

DCC168 Teste de Software

Trabalho Parte II – Teste Estrutural

Débora Izabel Duarte
Fabrício Guidine
Laura Neves

DCC168 Teste de Software

Trabalho Parte II – Teste Estrutural

Débora Izabel Duarte
Fabrício Guidine
Laura Neves

Relatório com a descrição da implementação do Jogo Aquário, ou seja, da arquitetura (classes e relacionamentos) e código fonte das operações desenvolvidas, dos resultados da implementação e execução dos casos de teste utilizando as ferramentas JUnit, Eclemma e Baduino para atender aos critérios de Teste Funcional e de Teste Estrutural baseados em Fluxo de Controle e Fluxo de Dados para atender aos requisitos da Parte II do trabalho da disciplina.

Juiz de Fora

Novembro, 2023

Sumário

1. Introdução.....	3
2. Arquitetura e Implementação do Programa.....	4
i. Diagrama de Classes	4
ii. Classe <i>Aquario</i>	5
iii. Classe <i>Configuracoes</i>	10
iv. Classe <i>Jogo</i>	14
v. Classe <i>Peixe</i>	22
vi. Classe <i>PeixeA</i>	26
vii. Classe <i>PeixeB</i>	32
viii. Classe <i>PosicaoAdjacente</i>	37
3. Teste Funcional	39
a. Critérios de Teste Funcional: Requisitos e Casos de Teste.....	39
b. Implementação dos Casos de Teste	46
i. Caso de Teste: <i>CT01</i>	46
ii. Caso de Teste: <i>CT09</i>	46
iii. Caso de Teste: <i>CT23</i>	47
c. Resultados da Execução dos Casos de Teste	47
d. Análise de Cobertura dos Casos de Teste	49
i. Análise de Cobertura de Fluxo de Controle	49
ii. <i>PosicaoAdjacente</i> : <i>PosicaoAdjacente()</i>	50
iii. <i>Jogo</i> : <i>setValor()</i>	51
iv. <i>Aquario</i>	52
v. <i>Configuracoes</i>	53
e. Análise de Cobertura de Fluxo de Dados	56
i. Método <i>auxIniciarJogo(int[] valores)</i> da classe <i>Jogo</i>	56
ii. Método <i>moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] posAoRedor)</i> da classe <i>PeixeA</i>	59
f. Resultados	61
4. Teste Estrutural - Critérios Baseados em Fluxo de Controle	62
a. Fluxo de Controle: Requisitos e Casos de Teste.....	62
i. Grafo de Fluxo de Controle	62
ii. Critérios Todos os Nós e Todos os Arcos	64
iii. Critérios Todos os Caminhos Simples e Livres de Laço	64
iv. Critérios Todos os Caminhos.....	64
b. Implementação dos Casos de Teste	65
i. Caso de Teste: <i>CT27</i>	65
ii. Caso de Teste: <i>CT28</i>	65
iii. Caso de Teste: <i>CT29</i>	66
iv. Caso de Teste: <i>CT30</i>	66

v.	Caso de Teste: CT31	67
c.	Resultados da Execução dos Casos de Teste	67
d.	Análise de Cobertura dos Casos de Teste.....	69
i.	Análise de Cobertura de Fluxo de Controle	69
ii.	Análise de Cobertura de Fluxo de Dados	70
e.	Resultados	74
5.	Teste Estrutural - Critérios Baseados em Fluxo de Dados	75
a.	Fluxo de Dados: Requisitos e Casos de Teste	75
i.	Grafo Def-Uso	75
b.	Requisitos de Teste Todas as Definições, Todos os P-Usos e C-Usos	79
c.	Requisitos de Teste Todos os Caminhos Definições-Usos.....	82
d.	Implementação dos Casos de Teste	83
i.	Caso de Teste CT32	83
ii.	Caso de Teste CT33	83
iii.	Caso de Teste CT34	84
e.	Resultados da Execução dos Casos de Teste	84
f.	Análise de Cobertura dos Casos de Teste	86
i.	Análise de Cobertura de Fluxo de Controle	86
g.	Análise de Cobertura de Fluxo de Dados	91
h.	Resultados	97
6.	Conclusões	98

1. Introdução

O presente relatório refere-se à disciplina *Teste de Software (DCC168)* da Universidade Federal de Juiz de Fora, abordando a Parte II do Trabalho sobre Teste Estrutural.

O foco do relatório é a descrição da implementação do jogo Aquário, incluindo a arquitetura (classes e relacionamentos) e o código-fonte das operações desenvolvidas. A execução dos casos de teste é realizada utilizando as ferramentas *JUnit*, *EclEmma* e *Baduino*, atendendo aos critérios de *Teste Funcional* e de *Teste Estrutural baseados em Fluxo de Controle e Fluxo de Dados*.

O documento está organizado em seis seções, começando pela introdução que apresenta a estrutura geral do relatório. Em seguida, são abordadas a Arquitetura e Implementação do Programa/Jogo, seguida pelo Teste Funcional, que inclui critérios, implementação dos casos de teste e análise de cobertura. Posteriormente, são detalhados os testes estruturais baseados em Fluxo de Controle e Fluxo de Dados, com seus respectivos critérios, implementação, resultados e análises de cobertura.

A seção de Conclusões encerra o relatório com reflexões sobre a eficiência das técnicas aplicadas, influência da criação de casos de teste funcionais, benefícios e limitações das ferramentas de automação utilizadas, além de desafios enfrentados e lições aprendidas ao longo do desenvolvimento do trabalho. As referências bibliográficas completam o documento.

2. Arquitetura e Implementação do Programa

A implementação foi feita utilizando a linguagem *Java*, através da *IDE Eclipse* na versão *Neon.3 Release (4.6.3)*. O código foi dividido em sete classes: *Jogo*, *Aquario*, *Peixe*, *PeixeA*, *PeixeB*, *Configuracoes* e *PosicaoAdjacente*.

O controle de versão foi feito através do *GitHub*, e o repositório pode ser acessado através do endereço https://github.com/dduartevla/DCC168_JogoDoAquario.git.

i. Diagrama de Classes

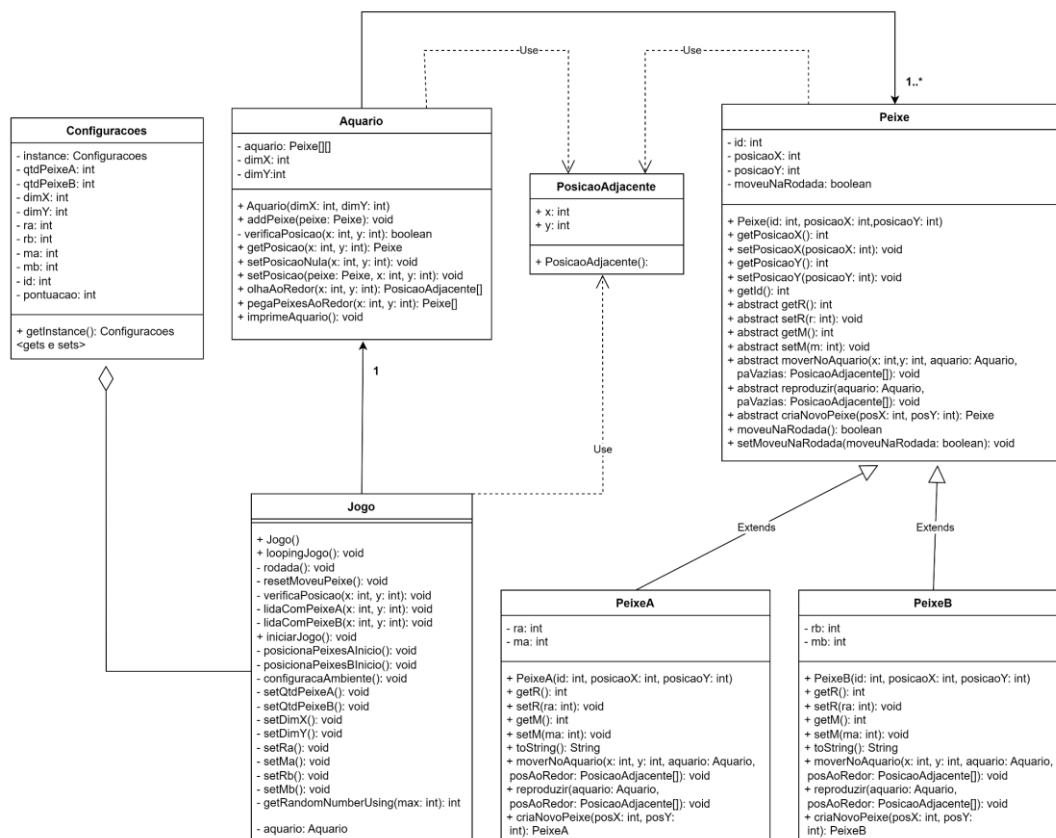


Figura: Diagrama UML do programa Aquario.

ii. Classe *Aquario*

A classe *Aquario* representa um aquário virtual onde os objetos da classe *Peixe* podem ser colocados em diferentes posições. A seguir, são detalhados os atributos, métodos e o comportamento da classe.

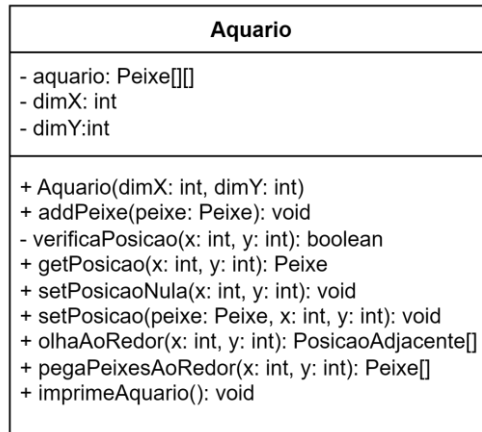


Figura: Diagrama UML da classe *Aquario*.

Método 1: <i>Aquario(int dimX, int dimY)</i>	
Descrição	Construtor da classe <i>Aquario</i> que inicializa o aquário com as dimensões especificadas.
Parâmetros	<ul style="list-style-type: none"> <i>dimX</i>: largura do aquário; <i>dimY</i>: altura do aquário.
Comportamento	O construtor da classe <i>Aquario</i> é responsável por criar uma instância do aquário com as dimensões fornecidas.
Código	<pre> public Aquario (int dimX, int dimY) { this.dimX = dimX; this.dimY = dimY; this.aquario = new Peixe[dimX][dimY]; } </pre>
Método 2: <i>addPeixe(Peixe peixe)</i>	
Descrição	Adiciona um objeto <i>Peixe</i> ao aquário em uma posição específica.
Parâmetro	<i>peixe</i> : objeto <i>Peixe</i> a ser adicionado ao aquário.
Comportamento	1. Verifica se a posição do peixe é válida no aquário;

	2. Se válido, adiciona o peixe na posição especificada; 3. Se inválido, lança uma exceção <i>"IllegalArgumentException"</i> com a mensagem <i>"POSIÇÃO FORA DOS LIMITES DO AQUÁRIO"</i> .
Código	<pre> public void addPeixe(Peixe peixe){ if (verificaPosicao(peixe.getPosicaoX(), peixe.getPosicaoY())) { aquario[peixe.getPosicaoX()] [peixe.getPosicaoY()] = peixe; } else { throw new IllegalArgumentException("POSIÇÃO FORA DOS LIMITES DO AQUÁRIO"); } } </pre>
Método 3: <i>verificaPosicao(int x, int y)</i>	
Descrição	Verifica se a posição x e y estão dentro dos limites do aquário.
Parâmetros	<ul style="list-style-type: none"> • x: coordenada x da posição a ser verificada; • y: coordenada y da posição a ser verificada.
Comportamento	<ul style="list-style-type: none"> • Retorna <i>true</i> se a posição está dentro dos limites do aquário; • Retorna <i>false</i> se a posição está fora dos limites do aquário.
Código	<pre> private boolean verificaPosicao(int x, int y){ if (x >= dimX y >= dimY){ return false; } return true; } </pre>
Método 4: <i>getPosicao(int x, int y)</i>	
Descrição	Obtém o objeto <i>Peixe</i> na posição especificada.
Parâmetros	<ul style="list-style-type: none"> • x: coordenada x da posição a ser obtida; • y: coordenada y da posição a ser obtida.
Comportamento	Retorna o objeto <i>Peixe</i> na posição especificada ou <i>null</i> se a posição estiver fora dos limites do aquário.
Código	<pre> public Peixe getPosicao (int x, int y){ if (verificaPosicao(x,y)) { return aquario[x][y]; } } </pre>

	<pre> } else { return null; } } </pre>
Método 5: <i>setPosicaoNula(int x, int y)</i>	
Descrição	Define a posição especificada no aquário como nula (sem peixe).
Parâmetros	<ul style="list-style-type: none"> • x: coordenada x da posição a ser definida como nula; • y: coordenada y da posição a ser definida como nula.
Comportamento	É responsável por modificar o estado de uma posição específica no aquário, marcando-a como nula, o que significa que não há peixe naquela posição.
Código	<pre> public void setPosicaoNula(int x, int y){ //quando o peixe morre de inanição if (verificaPosicao(x,y)) { aquario[x][y]=null; } } </pre>
Método 6: <i>setPosicao(Peixe peixe, int x, int y)</i>	
Descrição	Define a posição especificada no aquário com o objeto <i>Peixe</i> fornecido.
Parâmetros	<ul style="list-style-type: none"> • x: coordenada x da posição a ser definida; • y: coordenada y da posição a ser definida.
Comportamento	<ol style="list-style-type: none"> 1. Verifica se a posição é válida no aquário; 2. Se válido, define a posição com o objeto <i>Peixe</i>; 3. Se inválido, lança uma exceção <i>IllegalArgumentException</i> com a mensagem "POSIÇÃO FORA DOS LIMITES DO AQUÁRIO".
Código	<pre> public void setPosicao(Peixe peixe, int x, int y) { if (verificaPosicao(x, y)) { aquario[x][y] = peixe; } else { throw new IllegalArgumentException("POSIÇÃO FORA DOS LIMITES DO AQUÁRIO"); } } </pre>

Método 7: <i>olhaAoRedor(int x, int y)</i>	
Descrição	Retorna um array de objetos <i>PosicaoAdjacente</i> representando as posições adjacentes à posição especificada.
Parâmetros	<ul style="list-style-type: none"> • x: coordenada x da posição central; • y: coordenada y da posição central.
Comportamento	Retorna um array de objetos <i>PosicaoAdjacente</i> contendo as posições adjacentes válidas.
Código	<pre> public PosicaoAdjacente[] olhaAoRedor(int x, int y){ PosicaoAdjacente[] pa; List<PosicaoAdjacente> lPa = new ArrayList<>(); for (int i = Math.max(0, x - 1); i <= Math.min(dimX - 1, x + 1); i++) { for (int j = Math.max(0, y - 1); j <= Math.min(dimY - 1, y + 1); j++){ if (i != x j != y) { PosicaoAdjacente p; p = new PosicaoAdjacente(i, j); lPa.add(p); } } } pa = lPa.toArray(new PosicaoAdjacente[lPa.size()]); return pa; } </pre>
Método 8: <i>pegaPeixesAoRedor(int x, int y)</i>	
Descrição	Retorna um array de objetos <i>Peixe</i> representando os peixes nas posições adjacentes à posição especificada.
Parâmetros	<ul style="list-style-type: none"> • x: coordenada x da posição central; • y: coordenada y da posição central.
Comportamento	Retorna um array de objetos <i>Peixe</i> contendo os peixes nas posições adjacentes válidas.
Código	<pre> public Peixe[] pegaPeixesAoRedor(int x, int y){ PosicaoAdjacente[] posAdjacente = olhaAoRedor(x, y); Peixe[] pa = new Peixe[posAdjacente.length]; for (int i=0; i< posAdjacente.length; i++){ pa[i] = aquario[posAdjacente[i].x] [posAdjacente[i].y]; } return pa; } </pre>

Método 9: <i>imprimeAquario()</i>	
Descrição	Imprime uma representação visual do aquário no console.
Parâmetros	Não há.
Comportamento	Para cada posição no aquário, imprime "[posição peixe]", onde 'posição' é a coordenada da posição no aquário e 'peixe' é a representação do objeto <i>Peixe</i> , ou uma <i>String</i> indicando que a posição está vazia.
Código	<pre> public void imprimeAquario(){ for (int i = 0; i < aquario.length; i++) { for (int j = 0; j < aquario[i].length; j++) { if (aquario[i][j] == null) { System.out.printf(" ["+ i + j + " " + "] "); } else { System.out.printf(" ["+ i + j + " "+aquario[i][j].toString() + "] "); } } System.out.println(); } } </pre>

iii. Classe Configuracoes

A classe *Configuracoes* é uma classe singleton que mantém as configurações do jogo, podendo ser acessada de qualquer parte do projeto. A seguir são detalhados os atributos, métodos e comportamentos da classe.

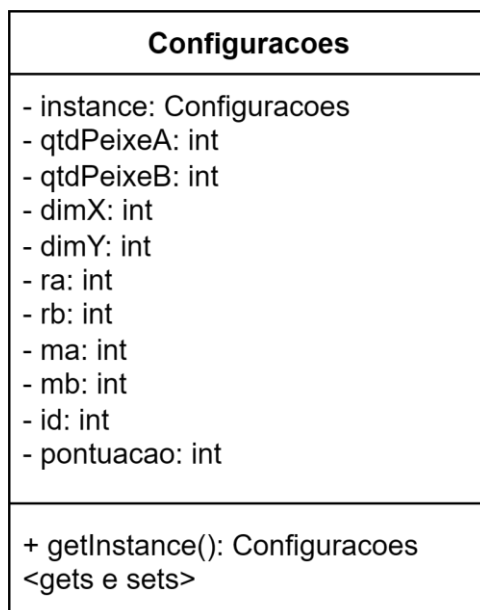


Figura: Diagrama UML da classe *Configuracoes*.

Método 1: <i>Configuracoes()</i>	
Descrição	O construtor é privado para garantir que a classe seja um singleton, ou seja, haverá apenas uma instância dela, a instância única é mantida como uma variável estática privada da própria classe.
Parâmetros	Não há.
Comportamento	O construtor da classe <i>Configuracoes</i> é responsável por criar uma instância do aquário com as dimensões fornecidas.
Código	<pre>private static Configuracoes instance= new Configuracoes();</pre>
Método 2: <i>getInstance()</i>	
Descrição	Método estático que retorna a única instância da classe <i>Configuracoes</i> .

Parâmetros	Não há.
Comportamento	Retorna a instância única da classe.
Código	<pre> ❑ public static Configuracoes getInstance() { return instance; } ❑ </pre>
Métodos 2 e 3: <i>addPontuacao()</i> e <i>addId()</i>	
Descrição	Incrementam em 1 os valores de <i>Id</i> e pontuação.
Parâmetros	Não há.
Código	<pre> ❑ public void addPontuacao(){ this.pontuacao = pontuacao+1; } public void addId(){ id = id+1; } ❑ </pre>
Método 4: <i>inicialDePontuacao()</i>	
Descrição	Inicializa o <i>Id</i> como 0 e a pontuação como 1.
Parâmetros	Não há.
Código	<pre> ❑ public void iniciaIDPontuacao(){ id = 0; pontuacao = 1; } ❑ </pre>
Métodos 5 ao 12: Controle da Quantidade de Peixes A e B	
<i>setQtdPeixeA(int qtdPeixeA);</i>	<p><i>qtdPeixeA</i>: nova quantidade de peixes do tipo A.</p> <pre> ❑ public void setQtdPeixeA(int qtdPeixeA) { this.qtdPeixeA = qtdPeixeA; } ❑ </pre>
<i>setQtdPeixeB(int qtdPeixeB);</i>	<p><i>qtdPeixeB</i>: nova quantidade de peixes do tipo B.</p> <pre> ❑ public void setQtdPeixeB(int qtdPeixeB) { this.qtdPeixeB = qtdPeixeB; } </pre>

	<input type="checkbox"/>
<i>getQtdPeixeA();</i>	<p>Retorna a quantidade de peixes do tipo A.</p> <pre> <input type="checkbox"/> public int getQtdPeixeA() { return qtdPeixeA; } <input type="checkbox"/> </pre>
<i>getQtdPeixeB();</i>	<p>Retorna a quantidade de peixes do tipo B.</p> <pre> <input type="checkbox"/> public int getQtdPeixeB() { return qtdPeixeB; } <input type="checkbox"/> </pre>
<i>addQtdPeixeA();</i>	<p>Incrementa a quantidade de peixes do tipo A por 1.</p> <pre> <input type="checkbox"/> public void addQtdPeixeA(){ qtdPeixeA = qtdPeixeA+1; } <input type="checkbox"/> </pre>
<i>subQtdPeixeA();</i>	<p>Decrementa a quantidade de peixes do tipo A por 1.</p> <pre> <input type="checkbox"/> public void subQtdPeixeA(){ qtdPeixeA = qtdPeixeA-1; } <input type="checkbox"/> </pre>
<i>addQtdPeixeB();</i>	<p>Incrementa a quantidade de peixes do tipo B por 1.</p> <pre> <input type="checkbox"/> public void addQtdPeixeB(){ qtdPeixeB = qtdPeixeB+1; } <input type="checkbox"/> </pre>
<i>subQtdPeixeB();</i>	<p>Decrementa a quantidade de peixes do tipo B por 1.</p> <pre> <input type="checkbox"/> public void subQtdPeixeB(){ qtdPeixeB = qtdPeixeB-1; } <input type="checkbox"/> </pre>
Métodos 12 ao 18: Dimensões do Aquário e Condições do Ambiente	
<ul style="list-style-type: none"> <i>getRA();</i> <i>getRB();</i> <i>setRA();</i> 	<p>Definem e retornam os valores que controlam as condições de reprodução dos peixes.</p> <pre> <input type="checkbox"/> public void setRa(int ra) { this.ra = ra; } </pre>

<ul style="list-style-type: none"> • setRB(); 	<pre> } public void setRb(int rb) { this.rb = rb; } public int getRa() { return ra; } public int getRb() { return rb; } } </pre>
<ul style="list-style-type: none"> • getMA(); • getMB(); • setMA(); • setMB(); 	<p>Definem e retornam os valores que controlam as condições para morte dos peixes.</p> <pre> public void setMa(int ma) { this.ma = ma; } public void setMb(int mb) { this.mb = mb; } public int getMa() { return ma; } public int getMb() { return mb; } </pre>
<ul style="list-style-type: none"> • getDimX(); • getDimY(); • setDimX(); • setDimY(); 	<p>Definem e retornam as dimensões do aquário.</p> <pre> public void setDimX(int dimX) { this.dimX = dimX; } public void setDimY(int dimY) { this.dimY = dimY; } </pre>

	<pre> } public int getDimX() { return dimX; } public int getDimY() { return dimY; } } </pre>
--	---

iv. Classe *Jogo*

A classe *Jogo* é responsável pela lógica do jogo, controlando o loop do jogo, as rodadas e a interação com o usuário. Abaixo estão as descrições detalhadas dos métodos e do funcionamento da classe.

