Especificação e Verificação de Programas

Trabalho 1 - Testes de Software

Este trabalho é dividido em três partes:

- [3,5 pontos] Teste manual de caixa preta
- [3,5 pontos] Teste manual de caixa branca
- [3 pontos] Testes baseados em propriedades (funções sem estado).

A linguagem de programação das atribuições é Java. Ao derivar casos de teste, você deve usar JUnit para organizar e executar seus casos de teste.

O trabalho pode ser realizado individualmente, ou em equipes com DOIS alunos. Indique os componentes de sua equipe ao professor até o dia

1. Teste de caixa preta

Sua tarefa é derivar casos de teste para uma série de métodos usando suas especificações. Analise as especificações para dividir o conjunto de possíveis entradas em diferentes situações. Em seguida, escreva um conjunto de casos de teste para cada método, que cobre bem os diferentes casos.

A tarefa é sobre uma classe chamada WorkSchedule. O objetivo da classe é gerenciar o cronograma para os funcionários de uma empresa. O tempo é dividido em unidades de 1 hora e as horas são simplesmente identificadas por números inteiros (Observe que esta é uma simplificação irrealista). Para cada hora, o horário armazena o número de funcionários que é necessário naquele momento. Por exemplo, pode ser que durante as semanas a necessidade é de que 5 funcionários trabalhem ao mesmo tempo, mas menos durante as noites e o fim de semana. Para cada hora, o cronograma também armazena os nomes dos funcionários que foram designados para trabalhar naquela hora.

O construtor e os métodos da classe são:

```
public WorkSchedule(int size) { ... }
```

cria um cronograma que contém as horas 0,1,2, ..., tamanho - 1, onde, para cada hora, o número de funcionários necessários é definido como zero e não há empregados atribuídos a ele

```
public Hour readSchedule(int time) { ... }
```

Retorna um objeto da classe Hour, que tem dois campos:

requiredNumber do tipo int é o número necessário de funcionários que trabalham na hora da hora.
workingEmployees do tipo String [] é o nome dos funcionários que até agora foram designados para trabalhar na hora da hora.

`public void setRequiredNumber(int nemployee, int starttime, int endtime) { ... }` define o número de empregados trabalhando necessários para o funcionário durante todas as horas na hora de início do intervalo até o fim do tempo.
`public boolean addWorkingPeriod(String employee, int starttime, int endtime) { ... }` agenda o funcionário para trabalhar durante as horas desde o início até o fim do tempo.
`public String[] workingEmployees(int starttime, int endtime) { ... }` retorna uma lista de todos os funcionários que trabalham em algum ponto durante o intervalo de início ao fim do tempo.
`public int nextIncomplete(int currenttime) { ... }` retorna o tempo mais próximo a partir do horário atual para o qual a quantidade necessária de funcionários ainda não foi agendada.

Esperasse que o construtor (WorkSchedule) e o método readSchedule estejam corretos e, por esta razão, não precisam ser testados. Observe que readSchedule pode ser usado para examinar o estado de uma instância de classe.

Uma implementação compilada da classe pode ser encontrada no seguinte arquivo https://goo.gl/tk1oM7

Você deve executar seus casos de teste nos métodos no arquivo .jar . Observe que alguns métodos podem ser implementados corretamente, enquanto outros podem conter bugs.

Sua tarefa: para cada especificação:

1. Especifique (o domínio e) o espaço de entrada para cada argumento para o método (incluindo o objeto desse método).
2. Divida este espaço de entrada em partições, com base na especificação.
3. Implementar pelo menos um caso de teste para cada partição, e possivelmente casos de borda adicionais.

Para cada caso de teste, dê um nome de comentário ou método que descreva a situação que ele testa. Em um arquivo de relatório, você **deve** explicar e motivar por que você dividiu os casos de teste da maneira que você fez. Você **deve** mostrar como você definiu o espaço de entrada e como você partiu.

