B.incScore();
8          /*3*/      }if(ball.x > (Game.width-5)) {
9              /*4*/          start();
10             /*4*/          A.incScore();
11         /*5*/      }else if(ball.y < 80) {
12             /*6*/          yVel = 4;
13         /*7*/      }else if(ball.y > (Game.height - 15)){
14             /*8*/          yVel = -4;
15         /*9*/      }else if(A.intersects(ball)){
16             /*10*/          xVel = 4;
17         /*11*/      }else if(B.intersects(ball)) {
18             /*12*/          xVel = -4;
19         /*12*/      }
20         /*13*/  }
21
22     // [...]
23 }
```

## Grafo de fluxo de controle

GameState.limits



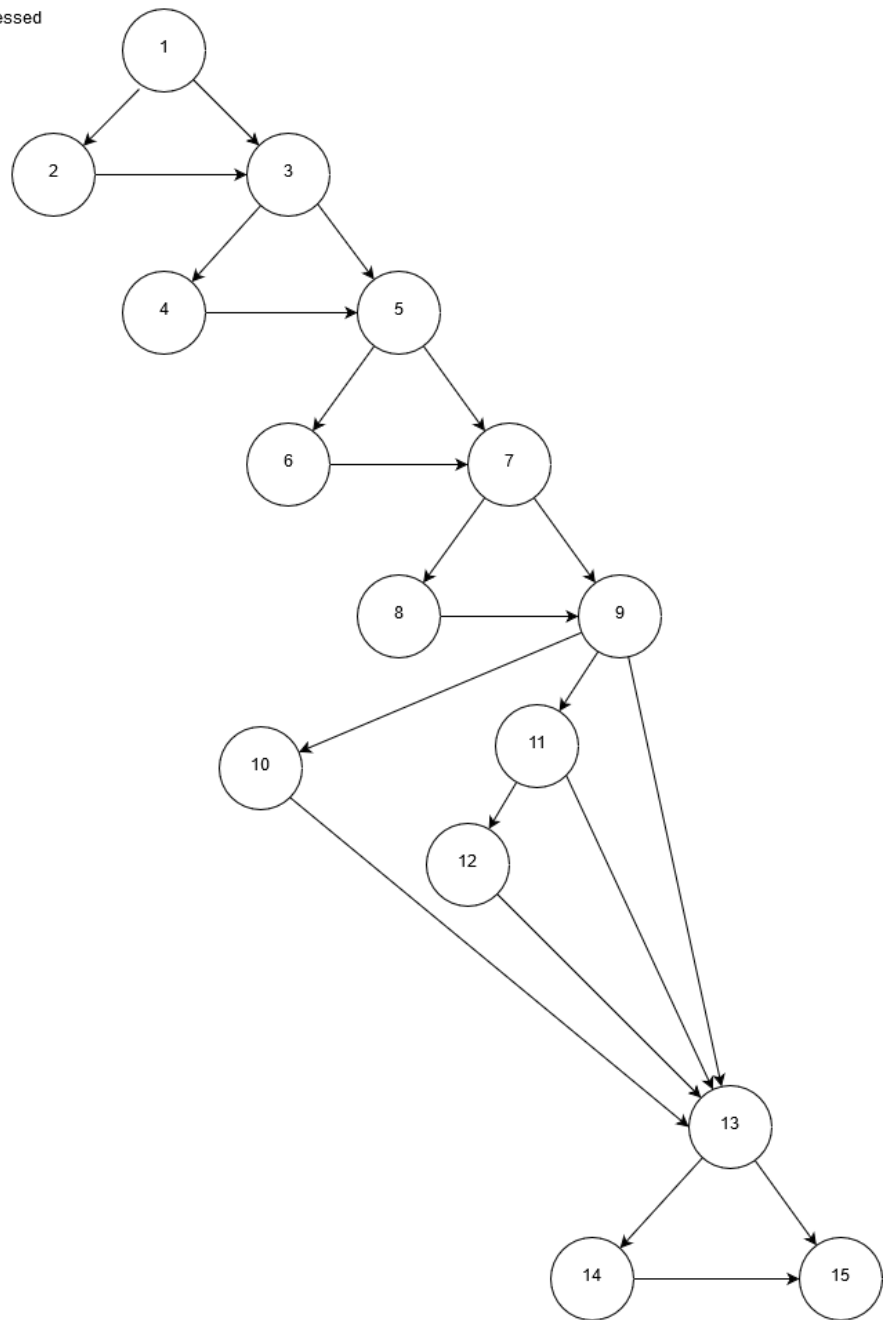


- GameState.checkKeyPressed()

```
55 public class GameState implements States {
56     // [...]