Jogo
- aquario: Aquario
+ Jogo() + loopingJogo(): void - rodada(): void - resetMoveuPeixe(): void - verificaPosicao(x: int, y: int): void - lidaComPeixeA(x: int, y: int): void - lidaComPeixeB(x: int, y: int): void + iniciarJogo(): void - posicionaPeixesAInicio(): void - posicionaPeixesBInicio(): void - configuraAmbiente(): void - setQtdPeixeA(): void - setQtdPeixeB(): void - setDimX(): void - setDimY(): void - setRa(): void - setMa(): void - setRb(): void - setMb(): void - getRandomNumberUsing(max: int): int

Figura: Diagrama UML da classe *Jogo*.

Método: <i>Jogo()</i>	
Descrição	Inicializa a instância da classe <i>Jogo</i> e configura as

	pontuações iniciais.
Parâmetros	Não há.
Comportamento	Chama o método <i>inicialDPontuacao()</i> da classe <i>Configuracoes</i> para configurar as pontuações iniciais.
Código	
<pre> ❑ publicJogo() { Configuracoes.getInstance().iniciaIDPontuacao(); } ❑ </pre>	
Método: <i>loopingJogo()</i>	
Descrição	Controla o loop principal do jogo.
Parâmetros	Não há.
Comportamento	Inicia o jogo chamando o método <i>iniciarJogo()</i> e, em seguida, entra em um loop onde cada iteração representa uma rodada do jogo. Após cada rodada, pergunta ao usuário se deseja encerrar o jogo.
Código	
<pre> ❑ public void loopingJogo(){ boolean encerrar = false; iniciarJogo(); while (Configuracoes.getInstance().getQtdPeixeB() > 0 && !encerrar){ rodada(); System.out.println("Deseja encerrar? Digite \"s\" para sim e \n\" para não."); String str = sc.next().trim(); if (str.equals("s") str.equals("S")) encerrar = true; } } ❑ </pre>	
Método: <i>rodada()</i>	
Descrição	Incrementam em 1 os valores de <i>id</i> e pontuação.
Parâmetros	Não há.
Código	
<pre> ❑ private void rodada(){ </pre>	

```

System.out.println("\n=====
=====\\n");
System.out.println("RODADA " + Configuracoes.getInstance().getPontuacao() +
"\\n");
for (int i = 0; i < Configuracoes.getInstance().getDimX(); i++) {
    for (int j = 0; j < Configuracoes.getInstance().getDimY(); j++) {
        verificaPosicao(i,j);
        if (aquario.getPosicao(i,j) != null && !aquario.getPosicao(i,
cj).moveuNaRodada()){

            aquario.getPosicao(i,j).setM(aquario.getPosicao(i,j).getM() +1);
        }
    }
}
resetMoveuPeixe();
aquario.imprimeAquario();
Configuracoes.getInstance().addPontuacao();
}

```

Método: resetMoveuPeixe()

Descrição	Reseta o indicador de movimento de todos os peixes no aquário para falso.
Parâmetros	Não há.
Comportamento	Itera sobre todas as posições no aquário, definindo o indicador de movimento para falso para cada peixe presente.

Código

```

private void resetMoveuPeixe(){
    for (int i = 0; i < Configuracoes.getInstance().getDimX(); i++) {
        for (int j = 0; j < Configuracoes.getInstance().getDimY(); j++)
        {
            if (aquario.getPosicao(i,j) != null){
                aquario.getPosicao(i,j).setMoveuNaRodada(false);
            }
        }
    }
}

```

Método: verificaPosicao(int x, int y)

Descrição	Verifica o tipo de peixe em uma determinada posição e executa a lógica correspondente.
Parâmetros	<ul style="list-style-type: none"> • x: posição x; • y: posição y.
Comportamento	Chama <i>lidaComPeixeA</i> ou <i>lidaComPeixeB</i> dependendo do tipo de peixe na posição.

Código	
<pre> ❑private void verificaPosicao(int x, int y){ if (aquario.getPosicao(x,y) != null){ if (aquario.getPosicao(x,y) instanceof PeixeA){ lidaComPeixeA(x,y); } else { lidaComPeixeB(x,y); } } } ❑ </pre>	
Método: <i>lidaComPeixeA(int x, int y)</i>	
Descrição	Lida com a lógica específica para peixes do tipo A.
Parâmetros	<ul style="list-style-type: none"> • x: posição x; • y: posição y.
Comportamento	Verifica se o peixe morreu sozinho, se está apto para reproduzir ou se deve mover no aquário.
Código	
<pre> ❑private void lidaComPeixeA(int x, int y){ if (aquario.getPosicao(x,y).getM() == Configuracoes.getInstance().getMa()){ // nesse caso peixe morreu sozinho aquario.setPosicaoNula(x,y); Configuracoes.getInstance().subQtdPeixeA(); } else { PosicaoAdjacente[] pa = aquario.olhaAoRedor(x,y); if (aquario.getPosicao(x,y).getR() == Configuracoes.getInstance().getRa()){ //peixe esta apto a reproduzir aquario.getPosicao(x,y).reproduzir(aquario,pa); } else { aquario.getPosicao(x,y).moverNoAquario(x,y,aquario,pa); } } } ❑ </pre>	
Método: <i>lidaComPeixeB(int x, int y)</i>	
Descrição	Lida com a lógica específica para peixes do tipo B.
Parâmetros	<ul style="list-style-type: none"> • x: posição x; • y: posição y.
Comportamento	Verifica se o peixe morreu sozinho, se está apto para

	reproduzir ou se deve mover no aquário.
Código	
<pre> ❑private void lidaComPeixeB(int x, int y){ if (aquario.getPosicao(x,y).getM() == Configuracoes.getInstance().getMb()){ // nesse caso peixe morreu sozinho aquario.setPosicaoNula(x,y); Configuracoes.getInstance().subQtdPeixeB(); } else { PosicaoAdjacente[] pa = aquario.olhaAoRedor(x,y); if (aquario.getPosicao(x,y).getR() == Configuracoes.getInstance().getRb()){ //peixe esta apto a reproduzir aquario.getPosicao(x,y).reproduzir(aquario,pa); } else { aquario.getPosicao(x,y).moverNoAquario(x,y,aquario,pa); } } } ❑ </pre>	
Método: <i>iniciarJogo()</i>	
Descrição	Configura o ambiente inicial do jogo, cria o aquário e posiciona os peixes.
Parâmetros	Não há.
Comportamento	Chama métodos para configurar o ambiente e posicionar os peixes A e B no aquário.
Código	
<pre> ❑private void iniciarJogo(){ configuracaAmbiente(); this.aquario = new Aquario(Configuracoes.getInstance().getDimX(), Configuracoes.getInstance().getDimY()); posicionaPeixesAInicio(); posicionaPeixesBInicio(); aquario.imprimeAquario(); } ❑ </pre>	
Métodos: <i>posicionaPeixesAInicio()</i> e <i>posicionaPeixesBInicio()</i>	
Descrição	Posiciona peixes do tipo A ou B no início do jogo.
Parâmetros	Não há.
Comportamento	Gera peixes aleatórios e os posiciona no aquário, garantindo que não haja sobreposição.

Código	
<pre> private void posicionaPeixesAInicio(){ // so usado quando inicia ao jogo for (int i=0; i<Configuracoes.GetInstance().getQtdPeixeA(); i++){ boolean add=false; while (!add) { // não coloca peixe onde já tem peixe PeixeA peixe = new PeixeA(Configuracoes.GetInstance().getId(), getRandomNumberUsing(Configuracoes.GetInstance().getDimX()- 1), getRandomNumberUsing(Configuracoes.GetInstance().getDimY() - 1)); Configuracoes.GetInstance().addId(); if (aquario.getPosicao(peixe.getPosicaoX(),peixe.getPosicaoY())==null) { aquario.addPeixe(peixe); add = true; } } } } private void posicionaPeixesBInicio(){ // so usado quando inicia o jogo for (int i=0; i<Configuracoes.GetInstance().getQtdPeixeB(); i++){ boolean add=false; while (!add) { // não coloca peixe onde já tem peixe PeixeB peixe = new PeixeB(Configuracoes.GetInstance().getId(), getRandomNumberUsing(Configuracoes.GetInstance().getDimX() - 1), getRandomNumberUsing(Configuracoes.GetInstance().getDimY() - 1)); Configuracoes.GetInstance().addId(); if (aquario.getPosicao(peixe.getPosicaoX(),peixe.getPosicaoY())==null) { aquario.addPeixe(peixe); add = true; } } } } </pre>	
Método:configuracaAmbiente()	
Descrição	Configura o ambiente do aquário com base nas escolhas do usuário.
Parâmetros	Não há.
Comportamento	Solicita ao usuário as configurações iniciais do ambiente, valida as entradas e exibe a configuração final do ambiente.
Código	

```

❑private void configuracaAmbiente(){
    System.out.println(">>>>CONFIGURAÇÃO DO AMBIENTE DO AQUÁRIO<<<<");
    setDimX();
    setDimY();
    setQtdPeixeA();
    setQtdPeixeB();
    setRa();
    setMa();
    setRb();
    setMb();

    System.out.println("\n\n0 ambiente foi configurado com a seguinte
configuração: \n" +
        "    - Tamanho do aquário: " +
Configuracoes.getInstance().getDimX() + "x" +
Configuracoes.getInstance().getDimY()+ "\n" +
        "    - Quantidade peixe A: " +
Configuracoes.getInstance().getQtdPeixeA() + "\n" +
        "    - Quantidade peixe B: " +
Configuracoes.getInstance().getQtdPeixeB() + "\n" +
        "    - Resistencia peixe A: " +
Configuracoes.getInstance().getMa() + "\n" +
        "    - Limite reprodução peixe A: " +
Configuracoes.getInstance().getRa() + "\n" +
        "    - Resistencia peixe B: " +
Configuracoes.getInstance().getMb() + "\n" +
        "    - Limite reprodução peixe B: " +
Configuracoes.getInstance().getRb() + "\n" +

"=====
===\n");
}
❑

```

Métodos: *setQtdPeixeA()*, *setQtdPeixeB()*, *setDimX()*, *setDimY()*, *setRa()*, *setMa()*, *setRb()*, *setMb()*

Descrição	Métodos que recebem do usuário as dimensões do aquário e as condições do ambiente.
Parâmetros	Não ha.
Comportamento	Solicita ao usuário os valores correspondentes e os configura na instância de <i>Configuracoes</i> .

Código de *setDimX()*

```

❑private void setDimX(){
    boolean validacao = false;
    int in;
    while (!validacao) {
        System.out.println("Informe a dimensão \"x\" do aquário: ");
        try {
            in = sc.nextInt();
        } catch (InputMismatchException e) {
            System.out.println("Informe apenas números.");
        }
    }
}

```

<pre> sc.next(); in=-1; } if (in > 0) { validacao = true; Configuracoes.getInstance().setDimX(in); } } } } </pre>	
Método: <i>getRandomNumber Using(int max)</i>	
Descrição	Gera um número aleatório entre 0 e um valor máximo especificado.
Parâmetros	<i>max</i> : valor máximo para a geração aleatória.
Comportamento	Gera e retorna um número aleatório entre 0 e <i>max</i> .
Código	
<pre> private int getRandomNumberUsing(int max) { Random random = new Random(); return random.nextInt(max); } </pre>	

v. Classe *Peixe*

A classe abstrata *Peixe* representa um peixe no contexto do jogo. Possui quatro atributos: *id*, que é o identificador único de cada peixe, os inteiros *posicaoX* e *posicaoY* que indicam a posição do peixe no aquário e o booleano *moveuNaRodada* que indica que se o peixe já se moveu ou se reproduziu em uma rodada. Abaixo são detalhados os métodos dessa classe e seu comportamento.

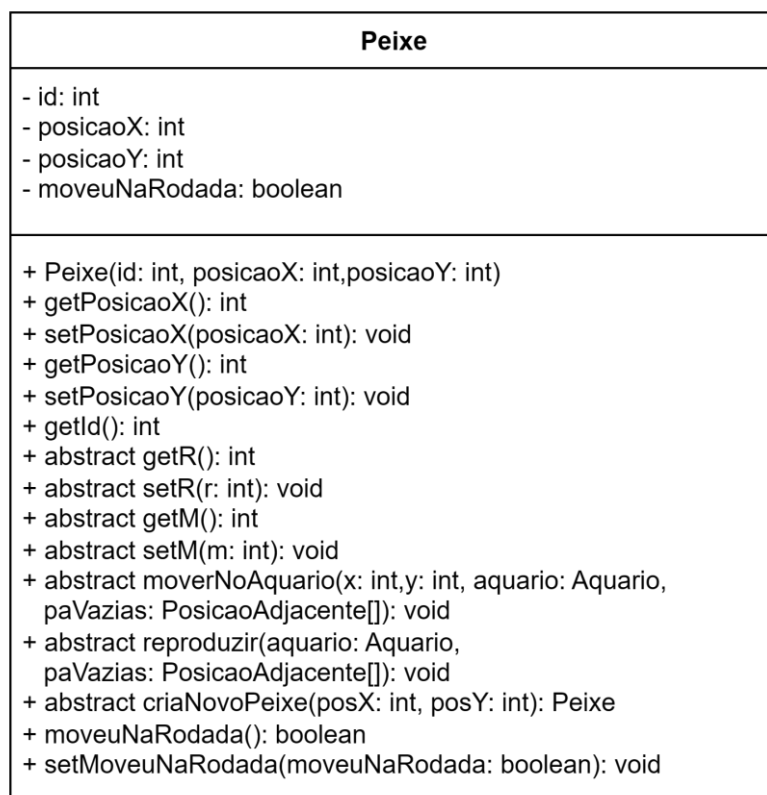


Figura: Diagrama UML da classe *Peixe*.

Método: <i>Peixe</i> (int id, int posicaoX, int posicaoY)	
Descrição	Inicializa uma instância da classe <i>Peixe</i> .
Parâmetros	<ul style="list-style-type: none">• <i>id</i>: identificador único do peixe;• <i>posicaoX</i>: posição x inicial do peixe no aquário;• <i>posicaoY</i>: posição y inicial do peixe no aquário.
Comportamento	Inicializa os atributos <i>id</i> , <i>posicaoX</i> , <i>posicaoY</i> e <i>moveuNaRodada</i> .
Código	
<pre>public Peixe(int id, int posicaoX, int posicaoY){</pre>	

<pre> this.id = id; this.posicaoX = posicaoX; this.posicaoY = posicaoY; this.moveuNaRodada = false; } nfiguracoes.getInstance().iniciaIDPontuacao(); } </pre>	
Métodos: <i>getPosicaoX()</i> , <i>setPosicaoX(int posicaoX)</i> , <i>getPosicaoY()</i> , <i>setPosicaoY(int posicaoY)</i>	
Descrição	Os métodos que definem e retornam a posição do peixe no aquário.
Parâmetros	<ul style="list-style-type: none"> <i>posicaoX</i>: nova posição x do peixe; <i>posicaoY</i>: nova posição y do peixe.
Código	
<pre> public int getPosicaoX() { return posicaoX; } public void setPosicaoX(int posicaoX) { this.posicaoX = posicaoX; } public int getPosicaoY() { return posicaoY; } public void setPosicaoY(int posicaoY) { this.posicaoY = posicaoY; } </pre>	
Método: getId()	
Descrição	Obtém o identificador único do peixe.
Parâmetros	Não há.
Comportamento	Retorna o identificador único do peixe.
Código	
<pre> public int getId(){ return id; } </pre>	

Método: <i>getR()</i>	
Descrição	Obtém o valor do atributo de condições de reprodução do peixe.
Parâmetros	Não há.
Comportamento	Método abstrato a ser implementado nas subclasses para retornar as condições de reprodução específicas do tipo de peixe.
Método: <i>setR(int r)</i>	
Descrição	Define o valor do atributo de condições de reprodução do peixe.
Parâmetros	<i>r</i> : novo valor de condição de reprodução.
Comportamento	Método abstrato a ser implementado nas subclasses para definir as condições de reprodução específicas do tipo de peixe.
Método: <i>getM()</i>	
Descrição	Obtém o valor do atributo de resistência do peixe.
Parâmetros	Não há.
Comportamento	Método abstrato a ser implementado nas subclasses para retornar a resistência específica do tipo de peixe.
Método: <i>setM(int m)</i>	
Descrição	Define o valor do atributo de resistência do peixe.
Parâmetros	<i>m</i> : novo valor de resistência.
Comportamento	Método abstrato a ser implementado nas subclasses para definir a resistência específica do tipo de peixe.
Método: <i>moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] paVazias)</i>	
Descrição	Implementa a lógica de movimentação do peixe no aquário.
Parâmetros	<ul style="list-style-type: none"> • <i>x</i>: nova posição x desejada; • <i>y</i>: nova posição y desejada; • <i>aquario</i>: instância do aquário; • <i>paVazias</i>: lista de posições adjacentes vazias.

Comportamento	Método abstrato a ser implementado nas subclasses para realizar a movimentação específica do tipo de peixe no aquário.
Métodos: <i>reproduzir(Aquario aquario, PosicaoAdjacente[] paVazias)</i>	
Descrição	Implementa a lógica de reprodução do peixe.
Parâmetros	<ul style="list-style-type: none"> • <i>aquario</i>: instância do aquário; • <i>paVazias</i>: lista de posições adjacentes vazias.
Comportamento	Método abstrato a ser implementado nas subclasses para realizar a reprodução específica do tipo de peixe.
Método: <i>criaNovoPeixe(int posX, int posY)</i>	
Descrição	Cria uma nova instância do peixe com base nas posições fornecidas.
Parâmetros	<ul style="list-style-type: none"> • <i>posX</i>: posição x da nova instância do peixe; • <i>posY</i>: posição y da nova instância do peixe.
Comportamento	Método abstrato a ser implementado nas subclasses para criar uma nova instância do tipo de peixe específico.
Métodos: <i>moveuNaRodada()</i>	
Descrição	Verifica se o peixe moveu na rodada atual.
Parâmetros	Não ha.
Comportamento	Retorna verdadeiro se o peixe moveu na rodada, falso caso contrário.
Código de <i>setDimX()</i> para exemplo	
<pre> ❑ public boolean moveuNaRodada(){ return this.moveuNaRodada; } ❑ </pre>	
Método: <i>setMoveuNaRodada(boolean moveuNaRodada)</i>	
Descrição	Define se o peixe moveu na rodada atual.
Parâmetros	<i>moveuNaRodada</i> : valor booleano indicando se o peixe moveu na rodada.

Comportamento	Define o status de movimento do peixe na rodada.
Código	
<pre> ❑ public void setMoveuNaRodada(boolean moveuNaRodada){ this.moveuNaRodada = moveuNaRodada; } ❑ </pre>	

vi. Classe *PeixeA*

A classe *PeixeA* representa um peixe do tipo A no contexto do jogo e é uma subclasse da classe abstrata *Peixe*. Ela estende a classe *Peixe*, herdando os atributos e métodos definidos nela, além de implementar os métodos abstratos especificados. Essa estrutura de herança permite que a classe *PeixeA* compartilhe características comuns com outras classes derivadas de *Peixe*, ao mesmo tempo em que proporciona flexibilidade para adicionar comportamentos específicos aos peixes do tipo A. Abaixo são detalhados os métodos dessa classe e seu comportamento.

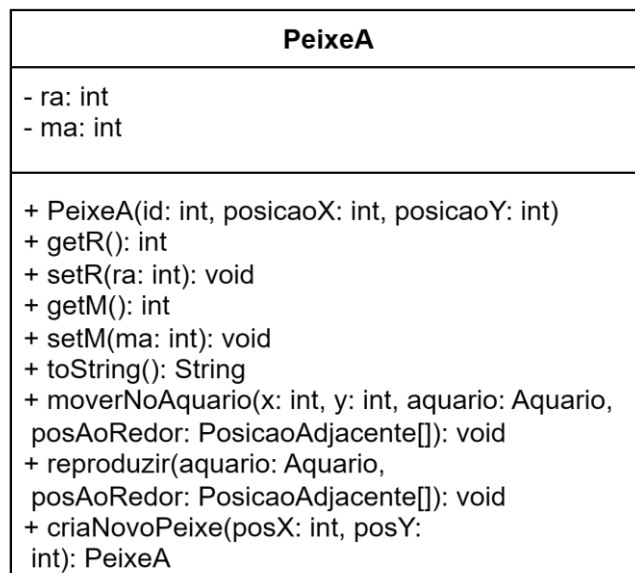


Figura: Diagrama UML da classe *PeixeA*.

Método: <i>PeixeA(int id, int posicaoX, int posicaoY)</i>	
Descrição	Inicializa uma instância da classe <i>PeixeA</i> .

