Teste e Inspeção de Software

Alyson Matheus Maruyama Nascimento - 8532269

Cristiano Di Maio Chiaramelli - 9293053

Felipe Tiago De Carli - 10525686

Rafael Gongora Bariccatti - 10892273

Projeto 1 - Teste Funcional

Universidade de São Paulo - São Carlos

Introdução

Este documento tem como objetivo relatar o desenvolvimento do do primeiro projeto da disciplina SSC0721 - Teste e Inspeção de Software, ministrada por Simone Senger de Souza.

O sistema a ser implementado em linguagem Java e testado utilizando técnicas de teste funcional é o Jogo da Vida, cujas especificações podem ser encontradas no seguinte enunciado:

"O jogo da vida corresponde a um tabuleiro plano 6x6, em que cada posição possui um valor: 1 – corresponde a uma célula viva e 0 – corresponde a uma célula morta.

O jogo começa com uma configuração inicial, gerada aleatoriamente. A partir dessa configuração, a cada passo uma nova geração é obtida, de acordo com as seguintes regras:

• Qualquer célula viva com menos de dois vizinhos vivos morre de solidão.

• Qualquer célula viva com mais de três vizinhos vivos morre de superpopulação.

• Qualquer célula morta com exatamente três vizinhos vivos se torna uma célula viva.

• Qualquer célula com dois vizinhos vivos continua no mesmo estado para a próxima geração.

O jogo não tem fim, assim, o usuário pode, a cada passo escolher uma nova geração ou finalizar o jogo. A cada passo é mostrado ao usuário a geração anterior e a geração atual".

Vale notar que, conforme combinado com o monitor da disciplina, para casos de célula não cobertos nos itens acima o jogo deve manter o mesmo valor da célula para o próximo estado do tabuleiro. Como exemplo de caso não coberto pelas regras do jogo podemos citar uma célula morta com mais de 3 vizinhos.

Aplicação do Teste Funcional

Nesta seção vamos aplicar duas técnicas de testes funcionais: Particionamento em Classes Equivalência e Análise dos Valores Limite.

Particionamento em Classes de Equivalência

Podemos dividir o domínio da entrada do nosso jogo nas seguintes classes de equivalência:

Nome da Classe de Equivalência	**Valor da Célula**	**Quantidade de vizinhos (qtd)**	**Saída esperada**
V1	1	$0 \leq$ qtd < 2	0
V2	1	qtd == 2	1
V3	1	qtd == 3	1
V4	1	qtd > 3	0
M1	0	$0 \leq$ qtd ≤ 2	0
M2	0	qtd == 3	1
M3	0	qtd > 3	0

Para cada tipo de célula (vivo, morta ou ambas), cada valor dentro do intervalo acima tem o mesmo grau de importância para o teste, ou seja, precisamos apenas implementar alguns casos de testes que cubram ao menos um valor dentro de cada intervalo.

Observação: a coluna “Nome da Classe de Equivalência” foi inserida para simplificarmos a identificação das respectivas classes nas seções seguintes. Sendo assim, ao longo das próximas seções utilizaremos este campo para indicar determinada classe de equivalência.

Análise dos Valores Limite

Como a análise dos valores limites atua como complemento do particionamento em classes de equivalência, dada a tabela de classes anterior, podemos definir os seguintes valores limites a serem testados:

Quantidade de Vizinhos:

Para células vivas:

- { 0 }
- { 1, 2 }
- { 2, 3 }
- { 3, 4 }

Para células mortas:

- { 0 }
- { 2, 3 }
- { 3, 4 }

Conjunto de Casos de Teste

Dados os valores limite e as classes de equivalência anteriores, podemos derivar o conjunto de casos de teste ***TestSet-Func*** que pode ser representado na tabela abaixo:

Caso de Teste	Entrada (estado do tabuleiro)	Saída esperada
t0	{ 0, 0, 0, 0, 0, 0 } { 0, **1**, **1**, 0, 0, 0 } { 0, **0**, 0, **0**, 1, 1 } { 1, **0**, 0, 0, 1, **0** } { **1**, **1**, 0, 0, 0, 0 } { 1, 1, 0, 0, 0, **1** }	{ 0, 0, 0, 0, 0, 0 } { 0, **0**, **0**, 0, 0, 0 } { 0, **1**, 0, **1**, 1, 1 } { 1, **1**, 0, 0, 1, **1** } { **0**, **0**, 0, 0, 0, 0 } { 1, 1, 0, 0, 0, **0** }
t1	{ 0, 0, 0, 1, 0, 1 } { 0, 0, 0, 0, **0**, 0 } { 0, 0, 0, 1, 0, 1 } { 1, 0, 0, 0, 0, 0 }	{ 0, 0, 0, 0, 0, 0 } { 0, 0, 0, 0, 0, 0 } { 0, 0, 0, 0, 0, 0 } { 1, 1, 0, 0, 0, 0 }

	{ **1**, 1, 0, 0, 0, 0 } { 1, 1, 0, 0, 0, 0 }	{ 0, 0, 0, 0, 0, 0 } { 1, 1, 0, 0, 0, 0 }

Vale ressaltar que para cada teste fornecemos ao programa uma entrada, que consiste em um estado fixo do tabuleiro, e a saída esperada corresponde ao estado do tabuleiro após rodar uma iteração/rodada do jogo da vida.

Como o tabuleiro possui dimensão 6x6, os casos de testes gerados acima são capazes de testar múltiplas classes de equivalência simultâneamente. A tabela abaixo demonstra quais classes cada caso de teste é capaz de testar:

Caso de Teste	**Classes Testadas**
t0	V1, V2, V3, M1, M2
t1	V4, M3

Programa a ser testado

A implementação do programa “Jogo da Vida” foi realizada em linguagem Java utilizando-se como ferramenta o *Eclipse IDE* para a criação e inicialização do projeto.

Os arquivos contendo o código e projeto completos podem ser encontrados no mesmo diretório deste documento.

Automatização do Teste Funcional

Os casos de testes gerados pelo teste funcional (*TestSet-Func*) descritos nas seções anteriores foram implementados utilizando a ferramenta *JUnit.* Os mesmos também podem ser encontrados no mesmo diretório deste documento dentro do projeto gerado pelo Eclipse.

Durante essa fase pudemos verificar que o conjunto de testes foi capaz de alcançar uma cobertura de 83,7%. Nenhum erro no código foi encontrado durante o desenvolvimento dos testes.

Aplicação do Teste Estrutural

Utilizamos a ferramenta de teste estrutural *EclEmma* para executar os casos de teste gerados anteriormente (casos de teste adequados ao teste funcional). Para esses testes, a ferramenta obteve uma cobertura de 98,6%. Vale notar que o valor de 100% de cobertura não pôde ser alcançado pois não é possível testar a linha do código que define a classe *Main*. O relatório de cobertura gerado foi entregue junto à atividade. Nenhum defeito foi encontrado durante o desenvolvimento desses testes.

Conclusão

O uso destas técnicas para guiar o desenvolvimento dos testes certamente ajudou a reduzir o número de casos escritos para alcançar uma alta cobertura. As ferramentas JUnit e Eclemma também ajudam a aumentar a produtividade.

Além disso, as técnicas aplicadas de teste funcional e teste estrutural nos auxiliaram a definir (antes da implementação dos testes) quais testes devem ser implementados a fim de minimizar a quantidade de testes e ao mesmo tempo maximizar a quantidade de fluxos e linhas de código cobertos.