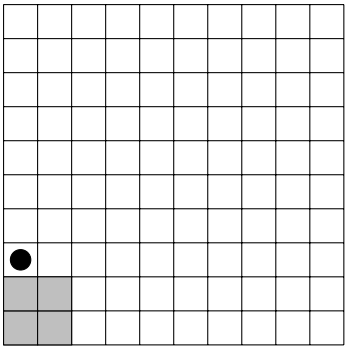
Primeiro Exercício-Programa

Norton Trevisan Roman

18 de setembro de 2018

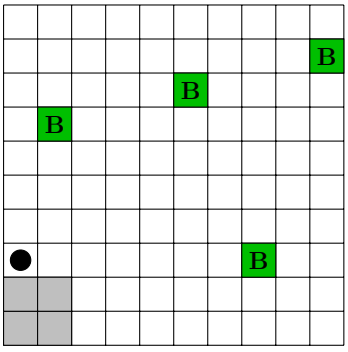
1 Robô em uma Sala

Você agora é o responsável por programar o comportamento de um robô em uma sala. Imagine seu robô (a bolinha preta) em uma sala de 10 × 10 passos, como esta:



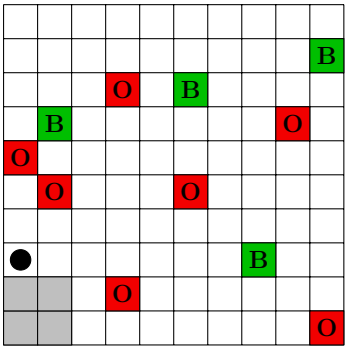
Seu robô é livre para se movimentar tanto verticalmente quanto horizontalmente, mas não nas diagonais. Assim, a cada momento ele pode escolher entre, no máximo, 4 posições para caminhar.

Nesta sala, a área cinza é chamada de Área de Armazenagem, e será usada para armazenar 4 blocos que estarão espalhados pela sala (e cuja posição o robô **desconhece**). Os blocos (representados por um 'B') podem estar em qualquer lugar, sendo sua posição definida pelo usuário do sistema. Dessa forma, uma possível configuração seria:



Seu robô é livre para se movimentar pelos quadrados dos blocos (mas não pela área de armazenagem), ou seja, os blocos não impedem o trânsito dele. Na sala (fora da área de

armazenagem) pode haver alguns obstáculos (representados por um 'O'), cuja quantidade e posição também são definidas pelo usuário do sistema. Assim, podemos ter, por exemplo:



Desta vez o robô não pode passar por quadrados com obstáculos. Sendo assim, quadrados com obstáculos não podem ser considerados quando da escolha dos movimentos do robô.

2 Tarefa

Sua tarefa é escrever um sistema que permita ao usuário criar o robô, definindo também as posições dos 4 blocos, além do número e posições dos obstáculos (a quantidade de obstáculos é deixada a cargo do usuário). Seu programa deve fazer o robô andar pela sala em busca dos blocos, evitando os obstáculos. O tamanho da sala é sempre 10×10. Cada ação executada pelo robô deve ser registrada em um log, que será entregue implementado a você.

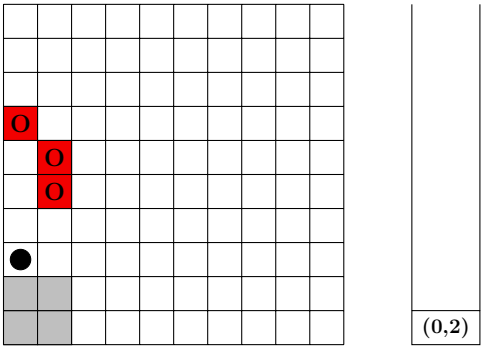
Para auxiliá-lo, foram definidas interfaces Java, contendo os métodos que seu robô e sua sala (que você deverá implementar também) devem fornecer. Do ponto de vista do usuário do sistema, o robô será criado via uma chamada ao **construtor padrão** deste (`Robo()`). Nessa chamada, a sala e o mensageiro devem ser criados automaticamente, sem a necessidade de seu conhecimento pelo usuário. A sala, quando criada, tem suas posições inicialmente vazias, ou seja, contendo o valor `POSICAO_VAZIA`, definido na interface `ISala`.