Parâmetros	<ul style="list-style-type: none"> • <i>id</i>: identificador único do peixe A; • <i>posicaoX</i>: posição x inicial do peixe A no aquário; • <i>posicaoY</i>: posição y inicial do peixe A no aquário.
Comportamento	Chama o construtor da classe pai (<i>Peixe</i>) passando os parâmetros correspondentes e inicializa os atributos específicos de <i>PeixeA</i> (<i>ra</i> e <i>ma</i>) com zero.
Código	
<pre> public PeixeA(int id, int posicaoX, int posicaoY){ super(id, posicaoX, posicaoY); ra = 0; ma = 0; } </pre>	
Método: <i>getR()</i>	
Descrição	Obtém o valor do atributo de condições de reprodução do peixe.
Parâmetros	Não há.
Comportamento	Rtorna as condições de reprodução específicas do tipo de peixe.
Código	
<pre> public int getR() { return ra; } </pre>	
Método: <i>setR(int r)</i>	
Descrição	Define o valor do atributo de condições de reprodução do peixe.
Parâmetros	<i>r</i> : novo valor de condição de reprodução.
Comportamento	Define as condições de reprodução específicas do tipo de peixe.
Código	
<pre> public void setR(int ra) { this.ra = ra; } </pre>	

Método: <i>getM()</i>	
Descrição	Obtém o valor do atributo de resistência do peixe.
Parâmetros	Não há.
Comportamento	Retorna a resistência específica do tipo de peixe.
Código	
<pre> public int getM() { return ma; } </pre>	
Método: <i>setM(int m)</i>	
Descrição	Define o valor do atributo de resistência do peixe.
Parâmetros	<i>m</i> : novo valor de resistência.
Comportamento	Define a resistência específica do tipo de peixe.
Código	
<pre> public void setM(int ma) { this.ma = ma; } </pre>	
Método: <i>moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] posAoRedor)</i>	
Descrição	O método <i>moverNoAquario</i> é responsável por mover um peixe do tipo A no aquário, considerando as posições adjacentes disponíveis.
Parâmetros	<ul style="list-style-type: none"> <i>int x, int y</i>: representam as coordenadas atuais do peixe no aquário; <i>Aquario aquario</i>: a instância do aquário onde a operação ocorrerá; <i>PosicaoAdjacente[] posAoRedor</i>: vetor que contém as posições adjacentes ao peixe.
Comportamento	<ol style="list-style-type: none"> Laço de repetição: o método inicia um loop que percorre todas as posições adjacentes ao peixe, conforme fornecido no vetor <i>posAoRedor</i>; Verificação da validade da posição: para cada posição adjacente (<i>posAoRedor[i]</i>), verifica-se se ela é válida

	<p>(não nula);</p> <ol style="list-style-type: none"> Obtenção do peixe na posição atual: obtém-se o peixe atual (<i>auxPeixe</i>) na posição atual (x, y) do aquário usando o método <i>aquario.getPosicao(x, y)</i>; Verificação de movimento na rodada: verifica se o peixe atual (<i>auxPeixe</i>) não moveu na rodada usando o método <i>auxPeixe.moveuNaRodada()</i>; Verificação de posição vazia: se a posição adjacente está vazia (nula) conforme verificado por <i>aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y) == null</i>, o peixe pode ser movido para lá; Atualização do aquário: a posição atual do peixe ((x, y)) é apagada no aquário usando <i>aquario.setPosicaoNula(x, y)</i>. O peixe (<i>auxPeixe</i>) é movido para a nova posição adjacente usando <i>aquario.setPosicao(auxPeixe, posAoRedor[i].x, posAoRedor[i].y)</i>; Atualização do contador de reprodução (R): o atributo de reprodução (R) do peixe é incrementado em 1 usando <i>auxPeixe.setR(auxPeixe.getR() + 1)</i>; Marcação de movimento na rodada: a posição adjacente movida é marcada como tendo sido ocupada na rodada usando <i>aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y).setMoveuNaRodada(true)</i>. O loop é interrompido usando <i>break</i> após a movimentação bem-sucedida.
Código	
<pre> public void moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] posAoRedor) { for(int i=0; i<posAoRedor.length;i++){ if (posAoRedor[i] !=null) { Peixe auxPeixe = aquario.getPosicao(x,y); if (!auxPeixe.moveuNaRodada()) { if (aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y) == null) { aquario.setPosicaoNula(x, y); aquario.setPosicao(auxPeixe, posAoRedor[i].x, posAoRedor[i].y); auxPeixe.setR(auxPeixe.getR() + 1); aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y).setMoveuNaRodada(true); break; } } } } } </pre>	
Métodos: <i>reproduzir(Aquario aquario, PosicaoAdjacente[] paVazias)</i>	
Descrição	<p>O método <i>reproduzir</i> é responsável por permitir que um peixe do tipo A se reproduza no aquário, gerando uma nova instância de peixe nas posições adjacentes disponíveis.</p>

Parâmetros	<ul style="list-style-type: none"> • <i>Aquario aquario</i>: é a instância do aquário onde a reprodução ocorrerá. • <i>PosicaoAdjacente[] posAoRedor</i>: é um vetor que contém as posições adjacentes ao peixe.
Comportamento	<ol style="list-style-type: none"> 1. Laço de repetição: o método inicia um loop que percorre todas as posições adjacentes ao peixe, conforme fornecido no vetor <i>posAoRedor</i>; 2. Verificação da validade da posição: para cada posição adjacente (<i>posAoRedor[i]</i>), verifica-se se ela é válida (não nula); 3. Obtenção do peixe na posição adjacente: obtém-se o peixe (<i>peixe</i>) na posição adjacente (<i>posAoRedor[i].x</i>, <i>posAoRedor[i].y</i>) do aquário usando o método <i>aquario.getPosicao()</i>; 4. Verificação de posição vazia: se a posição adjacente está vazia (nula) conforme verificado por <i>peixe == null</i>, a reprodução pode ocorrer nessa posição; 5. Obtenção das coordenadas da posição vazia: obtém-se as coordenadas (<i>x</i>, <i>y</i>) da posição adjacente vazia (<i>posAoRedor[i].x</i>, <i>posAoRedor[i].y</i>); 6. Criação de um novo peixe: Utiliza o método <i>criaNovoPeixe(x, y)</i> para criar uma nova instância de peixe nas coordenadas obtidas; 7. Inserção do novo peixe no aquário: insere o novo peixe no aquário na posição adjacente vazia usando <i>aquario.setPosicao(criaNovoPeixe(x, y), x, y)</i>; 8. Interrupção do loop: o loop é interrompido usando <i>break</i> após a reprodução bem-sucedida em uma posição adjacente.
Código	
<pre> □ public void reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor) { for(int i=0; i<posAoRedor.length;i++){ if (posAoRedor[i] !=null) { Peixe peixe = aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y); if (peixe == null) { int x = posAoRedor[i].x; int y = posAoRedor[i].y; aquario.setPosicao(criaNovoPeixe(x,y), x, y); break; } } } } □ </pre>	
Método: <i>criaNovoPeixe(int posX, int posY)</i>	

Descrição	Cria uma nova instância do peixe A com base nas posições fornecidas.
Parâmetros	<ul style="list-style-type: none"> • <i>posX</i>: posição x da nova instância do peixe; • <i>posY</i>: posição y da nova instância do peixe.
Comportamento	Cria uma nova instância de <i>PeixeA</i> , inicializa seus atributos e incrementa o contador de identificadores e a quantidade de peixes A no aquário. Retorna a nova instância de peixe A.
Código	
<pre> ❑ public Peixe criaNovoPeixe(int posX, int posY) { PeixeA peixe = new PeixeA(Configuracoes.getInstance().getId(), posX, posY); peixe.setR(0); Configuracoes.getInstance().addId(); Configuracoes.getInstance().addQtdPeixeA(); return peixe; } ❑ </pre>	

vii. Classe *PeixeB*

A classe *PeixeB* representa um peixe do tipo B no contexto do jogo e é uma subclasse da classe abstrata *Peixe*. Ela estende a classe *Peixe*, herdando os atributos e métodos definidos nela, além de implementar os métodos abstratos especificados. Essa estrutura de herança permite que a classe *PeixeB* compartilhe características comuns com outras classes derivadas de *Peixe*, ao mesmo tempo em que proporciona flexibilidade para adicionar comportamentos específicos aos peixes do tipo B. Abaixo são detalhados os métodos dessa classe e seu comportamento.

PeixeB
- rb: int - mb: int
+ PeixeB(id: int, posicaoX: int, posicaoY: int) + getR(): int + setR(ra: int): void + getM(): int + setM(ma: int): void + toString(): String + moverNoAquario(x: int, y: int, aquario: Aquario, posAoRedor: PosicaoAdjacente[]): void + reproduzir(aquario: Aquario, posAoRedor: PosicaoAdjacente[]): void + criaNovoPeixe(posX: int, posY: int): PeixeB

Figura: Diagrama UML da classe *PeixeB*.

Método: <i>PeixeB(int id, int posicaoX, int posicaoY)</i>	
Descrição	Inicializa uma instância da classe <i>PeixeB</i> .
Parâmetros	<ul style="list-style-type: none"> • <i>id</i>: identificador único do peixe B. • <i>posicaoX</i>: posição x inicial do peixe B no aquário. • <i>posicaoY</i>: posição y inicial do peixe B no aquário.
Comportamento	Chama o construtor da classe pai (<i>Peixe</i>) passando os parâmetros correspondentes e inicializa os atributos específicos de <i>PeixeB</i> (<i>rb</i> e <i>mb</i>) com zero.
Código	
<pre> public PeixeB(int id, int posicaoX, int posicaoY){ super(id, posicaoX, posicaoY); rb = 0; mb = 0; </pre>	

} □	
Método: <i>getR()</i>	
Descrição	Obtém o valor do atributo de condições de reprodução do peixe.
Parâmetros	Não há.
Comportamento	Rtorna as condições de reprodução específicas do tipo de peixe.
Código	
□ <pre>public int getR() { return rb; }</pre> □	
Método: <i>setR(int r)</i>	
Descrição	Define o valor do atributo de condições de reprodução do peixe.
Parâmetros	<i>r</i> : novo valor de condição de reprodução.
Comportamento	Define as condições de reprodução específicas do tipo de peixe.
Código	
□ <pre>public void setR(int rb) { this.rb = rb; }</pre> □	
Método: <i>getM()</i>	
Descrição	Obtém o valor do atributo de resistência do peixe.
Parâmetros	Não há.
Comportamento	Retorna a resistência específica do tipo de peixe.
Código	
□ <pre>public int getM() {</pre>	

<pre> return mb; } </pre>	
Método: <i>setM(int m)</i>	
Descrição	Define o valor do atributo de resistência do peixe.
Parâmetros	<i>m</i> : novo valor de resistência.
Comportamento	Define a resistência específica do tipo de peixe.
Código	
<pre> public void setM(int mb) { this.mb = mb; } </pre>	
Método: <i>moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] paVazias)</i>	
Descrição	O método <i>moverNoAquario</i> é responsável por mover um peixe do tipo B no aquário, considerando as posições adjacentes disponíveis.
Parâmetros	<ul style="list-style-type: none"> • <i>int x, int y</i>: representam as coordenadas atuais do peixe no aquário; • <i>Aquario aquario</i>: é a instância do aquário onde a operação ocorrerá; • <i>PosicaoAdjacente[] posAoRedor</i>: é um vetor que contém as posições adjacentes ao peixe.
Comportamento	<ol style="list-style-type: none"> 9. O método utiliza um loop para percorrer as posições adjacentes ao peixe.; 10. Para cada posição adjacente, verifica se é uma posição válida (não nula); 11. Obtém o peixe que será movido (<i>auxPeixe</i>) utilizando o método <i>aquario.getPosicao(x, y)</i>; 12. Verifica se o peixe não moveu na rodada atual (<i>!auxPeixe.moveuNaRodada()</i>); 13. Se a posição adjacente contiver um peixe do tipo "A": <ol style="list-style-type: none"> a. Move o peixe para a posição adjacente; b. Define a posição anterior como nula; c. Marca a posição adjacente como já movida na rodada; d. Incrementa o atributo de reprodução (<i>R</i>) do peixe movido. 14. Se a posição adjacente estiver vazia (nula): <ol style="list-style-type: none"> a. Move o peixe para a posição adjacente; b. Incrementa o atributo de resistência (<i>M</i>) do peixe. 15. Encerra o loop.

Código	
<pre> ❑ public void moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] posAoRedor) { for(int i=0; i<posAoRedor.length;i++){//percorre vetor de posições adjacentes ao peixe selecionado if (posAoRedor[i] !=null) {//ve se tem uma posição válida Peixe auxPeixe = aquario.getPosicao(x,y);//auxPeixe recebe o peixe que vai ser movido try{ if (!auxPeixe.moveuNaRodada()) { if (aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y) instanceof PeixeA) { aquario.setPosicaoNula(x, y);//apaga o peixe da posicao anterior aquario.setPosicao(auxPeixe, posAoRedor[i].x, posAoRedor[i].y); aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y).setMoveuNaRodada(true); auxPeixe.setR(auxPeixe.getR() + 1); } else if (aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y) == null){ aquario.setPosicaoNula(x, y);//apaga o peixe da posicao anterior aquario.setPosicao(auxPeixe, posAoRedor[i].x, posAoRedor[i].y); auxPeixe.setM(auxPeixe.getM()+1); break; } } } catch (NullPointerException e){} } } } ❑ </pre>	
Métodos: <i>reproduzir(Aquario aquario, PosicaoAdjacente[] paVazias)</i>	
Descrição	Este método permite que o peixe do tipo "B" se reproduza no aquário, gerando uma nova instância de peixe nas posições adjacentes disponíveis.
Parâmetros	<ul style="list-style-type: none"> • <i>Aquario aquario</i>: instância do aquário onde ocorrerá a reprodução; • <i>PosicaoAdjacente[] posAoRedor</i>: vetor que contém as posições adjacentes ao peixe.

Comportamento	<ol style="list-style-type: none"> 16. Obtém a lista de peixes ao redor do peixe do tipo "B" usando <i>aquario.pegPeixesAoRedor()</i>; 17. Verifica se há peixes do tipo "B" nas posições adjacentes, definindo a variável <i>podeReproduzir</i>; 18. Se <i>podeReproduzir</i> for verdadeiro, indica que não há peixes do tipo "B" nas posições adjacentes e permite a reprodução; 19. Itera sobre as posições adjacentes para encontrar a primeira posição vazia; 20. Se encontra uma posição vazia: <ol style="list-style-type: none"> a. Cria uma nova instância de peixe do tipo "B" usando <i>criaNovoPeixe(x, y)</i>; b. Insere o novo peixe no aquário na posição adjacente vazia; c. Incrementa o atributo de resistência (<i>M</i>) do peixe pai.
Código	
<pre> public void reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor) { //pegando os peixes ao redor Peixe[] peixesAoRedor = aquario.pegPeixesAoRedor(this.getPosicaoX(), this.getPosicaoY()); //verificando se tem peixes tipoB ao redor boolean podeReproduzir = true; for (int i=0; i<peixesAoRedor.length; i++){ if (peixesAoRedor[i] instanceof PeixeB) { podeReproduzir = false; break; } } if (podeReproduzir) { //faz a reprodução do peixe B para a primeira célula livre disponível for (int i = 0; i < posAoRedor.length; i++) { if (posAoRedor[i] !=null) { Peixe peixe = aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y); if (peixe == null) { int x = posAoRedor[i].x; int y = posAoRedor[i].y; aquario.setPosicao(criaNovoPeixe(x, y), x, y); this.mb = this.mb +1; break; } } } } } </pre>	
Método: <i>criaNovoPeixe(int posX, int posY)</i>	

Descrição	Cria uma nova instância do peixe B com base nas posições fornecidas.
Parâmetros	<ul style="list-style-type: none"> • <i>posX</i>: posição x da nova instância do peixe; • <i>posY</i>: posição y da nova instância do peixe.
Comportamento	Cria uma nova instância de <i>PeixeB</i> , inicializa seus atributos e incrementa o contador de identificadores e a quantidade de peixes B no aquário. Retorna a nova instância de peixe B.
Código	
<pre> ❑public Peixe criaNovoPeixe(int posX, int posY) { PeixeB peixe = new PeixeB(Configuracoes.getInstance().getId(), posX, posY); Configuracoes.getInstance().addId(); Configuracoes.getInstance().addQtdPeixeB(); return peixe; } ❑ </pre>	

viii. Classe *PosicaoAdjacente*

A classe *PosicaoAdjacente* desempenha a função crucial de representar coordenadas na forma de um par de inteiros (x, y) dentro da matriz bidimensional que simboliza o aquário no contexto do jogo em questão. O parâmetro do construtor consiste nos inteiros x e y, que se referem, respectivamente, à linha e à coluna na matriz. A escolha de tornar os atributos públicos elimina a necessidade de métodos de encapsulamento. Essa classe desempenha um papel fundamental na identificação e manipulação de posições no aquário durante a execução do jogo.

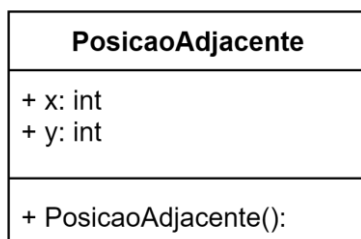


Figura: Diagrama UML da classe *PosicaoAdjacente*.

Código
<pre> ❑public class PosicaoAdjacente { </pre>

```
public int x, y;  
public PosicaoAdjacente(int x, int y){  
    this.x = x;  
    this.y = y;  
}  
  
public PosicaoAdjacente(){  
  
}  
}  
□
```


3. Teste Funcional

a. Critérios de Teste Funcional: Requisitos e Casos de Teste

Tabelas com a especificação dos requisitos de teste e casos de teste para atender aos critérios de *Teste Funcional* e *Particionamento em Classes de Equivalência*, *Análise de Valor Limite* e *Grafo Causa-Efeito*.

Tabela – Classes de Equivalência		
Configurações do Ambiente	Dimensões do Aquário	
	CE1	Dimensões maiores que 0: $m > 0 \ \&\& \ n > 0$
	CE2	Dimensões menores que 1: $m < 1 \ \parallel \ n < 1$
	Quantidade Inicial de peixes A e B	
	CE3	Números inteiros positivos: $x > 0 \ \&\& \ y > 0$
	CE4	Números negativos, não inteiros: $x < 0 \ \parallel \ y < 0 \ \parallel \ x, y \notin \mathbb{Z}^*$
	CE5	Quantidade de peixes maior que capacidade do aquário: $x > m * n \ \parallel \ y > m * n$
	Resistência e Condições de Reprodução	
	CE6	Números inteiros positivos: $Ra > 0 \ \&\& \ Ma > 0 \ \&\& \ Rb > 0 \ \&\& \ Mb > 0$
	CE7	Números negativos, não inteiros: $Ra < 0 \ \parallel \ Ma < 0 \ \parallel \ Rb < 0 \ \parallel \ Mb < 0$ $\parallel \ Ra, Ma, Rb, Mb \notin \mathbb{Z}^*$
Movimentação dos Peixes	Peixe A: Movimento	
	CE8	Posições adjacentes vazias
	CE9	Posições adjacentes ocupadas
	CE10	Movimentar duas vezes na mesma rodada
	Peixe A: Reprodução	
	CE11	Posições adjacentes vazias

	CE12	Posições adjacentes ocupadas
	CE13	Reprodução e Movimento na mesma rodada
	Peixe B: Movimento	
	CE14	Posições adjacentes vazias ou ocupadas por <i>PeixeA</i>
	CE15	Tentativa de movimentação para posição ocupada por <i>PeixeB</i>
	CE16	Movimentar duas vezes na mesma rodada
	Peixe B: Reprodução	
	CE17	Posições adjacentes vazias
	CE18	Posições adjacentes ocupadas
	CE19	Uma posição adjacente ocupada por peixe do tipo B
	CE20	Tentativa de reprodução após movimento na mesma rodada

Tabela – Casos de teste para o critério de particionamento em classes de equivalência.			
ID	Condições de Entrada	Saída Esperada	Classes Exercitadas (Requisitos de Teste)
CT1	<i>M</i> (número de linhas) <i>M = 5</i>	"Ambiente criado com sucesso."	CE1
CT2	<i>N</i> (número de colunas) <i>N=3</i>	"Ambiente criado com sucesso."	CE1
CT3	<i>X</i>	"Ambiente criado com sucesso."	CE1

	<p>(número de peixes do tipo A)</p> <p>$X=3$</p>		
CT4	<p>RA</p> <p>(movimentos seguidos para reprodução de peixes A)</p> <p>$RA = 5$</p>	"Ambiente criado com sucesso."	CE1
CT5	<p>MA</p> <p>(movimentos seguidos sem movimento de peixes A para morrer de fome)</p> <p>$MA = 10$</p>	"Ambiente criado com sucesso."	CE1
CT6	<p>RB</p> <p>(quantidade de peixes A que peixe B deve comer para reprodução)</p> <p>$RB = 5$</p>	"Ambiente criado com sucesso."	CE1
CT7	<p>MB</p> <p>(movimentos seguidos sem comer peixes A para morrer de fome)</p> <p>$MB = 10$</p>	"Ambiente criado com sucesso."	CE1

Tabela – Casos de teste para o critério análise de valor limite.			
ID	Condições de Entrada	Saída Esperada	Classes Exercitadas (Requisitos de Teste)
CT8	$\langle M = 5, N = 5, X = 10, Y = 15, RA = 3, MA = 5, RB = 2, MB = 4 \rangle$	"Ambiente criado com sucesso."	CE1, CE3, CE6
CT9	$\langle M = 0, N = 5, X = 3, Y = 8, RA = 2, MA = 3, RB = 1, MB = 5 \rangle$	Erro: Tamanho inválido de aquário	CE2
CT10	$\langle M = 4, N = 4, X = 20, Y = 16, RA = 2, MA = -1, RB = 3, MB = 6 \rangle$	Erro: "Quantidade de peixes inválida"	CE5
CT11	$\langle M = 4, N = -3, X = 20, Y = 16, RA = 2, MA = -1, RB = 3, MB = 6 \rangle$	Erro: Tamanho inválido de aquário	CE2
CT12	$\langle M = 6, N = 6, X = 15, Y = 10, RA = 4, MA = 2, RB = 5, MB = -2 \rangle$	Erro: "Condições do ambiente inválidas."	CE7

CT13	$\langle M = 3, N = 3, X = 0, Y = 0, RA = 0, MA = 0, RB = 0, MB = 0 \rangle$	Erro: "Quantidade de peixes inválida"	CE4
CT14	$\langle M = 20, N = 20, X = 400, Y = 400, RA = 10, MA = 10, RB = 20, MB = 20 \rangle$	"Ambiente criado com sucesso."	CE1, CE3, CE6
CT15	$\langle M=5, N=0, X=3, Y=3, RA=5, MA=10, RB=5, MB=10 \rangle$	"Tamanho inválido de aquário"	CE2
CT16	$\langle M=5, N=3, X=3, Y=0, RA=5, MA=10, RB=5, MB=10 \rangle$	Erro: "Quantidade de peixes inválida"	CE4
CT17	$\langle M=4, N=4, X=15, Y=17, RA=2, MA=1, RB=3, MB=6 \rangle$	Erro: "Quantidade de peixes inválida"	CE5
CT18	$\langle M=4, N=4, X=15, Y=0, RA=2, MA=1, RB=3, MB=6 \rangle$	Erro: "Quantidade de peixes inválida"	CE4

CT19	<M=4, N=4, X=5, Y=5, RA=0, MA=1, RB=3, MB=6>	Erro: "Condições do ambiente inválidas."	CE7
CT20	<M=4, N=4, X=5, Y=5, RA=2, MA=0, RB=3, MB=6>	Erro: "Condições do ambiente inválidas."	CE7
CT21	<M=4, N=4, X=5, Y=5, RA=2, MA=1, RB=0, MB=6>	Erro: "Condições do ambiente inválidas."	CE7
CT22	<M=4, N=4, X=5, Y=5, RA=2, MA=1, RB=3, MB=0>	Erro: "Condições do ambiente inválidas."	CE7

Técnica de Grafo de Causa-Efeito para o método *moverNoAquario()*:

Tabela – Determinação das Causas e Efeitos.			
Causas		Efeitos	
C1	Condição de Loop ($i < posAoRedor.length$)	E1	Guardar peixe a ser movido em var auxiliar
C2	$posicao[i]$ é válida (dentro da matriz)	E2	Mover o peixe para $posicao[i]$
C3	peixe não moveu na rodada	E3	Finaliza o método

C4	<i>posicao[i]</i> está vazia		
----	------------------------------	--	--

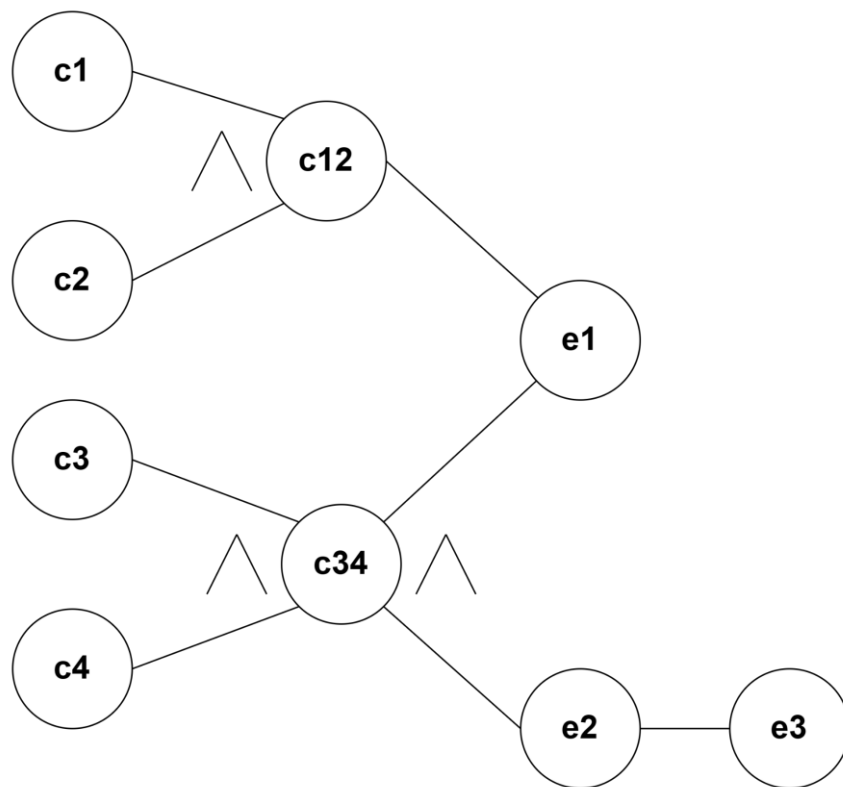


Figura: Grafo de causa e Efeito para o método *moverNoAquario()*.