Exemplo: Dado o método:

public String m (int x);

Com a especificação:

```
requer:  x != 0
garante:   if x > 0 then
```

```
            o método retorna o String "High"
          otherwise
           o método retorna o String "Low"
```

Seu relatório deve ser algo como:

```
Input space: x != 0 (or: "x" in { MIN_INT, ..., -1, 1, ..., MAX_INT } )

 Partition #1: x > 0 (or: "x" in { 1, ..., MAX_INT } )
  test_m_part1: input: "x = 155", expected: "m(x) = "High""
 Partition #2: x < 0 (or: "x" in { MIN_INT, ..., -1 } )
  test_m_part2: input: "x = -18", expected: "m(x) = "Low""

 Border cases:
  test_m_border1: input: "x = MIN_INT", expected: "m(x) = "Low""
  test_m_border2: input: "x = -1",      expected: "m(x) = "Low""
  test_m_border3: input: "x = 1",       expected: "m(x) = "High""
  test_m_border4: input: "x = MAX_INT", expected: "m(x) = "High""

Found bugs: None.
```

Abaixo a especificação dos métodos

a) addWorkingPeriod
Requer:
O funcionário é uma cadeia não-nula
Garante:
se o tempo de início <0 ou o tempo de término> = tamanho ou tempo de início> tempo de término, *então*
retorna falso e o cronograma permanece inalterado
de outra forma
se, durante qualquer hora no intervalo, o tempo de início até o final do tempo, o comprimento de workingEmployees é igual a requiredNumber, *então*
retorna falso e o cronograma permanece inalterado
de outra forma
se, durante qualquer hora do intervalo, o tempo de início até o fim do tempo, existe uma string em workingEmployees que é igual a empregado, *então*
retorna falso e o cronograma permanece inalterado
de outra forma
retorna verdadeiro
para eu entre o horário de início e o tempo de término, workingEmployees contém uma string igual ao empregado e o resto do cronograma permanece inalterado

b) workingEmployees
Requer:
Hora de início> = 0 e tempo de término <tamanho
Garante:
se starttime <= time time *then*
retorna uma matriz com cordas distintas - uma cadeia aparece na matriz de retorno se e somente se aparece no workingEmployees de pelo menos uma hora no intervalo starttime até o fim do tempo
de outra forma
retorna uma matriz vazia

e, em ambos os casos, o cronograma é inalterado

---

2. Teste de caixa branca

Para as atribuições da parte 2, você usará o código-fonte dos métodos. Para cada método, você deve primeiro derivar um conjunto de casos de teste que juntos possuem cobertura de declaração completa do código-fonte. Em seguida, você deve escrever casos de teste adicionais (se for necessário) que junto com os que já possuem cobertura de ramo completa do código. Os casos de teste devem ser executáveis com JUnit, como na parte 1 do laboratório. Se algum dos casos de teste de um método falhar, tente encontrar o bug e corrigir o código-fonte.

Tenha em mente que você pode precisar de um método, A, para configurar o teste de outro método, B. Se o método A tiver um erro, pode ser muito complicado testar o método B. Sempre que possível, você pode primeiro corrigir o erro em um método e depois faça os testes que o utilizam na configuração (ou oracle de teste). Também tenha em mente que um erro na implementação pode implicar que a declaração completa ou cobertura de agência não é possível. Isto é, por si só, uma indicação da existência do bug. Nesse caso, é suficiente se você tiver uma cobertura completa depois de corrigir o erro.
A atribuição é sobre a classe Set .

Esta questão é sobre a classe Set (https://goo.gl/HbKbyn).

O conjunto representa conjuntos de números inteiros. Os elementos de um conjunto são armazenados em um ArrayList . Eles são classificados e sem duplicatas para acelerar algumas operações. Dois métodos podem precisar de uma explicação:

seção **void pública** (Set s) { . . . } remove deste conjunto qualquer elemento que seja igual a um elemento em s .
booleano público contémArithTriple ( ) { . . . } retorna verdadeiro se h h três elementos, x, y e z, neste conjunto, de forma que y - x = z-y.