57
58     /*1*/ public void checkKeyPressed(int cod) {
59     /*1*/     if(KeyManager.w)
60     /*2*/         A.move(1);
61     /*3*/     if(KeyManager.s)
62     /*4*/         A.move(0);
63     /*5*/     if(KeyManager.up)
64     /*6*/         B.move(1);
65     /*7*/     if(KeyManager.down)
66     /*8*/         B.move(0);
67
68     /*9*/     if(KeyManager.speed && xVel<0)
69     /*10*/         xVel = -10;
70     /*11*/     else if(!KeyManager.speed && xVel<0)
71     /*12*/         xVel = -4;
72
73     /*13*/     if(KeyManager.esc)
74     /*14*/         StateManager.setState(StateManager.MENU);
75     /*15*/ }
76
77     // [...]
78 }
```

## Grafo de fluxo de controle

GameState.checkKeyPressed



- Game.run()

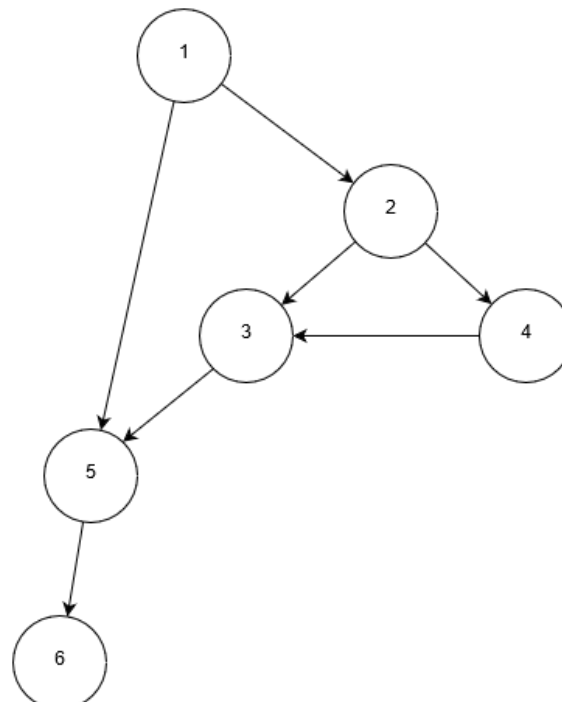
```

26 public class Game implements Runnable{
27     // [...]
28
29     /*1*/ public void run() {
30         /*1*/     init();
31
32         /*1*/     int FPS = 60;
33         /*1*/     double timePerTick = 1000000000/FPS;
34         /*1*/     double delta = 0;
35         /*1*/     long now;
36         /*1*/     long lastTime = System.nanoTime();
37
38         /*1*/     while(running) {
39             /*2*/         now = System.nanoTime();
40             /*2*/         delta += (now - lastTime)/timePerTick;
41             /*2*/         lastTime = now;
42
43             /*2*/         if(delta >=1) {
44                 /*3*/             update();
45                 /*3*/             render();
46                 /*3*/             delta--;
47             /*4*/         }
48             /*5*/     }
49             /*5*/     stop();
50         /*6*/ }
51
52     // [...]
53 }

```















### Grafo de fluxo de controle

Game.run



A implementação dos casos de teste estrutural foi feita, dentro do projeto base do jogo PongGame, no pacote test, na classe StructuralTest.java.

Com a implementação destes testes, a taxa de cobertura de testes, gerada pelo EcEmma chegou a 96,3%

Element	Missed Instructions	Cov.
 <a href="#">test</a>		94%
 <a href="#">pong_game</a>		89%
 <a href="#">pong_states</a>		98%
 <a href="#">pong_main</a>		0%
 <a href="#">pong_view</a>		100%
 <a href="#">pong_input</a>		100%
 <a href="#">pong_elements</a>		100%
Total	76 of 2.081	96%

PongGame (02/12/2019 01:03:35)

## 5 - Teste de mutação

A ferramenta PIT Mutations entrava em loop infinito ao ser rodada para geração de mutantes. Em conversa com outros grupos que também utilizaram o PongGame como base para o trabalho, o ocorrido se repete.

## 6 - Referências

- 1 – PongGame - <https://github.com/ArthurK12/PongGame>
- 2 – Eclipse IDE - <https://www.eclipse.org/>
- 3 – Junit 5 - <https://junit.org/junit5/>
- 4 – PIT Mutations - <https://pitest.org/>