Tabela – Casos de teste para atender os requisitos de teste causa-efeito.			
ID	Condições de Entrada	Saída Esperada	Requisito de Teste
CT23	<ul style="list-style-type: none"> <i>posAoRedor</i> com posições válidas; peixe não moveu na rodada; <i>posicao[i]</i> está vazia. 	Peixe na <i>posicao (x,y)</i> agora está na posição <i>posAoRedor[i]</i>	$c1 > c2 > c12 > e1 > e$
CT24	<ul style="list-style-type: none"> <i>posAoRedor</i> com posições válidas; peixe já moveu na rodada. 	Peixe não se move	$c1 > c2 > c12 > !e1 > e$
CT25	<ul style="list-style-type: none"> <i>posAoRedor</i> está vazio. 	Peixe não se move	$!c1$
CT26	<ul style="list-style-type: none"> <i>posAoRedor</i> com posições válidas; peixe não moveu na rodada; <i>posicao[i]</i> não está vazia. 	Peixe não se move, busca outra posição em <i>posAoRedor</i>	$c1 > c2 > e1 > c3 > !c4 > e3$

b. Implementação dos Casos de Teste

i. Caso de Teste: CT01

```
00  @Test
01  public void ct1(){
02      Jogo jogo = new Jogo();
03
04      int[] valores = {5,3,3,3,5,10,5,10};
05
06      jogo.auxIniciarJogo(valores);
07
08      assertEquals(5,Configuracoes.getInstance().getDimX());
09  }
```

Na linha 02 é iniciada um nova instância do objeto Jogo, em seguida, na linha 04 é criado um vetor de inteiros “valores” com os parâmetros iniciais do ambiente do jogo: $M=5$, $N=3$, $X=3$, $Y=3$, $RA=5$, $MA=10$, $RB=5$ e $MB=10$. Na linha 06 é chamado o método *auxIniciarJogo(int[] valores)* que é responsável por iniciar os parâmetros do ambiente do aquário e armazená-los na classe *Configuracoes*. Na linha 08 é chamado o método *assertEquals(int, int)* para verificar se o valor 5 foi armazenado na variável *dimX* da classe *Configuracoes*.

ii. Caso de Teste: CT09

```
00  @Test
01  public void ct9(){
02      Jogo jogo = new Jogo();
03      int[] valores = {0,5,3,8,2,3,1,5};
04
05      String str = jogo.auxIniciarJogo(valores);
06
07      assertEquals("Tamanho inválido de aquário", str);
08  }
```

Na linha 02 é iniciada um nova instância do objeto Jogo, em seguida, na linha 03 é criado um vetor de inteiros “valores” com os parâmetros iniciais do ambiente do jogo: $M=0$, $N=5$, $X=3$, $Y=8$, $RA=2$, $MA=3$, $RB=1$ e $MB=5$. Na linha 05 a variável *str* do tipo *String* recebe o resultado do método *auxIniciarJogo(int[] valores)*. O valor de $M=0$ deve disparar uma mensagem de erro dizendo “*Tamanho Inválido de aquário*”. Na linha 07 é chamado o método *assertEquals(String, String)* que verifica se a mensagem de erro foi disparada e armazenada na variável *str*.

iii. Caso de Teste: CT23

```
00  @Test
01  public void ct23(){
02      Aquario aquario = new Aquario(3,3);
03      Peixe peixe = new PeixeA(23,1,1);
04
05      aquario.addPeixe(peixe);
06
07      PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1,1);
08
09      peixe.moverNoAquario(1,1,aquario,posAoRedor);
10
11      assertEquals(23,aquario.getPosicao(0,0).getId());
12  }
```

Na linha 02 é iniciada uma nova instância do objeto *Aquario* com as dimensões $M=3$ e $N=3$. Em seguida na linha 03 é criado um objeto do tipo *PeixeA* com $Id=23$ e posição $x=1$ e $y=1$. Na linha 05 o peixe é adicionado no aquário na posição $(1,1)$. Na linha 07, as posições válidas ao redor do peixe 23 são armazenadas no vetor *posAoRedor*. Na linha 09 é chamado o método que move o peixe para uma nova posição no aquário, passando como parâmetros a posição $(1,1)$, o objeto *aquario* e o vetor com as posições válidas ao redor. Na linha 11 o método *assertEquals(int, int)* verifica se o peixe 23 foi movido para a posição $(0,0)$ n aquário.

c. Resultados da Execução dos Casos de Teste

Tabela com a descrição dos resultados da execução de cada caso de teste para os critérios de teste funcional.

Tabela – Resultados da execução dos testes para atender aos critérios de teste funcional.				
ID	Condições de Entrada	Saída Esperada	Saída Obtida	Requisitos de Teste
CT01	$M=5$	$dimX = 5$	$dimX = 5$	CE1
CT02	$N=3$	$dimY = 3$	$dimY = 3$	CE1
CT03	$X=3$	$qtdPeixeA = 3$	$qtdPeixeA = 3$	CE1
CT04	$Ra=5$	$ra = 5$	$ra = 5$	CE1
CT05	$Ma=10$	$ma = 10$	$ma = 10$	CE1
CT06	$Rb=5$	$rb =5$	$rb =5$	CE1

CT07	<i>Mb=10</i>	<i>mb = 10</i>	<i>mb = 10</i>	CE1
CT08	{5,5,10,15,3,5,2,4}	"Ambiente criado com sucesso."	"Ambiente criado com sucesso."	CE1, CE3, CE6
CT09	{0,5,3,8,2,3,1,5}	"Tamanho inválido de aquário"	"Tamanho inválido de aquário"	CE2
CT10	{4,4,20,16,2,-1,3,6}	"Quantidade de peixes inválida"	"Quantidade de peixes inválida"	CE5
CT11	{4,4,20,16,2,-1,3,6}	"Condições do ambiente inválidas."	"Condições do ambiente inválidas."	CE2
CT12	{6,6,15,10,4,2,5,-2}	"Condições do ambiente inválidas."	"Condições do ambiente inválidas."	CE7
CT13	{3,3,0,0,0,0,0,0}	"Quantidade de peixes inválida"	"Quantidade de peixes inválida"	CE4
CT14	{20,20,400,400,10,10,20,20}	"Ambiente criado com sucesso."	"Ambiente criado com sucesso."	CE1, CE3,CE6
CT15	{5,0,3,3,5,10,5,10}	"Tamanho inválido de aquário"	"Tamanho inválido de aquário"	CE2
CT16	{5,3,3,0,5,10,5,10}	"Quantidade de peixes inválida"	"Quantidade de peixes inválida"	CE4
CT17	{4,4,15,17,2,1,3,6}	"Quantidade de peixes inválida"	"Quantidade de peixes inválida"	CE5
CT18	{4,4,15,0,2,1,3,6}	"Quantidade de peixes inválida"	"Quantidade de peixes inválida"	CE4
CT19	{4,4,5,5,0,1,3,6}	"Condições do ambiente inválidas."	"Condições do ambiente inválidas."	CE7
CT20	{4,4,5,5,3,0,3,6}	"Condições do ambiente inválidas."	"Condições do ambiente inválidas."	CE7
CT21	{4,4,5,5,3,3,0,6}	"Condições do ambiente inválidas."	"Condições do ambiente inválidas."	CE7
CT22	{4,4,5,5,3,1,3,0}	"Condições do	"Condições do	CE7

		<i>ambiente inválidas."</i>	<i>ambiente inválidas."</i>	
CT23	<ul style="list-style-type: none"> • aquário 3x3 • peixe: id 23, na posição 1,1 	23 Id do peixe na nova posição (0,0)	23 Id do peixe na nova posição (0,0)	$c1 > c2 > c12 > e1 > e3$
CT24	<ul style="list-style-type: none"> • aquário 3x3 • peixe: id 23, na posição 1,1 • peixe.moveu NaRodada = true 	23 Id do peixe na posição antiga (1,1)	23 Id do peixe na posição antiga (1,1)	$c1 > c2 > c12 > !e1 > e3$
CT25	<ul style="list-style-type: none"> • aquário 3x3 • peixe: id 23, na posição 1,1 • posAoRedor está vazio 	23 Id do peixe na posição antiga (1,1)	23 Id do peixe na posição antiga (1,1)	$!c1$
CT26	<ul style="list-style-type: none"> • aquário 3x3 • peixe: id 23, na posição 1,1 • posicao[i] não está vazia 	23 Id do peixe na nova posição (0,1)	23 Id do peixe na nova posição (0,1)	$c1 > c2 > e1 > c3 > !c4 > e3$

d. Análise de Cobertura dos Casos de Teste

i. Análise de Cobertura de Fluxo de Controle

Abaixo serão analisados alguns dos métodos cobertos pela ferramenta *EclEmma* e como foi a cobertura dos testes no *Eclipse*, seguido com uma breve explicação do passo a passo para execução de cada método conforme o programa vai sendo compilado.

A primeira figura ilustra os casos de teste com os campos *Coverage*, *Covered Instructions*, *Missed Instructions* e *Total Instructions* de todas as classes do programa *Aquario*.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
▼ DCC168_JogoDoAquario-main	72,2 %	1.930	744	2.674
▼ src	72,2 %	1.930	744	2.674
▼ logica	52,9 %	777	692	1.469
> Jogo.java	49,6 %	314	319	633
> PeixeB.java	10,0 %	23	207	230
> PeixeA.java	58,2 %	96	69	165
> Aquario.java	82,2 %	221	48	269
> Configuracoes.java	68,6 %	83	38	121
> Peixe.java	79,5 %	31	8	39
> PosicaoAdjacente.java	75,0 %	9	3	12
▼ trabalhodcc168	0,0 %	0	52	52
> TrabalhoDCC168.java	0,0 %	0	52	52
▼ testes	100,0 %	1.153	0	1.153
> AquarioTest.java	100,0 %	1.153	0	1.153

Figura: Resultados do Eclemma para todas as classes do programa Aquario.

ii. PosicaoAdjacente: PosicaoAdjacente()

A classe *PosicaoAdjacente* é uma representação de coordenadas em um plano bidimensional. Possui dois atributos públicos, *x* e *y* (linha 4), que armazenam as coordenadas da posição. O construtor com parâmetros (linha 5) permite a inicialização imediata das coordenadas ao criar uma instância, enquanto o construtor padrão permite criar um objeto e atribuir valores posteriormente. Nas linhas 10-12 temos um construtor vazio que não foi coberto por não ter sido utilizado em nenhum lugar do programa e é útil apenas para fins de padrão do projeto.

A cobertura dos testes foi de 75% , como mostra na imagem no início da seção.

```

1 package logica;
2 public class PosicaoAdjacente {
3
4     public int x, y;
5     public PosicaoAdjacente(int x, int y){
6         this.x = x;
7         this.y = y;
8     }
9
10    public PosicaoAdjacente(){
11
12    }
13 }
14

```

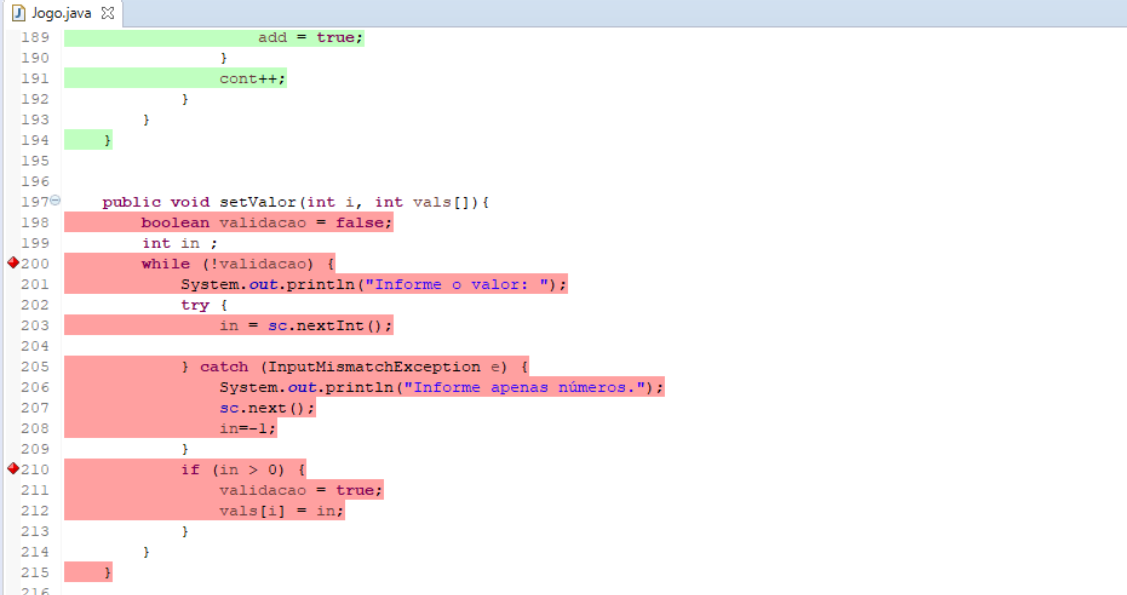
Figura: Classe *PosicaoAdjacente* com o método construtor sendo coberto pelo Eclemma.

iii. Jogo: setValor()

A classe Jogo tem cobertura do seu construtor nas linhas 7-13, depois os métodos *loopingJogo*, *rodada*, *resetMoveuPeixe*, *verificaPosicao*, *lidaComPeixeA*, *lidaComPeixeB* e *iniciarJogo* até a linha 99, não foram cobertos. O método *auxIniciarJogo* até a linha 194 é coberto 100% pela ferramenta.

O método analisado começa na linha 197, onde é responsável por receber um índice *i* e um array de inteiros *vals*. Durante a execução, ele utiliza um loop *while* para solicitar ao usuário a inserção de um valor positivo. A lógica de validação é implementada dentro do loop, onde é exibida uma mensagem pedindo o valor. O código utiliza um bloco *try-catch* para capturar possíveis exceções relacionadas à entrada do usuário, como *InputMismatchException*. Caso a entrada não seja um número inteiro, o programa exibe uma mensagem de erro e continua aguardando uma entrada válida. Uma vez que um valor positivo seja inserido, a validação é marcada como verdadeira, e o valor é atribuído ao array *vals* no índice *i*. Esse método é útil para garantir que apenas valores positivos sejam aceitos em uma determinada posição do array, proporcionando robustez à entrada do usuário.

A cobertura de testes foi de 49,6%.



```
189         add = true;
190     }
191     cont++;
192 }
193 }
194 }
195
196
197 public void setValor(int i, int vals[]){
198     boolean validacao = false;
199     int in ;
200     while (!validacao) {
201         System.out.println("Informe o valor: ");
202         try {
203             in = sc.nextInt();
204         } catch (InputMismatchException e) {
205             System.out.println("Informe apenas números.");
206             sc.next();
207             in=-1;
208         }
209     }
210     if (in > 0) {
211         validacao = true;
212         vals[i] = in;
213     }
214 }
215 }
216 }
```

Figura: Classe Jogo com o método *setValor* não sendo coberto pelo Eclemma.

iv. Aquario

A classe *Aquario* representa um aquário em um contexto de simulação de peixes, com foco na interação e controle desses animais. A matriz aquario é o componente central, armazenando instâncias da classe *Peixe* em suas células. O construtor da classe inicializa o aquário com dimensões especificadas.

O método *addPeixe* adiciona um peixe ao aquário, verificando se a posição está dentro dos limites, foi parcialmente coberto (linha 18). O método *verificaPosicao* (linha 19) é utilizado para validar se uma posição específica está dentro das dimensões do aquário.

Outros métodos, como *getPosicao*, *setPosicaoNula* e *setPosicao*, manipulam informações sobre a posição e presença de peixes no aquário. As operações *olhaAoRedor* e *pegaPeixesAoRedor* fornecem informações sobre as posições adjacentes válidas e os peixes nelas, respectivamente.

A função *imprimeAquario* exibe o estado atual do aquário no console, facilitando a visualização do ambiente de simulação. Na condição *else* na linha 23-24 a *exception*, não foi exercitada por não ter criado um caso de teste para cobrir a mesma. A classe utiliza variáveis locais e membros, como *dimX*, *dimY* e *aquario*, com associações definidas nos métodos para cumprir suas funcionalidades. Essa estrutura oferece uma base sólida para a implementação e controle de um sistema simulado de aquário com peixes.

A cobertura do Eclemma foi de 82.2%.

```

1 package logica;
2 import java.util.ArrayList;
3
4
5 public class Aquario {
6
7     private Peixe[][] aquario;
8
9     private final int dimX;
10    private final int dimY;
11
12    public Aquario (int dimX, int dimY){
13        this.dimX = dimX;
14        this.dimY = dimY;
15        this.aquario = new Peixe[dimX][dimY];
16    }
17
18    public void addPeixe(Peixe peixe){
19        if (verificaPosicao(peixe.getPosicaoX(), peixe.getPosicaoY())) {
20            aquario[peixe.getPosicaoX()][peixe.getPosicaoY()] = peixe;
21        }
22        else {
23            throw new IllegalArgumentException("POSIÇÃO FORA DOS LIMITES DO AQUÁRIO");
24        }
25    }
26
27    private boolean verificaPosicao(int x, int y){
28        if (x >= dimX || y >= dimY){
29            return false;
30        }
31        return true;
32    }
33
34    public Peixe getPosicao (int x, int y){
35        if (verificaPosicao(x,y)) {
36            return aquario[x][y];
37        }
38    }

```

Figura: Classe Aquario com alguns métodos sendo parcialmente cobertos pelo Eclemma.

v. Configuracoes

A classe *Configuracoes* é projetada para gerenciar as configurações em um contexto de simulação. Na linha 1, implementa um padrão singleton ao criar uma instância única da classe *Configuracoes*. O método estático *getInstance()* (linha 5) permite o acesso a essa instância global. As variáveis *qtdPeixeA* e *qtdPeixeB* (linha 7) armazenam a quantidade de peixes dos tipos A e B.

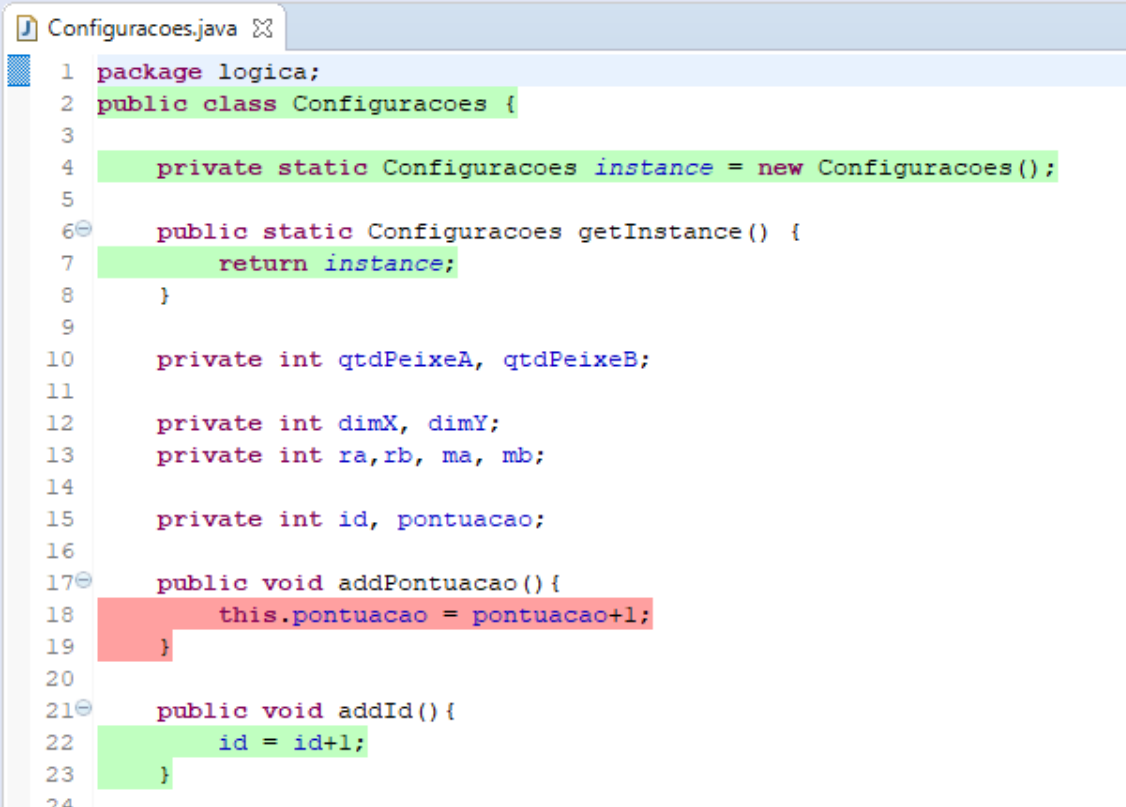
A classe possui variáveis para armazenar informações cruciais, como a quantidade de peixes dos tipos A e B, as dimensões do ambiente de simulação (*dimX* e *dimY*), e parâmetros específicos (*ra*, *rb*, *ma*, *mb*).

A função *inicialDPontuacao* é responsável por inicializar as variáveis de *Id* e pontuação. Essa classe encapsula as configurações essenciais para a simulação, oferecendo uma interface para acessar e modificar esses parâmetros de maneira organizada e centralizada. O uso de um padrão singleton assegura que apenas uma instância global dessas configurações exista no programa.

Operações para modificar configurações incluem métodos como *addPontuacao()* (linha 13), que incrementa a pontuação. A função *inicialDPontuacao()* (linha 21) inicializa as variáveis de *Id* e pontuação. Métodos *getters* (por exemplo, *getQtdPeixeA()*, linha 29) permitem a obtenção dos valores dessas configurações.

Essa classe centraliza e organiza as informações de configuração, facilitando o controle dinâmico desses parâmetros. O padrão *singleton* assegura que haja apenas uma instância global das configurações, promovendo consistência e coesão no programa. Métodos estáticos são disponibilizados para acessar essa instância única, garantindo uma configuração global no programa.