Observe que você pode usar para examinar para examinar facilmente o estado de uma instância de classe.
a) Use a declaração e a cobertura do ramo para derivar os casos de teste para a inserção .
b) Use a declaração e a cobertura de derivação para derivar os casos de teste para o membro .
c) Use declaração e cobertura de derivação para derivar casos de teste para seção .
d) Use a declaração e a cobertura do ramo para derivar os casos de teste para contémArithTriple .

e) Se você encontrar algum erro, tente corrigi-los na implementação. Também tenha em mente que pode ser sua especificação incorreta. Faça uma nova versão do Set.java que passa todos os testes.

---

3. Testes baseados na propriedade de funções sem estado

Para cada uma das seguintes classes, implemente funções que geram entrada para casos de teste e escreva as propriedades que devem ser testadas. Informe e corrija todos os erros que você encontrar.

1. **Pesquisa binária**
   Quando um array é ordenado, encontrar um elemento nessa matriz torna-se possível no tempo logarítmico ao realizar uma pesquisa binária . Esta classe implementa uma pesquisa binária para array de números inteiros (https://en.wikipedia.org/wiki/Binary_search_algorithm).
   - Realize o download do arquivo Java em https://goo.gl/AVLVus
   - Implementar uma função que gere arrays aleatórios.
   - Defina propriedades que testem a função e corrijam todos os erros que você encontrar.
2. **Merge sort**
   Esta classe implementa o merge sort (https://en.wikipedia.org/wiki/Merge_sort).
   - Realize o download do arquivo Java em https://goo.gl/Z81qq9
   - Implemente uma função que gere arrays aleatórios a serem ordenadas.
   - Defina propriedades que devem ser satisfeitas em arrays ordenados e corrigir todos os erros que você encontrar.

Relatórios
Envie um arquivo TRAB1.zip (ou rar, tar.gz, tar como desejar) usando o SIGAA. O arquivo deve ter a seguinte estrutura:

```
TRAB1.zip
 |
 |  -ex1
 |  | - <sua_classe_teste_1> .java
 |  | - ...
 |  | - <sua_classe_teste_n> .java
 |  | - WorkScheduleTestSuite.java
 |  \ - report.txt
 |
 |  -ex2
 |  | - <sua_classe_teste_1> .java
 |  | - ...
```

| | - <sua_classe_teste_m> .java
| | - SetTestSuite.java
| | - Set.java
| \ - report.txt
|
\ -ex3
  | - BinarySearchTest.java
  | - MergeSortTest.java
  \ - report.txt

- ex1, ex2 e ex3 são diretórios
- Nos diretórios ex1 e ex2, <your_1st to nth_test_class> .java são todas as classes que implementam casos de teste para o exercício (substituindo o conteúdo entre colchetes <> com seus próprios nomes de classe)
- Quando indicado no exercício, a classe com sufixo TestSuite.java representará o conjunto de teste para o exercício . Utilize o texto entre colchetes <> com os respectivos nomes das classes de test, contidas no mesmo diretório:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith (Suite.class)
@ Suite.SuiteClasses ({
        <your_1st_test_class> .class,
        ...
        <your_nth_test_class> .class
})
```

- Executando com JUnit, a suite de teste deve fazer o JUnit executar os casos de teste contidos nas classes especificadas nos argumentos da anotação @ Suite.SuiteClasses
- report.txt é o arquivo de relatório (onde você precisa motivar como você criou o teste e denunciou os erros encontrados / corrigidos).

Observação: não precisa ser mais complexo do que isso; evite declarações de pacotes, evite adicionar arquivos desnecessários, como arquivos ocultos ou pastas produzidos por seu editor de texto, por exemplo.