A cobertura de testes foi de 68,6%.



```
1 package logica;
2 public class Configuracoes {
3
4     private static Configuracoes instance = new Configuracoes();
5
6     public static Configuracoes getInstance() {
7         return instance;
8     }
9
10    private int qtdPeixeA, qtdPeixeB;
11
12    private int dimX, dimY;
13    private int ra,rb, ma, mb;
14
15    private int id, pontuacao;
16
17    public void addPontuacao() {
18        this.pontuacao = pontuacao+1;
19    }
20
21    public void addId() {
22        id = id+1;
23    }
24 }
```

Figura: Classe *Configuracoes* com o método *addPontuacao()* e *addId()*, sendo um coberto e outro não, pelo *EclEmma*.

Abaixo temos a tabela que mostra as comparações com os testes executados pela ferramenta EclEmma com os resultados da ferramenta Baduino que será utilizada na seção 3.4.2 no relatório: Análise de Cobertura de Fluxo de Dados, que visa identificar possíveis problemas relacionados à manipulação de dados, como inconsistências, vazamentos de informações, condições de corrida e outras questões que podem afetar a integridade e a segurança dos dados.

Tabela – Comparação de cobertura entre as ferramentas EclEmma e Baduino.		
Classe	EclEmma	Baduino
Jogo	49,6%	75,80%
PeixeA	58,2%	0%
PeixeB	10%	0%
Aquario	82,2%	40,13%
Configuracoes	68,6%	-
Peixe	79,5%	-
PosicaoAdjacente	75%	-

e. Análise de Cobertura de Fluxo de Dados

i. Método `auxIniciarJogo(int[] valores)` da classe `Jogo`

O caso de teste `ct01` atingiu a cobertura de 65,85% das associações *def-
uso* do método `auxIniciarJogo()`. A ferramenta Baduino não destacou as a linhas de código que foram cobertas nesse caso de teste.

Abaixo está a lista de associações são cobertas nesse caso de teste. As linhas de código estão demonstradas na figura a seguir.

```
100
101 public String auxIniciarJogo(int[] valores){
102     int dimX, dimY, qtdA, qtdB, ra, ma, rb, mb;
103     dimX = valores[0];
104     dimY = valores[1];
105     qtdA = valores[2];
106     qtdB = valores[3];
107     ra = valores[4];
108     ma = valores[5];
109     rb = valores[6];
110     mb = valores[7];
111     if (dimX<1 || dimY<1){
112         return "Tamanho inválido de aquário";
113     } else if (qtdA<1 || qtdB<1 || qtdA > dimX*dimY || qtdB > dimX*dimY){
114         return "Quantidade de peixes inválida";
115     } else if (ra<1 || ma<1 || rb<1 || mb<1){
116         return "Condições do ambiente inválidas.";
117     }else{
118         Configuracoes.getInstance().setDimX(dimX);
119         Configuracoes.getInstance().setDimY(dimY);
120         Configuracoes.getInstance().setQtdPeixeA(qtdA);
121         Configuracoes.getInstance().setQtdPeixeB(qtdB);
122         Configuracoes.getInstance().setRa(ra);
123         Configuracoes.getInstance().setMa(ma);
124         Configuracoes.getInstance().setRb(rb);
125         Configuracoes.getInstance().setMb(mb);
126
127         System.out.println("\n\nO ambiente foi configurado com a seguinte configuração: \n" +
128             " - Tamanho do aquário: " + Configuracoes.getInstance().getDimX() + "x" +
129             Configuracoes.getInstance().getDimY()+ "\n" +
130             " - Quantidade peixe A: " + Configuracoes.getInstance().getQtdPeixeA() + "\n" +
131             " - Quantidade peixe B: " + Configuracoes.getInstance().getQtdPeixeB() + "\n" +
132             " - Resistencia peixe A: " + Configuracoes.getInstance().getMa() + "\n" +
133             " - Limite reprodução peixe A: " + Configuracoes.getInstance().getRa() + "\n" +
134             " - Resistencia peixe B: " + Configuracoes.getInstance().getMb() + "\n" +
135             " - Limite reprodução peixe B: " + Configuracoes.getInstance().getRb() + "\n" +
136             "===== \n");
137
138         this.aquario = new Aquario(Configuracoes.getInstance().getDimX(),
139             Configuracoes.getInstance().getDimY());
140         posicionaPeixesAInicio();
141         posicionaPeixesBInicio();
142         aquario.imprimeAquario();
143         return "Ambiente criado com sucesso.";
144     }
145 }
```

Figura: Print do método `auxIniciarJogo()`.

▼ String auxIniciarLogo(int[])	🕒 (27/41) (65,85%)
🍷 (103, 138, this)	✅ true
🍷 (103, 140, this)	✅ true
🍷 (103, 141, this)	✅ true
🍷 (103, 142, this)	✅ true
🍷 (103, 127, out)	✅ true
🍷 (103, (111, 111), dimX)	✅ true
🍷 (103, (111, 112), dimX)	❌ false
🍷 (103, (113, 113), dimX)	✅ true
🍷 (103, (113, 114), dimX)	❌ false
🍷 (103, (113, 114), dimX)	❌ false
🍷 (103, (113, 115), dimX)	✅ true
🍷 (103, 118, dimX)	✅ true
🍷 (104, (111, 112), dimY)	❌ false
🍷 (104, (111, 113), dimY)	✅ true
🍷 (104, (113, 113), dimY)	✅ true
🍷 (104, (113, 114), dimY)	❌ false
🍷 (104, (113, 114), dimY)	❌ false
🍷 (104, (113, 115), dimY)	✅ true
🍷 (104, 119, dimY)	✅ true
🍷 (105, (113, 113), qtdA)	✅ true
🍷 (105, (113, 114), qtdA)	❌ false
🍷 (105, (113, 113), qtdA)	✅ true
🍷 (105, (113, 114), qtdA)	❌ false
🍷 (105, 120, qtdA)	✅ true
🍷 (106, (113, 113), qtdB)	✅ true
🍷 (106, (113, 114), qtdB)	❌ false
🍷 (106, (113, 114), qtdB)	❌ false
🍷 (106, (113, 115), qtdB)	✅ true
🍷 (106, 121, qtdB)	✅ true
🍷 (107, (115, 115), ra)	✅ true
🍷 (107, (115, 116), ra)	❌ false
🍷 (107, 122, ra)	✅ true
🍷 (108, (115, 115), ma)	✅ true
🍷 (108, (115, 116), ma)	❌ false
🍷 (108, 123, ma)	✅ true
🍷 (109, (115, 115), rb)	✅ true
🍷 (109, (115, 116), rb)	❌ false
🍷 (109, 124, rb)	✅ true
🍷 (110, (115, 116), mb)	❌ false
🍷 (110, (115, 118), mb)	✅ true
🍷 (110, 125, mb)	✅ true
> void configuracaAmbiente()	🕒 (0/12) (0,00%)
> void posicionaPeixesAlnicio()	🕒 (17/28) (60,71%)
> void posicionaPeixesBlnicio()	🕒 (22/28) (78,57%)

Figura: Associações Def-Uso cobertas pelo Caso de Teste CT01.

Project	DUAs Coverage
▼ trabalhodcc168	🕒 (129/645) (20,00%)
▼ logica	🕒 (129/633) (20,38%)
> logica.Aquario	🕒 (63/157) (40,13%)
▼ logica.Jogo	🕒 (66/303) (21,78%)
> void loopingJogo()	🕒 (0/12) (0,00%)
> void rodada()	🕒 (0/50) (0,00%)
> void resetMoveuPeixe()	🕒 (0/30) (0,00%)
> void verificaPosicao(int, int)	🕒 (0/22) (0,00%)
> void lidaComPeixeA(int, int)	🕒 (0/34) (0,00%)
> void lidaComPeixeB(int, int)	🕒 (0/34) (0,00%)
> String auxIniciarJogo(int[])	🕒 (27/41) (65,85%)
> void configuracaAmbiente()	🕒 (0/12) (0,00%)
> void posicionaPeixesAlnicio()	🕒 (17/28) (60,71%)
> void posicionaPeixesBlnicio()	🕒 (22/28) (78,57%)
> void setValor(int, int[])	🕒 (0/12) (0,00%)
> logica.PeixeA	🕒 (0/68) (0,00%)
> logica.PeixeB	🕒 (0/105) (0,00%)
> trabalhodcc168	🕒 (0/12) (0,00%)

Figura: Cobertura do Caso de Teste CT01.

Ao todo foram implementados vinte e dois casos de teste para os métodos envolvendo a inicialização do jogo. Com eles, foi atingida 100% de cobertura no método *auxIniciarJogo()*, 85% nos métodos de posicionamento inicial dos peixes e 41% de cobertura na classe *Jogo*. Abaixo está o resultado da cobertura das associações def-uso utilizando a ferramenta baduino.

Project	DUAs Coverage
▼ trabalhodcc168	🕒 (154/645) (23,88%)
▼ logica	🕒 (154/633) (24,33%)
> logica.Aquario	🕒 (65/157) (41,40%)
▼ logica.Jogo	🕒 (89/303) (29,37%)
> void loopingJogo()	🕒 (0/12) (0,00%)
> void rodada()	🕒 (0/50) (0,00%)
> void resetMoveuPeixe()	🕒 (0/30) (0,00%)
> void verificaPosicao(int, int)	🕒 (0/22) (0,00%)
> void lidaComPeixeA(int, int)	🕒 (0/34) (0,00%)
> void lidaComPeixeB(int, int)	🕒 (0/34) (0,00%)
> String auxIniciarJogo(int[])	🕒 (41/41) (100,00%)
> void configuracaAmbiente()	🕒 (0/12) (0,00%)
> void posicionaPeixesAlnicio()	🕒 (24/28) (85,71%)
> void posicionaPeixesBlnicio()	🕒 (24/28) (85,71%)
> void setValor(int, int[])	🕒 (0/12) (0,00%)
> logica.PeixeA	🕒 (0/68) (0,00%)
> logica.PeixeB	🕒 (0/105) (0,00%)
> trabalhodcc168	🕒 (0/12) (0,00%)

Figura: Cobertura total da classe Jogo.

ii. Método `moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] posAoRedor)` da classe `PeixeA`

O caso de teste *CT23* atingiu a cobertura de 65,85% das associações *def-
uso* do método *auxIniciarJogo()*. A ferramenta Baduino não destacou as linhas de código que foram cobertas nesse caso de teste.

Abaixo está a lista de associações não cobertas nesse caso de teste. As linhas de código estão demonstradas na figura a seguir.

```
31 public void moverNoAquario(int x, int y, Aquario aquario, PosicaoAdjacente[] posAoRedor) {  
32     //percorre vetor de posições adjacentes ao peixe selecionado  
33     for(int i=0; i<posAoRedor.length;i++){  
34         if (posAoRedor[i] !=null) { //ve se tem uma posição válida  
35             Peixe auxPeixe = aquario.getPosicao(x,y); //auxPeixe recebe o peixe que vai ser movido  
36             if (!auxPeixe.moveuNaRodada()) {  
37                 //verifica se a posição onde vai inserir está vazia  
38                 if (aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y) == null) {  
39                     aquario.setPosicaoNula(x, y); //apaga o peixe da posicao anterior  
40                     aquario.setPosicao(auxPeixe, posAoRedor[i].x, posAoRedor[i].y);  
41                     auxPeixe.setR(auxPeixe.getR() + 1);  
42                     aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y).setMoveuNaRodada(true);  
43                     break;  
44                 }  
45             }  
46         }  
47     }  
48 }  
49
```

Figura: Print do método `moverNoAquario()`.

logica.PeixeA	(22/68) (32,35%)
void moverNoAquario(int, int, Aquario, PosicaoAdjacente[])	(22/40) (55,00%)
(32, 34, x)	true
(32, 37, x)	true
(32, 34, y)	true
(32, 37, y)	true
(32, 34, aquario)	true
(32, (36, 37), aquario)	true
(32, (36, 32), aquario)	false
(32, 37, aquario)	true
(32, 38, aquario)	true
(32, 40, aquario)	true
(32, (32, 46), posAoRedor)	false
(32, (32, 33), posAoRedor)	true
(32, (33, 34), posAoRedor)	true
(32, (33, 32), posAoRedor)	false
(32, (36, 37), posAoRedor)	true
(32, (36, 32), posAoRedor)	false
(32, 38, posAoRedor)	true
(32, 40, posAoRedor)	true
(32, (32, 46), i)	false
(32, (32, 33), i)	true
(32, (33, 34), i)	true
(32, (33, 32), i)	false
(32, 32, i)	false
(32, (36, 37), i)	true
(32, (36, 32), i)	false
(32, 38, i)	true
(32, 40, i)	true
(34, (35, 36), auxPeixe)	true
(34, (35, 32), auxPeixe)	false
(34, 38, auxPeixe)	true
(34, 39, auxPeixe)	true
(32, (32, 46), i)	false
(32, (32, 33), i)	false
(32, (33, 34), i)	false
(32, (33, 32), i)	false
(32, 32, i)	false
(32, (36, 37), i)	false
(32, (36, 32), i)	false
(32, 38, i)	false
(32, 40, i)	false
void reproduzir(Aquario, PosicaoAdjacente[])	(0/28) (0,00%)

Figura: Associações Def-Uso cobertas pelo Caso de Teste CT23.

Ao todo foram implementados quatro casos de teste para o método `moverNoAquario()` da classe `peixeA`. Com eles, foi atingida 90% de cobertura no método `moverNoAquario()`, 52,94% de cobertura na classe `PeixeA`. Abaixo está o resultado da cobertura das associações def-uso utilizando a ferramenta Baduino.

Project	DUAs Coverage
▼ trabalhodcc168	(244/645) (37,83%)
▼ logica	(244/633) (38,55%)
> logica.Aquario	(119/157) (75,80%)
> logica.Jogo	(89/303) (29,37%)
▼ logica.PeixeA	(36/68) (52,94%)
> void moverNoAquario(int, int, Aquario, PosicaoAdjacente[])	(36/40) (90,00%)
> void reproduzir(Aquario, PosicaoAdjacente[])	(0/28) (0,00%)
> logica.PeixeB	(0/105) (0,00%)
> trabalhodcc168	(0/12) (0,00%)

Figura: Cobertura total da classe PeixeA.

f. Resultados

Nos casos de teste implementados não foram identificados defeitos.

Abaixo temos a tabela que mostra as comparações com os testes executados pela ferramenta Eclemma com os resultados da ferramenta Baduino que foi utilizada na seção 3.4.2 no relatório: Análise de Cobertura de Fluxo de Dados, que visa identificar possíveis problemas relacionados à manipulação de dados, como inconsistências, vazamentos de informações, condições de corrida e outras questões que podem afetar a integridade e a segurança dos dados.

Tabela – Comparação de cobertura entre as ferramentas Eclemma e Baduino.		
Classe	Eclemma	Baduino
Jogo	49,6%	29,37%
PeixeA	58,2%	52,94%
PeixeB	10%	0%
Aquario	82,2%	75,80%
Configuracoes	68,6%	-
Peixe	79,5%	-
PosicaoAdjacente	75%	-

4. Teste Estrutural - Critérios Baseados em Fluxo de Controle

a. Fluxo de Controle: Requisitos e Casos de Teste

i. Grafo de Fluxo de Controle

Para aplicar os critérios baseados em fluxo de controle escolhemos o método `olhaAoRedor(int x, int y)` da classe `Aquário`. O código fonte do método está detalhado na seção i do capítulo 2.

```
01  /*1*/ public PosicaoAdjacente[] olhaAoRedor(int x, int y){
02  /*1*/      PosicaoAdjacente[] pa;
03  /*1*/      List<PosicaoAdjacente> IPa = new ArrayList<>();
04  /*1,7*/      for (int i = Math.max(0, x - 1); i <= Math.min(dimX - 1, x + 1); i++) {
05  /*2,6*/          for (int j = Math.max(0, y - 1); j <= Math.min(dimY - 1, y + 1); j++) {
06  /*3*/              if (i != x || j != y) {
07  /*4*/                  PosicaoAdjacente p = new PosicaoAdjacente(i,j);
08  /*4*/                  IPa.add(p);
09  /*5*/              }
10  /*6*/          }
11  /*7*/      }
12  /*8*/      pa = IPa.toArray(new PosicaoAdjacente[IPa.size()]);
13  /*8*/      return pa;
14  /*9*/ }
```

Figura: Código fonte do método `olhaAoRedor()`.

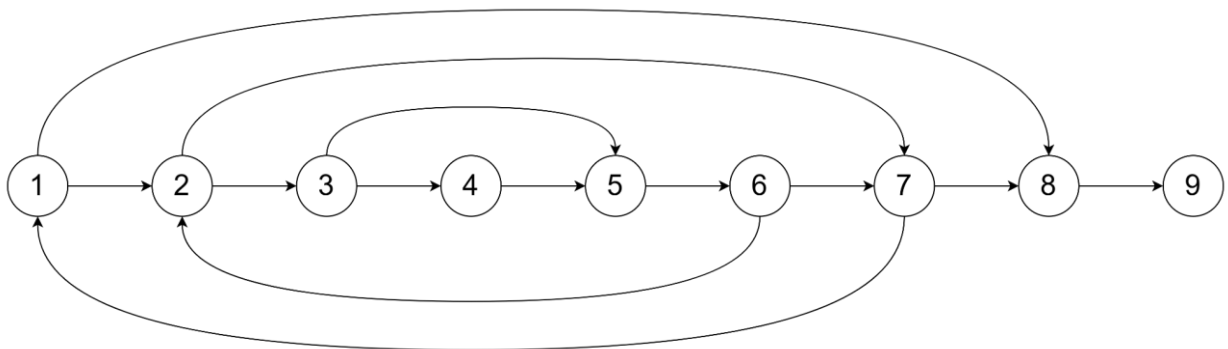


Figura: Grafo de Fluxo de Controle do método `olhaAoRedor()`.

Esse é um passo a passo simplificado do método *olhaAoRedor()*, destacando os principais pontos em cada parte do código. Cada nó representa uma ação ou conjunto de ações executadas sequencialmente no método:

21. Início do Método /*1*/

- a. Declaração do método *olhaAoRedor()* com parâmetros x e y;
- b. Inicialização do array pa;
- c. Inicialização da lista IPa para armazenar posições adjacentes;
- d. Início do loop externo para percorrer as linhas da matriz(coordenada i);
- e. Cálculo da condição do loop para a coordenada i.

22. Loop interno /*2*/

- a. Início do loop interno para percorrer as colunas da matriz (coordenada j);
- b. Cálculo da condição do loop para a coordenada j.

23. Condição de Exclusão da Posição Central /*3*/

- a. Verificação se a posição atual (i, j) não é igual à posição central (x, y);

24. Criação e Adição da Posição Adjacente /*4*/

- a. Criação do objeto PosicaoAdjacente para a posição (i, j);
- b. Adição do objeto à lista IPa;

25. Fim da Condição de Exclusão da Posição Central

26. Fim do Loop Interno Para a Coordenada j /*5*/

27. Fim do Loop Externo Para a Coordenada i /*6*/

28. Conversão e Retorno

- a. Conversão da lista para um array de PosicaoAdjacente;
- b. Retorno do array de posições adjacentes;

29. Fim do Método

ii. Critérios Todos os Nós e Todos os Arcos

Tabela – Requisitos e casos de teste todos os nós.			
Requisito de Teste	Caso de Teste *		
	ID	Entrada	Saída Esperada
<1,2,3,4,5,6,7,8,9>			
	ct27	x=1 ,y=1	"0,0 0,1 0,2 1,0 1,2 2,0 2,1 2,2 "
	ct28	x=0, y=0	"0,1 1,0 1,1 "

*considerando uma matriz 3x3

Tabela – Requisitos e casos de teste todos os arcos.			
Requisito de Teste	Caso de Teste *		
	ID	Entrada	Saída Esperada
<(1,2), (1,8), (2,3), (2,7), (3,4), (3,5), (4,5), (5,6), (6,2), (6,7), (7,1), (7,8), (8,9)>			
	ct29	x=1 ,y=2	"0,1 0,2 0,3 1,1 1,3 2,1 2,2 2,3 "
	ct30	x=2, y=2	"1,1 1,2 1,3 2,1 2,3 "

*considerando uma matriz 3x4

iii. Critérios Todos os Caminhos Simples e Livres de Laço

Tabela – Requisitos e casos de teste todos os caminhos simples.			
Requisito de Teste	Caso de Teste*		
	ID	Entrada	Saída Esperada
<1,8,9>	ct31	x=0, y=0	""

*considerando uma matriz 0x0

Tabela – Requisitos e casos de teste todos os caminhos livres de laço.			
Requisito de Teste	Caso de Teste*		
	ID	Entrada	Saída Esperada
<1,8,9>	ct31	x=0, y=0	""

*considerando uma matriz 0x0

iv. Critérios Todos os Caminhos

Tabela – Requisitos e casos de teste todos os caminhos completos.			
Requisito de Teste	Caso de Teste*		
	ID	Entrada	Saída Esperada
<(1,2), (1,8), (2,3), (2,7), (3,4), (3,5), (4,5), (5,6), (6,2), (6,7), (7,1), (7,8), (8,9)>	ct27	x=1 ,y=1	"0,0 0,1 0,2 1,0 1,2 2,0 2,1 2,2 "

*considerando uma matriz 3x3

b. Implementação dos Casos de Teste

i. Caso de Teste: CT27

```
01 @Test
02 public void ct27() {
03     Aquario aquario = new Aquario(3,3);
04     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 1);
05     String str = "";
06     for (int i=0; i< posAoRedor.length; i++){
07         str = str + posAoRedor[i].x + ", " + posAoRedor[i].y + " | ";
08     }
09     assertEquals("0,0 | 0,1 | 0,2 | 1,0 | 1,2 | 2,0 | 2,1 | 2,2 | ",
09         str);
10 }
```

Na linha 03, uma nova instância do objeto Aquario é criada com dimensões 3x3. Em seguida, na linha 04, o método olhaAoRedor é invocado no objeto aquario, passando as coordenadas (1, 1) como argumento. Esse método provavelmente retorna um array de objetos PosicaoAdjacente, representando as posições ao redor da coordenada (1, 1). A partir da linha 05 até a linha 08, um loop for itera sobre o array posAoRedor, construindo uma string str que contém as coordenadas x e y de cada posição ao redor, separadas por vírgula e seguidas por "|". Essa string é então comparada, na linha 09, usando o método assertEquals, com a string esperada "0,0 | 0,1 | 0,2 | 1,0 | 1,2 | 2,0 | 2,1 | 2,2 |".

ii. Caso de Teste: CT28

```
01 @Test
02 public void ct28() {
03     Aquario aquario = new Aquario(3,3);
04     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(0, 0);
05     String str = "";
06     for (int i=0; i< posAoRedor.length; i++){
07         str = str + posAoRedor[i].x + ", " + posAoRedor[i].y + " | ";
08     }
09     assertEquals("0,1 | 1,0 | 1,1 | ", str);
10 }
```

Na linha 03, uma nova instância do objeto Aquario é criada com dimensões 3x3. Em seguida, na linha 04, o método olhaAoRedor é invocado no objeto aquario, passando as coordenadas (0, 0) como argumento. Esse método provavelmente retorna um array de objetos PosicaoAdjacente, representando as posições ao redor da coordenada (0, 0). A partir da linha 05 até a linha 08, um loop for itera sobre o array posAoRedor, construindo uma string str que contém as coordenadas x e y de cada posição ao redor, separadas por vírgula e

seguidas por "|". Essa string é então comparada, na linha 09, usando o método `assertEquals`, com a string esperada "0,1 | 1,0 | 1,1 |".

iii. Caso de Teste: CT29

```
01 @Test
02 public void ct29() {
03     Aquario aquario = new Aquario(3,4);
04     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 2);
05     String str = "";
06     for (int i=0; i< posAoRedor.length; i++){
07         str = str + posAoRedor[i].x + ", " + posAoRedor[i].y + " | ";
08     }
09     assertEquals("0,1 | 0,2 | 0,3 | 1,1 | 1,3 | 2,1 | 2,2 | 2,3 | ",
09         str);
10 }
```

Na linha 03, uma nova instância do objeto `Aquario` é criada com dimensões 3x4. Em seguida, na linha 04, o método `olhaAoRedor` é invocado no objeto `aquario`, passando as coordenadas (1, 2) como argumento. Esse método provavelmente retorna um array de objetos `PosicaoAdjacente`, representando as posições ao redor da coordenada (1, 2). A partir da linha 05 até a linha 08, um loop `for` itera sobre o array `posAoRedor`, construindo uma string `str` que contém as coordenadas `x` e `y` de cada posição ao redor, separadas por vírgula e seguidas por "|". Essa string é então comparada, na linha 09, usando o método `assertEquals`, com a string esperada "0,1 | 0,2 | 0,3 | 1,1 | 1,3 | 2,1 | 2,2 | 2,3 |".

iv. Caso de Teste: CT30

```
01 @Test
02 public void ct30() {
03     Aquario aquario = new Aquario(3,4);
04     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(2, 2);
05     String str = "";
06     for (int i=0; i< posAoRedor.length; i++){
07         str = str + posAoRedor[i].x + ", " + posAoRedor[i].y + " | ";
08     }
09     assertEquals("1,1 | 1,2 | 1,3 | 2,1 | 2,3 | ", str);
10 }
```

Na linha 03, uma nova instância do objeto `Aquario` é criada com dimensões 3x4. Em seguida, na linha 04, o método `olhaAoRedor` é invocado no objeto `aquario`, passando as coordenadas (2, 2) como argumento. Esse método provavelmente retorna um array de objetos `PosicaoAdjacente`, representando as

posições ao redor da coordenada (2, 2). A partir da linha 05 até a linha 08, um loop for itera sobre o array posAoRedor, construindo uma string str que contém as coordenadas x e y de cada posição ao redor, separadas por vírgula e seguidas por "|". Essa string é então comparada, na linha 09, usando o método assertEquals, com a string esperada "1,1 | 1,2 | 1,3 | 2,1 | 2,3 |".

v. Caso de Teste: CT31

```

01 @Test
02 public void ct31() {
03     Aquario aquario = new Aquario(0, 0);
04     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(0, 0);
05     String str = "";
06     for (int i=0; i< posAoRedor.length; i++){
07         str = str + posAoRedor[i].x + ", " + posAoRedor[i].y + " | ";
08     }
09     assertEquals("", str);
10 }

```

Na linha 03, uma nova instância do objeto Aquario é criada com dimensões 0x0, indicando um aquário vazio ou sem posições válidas. Na linha 04, o método olhaAoRedor é invocado no objeto aquario, passando as coordenadas (0, 0) como argumento. Esse método provavelmente retorna um array de objetos PosicaoAdjacente, representando as posições ao redor da coordenada (0, 0). A partir da linha 05 até a linha 08, um loop for itera sobre o array posAoRedor, construindo uma string str que contém as coordenadas x e y de cada posição ao redor, separadas por vírgula e seguidas por "|". Na linha 09, a string str resultante da iteração é comparada com a string esperada "". Isso significa que o teste está verificando se, para um aquário de dimensões 0x0, o método olhaAoRedor retorna um array vazio.

c. Resultados da Execução dos Casos de Teste

Tabela – Resultados da execução dos testes para atender aos critérios todos os caminhos.				
ID	Condições de Entrada	Saída Esperada	Saída Obtida	Requisito de Teste
ct27	x=1 ,y=1	"0,0 0,1 0,2 1,0 1,2 2,0 2,1 2,2 "	"0,0 0,1 0,2 1,0 1,2 2,0 2,1 2,2 "	<1,2,3,4,5,6,7,8,9>
ct28	x=0, y=0	"0,1 1,0 1,1 "	"0,1 1,0 1,1 "	
ct29	x=1 ,y=2	"0,1 0,2 0,3 1,1 1,3 2,1 2,2 2,3 "	"0,1 0,2 0,3 1,1 1,3 2,1 2,2 2,3 "	<(1,2), (1,8), (2,3), (2,7), (3,4), (3,5), (4,5), (5,6), (6,2),

ct30	x=2, y=2	"1,1 1,2 1,3 2,1 2,3 "	"1,1 1,2 1,3 2,1 2,3 "	(6,7), (7,1), (7,8), (8,9)>
ct31	x=0, y=0	""	""	<1,8,9>

d. Análise de Cobertura dos Casos de Teste

i. Análise de Cobertura de Fluxo de Controle

O método 'olhaAoRedor' tem a finalidade de retornar um conjunto de objetos 'PosicaoAdjacente', os quais representam as posições adjacentes válidas à coordenada específica (x, y) em um aquário. A verificação é feita para garantir que posições inválidas sejam representadas como nulas no array resultante.

O método inicia declarando um array de 'PosicaoAdjacente' chamado 'pa' e uma lista 'IPa' do tipo 'ArrayList' de 'PosicaoAdjacente', utilizada para armazenar as posições adjacentes válidas.

Em seguida, dois loops 'for' aninhados são empregados para percorrer as posições ao redor da coordenada (x, y). O primeiro loop ('i') varia dentro dos limites do aquário nas dimensões x, enquanto o segundo loop ('j') varia dentro dos limites do aquário nas dimensões y.

Dentro dos loops, é verificado se a posição atual (i, j) é diferente da posição de origem (x, y), assegurando que a posição atual não seja incluída como adjacente a si mesma.

Para cada posição adjacente válida, um novo objeto 'PosicaoAdjacente' é criado com coordenadas (i, j) e adicionado à lista 'IPa'.

Ao término dos loops, a lista de posições adjacentes ('IPa') é convertida para um array de 'PosicaoAdjacente' ('pa'), que é então retornado pelo método.

Resumidamente, o método encapsula a lógica de identificar e retornar as posições adjacentes válidas a partir de uma coordenada específica no aquário, considerando as dimensões do aquário (dimX, dimY).

```

58 public PosicaoAdjacente[] olhaAoRedor(int x, int y){
59     //retorna posicoes adjacentes válidas onde não é válido fica nulo
60     PosicaoAdjacente[] pa;
61     List<PosicaoAdjacente> lPa = new ArrayList<>();
62
63     for (int i = Math.max(0, x - 1); i <= Math.min(dimX - 1, x + 1); i++) {
64         for (int j = Math.max(0, y - 1); j <= Math.min(dimY - 1, y + 1); j++) {
65             // Verifica se a posição não é a posição atual
66             if (i != x || j != y) {
67                 PosicaoAdjacente p = new PosicaoAdjacente(i,j);
68                 lPa.add(p);
69             }
70         }
71     }

```

Figura: Print do método olhaAoRedor().

ii. Análise de Cobertura de Fluxo de Dados

A partir dos critérios baseados em Fluxo de Controle foram implementados cinco casos de testes para o método olhaAoRedor(intx, int y) da classe Aquário. A ferramenta escolhida para a análise de cobertura de fluxo de dados foi a ferramenta Baduíno.

O caso de teste ct27 atingiu a cobertura de 90,70% das associações de uso do método olhaAoRedor(), as linhas não cobertas estão destacadas no print abaixo.

```

58 public PosicaoAdjacente[] olhaAoRedor(int x, int y){
59     //retorna posicoes adjacentes válidas onde não é válido fica nulo
60     PosicaoAdjacente[] pa;
61     List<PosicaoAdjacente> lPa = new ArrayList<>();
62
63     for (int i = Math.max(0, x - 1); i <= Math.min(dimX - 1, x + 1); i++) {
64         for (int j = Math.max(0, y - 1); j <= Math.min(dimY - 1, y + 1); j++) {
65             // Verifica se a posição não é a posição atual
66             if (i != x || j != y) {
67                 PosicaoAdjacente p = new PosicaoAdjacente(i,j);
68                 lPa.add(p);
69             }
70         }
71     }
72
73     pa = lPa.toArray(new PosicaoAdjacente[lPa.size()]);
74     return pa;
75 }

```

Figura: Print do método olhaAoRedor().

object	DUAs Coverage
▼ ● PosicaoAdjacente[] olhaAoRedor(int, int)	🕒 (39/43) (90,70%)
🔗 (60, (62, 72), this)	✅ true
🔗 (60, (62, 63), this)	✅ true
🔗 (60, (63, 62), this)	✅ true
🔗 (60, (63, 65), this)	✅ true
🔗 (60, (62, 72), x)	✅ true
🔗 (60, (62, 63), x)	✅ true
🔗 (60, (65, 65), x)	✅ true
🔗 (60, (65, 66), x)	✅ true
🔗 (60, 63, y)	✅ true
🔗 (60, (63, 62), y)	✅ true
🔗 (60, (63, 65), y)	✅ true
🔗 (60, (65, 66), y)	✅ true
🔗 (60, (65, 63), y)	✅ true
🔗 (60, (63, 62), this.dimY)	✅ true
🔗 (60, (63, 65), this.dimY)	✅ true
🔗 (60, (62, 72), this.dimX)	✅ true
🔗 (60, (62, 63), this.dimX)	✅ true
🔗 (60, 67, lPa)	✅ true
🔗 (60, 72, lPa)	✅ true
🔗 (62, (62, 72), i)	❌ false
🔗 (62, (62, 63), i)	✅ true
🔗 (62, (65, 65), i)	❌ false
🔗 (62, (65, 66), i)	✅ true
🔗 (62, 66, i)	✅ true
🔗 (62, 62, i)	✅ true
🔗 (63, (63, 62), j)	❌ false
🔗 (63, (63, 65), j)	✅ true
🔗 (63, 66, j)	✅ true
🔗 (63, 63, j)	✅ true
🔗 (63, (65, 66), j)	✅ true
🔗 (63, (65, 63), j)	❌ false
🔗 (63, (63, 62), j)	✅ true
🔗 (63, (63, 65), j)	✅ true
🔗 (63, 66, j)	✅ true
🔗 (63, 63, j)	✅ true
🔗 (63, (65, 66), j)	✅ true
🔗 (63, (65, 63), j)	✅ true
🔗 (62, (62, 72), i)	✅ true
🔗 (62, (62, 63), i)	✅ true
🔗 (62, (65, 65), i)	✅ true
🔗 (62, (65, 66), i)	✅ true
🔗 (62, 66, i)	✅ true
🔗 (62, 62, i)	✅ true
> ● Peixe[] pegaPeixesAoRedor(int, int)	🕒 (0/15) (0,00%)

Figura: Associações Def-Uso cobertas pelo Caso de Teste ct27.

Os casos de teste ct28, ct29 e ct30 atingiram a cobertura de 95,35% das associações def-uso do método `olhaAoRedor()`, as linhas não cobertas estão destacadas no print abaixo.

```

58 public PosicaoAdjacente[] olhaAoRedor(int x, int y){
59     //retorna posicoes adjacentes válidas onde não é válido fica nulo
60     PosicaoAdjacente[] pa;
61     List<PosicaoAdjacente> lPa = new ArrayList<>();
62
63     for (int i = Math.max(0, x - 1); i <= Math.min(dimX - 1, x + 1); i++) {
64         for (int j = Math.max(0, y - 1); j <= Math.min(dimY - 1, y + 1); j++) {
65             // Verifica se a posição não é a posição atual
66             if (i != x || j != y) {
67                 PosicaoAdjacente p = new PosicaoAdjacente(i,j);
68                 lPa.add(p);
69             }
70         }
71     }
72
73     pa = lPa.toArray(new PosicaoAdjacente[lPa.size()]);
74     return pa;
75 }

```

Figura: Print do método olhaAoRedor().

PosicaoAdjacente[] olhaAoRedor(int, int)	(41/43) (95,35%)
(61, (63, 73), this)	✓ true
(61, (63, 64), this)	✓ true
(61, (64, 63), this)	✓ true
(61, (64, 66), this)	✓ true
(61, (63, 73), x)	✓ true
(61, (63, 64), x)	✓ true
(61, (66, 66), x)	✓ true
(61, (66, 67), x)	✓ true
(61, 64, y)	✓ true
(61, (64, 63), y)	✓ true
(61, (64, 66), y)	✓ true
(61, (66, 67), y)	✓ true
(61, (66, 64), y)	✓ true
(61, (64, 63), this.dimY)	✓ true
(61, (64, 66), this.dimY)	✓ true
(61, (63, 73), this.dimX)	✓ true
(61, (63, 64), this.dimX)	✓ true
(61, 68, lPa)	✓ true
(61, 73, lPa)	✓ true
(63, (63, 73), i)	✗ false
(63, (63, 64), i)	✓ true
(63, (66, 66), i)	✓ true
(63, (66, 67), i)	✓ true
(63, 67, i)	✓ true
(63, 63, i)	✓ true
(64, (64, 63), j)	✗ false
(64, (64, 66), j)	✓ true
(64, 67, j)	✓ true
(64, 64, j)	✓ true
(64, (66, 67), j)	✓ true
(64, (66, 64), j)	✓ true
(64, (64, 63), j)	✓ true
(64, (64, 66), j)	✓ true
(64, 67, j)	✓ true
(64, 64, j)	✓ true
(64, (66, 67), j)	✓ true
(64, (66, 64), j)	✓ true
(63, (63, 73), i)	✓ true
(63, (63, 64), i)	✓ true
(63, (66, 66), i)	✓ true
(63, (66, 67), i)	✓ true
(63, 67, i)	✓ true
(63, 63, i)	✓ true

Figura: Associações Def-Uso cobertas pelos Casos de Teste ct28, ct29 e ct30.

O caso de teste ct31 atingiu a cobertura de 97,67% das associações def-
uso do método *olhaAoRedor()*, as linhas não cobertas estão destacadas no print
abaixo.

```

58 public PosicaoAdjacente[] olhaAoRedor(int x, int y){
59     //retorna posicoes adjacentes válidas onde não é válido fica nulo
60     PosicaoAdjacente[] pa;
61     List<PosicaoAdjacente> lPa = new ArrayList<>();
62
63     for (int i = Math.max(0, x - 1); i <= Math.min(dimX - 1, x + 1); i++) {
64         for (int j = Math.max(0, y - 1); j <= Math.min(dimY - 1, y + 1); j++) {
65             // Verifica se a posição não é a posição atual
66             if (i != x || j != y) {
67                 PosicaoAdjacente p = new PosicaoAdjacente(i,j);
68                 lPa.add(p);
69             }
70         }
71     }
72
73     pa = lPa.toArray(new PosicaoAdjacente[lPa.size()]);
74     return pa;
75 }

```

Figura: Print do método *olhaAoRedor()*.

PosicaoAdjacente[] olhaAoRedor(int, int)		(42/43) (97,67%)
(61, (63, 73), this)		✓ true
(61, (63, 64), this)		✓ true
(61, (64, 63), this)		✓ true
(61, (64, 66), this)		✓ true
(61, (63, 73), x)		✓ true
(61, (63, 64), x)		✓ true
(61, (66, 66), x)		✓ true
(61, (66, 67), x)		✓ true
(61, 64, y)		✓ true
(61, (64, 63), y)		✓ true
(61, (64, 66), y)		✓ true
(61, (66, 67), y)		✓ true
(61, (66, 64), y)		✓ true
(61, (64, 63), this.dimY)		✓ true
(61, (64, 66), this.dimY)		✓ true
(61, (63, 73), this.dimX)		✓ true
(61, (63, 64), this.dimX)		✓ true
(61, 68, lPa)		✓ true
(61, 73, lPa)		✓ true
(63, (63, 73), i)		✓ true
(63, (63, 64), i)		✓ true
(63, (66, 66), i)		✓ true
(63, (66, 67), i)		✓ true
(63, 67, i)		✓ true
(63, 63, i)		✓ true
(64, (64, 63), j)		✗ false

Figura: Associações Def-Uso cobertas pelo Caso de Teste ct31.

e. Resultados

Nos casos de teste implementados não foram identificados defeitos.

De acordo com a ferramenta Baduino, os casos de testes ct27, ct28, ct29, ct30 e ct31 atingiram: 97,67% das associações def-uso do método *olhaAoRedor()*, 77,71% das associações def-uso da classe *Aquario* e 41,94% das associações def-uso do projeto completo.

▼ trabalhodcc168	🕒 (302/720) (41,94%)
▼ logica	🕒 (247/633) (39,02%)
▼ logica.Aquario	🕒 (122/157) (77,71%)
> void addPeixe(Peixe)	🕒 (5/7) (71,43%)
> boolean verificaPosicao(int, int)	🕒 (6/12) (50,00%)
> Peixe getPosicao(int, int)	🕒 (7/10) (70,00%)
> void setPosicaoNula(int, int)	🕒 (7/10) (70,00%)
> void setPosicao(Peixe, int, int)	🕒 (8/11) (72,73%)
> PosicaoAdjacente[] olhaAoRedor(int, int)	🕒 (42/43) (97,67%)
> Peixe[] pegaPeixesAoRedor(int, int)	🕒 (0/15) (0,00%)
> void imprimeAquario()	🕒 (47/49) (95,92%)
> logica.Jogo	🕒 (89/303) (29,37%)
> logica.PeixeA	🕒 (36/68) (52,94%)
> logica.PeixeB	🕒 (0/105) (0,00%)
> testes	🕒 (55/75) (73,33%)
> trabalhodcc168	🕒 (0/12) (0,00%)

Figura: Porcentagem das Associações Def-Usso Cobertas na Ferramenta Baduino.

Abaixo temos a tabela que mostra as comparações com os testes executados pela ferramenta EclEmma com os resultados da ferramenta Baduino.

Tabela – Comparação de cobertura entre as ferramentas EclEmma e Baduino.		
Classe	EclEmma	Baduino
Jogo	49,6%	29,37%
PeixeA	95,8%	52,94%
PeixeB	49,6%	0%
Aquario	94,8%	77,71%
Configuracoes	80,2%	-
Peixe	79,5%	-
PosicaoAdjacente	75%	-

5. Teste Estrutural - Critérios Baseados em Fluxo de Dados

Para aplicar os critérios baseados em fluxo de controle escolhemos o método `reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor)` da classe `PeixeB`.

a. Fluxo de Dados: Requisitos e Casos de Teste

i. Grafo Def-Uso

Tabela – Código fonte e definição dos nós do método <code>reproduzir()</code>		
L	Nó	Instruções
56	1	<code>public void reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor) {</code>
59	1	<code>Peixe[] peixesAoRedor = aquario.pegaPeixesAoRedor(this.getPosicaoX() , this.getPosicaoY());</code>
62	1	<code>boolean podeReproduzir = true;</code>
63	1,5	<code>for (int i=0; i<peixesAoRedor.length; i++){</code>
64	2	<code>if (peixesAoRedor[i] instanceof PeixeB) {</code>
65	3	<code>podeReproduzir = false;</code>
66	3	<code>break;</code>
67	4	<code>}</code>
68	5	<code>}</code>
70	6	<code>if (podeReproduzir) {</code>
72	7,13	<code>for (int i = 0; i < posAoRedor.length; i++) {</code>
73	8	<code>if (posAoRedor[i] !=null) {</code>
74	9	<code>Peixe peixe = aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y);</code>
75	9	<code>if (peixe == null) {</code>
76	10	<code>int x = posAoRedor[i].x;</code>
77	10	<code>int y = posAoRedor[i].y;</code>
78	10	<code>aquario.setPosicao(criaNovoPeixe(x, y), x, y);</code>
79	10	<code>this.mb = this.mb +1;</code>
80	10	<code>break;</code>
81	11	<code>}</code>

82	12	}
83	13	}
84	14	}
86	15	}

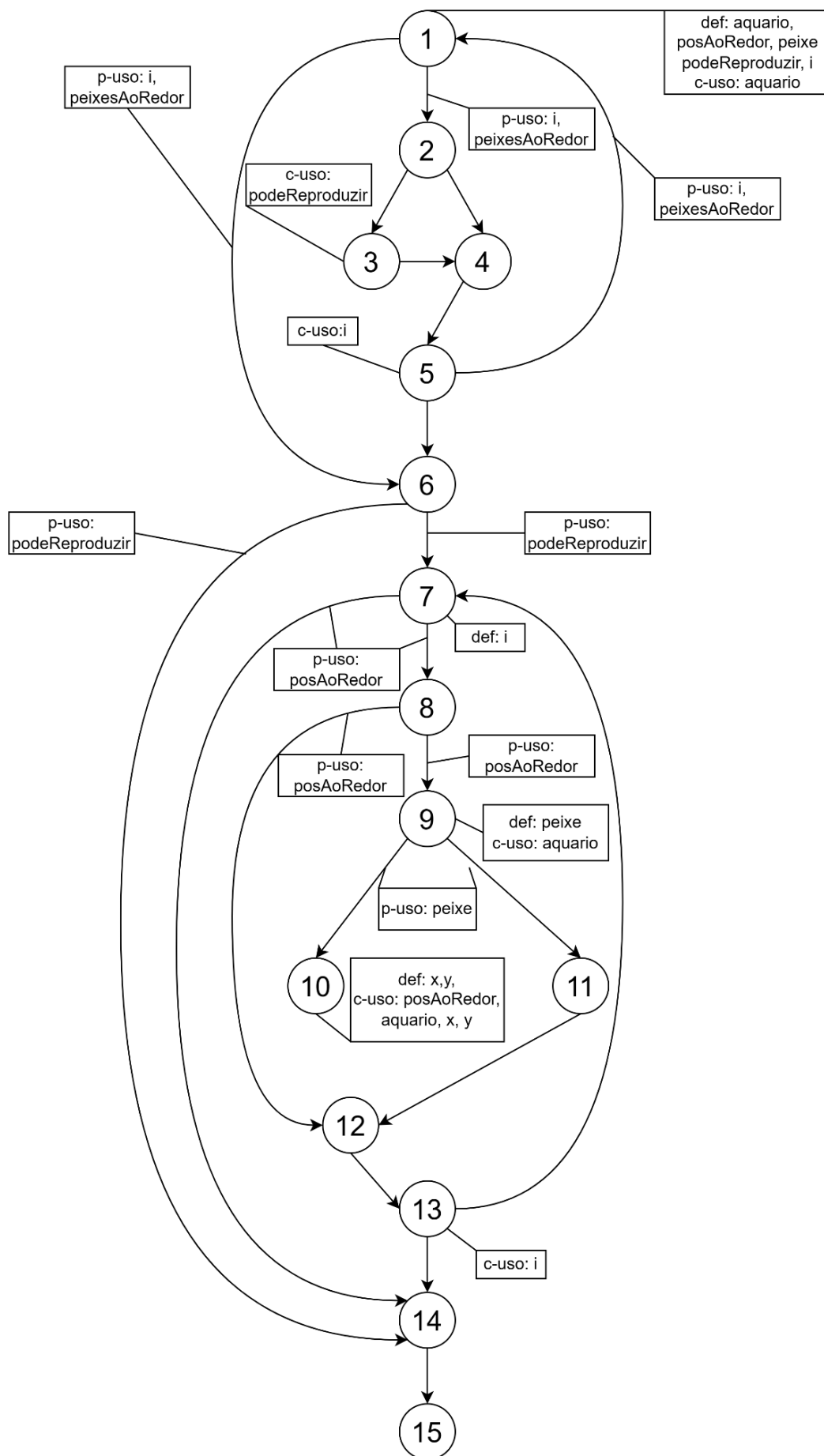


Figura: Grafo de Fluxo de Dados do método reproduzir().

Descrição textual do Grafo Def-Uso do método *reproduzir()*:

1. Início do Método /*1*/
 - a. Declaração do método *reproduzir()* com parâmetros *aquario* e *posAoRedor*;
 - b. Obtenção dos Peixes ao Redor da posição atual;
 - c. Chamada do método *pegaPeixesAoRedor* do objeto *aquario*.;
 - d. Atribuição do resultado à variável *peixesAoRedor*;
 - e. Inicialização da variável *podeReproduzir* como verdadeira;
 - f. Início do loop (for) para iterar sobre os peixes ao redor.
2. Condição para verificar se existem peixes do tipo B ao redor /*2*/
3. Se um peixe do tipo B é encontrado, *podeReproduzir* é definido como falso e o loop é interrompido /*3*/
4. Fim da condição /*4*/
5. Fim do loop (for) e incremento da variável *i* /*5*/
6. Verificação se *podeReproduzir* é verdadeiro /*6*/
7. Início do loop (for) para iterar sobre as posições ao redor /*7*/
8. Condição para verificar se a posição ao redor não é nula /*8*/
9. Obtenção do peixe na posição ao redor e condição para verificar se a posição obtida está livre /*9*/
10. Se a posição obtida é livre /*10*/
 - a. Obtenção das coordenadas da posição ao redor;
 - b. Criação de um novo peixe na posição ao redor;
 - c. Incremento da variável *mb* (resistencia do peixe);
 - d. Interrupção do loop (break);
11. Fim da condição do nó 9 /*11*/
12. Fim da condição do nó 8 /*12*/
13. Fim do Loop para Reprodução e incremento da variável *i* /*13*/
14. Fim da condição do nó 6 /*14*/
15. Fim do método /*15*/

b. Requisitos de Teste Todas as Definições, Todos os P-Usos e C-Usos

Tabela – Requisitos de teste para os critérios todas as definições, p-usos e c-usos.		
L	Nó	Instruções
56	1	public void reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor) {
59	1	Peixe[] peixesAoRedor = aquario.pegaPeixesAoRedor(this.getPosicaoX() , this.getPosicaoY());
62	1	boolean podeReproduzir = true;
63	1,5	for (int i=0; i<peixesAoRedor.length; i++){
64	2	if (peixesAoRedor[i] instanceof PeixeB) {
65	3	podeReproduzir = false;
66	3	break;
67	4	}
68	5	}
70	6	if (podeReproduzir) {
72	7,13	for (int i = 0; i < posAoRedor.length; i++) {
73	8	if (posAoRedor[i] !=null) {
74	9	Peixe peixe = aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y);
75	9	if (peixe == null) {
76	10	int x = posAoRedor[i].x;
77	10	int y = posAoRedor[i].y;
78	10	aquario.setPosicao(criaNovoPeixe(x, y), x, y);
79	10	this.mb = this.mb +1;
80	10	break;
81	11	}
82	12	}
83	13	}
84	14	}
86	15	}

Tabela – Requisitos de teste para os critérios todas as definições, p-usos e c-usos.				
L	Nó	Def	C-Uso	P-Uso
56	1	aquario, posAoRedor		
59	1	peixesAoRedor	aquario	
62	1	podeReproduzir		
63	1,5	i		i, peixesAoRedor
64	2			i, peixesAoRedor
65	3		podeReproduzir	
66	3	podeReprozir	-	-
67	4	-	-	-
68	5	-	i	-
70	6			podeReproduzir
72	7,13	i		posAoRedor
73	8			posAoRedor
74	9	peixe	aquario	
75	9			peixe
76	10	x	posAoRedor	
77	10	y	posAoRedor	
78	10		aquario, x, y	
79	10	-	-	-
80	10	-	-	-
81	11	-	-	-
82	12	-	-	-
83	13		i	
84	14	-	-	-
86	15	-	-	-

Tabela – Requisitos de teste para critério todas as definições e casos de teste.				
ID	Entradas	Saída Esp.	Variável	Requisito de Teste (Caminho)

CT32	peixe: id 24, na posição 1,1 aquario: 3x3	0 Id do novo peixe na posição 0,0	aquario, posAoRedor, peixesAoRedor, podeReproduzir, i, peixe, x, y	<1,7,9,10>
------	--	-----------------------------------	--	------------

Tabela – Requisitos de teste para critério todos os p-usos e casos de teste.				
ID	Entradas	Saída Esp.	Variável	Requisito de Teste (Caminho)
CT33	peixe1B: id 24, na posição 1,1 peixe2A: id 25, na posição 0,0 aquario: 3x3	0 Id do novo peixe na posição 0,0	i	<(1,2), (1,6), (5,1)>
CT34	aquario: 3x3 peixe1B: 0,0 peixe2B: 0,1	2 peixes no aquario	peixesAoRedor	<(1,2), (1,6), (5,1)>
CT33	peixe1B: id 24, na posição 1,1 peixe2A: id 25, na posição 0,0 aquario: 3x3	0 Id do novo peixe na posição 0,0	podeReproduzir	<(6,7), (6,14)>
			peixesAoRedor	<(7,8), (7,14), (8,12), (8,9)>
			peixe	<(9,10), (9,11)>

Tabela – Requisitos de teste para critério todos os c-usos e casos de teste.				
ID	Entradas	Saída Esp.	Variável	Requisito de Teste (Caminho)
CT32	peixe: id 24, na posição 1,1 aquario: 3x3	0 Id do novo peixe na posição 0,0	aquario	<1,9,10>
			podeReproduzir	<3>
			posAoRedor	<10>
			x	<10>
			y	<10>

CT33	peixe1: id 24, na posição 1,1 peixe2: id 25, na posição 0,0 aquario: 3x3	0 Id do novo peixe na posição 0,0	i	<5,13>
------	--	-----------------------------------	---	--------

c. Requisitos de Teste Todos os Caminhos Definições-Usos

Tabela – Requisitos de teste para critério todos os c-usos e casos de teste.			
Definição		Casos de Teste	
Assoc. Def-Use	Requisitos de Teste (Caminho)	Entradas	Saída Esperada
(1, (1, 9, 10), aquario)	<1, 2, 3, 4, 5, 6, 7, 8, 9, 10>	peixe: id 24, na posição 1,1 aquario: 3x3	0 Id do novo peixe na posição 0,0
(1, (7, 8, 10, 13), posAoRedor)	<1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 7, 8, 9, 10>	peixe1: id 24, na posição 1,1 peixe2: id 25, na posição 0,0 aquario: 3x3	0 Id do novo peixe na posição 0,0
(1, (1, 5), peixesAoRedor)	<1, 2, 3, 4, 5, 6, 7, 8, 9, 10>	peixe: id 24, na posição 1,1 aquario: 3x3	0 Id do novo peixe na posição 0,0
(1, (3), podeReproduzir)	<1, 2, 3, 4, 5, 6, 7, 8, 9, 10>	peixe: id 24, na posição 1,0 aquario: 3x3	0 Id do novo peixe na posição 0,0
(3, (6), podeReproduzir)	<1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 7, 8, 9, 10>	peixe1: id 24, na posição 1,1 peixe2: id 25, na posição 0,0 aquario: 3x3	0 Id do novo peixe na posição 0,0
(9, (9), peixe)	<1, 2, 3, 4, 5, 6, 7, 8, 9, 10>	peixe: id 24, na posição 1,0 aquario: 3x3	0 Id do novo peixe na posição 0,0
(10, (10), x)			
(10, (10), y)			
(1, (1, 2, 5), i)			
(7, (13), i)			

d. Implementação dos Casos de Teste

i. Caso de Teste CT32

Neste caso de teste, uma instância do objeto Aquario é criada com dimensões 3x3 na linha 03. Na linha 04, o método olhaAoRedor é invocado no objeto aquario, passando as coordenadas (1, 1) como argumento, resultando em um array de objetos PosicaoAdjacente. Nas linhas 05 a 07, um novo PeixeB é criado com identificação 24 e coordenadas (1, 1). O método reproduzir do PeixeB é chamado na linha 07, passando o aquário e as posições ao redor como argumentos. Finalmente, na linha 08, é realizado um teste de assert para verificar se a posição (0, 0) no aquário possui a identificação 0.

```
01 @Test
02 public void ct32() {
03     Aquario aquario = new Aquario(3,3);
04     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 1);
05     Peixe peixe1 = new PeixeB(24,1,1);
06     aquario.addPeixe(peixe1);
07     peixe1.reproduzir(aquario, posAoRedor);
08     assertEquals(0,aquario.getPosicao(0, 0).getId());
09 }
```

ii. Caso de Teste CT33

Neste caso de teste, um aquário 3x3 é criado na linha 03. Posições ao redor da coordenada (1, 1) são obtidas na linha 04. Dois peixes, um PeixeB e um PeixeA, são criados nas linhas 05 e 06, respectivamente. Ambos os peixes são adicionados ao aquário nas linhas 07 e 08. O método reproduzir do PeixeB é chamado na linha 09, passando o aquário e as posições ao redor como argumentos. O teste de assert na linha 10 verifica se a posição (0, 1) no aquário possui a identificação 0 após a reprodução.

```
01 @Test
02 public void ct33() {
03     Aquario aquario = new Aquario(3,3);
04     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 1);
05     Peixe peixe1 = new PeixeB(24,1,1);
06     Peixe peixe2 = new PeixeA(25,0,0);
07     aquario.addPeixe(peixe1);
08     aquario.addPeixe(peixe2);
09     peixe1.reproduzir(aquario, posAoRedor);
10     assertEquals(0,aquario.getPosicao(0, 1).getId());
11 }
```

iii. Caso de Teste CT34

Neste caso de teste, um aquário 3x3 é criado na linha 02. Posições ao redor da coordenada (1, 1) são obtidas na linha 03. Dois PeixeB são criados nas linhas 04 e 05, respectivamente, e ambos são adicionados ao aquário nas linhas 06 e 07. O método reproduzir do PeixeB é chamado na linha 08, passando o aquário e as posições ao redor como argumentos. O teste de assert na linha 16 verifica se existem 2 posições não nulas no aquário após a reprodução, percorrendo todas as posições no aquário com dois loops for nas linhas 10 a 15.

```
@Test
01 public void ct34() {
02     Aquario aquario = new Aquario(3,3);
03     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 1);
04     Peixe peixe1 = new PeixeB(24,0,0);
05     Peixe peixe2 = new PeixeB(25,0,1);
06     aquario.addPeixe(peixe1);
07     aquario.addPeixe(peixe2);
08     peixe1.reproduzir(aquario, posAoRedor);
09     int cont = 0;
10     for(int i=0; i<3; i++){
11         for(int j=0; j<3; j++){
12             if(aquario.getPosicao(i, j) != null)
13                 cont++;
14         }
15     }
16     assertEquals(2,cont);
17 }
```

e. Resultados da Execução dos Casos de Teste

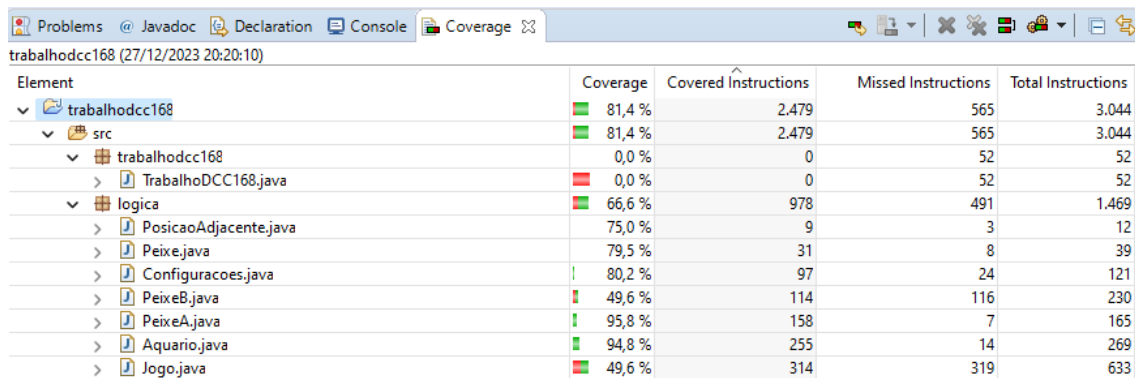
Tabela – Resultados da execução dos testes para atender aos critérios baseados em fluxo de dados.				
ID	Condições de Entrada	Saída Esperada	Saída Obtida	Associações Def-Use Exercitadas
CT32	peixe: id 24, na posição 1,1 aquario: 3x3	0 Id do novo peixe na posição 0,0	0 Id do novo peixe na posição 0,0	(1, (1, 9, 10), aquario) (1, (1, 5), peixesAoRedor) (1, (3), podeReproduzir) (9, (9), peixe) (10, (10), x) (10, (10), y) (1, (1, 2, 5), i) (7, (13), i)
CT33	peixe1: id 24, na posição 1,1 peixe2: id 25, na posição 0,0 aquario: 3x3	0 Id do novo peixe na posição 0,0	0 Id do novo peixe na posição 0,0	(1, (7, 8, 10, 13), posAoRedor) (3, (6), podeReproduzir)

CT34	aquario: 3x3 peixe1B: 0,0 peixe2B: 0,1	2 peixes no aquario	2 peixes no aquario	(1, (1), aqua (1, (1, 2, 5), peixesAoRedor) (1, (1, 2, 5), (1, (3), podeReproduzir) (3, (6), podeReproduzir)
------	--	------------------------	------------------------	--

f. Análise de Cobertura dos Casos de Teste

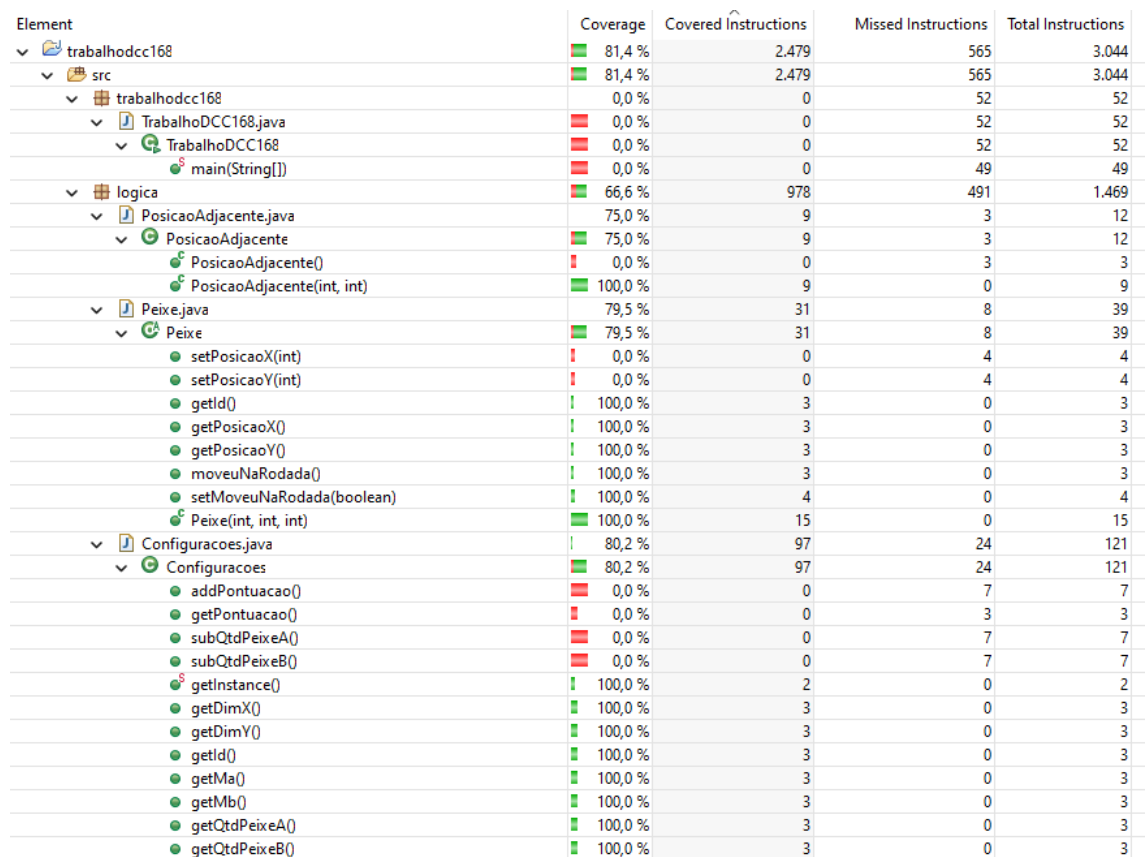
i. Análise de Cobertura de Fluxo de Controle

Abaixo serão analisados alguns dos métodos cobertos pela ferramenta *EclEmma* e como foi a cobertura dos testes no *Eclipse*, seguido com uma breve explicação do passo a passo para execução de cada método conforme o programa vai sendo compilado.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
trabalhodcc168	81,4 %	2.479	565	3.044
src	81,4 %	2.479	565	3.044
trabalhodcc168	0,0 %	0	52	52
TrabalhoDCC168.java	0,0 %	0	52	52
logica	66,6 %	978	491	1.469
PosicaoAdjacente.java	75,0 %	9	3	12
Peixe.java	79,5 %	31	8	39
Configuracoes.java	80,2 %	97	24	121
PeixeB.java	49,6 %	114	116	230
PeixeA.java	95,8 %	158	7	165
Aquario.java	94,8 %	255	14	269
Jogo.java	49,6 %	314	319	633

Figura: Coverage do EclEmma com os casos de testes CT32 ao 34.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
trabalhodcc168	81,4 %	2.479	565	3.044
src	81,4 %	2.479	565	3.044
trabalhodcc168	0,0 %	0	52	52
TrabalhoDCC168.java	0,0 %	0	52	52
TrabalhoDCC168	0,0 %	0	52	52
main(String[])	0,0 %	0	49	49
logica	66,6 %	978	491	1.469
PosicaoAdjacente.java	75,0 %	9	3	12
PosicaoAdjacente	75,0 %	9	3	12
PosicaoAdjacente()	0,0 %	0	3	3
PosicaoAdjacente(int, int)	100,0 %	9	0	9
Peixe.java	79,5 %	31	8	39
Peixe	79,5 %	31	8	39
setPosicaoX(int)	0,0 %	0	4	4
setPosicaoY(int)	0,0 %	0	4	4
getId()	100,0 %	3	0	3
getPosicaoX()	100,0 %	3	0	3
getPosicaoY()	100,0 %	3	0	3
moveuNaRodada()	100,0 %	3	0	3
setMoveuNaRodada(boolean)	100,0 %	4	0	4
Peixe(int, int, int)	100,0 %	15	0	15
Configuracoes.java	80,2 %	97	24	121
Configuracoes	80,2 %	97	24	121
addPontuacao()	0,0 %	0	7	7
getPontuacao()	0,0 %	0	3	3
subQtdPeixeA()	0,0 %	0	7	7
subQtdPeixeB()	0,0 %	0	7	7
getInstance()	100,0 %	2	0	2
getDimX()	100,0 %	3	0	3
getDimY()	100,0 %	3	0	3
getId()	100,0 %	3	0	3
getMa()	100,0 %	3	0	3
getMb()	100,0 %	3	0	3
getQtdPeixeA()	100,0 %	3	0	3
getQtdPeixeB()	100,0 %	3	0	3

Figura: Parte I do print de todos os métodos cobertos pelo EclEmma.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
● getRa()	100,0 %	3	0	3
● getRb()	100,0 %	3	0	3
● setDimX(int)	100,0 %	4	0	4
● setDimY(int)	100,0 %	4	0	4
● setMa(int)	100,0 %	4	0	4
● setMb(int)	100,0 %	4	0	4
● setQtdPeixeA(int)	100,0 %	4	0	4
● setQtdPeixeB(int)	100,0 %	4	0	4
● setRa(int)	100,0 %	4	0	4
● setRb(int)	100,0 %	4	0	4
● addId()	100,0 %	7	0	7
● addQtdPeixeA()	100,0 %	7	0	7
● addQtdPeixeB()	100,0 %	7	0	7
● inicialDPontuacao()	100,0 %	7	0	7
▼ PeixeB.java	49,6 %	114	116	230
▼ PeixeB	49,6 %	114	116	230
● getM()	0,0 %	0	3	3
● getR()	0,0 %	0	3	3
● moverNoAquario(int, int, Aquario, PosicaoAdjacente[])	0,0 %	0	101	101
● setM(int)	0,0 %	0	4	4
● setR(int)	0,0 %	0	4	4
● toString()	100,0 %	11	0	11
● PeixeB(int, int, int)	100,0 %	12	0	12
● criaNovoPeixe(int, int)	100,0 %	14	0	14
● reproduzir(Aquario, PosicaoAdjacente[])	98,7 %	77	1	78
▼ PeixeA.java	95,8 %	158	7	165
▼ PeixeA	95,8 %	158	7	165
● getM()	0,0 %	0	3	3
● setM(int)	0,0 %	0	4	4
● getR()	100,0 %	3	0	3
● setR(int)	100,0 %	4	0	4
● toString()	100,0 %	11	0	11
● PeixeA(int, int, int)	100,0 %	12	0	12
● criaNovoPeixe(int, int)	100,0 %	17	0	17
● reproduzir(Aquario, PosicaoAdjacente[])	100,0 %	45	0	45
● moverNoAquario(int, int, Aquario, PosicaoAdjacente[])	100,0 %	66	0	66
▼ Aquario.java	94,8 %	255	14	269
▼ Aquario	94,8 %	255	14	269
● verificaPosicao(int, int)	83,3 %	10	2	12
● getPosicao(int, int)	85,7 %	12	2	14
● setPosicaoNula(int, int)	100,0 %	13	0	13

Figura: Parte II do print de todos os métodos cobertos pelo EclEmma.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
● Aquario(int, int)	100,0 %	14	0	14
● setPosicao(Peixe, int, int)	73,7 %	14	5	19
● addPeixe(Peixe)	78,3 %	18	5	23
● pegaPeixesAoRedor(int, int)	100,0 %	34	0	34
● olhaAoRedor(int, int)	100,0 %	65	0	65
● imprimeAquario()	100,0 %	75	0	75
▼ Jogo.java	49,6 %	314	319	633
▼ Jogo	49,6 %	314	319	633
● configuraAmbiente()	0,0 %	0	22	22
● iniciarJogo()	0,0 %	0	3	3
● lidaComPeixeA(int, int)	0,0 %	0	54	54
● lidaComPeixeB(int, int)	0,0 %	0	54	54
● loopingJogo()	0,0 %	0	31	31
● resetMoveuPeixe()	0,0 %	0	30	30
● rodada()	0,0 %	0	70	70
● setValor(int, int[])	0,0 %	0	32	32
● verificaPosicao(int, int)	0,0 %	0	23	23
● getRandomNumberUsing(int)	100,0 %	8	0	8
● Jogo()	100,0 %	11	0	11
● posicionaPeixesAlInicio()	100,0 %	60	0	60
● posicionaPeixesBInicio()	100,0 %	60	0	60
● auxIniciarJogo(int[])	100,0 %	175	0	175

Figura: Parte III do print de todos os métodos cobertos pelo EclEmma.

Problems Javadoc Declaration Console Coverage				
trabalhodcc168 (27/12/2023 20:20:10)				
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ AquarioTest.java	98,6 %	1.501	22	1.523
▼ AquarioTest	98,6 %	1.501	22	1.523
ct13()	100,0 %	23	0	23
ct31()	52,2 %	24	22	46
ct32()	100,0 %	34	0	34
ct23()	100,0 %	36	0	36
ct25()	100,0 %	37	0	37
ct24()	100,0 %	39	0	39
ct15()	100,0 %	43	0	43
ct16()	100,0 %	43	0	43
ct18()	100,0 %	43	0	43
ct19()	100,0 %	43	0	43
ct20()	100,0 %	43	0	43
ct21()	100,0 %	43	0	43
ct22()	100,0 %	43	0	43
ct9()	100,0 %	43	0	43
ct33()	100,0 %	44	0	44
ct26()	100,0 %	46	0	46
ct27()	100,0 %	46	0	46
ct28()	100,0 %	46	0	46
ct29()	100,0 %	46	0	46
ct30()	100,0 %	46	0	46
ct10()	100,0 %	47	0	47
ct11()	100,0 %	47	0	47
ct12()	100,0 %	47	0	47
ct14()	100,0 %	47	0	47
ct17()	100,0 %	47	0	47
ct8()	100,0 %	47	0	47
ct1()	100,0 %	49	0	49
ct2()	100,0 %	49	0	49
ct3()	100,0 %	49	0	49
ct4()	100,0 %	49	0	49
ct5()	100,0 %	49	0	49
ct6()	100,0 %	49	0	49
ct7()	100,0 %	49	0	49
ct34()	100,0 %	62	0	62

Figura: Todos os casos de testes.

O método de teste '@Test public void ct32()' configura um ambiente virtual de aquário com dimensões 3x3. Ao chamar o método 'olhaAoRedor(1, 1)' no aquário, são obtidas as posições adjacentes à coordenada (1, 1) em um array de objetos PosicaoAdjacente. Em seguida, um novo peixe do tipo PeixeB é criado com identificação 24 e coordenadas (1, 1), sendo adicionado ao aquário através do método 'addPeixe(peixe1)'. O peixe recém-criado é então instruído a se reproduzir usando o método 'reproduzir(aquario, posAoRedor)', considerando as posições ao redor.

A verificação ocorre no final do teste, onde se utiliza o método 'assertEquals(0, aquario.getPosicao(0, 0).getId())' para confirmar se, após a reprodução, um novo peixe foi criado com identificação 0 na posição (0, 0) do aquário. Em resumo, o caso de teste CT32 avalia a capacidade do sistema em reproduzir e posicionar corretamente um novo peixe no aquário a partir de uma posição específica.

```

388 @Test
389 public void ct32() {
390     Aquario aquario = new Aquario(3,3);
391     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 1);
392     Peixe peixe1 = new PeixeB(24,1,1);
393     aquario.addPeixe(peixe1);
394     peixe1.reproduzir(aquario, posAoRedor);
395     assertEquals(0,aquario.getPosicao(0, 0).getId());
396 }
397

```

Figura: Coverage do EclEmma com os casos de testes CT32.

No método de teste '@Test public void ct32()', um ambiente simulado de aquário com dimensões 3x3 é configurado. Utilizando o método 'olhaAoRedor(1, 1)' no aquário, são obtidas as posições adjacentes à coordenada (1, 1) em um array de objetos PosicaoAdjacente. Posteriormente, um novo peixe da classe PeixeB é instanciado com a identificação 24 e as coordenadas (1, 1). Esse peixe é então adicionado ao aquário através do método 'addPeixe(peixe1)'.

O peixe recém-criado é direcionado a se reproduzir chamando o método 'reproduzir(aquario, posAoRedor)', levando em consideração as posições ao redor dele. A validação ocorre na última linha do teste, onde o método 'assertEquals(0, aquario.getPosicao(0, 0).getId())' é utilizado para verificar se, após a reprodução, um novo peixe foi corretamente criado na posição (0, 0) do aquário com uma identificação igual a 0.

Em resumo, o caso de teste CT32 avalia a funcionalidade de reprodução do aquário, verificando se um novo peixe é gerado e posicionado adequadamente após a reprodução a partir de uma posição específica no aquário.

```

399 public void ct33() {
400     Aquario aquario = new Aquario(3,3);
401     PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 1);
402     Peixe peixe1 = new PeixeA(24,1,1);
403     Peixe peixe2 = new PeixeB(25,0,0);
404     aquario.addPeixe(peixe1);
405     aquario.addPeixe(peixe2);
406     peixe1.reproduzir(aquario, posAoRedor);
407     assertEquals(1,aquario.getPosicao(0, 1).getId());
408 }
409

```

Figura: Coverage do EclEmma com os casos de testes CT33.

No método de teste '@Test public void ct34()', um ambiente virtual de aquário é inicializado com dimensões 3x3. Em seguida, são obtidas as posições ao redor da coordenada (1, 1) por meio do método 'olhaAoRedor(1, 1)'. Dois peixes da classe PeixeB são criados com identificações 24 e 25, respectivamente, e são adicionados ao aquário utilizando o método 'addPeixe(peixe1)' e 'addPeixe(peixe2)'.

Posteriormente, o método 'reproduzir(aquario, posAoRedor)' é chamado para o peixe1, indicando a tentativa de reprodução com base nas posições ao redor no aquário.

O teste continua com um loop duplo 'for' (linhas 8-15) que percorre todas as posições do aquário, verificando se cada posição não é nula ('aquario.getPosicao(i, j) != null'). Cada vez que uma posição não nula é encontrada, a variável 'cont' é incrementada.

Finalmente, é realizado um teste de assert ('assertEquals(2, cont)') para verificar se o número total de posições não nulas no aquário é igual a 2. Isso implica que a reprodução do peixe na posição (0, 0) gerou com sucesso duas novas posições no aquário.

Em resumo, o caso de teste CT34 avalia se a reprodução de um peixe na posição específica (0, 0) resulta na criação bem-sucedida de dois novos peixes no aquário.

```
411     public void ct34() {
412         Aquario aquario = new Aquario(3,3);
413         PosicaoAdjacente[] posAoRedor = aquario.olhaAoRedor(1, 1);
414         Peixe peixe1 = new PeixeB(24,0,0);
415         Peixe peixe2 = new PeixeB(25,0,1);
416         aquario.addPeixe(peixe1);
417         aquario.addPeixe(peixe2);
418         peixe1.reproduzir(aquario, posAoRedor);
419         int cont = 0;
420         for(int i=0; i<3; i++){
421             for(int j=0; j<3; j++){
422                 if(aquario.getPosicao(i, j) != null)
423                     cont++;
424             }
425         }
426         assertEquals(2,cont);
427     }
```

Figura: Coverage do Eclemma com os casos de testes CT34.

g. Análise de Cobertura de Fluxo de Dados

A partir dos critérios baseados em Fluxo de dados foram implementados três casos de teste para o método `reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor)` da classe `PeixeB`. A ferramenta escolhida para a análise de cobertura de fluxo de dados foi a ferramenta Baduino.

O caso de teste `ct32` atingiu a cobertura de 56,25% das associações de uso do método `reproduzir()`, as linhas não cobertas estão destacadas em vermelho.

```
56 /*1*/ public void reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor) {
57 /*1*/
58 /*1*/ //pegando os peixes ao redor
59 /*1*/ Peixe[] peixesAoRedor = aquario.pegaPeixesAoRedor(this.getPosicaoX(), this.getPosicaoY());
60 /*1*/
61 /*1*/ //verificando se tem peixes tipoB ao redor
62 /*1*/ boolean podeReproduzir = true;
63 /*1,5*/ for (int i=0; i<peixesAoRedor.length; i++){
64 /*2*/ if (peixesAoRedor[i] instanceof PeixeB) {
65 /*3*/ podeReproduzir = false;
66 /*3*/ break;
67 /*4*/ }
68 /*5*/ }
69 /*5*/
70 /*6*/ if (podeReproduzir) {
71 /*7*/ //faz a reprodução do peixe B para a primeira célula livre disponível
72 /*7,13*/ for (int i = 0; i < posAoRedor.length; i++) {
73 /*8*/ if (posAoRedor[i] != null) {
74 /*9*/ Peixe peixe = aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y);
75 /*9*/ if (peixe == null) {
76 /*10*/ int x = posAoRedor[i].x;
77 /*10*/ int y = posAoRedor[i].y;
78 /*10*/ aquario.setPosicao(criaNovoPeixe(x, y), x, y);
79 /*10*/ this.mb = this.mb +1;
80 /*10*/ break;
81 /*11*/ }
82 /*12*/ }
83 /*13*/ }
84 /*14*/ }
85 /*14*/ }
86 /*15*/ }
```

Figura: Print do método `reproduzir()`.

▼ void reproduzir(Aquario, PosicaoAdjacente[])	🕒 (27/48) (56,25%)
🍷 (59, 78, this)	✅ true
🍷 (59, 79, this)	✅ true
🍷 (59, 74, aquario)	✅ true
🍷 (59, 78, aquario)	✅ true
🍷 (59, (72, 86), posAoRedor)	❌ false
🍷 (59, (72, 73), posAoRedor)	✅ true
🍷 (59, (73, 74), posAoRedor)	✅ true
🍷 (59, (73, 72), posAoRedor)	❌ false
🍷 (59, 74, posAoRedor)	✅ true
🍷 (59, 76, posAoRedor)	✅ true
🍷 (59, 77, posAoRedor)	✅ true
🍷 (59, 79, this.mb)	✅ true
🍷 (59, (63, 70), peixesAoRedor)	✅ true
🍷 (59, (63, 64), peixesAoRedor)	✅ true
🍷 (59, (64, 65), peixesAoRedor)	❌ false
🍷 (59, (64, 63), peixesAoRedor)	✅ true
🍷 (62, (70, 72), podeReproduzir)	✅ true
🍷 (62, (70, 86), podeReproduzir)	❌ false
🍷 (63, (63, 70), i)	❌ false
🍷 (63, (63, 64), i)	✅ true
🍷 (63, (64, 65), i)	❌ false
🍷 (63, (64, 63), i)	✅ true
🍷 (63, 63, i)	✅ true
🍷 (65, (70, 72), podeReproduzir)	❌ false
🍷 (65, (70, 86), podeReproduzir)	❌ false
🍷 (63, (63, 70), i)	✅ true
🍷 (63, (63, 64), i)	✅ true
🍷 (63, (64, 65), i)	❌ false
🍷 (63, (64, 63), i)	✅ true
🍷 (63, 63, i)	✅ true
🍷 (72, (72, 86), i)	❌ false
🍷 (72, (72, 73), i)	✅ true
🍷 (72, (73, 74), i)	✅ true
🍷 (72, (73, 72), i)	❌ false
🍷 (72, 72, i)	❌ false
🍷 (72, 74, i)	✅ true
🍷 (72, 76, i)	✅ true
🍷 (72, 77, i)	✅ true
🍷 (74, (75, 76), peixe)	✅ true
🍷 (74, (75, 72), peixe)	❌ false
🍷 (72, (72, 86), i)	❌ false
🍷 (72, (72, 73), i)	❌ false
🍷 (72, (73, 74), i)	❌ false
🍷 (72, (73, 72), i)	❌ false
🍷 (72, 72, i)	❌ false
🍷 (72, 74, i)	❌ false
🍷 (72, 76, i)	❌ false
🍷 (72, 77, i)	❌ false

Figura: Associações Def-Usso cobertas pelo Caso de Teste CT32.

O caso de teste ct33 atingiu a cobertura de 70,83% das associações defuso do método `reproduzir()`, as linhas não cobertas estão destacadas em vermelho.

```

56 public void reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor) {
57
58     //pegando os peixes ao redor
59     Peixe[] peixesAoRedor = aquario.pegaPeixesAoRedor(this.getPosicaoX(), this.getPosicaoY());
60
61     //verificando se tem peixes tipoB ao redor
62     boolean podeReproduzir = true;
63     for (int i=0; i<peixesAoRedor.length; i++){
64         if (peixesAoRedor[i] instanceof PeixeB) {
65             podeReproduzir = false;
66             break;
67         }
68     }
69
70     if (podeReproduzir) {
71         //faz a reprodução do peixe B para a primeira célula livre disponível
72         for (int i = 0; i < posAoRedor.length; i++) {
73             if (posAoRedor[i] != null) {
74                 Peixe peixe = aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y);
75                 if (peixe == null) {
76                     int x = posAoRedor[i].x;
77                     int y = posAoRedor[i].y;
78                     aquario.setPosicao(criaNovoPeixe(x, y), x, y);
79                     this.mb = this.mb +1;
80                     break;
81                 }
82             }
83         }
84     }
85 }
86

```

Figura: Print do método `reproduzir()`.

void reproduzir(Aquario, PosicaoAdjacente[])	(34/48) (70,83%)
(59, 78, this)	true
(59, 79, this)	true
(59, 74, aquario)	true
(59, 78, aquario)	true
(59, (72, 86), posAoRedor)	false
(59, (72, 73), posAoRedor)	true
(59, (73, 74), posAoRedor)	true
(59, (73, 72), posAoRedor)	false
(59, 74, posAoRedor)	true
(59, 76, posAoRedor)	true
(59, 77, posAoRedor)	true
(59, 79, this.mb)	true
(59, (63, 70), peixesAoRedor)	true
(59, (63, 64), peixesAoRedor)	true
(59, (64, 65), peixesAoRedor)	false
(59, (64, 63), peixesAoRedor)	true
(62, (70, 72), podeReproduzir)	true
(62, (70, 86), podeReproduzir)	false
(63, (63, 70), i)	false
(63, (63, 64), i)	true
(63, (64, 65), i)	false
(63, (64, 63), i)	true
(63, 63, i)	true
(65, (70, 72), podeReproduzir)	false
(65, (70, 86), podeReproduzir)	false
(63, (63, 70), i)	true
(63, (63, 64), i)	true
(63, (64, 65), i)	false
(63, (64, 63), i)	true
(63, 63, i)	true
(72, (72, 86), i)	false
(72, (72, 73), i)	true
(72, (73, 74), i)	true
(72, 72, i)	true
(72, 74, i)	true
(72, 76, i)	true
(72, 77, i)	true
(74, (75, 76), peixe)	true
(74, (75, 72), peixe)	true
(72, (72, 86), i)	false
(72, (72, 73), i)	true
(72, (73, 74), i)	true
(72, (73, 72), i)	false
(72, 72, i)	false
(72, 74, i)	true
(72, 76, i)	true
(72, 77, i)	true

Figura: Associações Def-Usso cobertas pelo Caso de Teste CT33.

O caso de teste ct34 atingiu a cobertura de 77,08% das associações defuso do método *reproduzir()*, as linhas não cobertas estão destacadas em vermelho.

```

56 /*1*/ public void reproduzir(Aquario aquario, PosicaoAdjacente[] posAoRedor) {
57 /*1*/
58 /*1*/ //pegando os peixes ao redor
59 /*1*/ Peixe[] peixesAoRedor = aquario.pegaPeixesAoRedor(this.getPosicaoX(), this.getPosicaoY());
60 /*1*/
61 /*1*/ //verificando se tem peixes tipoB ao redor
62 /*1*/ boolean podeReproduzir = true;
63 /*1,5*/ for (int i=0; i<peixesAoRedor.length; i++){
64 /*2*/     if (peixesAoRedor[i] instanceof PeixeB) {
65 /*3*/         podeReproduzir = false;
66 /*3*/         break;
67 /*4*/     }
68 /*5*/ }
69 /*5*/
70 /*6*/ if (podeReproduzir) {
71 /*7*/     //faz a reprodução do peixe B para a primeira célula livre disponível
72 /*7,13*/ for (int i = 0; i < posAoRedor.length; i++) {
73 /*8*/         if (posAoRedor[i] != null) {
74 /*9*/             Peixe peixe = aquario.getPosicao(posAoRedor[i].x, posAoRedor[i].y);
75 /*9*/             if (peixe == null) {
76 /*10*/                 int x = posAoRedor[i].x;
77 /*10*/                 int y = posAoRedor[i].y;
78 /*10*/                 aquario.setPosicao(criaNovoPeixe(x, y), x, y);
79 /*10*/                 this.mb = this.mb +1;
80 /*10*/                 break;
81 /*11*/             }
82 /*12*/         }
83 /*13*/     }
84 /*14*/ }
85 /*14*/ }
86 /*15*/ }

```

Figura: Print do método *reproduzir()*.

▼ ● void reproduzir(Aquario, PosicaoAdjacente[])	🕒 (37/48) (77,08%)
🔗 (59, 78, this)	✅ true
🔗 (59, 79, this)	✅ true
🔗 (59, 74, aquario)	✅ true
🔗 (59, 78, aquario)	✅ true
🔗 (59, (72, 86), posAoRedor)	❌ false
🔗 (59, (72, 73), posAoRedor)	✅ true
🔗 (59, (73, 74), posAoRedor)	✅ true
🔗 (59, (73, 72), posAoRedor)	❌ false
🔗 (59, 74, posAoRedor)	✅ true
🔗 (59, 76, posAoRedor)	✅ true
🔗 (59, 77, posAoRedor)	✅ true
🔗 (59, 79, this.mb)	✅ true
🔗 (59, (63, 70), peixesAoRedor)	✅ true
🔗 (59, (63, 64), peixesAoRedor)	✅ true
🔗 (59, (64, 65), peixesAoRedor)	✅ true
🔗 (59, (64, 63), peixesAoRedor)	✅ true
🔗 (62, (70, 72), podeReproduzir)	✅ true
🔗 (62, (70, 86), podeReproduzir)	❌ false
🔗 (63, (63, 70), i)	❌ false
🔗 (63, (63, 64), i)	✅ true
🔗 (63, (64, 65), i)	✅ true
🔗 (63, (64, 63), i)	✅ true
🔗 (63, 63, i)	✅ true
🔗 (65, (70, 72), podeReproduzir)	❌ false
🔗 (65, (70, 86), podeReproduzir)	✅ true
🔗 (63, (63, 70), i)	✅ true
🔗 (63, (63, 64), i)	✅ true
🔗 (63, (64, 65), i)	❌ false
🔗 (63, (64, 63), i)	✅ true
🔗 (63, 63, i)	✅ true
🔗 (72, (72, 86), i)	❌ false
🔗 (72, (72, 73), i)	✅ true
🔗 (72, (73, 74), i)	✅ true
🔗 (72, (73, 72), i)	❌ false
🔗 (72, 72, i)	✅ true
🔗 (72, 74, i)	✅ true
🔗 (72, 76, i)	✅ true
🔗 (72, 77, i)	✅ true
🔗 (74, (75, 76), peixe)	✅ true
🔗 (74, (75, 72), peixe)	✅ true
🔗 (72, (72, 86), i)	❌ false
🔗 (72, (72, 73), i)	✅ true
🔗 (72, (73, 74), i)	✅ true
🔗 (72, (73, 72), i)	❌ false
🔗 (72, 72, i)	❌ false
🔗 (72, 74, i)	✅ true
🔗 (72, 76, i)	✅ true
🔗 (72, 77, i)	✅ true

Figura: Associações Def-Usso cobertas pelo Caso de Teste CT34.

h. Resultados

Nos casos de teste implementados não foram identificados defeitos.

De acordo com a ferramenta Baduino, os casos de testes ct32, ct33 e ct34 atingiram: 77,08% das associações def-uso do método reproduzir(), 35,24% das associações def-uso da classe PeixeB e 50,27% das associações def-uso do projeto completo.

▼ trabalho168	🌑 (375/746) (50,27%)
▼ logica	🌑 (298/633) (47,08%)
> 🟢 logica.Aquario	🌑 (136/157) (86,62%)
> 🟢 logica.Jogo	🌑 (89/303) (29,37%)
> 🟢 logica.PeixeA	🌑 (36/68) (52,94%)
> 🟢 logica.PeixeB	🌑 (37/105) (35,24%)
> testes	🌑 (77/101) (76,24%)
> trabalho168	🌑 (0/12) (0,00%)

Figura: Porcentagem das Associações Def-Usso Cobertas na Ferramenta Baduino.

Abaixo temos a tabela que mostra as comparações com os testes executados pela ferramenta EclEmma com os resultados da ferramenta Baduino.

Tabela – Comparação de cobertura entre as ferramentas EclEmma e Baduino.		
Classe	EclEmma	Baduino
Jogo	49,6%	29,37%
PeixeA	95,8%	52,94%
PeixeB	49,6%	35,24%
Aquario	94,8%	77,71%
Configuracoes	80,2%	-
Peixe	79,5%	-
PosicaoAdjacente	75%	-

6. Conclusões

A automação de testes, especialmente com ferramentas como EclEmma e Baduino, oferece benefícios significativos na identificação de áreas não testadas e na avaliação da cobertura do código. No entanto, é importante considerar que a automação pode ter limitações em lidar com casos complexos e específicos.

Ao aplicar técnicas de teste funcional e estrutural, enfrentamos desafios na criação de casos abrangentes que cobrissem todas as situações possíveis. A análise de cobertura revelou a importância de testar casos extremos para garantir robustez no código.

A implementação dos casos de teste e a análise de cobertura evidenciam a eficácia das técnicas aplicadas. A combinação de testes funcionais e estruturais proporcionou uma cobertura abrangente, identificando áreas críticas e garantindo a qualidade do código. O uso de ferramentas de automação complementou esse processo, facilitando a identificação de falhas e áreas não testadas. A experiência adquirida reforça a importância contínua dos testes no desenvolvimento de software.

Referências

GAMMA, Erich; BECK, Kent. **JUnit: Framework for Java Unit Testing**. JUnit 5. Eclipse Public License 2.0, 2023. Disponível em: <https://junit.org/junit5/>.

EclEmma: [Nome do desenvolvedor ou equipe de desenvolvimento]. **EclEmma: Java Code Coverage for Eclipse**. Versão 3.1.7, 2023. Disponível em: <https://www.eclEmma.org/>.

Baduino: [Nome do desenvolvedor ou equipe de desenvolvimento]. **Baduino: Bitwise-Algorithm Definition-Use associatiON visualizatiON**. Versão 0.3.13, 2023. Disponível em: <https://saeg.github.io/baduino/>.

Associação Brasileira de Normas Técnicas. **NBR ISO/IEC 25000: Sistemas e software de engenharia de software - Qualidade de produtos de software - Requisitos e avaliação (SQuaRE)**. Rio de Janeiro, 2014.

Associação Brasileira de Normas Técnicas. **NBR ISO/IEC/IEEE 29119: Engenharia de software e sistemas - Teste de software**. Rio de Janeiro, 2014.