Criado o robô, o próximo passo a ser dado pelo usuário é incluir os blocos no tabuleiro. Ele fará isso via chamadas ao método `adicionaBloco(x,y)` do robô criado, passando as coordenadas cartesianas do bloco em `x` e `y`. Cabe ao usuário do sistema escolher quantos blocos criar. Seu sistema, no entanto, trabalha com o pressuposto de que estes são 4. Mais adiante será detalhado como o sistema deve agir caso haja menos que 4 blocos.

Incluídos os blocos na sala, cabe ao usuário adicionar obstáculos, **se assim o quiser** (ou seja, obstáculos são totalmente opcionais). Para tal, ele fará chamadas ao método `adicionaObstaculo(x,y)` do robô, passando as coordenadas cartesianas do obstáculo em `x` e `y`. Terminado o povoamento da sala, o usuário pode então iniciar o passeio do robô, chamando o método `buscaBlocos()` deste.

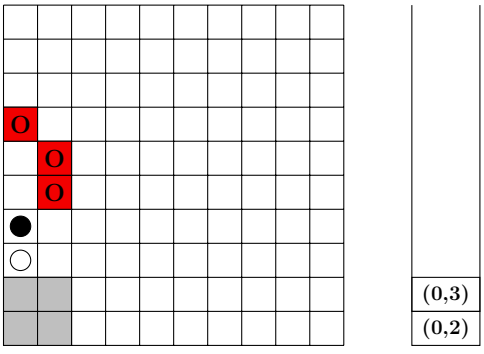
Durante cada movimento desse passeio, o robô deve escolher para onde ir. Para essa escolha, o conceito que iremos aplicar se chama busca em profundidade. Sua ideia básica é razoavelmente simples: o robô anda preferivelmente em uma mesma direção, marcando seus passos. Se não for mais possível andar nesta direção, ele escolhe a próxima de sua preferência.

Caso não seja mais possível andar em nenhuma nova direção, ele volta um passo, escolhendo novamente em que direção ir (dentre as ainda não visitadas). Caso mais uma vez não haja movimento possível, ele deve dar mais um passo atrás, e assim por diante. Como exemplo, considere a seguinte situação:

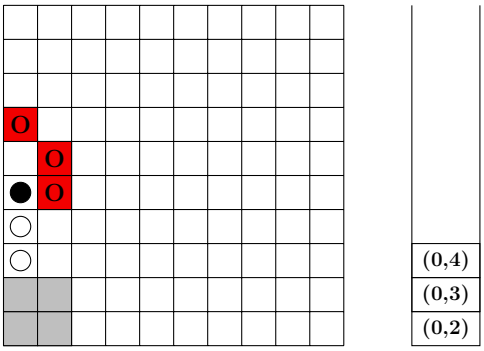


À esquerda temos o mapa da sala, enquanto que à direita temos uma estrutura que marca o caminho percorrido pelo robô ((x,y) cartesianos). Note que as coordenadas correspondem a um `x` na horizontal, variando de 0 a 9, e um `y` na vertical, também variando de 0 a 9, estando a origem (ponto (0,0)) no canto inferior esquerdo da sala. Como você escolherá representar isso é irrelevante para o problema, **desde que para o usuário ele se apresente como se fosse esse sistema de coordenadas**.

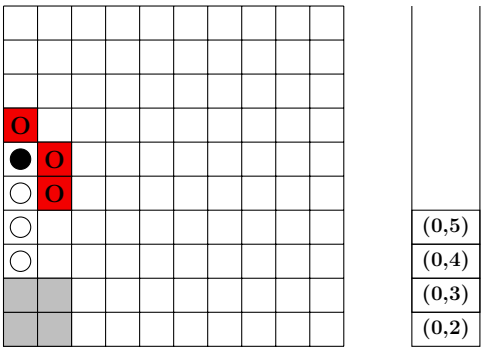
Agora suponha que nosso robô prefere a seguinte ordem de escolha de direções (ordem esta que também você deverá seguir): para cima, à direita, para baixo, à esquerda. Ou seja, ele sempre tenta ir para cima. Caso não possa, tenta à direita, e assim por diante. No caso acima, ele segue sua preferência e vai à casa imediatamente acima, armazenando sua nova posição na estrutura, acima do dado que lá estava:



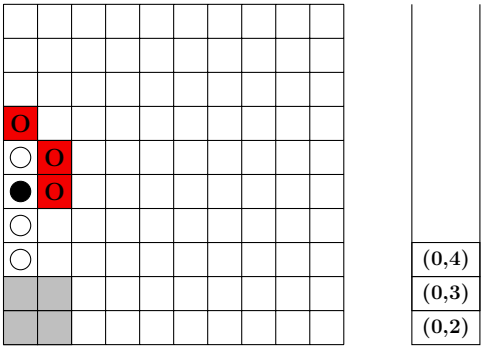
A implementação dessa estrutura fica a cargo de vocês. Note que, além de armazenar a posição, o robô também marca os quadros por onde passa (bola branca). Isso pode ser feito marcando a estrutura de dados que representa sala com o valor `MARCA_PRESENTE`, definido na interface `ISala`. Seguindo sua ordem de preferência, o robô novamente se move para cima, já que não há impedimentos, armazenando essa nova posição na estrutura:



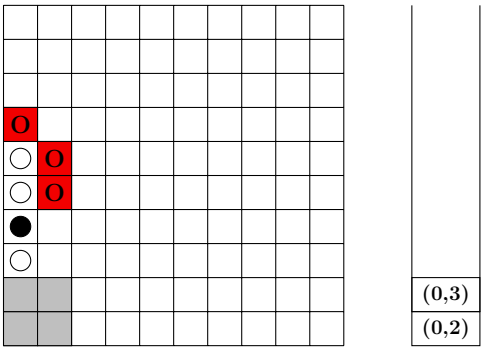
E mais uma vez acima...



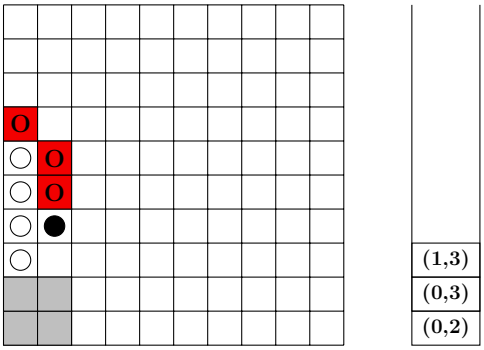
Aqui ele tenta ir para cima, para a direita, para baixo e para a esquerda, mas não há casa livre para onde ir (note que a de baixo não está livre por ele já ter passado por ela). Então ele volta uma casa e continua a busca de sua posição anterior. Para saber que posição era esta, ele deve remover da estrutura sua posição antiga – (0,5) – e olhar a última coordenada que lá ficou armazenada – (0,4):



Novamente ele tenta todas as direções e, como não obtém sucesso, executa o backtracking mais uma vez, removendo (0,4) – sua posição atual – da estrutura, e indo à posição anterior, ou seja, (0,3):



Tendo já tentado a casa de cima, ele passa à próxima de sua preferência: à direita. Uma vez que esta não foi tentada ainda, ele então se move para lá, armazenando a nova posição:



E assim continua, até achar um bloco ou percorrer a sala toda (caso em que volta à área de armazenagem, ou seja, à sua posição inicial em (0,2), avisando que não encontrou nada). Ao encontrar um bloco (ou percorrer a sala toda e não achar nenhum), o robô volta à área de armazenagem (cinza) seguindo sua própria trilha, e limpando as marcações que fez na pilha auxiliar. Note que, ao recomeçar a fase de busca, tanto a estrutura com o caminho atualmente percorrido quanto as marcações na sala devem estar limpas antes da próxima busca.

Essa estrutura auxiliar, em que novos elementos são adicionados ao seu topo, enquanto que, para retirarmos algo, devemos fazê-lo apenas desse topo, é conhecida como pilha. Você, contudo, não precisa implementá-la explicitamente. Basta, para tal, construir uma versão recursiva para a fase da busca. A grande vantagem, nesse caso, é que, durante a recursão, o computador acaba usando uma pilha. Assim, além do código ficar mais legível, você ainda poupa o tempo de criar a estrutura e gerenciá-la. Isso, contudo, é apenas uma sugestão de implementação.

É importante frisar que, a cada movimento do robô, o log deve ser atualizado via `Mensageiro`. Assim, o método `mensagem` do objeto mensageiro deve ser chamado quando:

- O robô visita uma posição. Para isso, faz log da mensagem do tipo `Mensageiro.BUSCA`
- O robô encontra um bloco. Para isso, faz log da mensagem do tipo `Mensageiro.CAPTURA`. Note que a captura vem sempre após uma `BUSCA` na mesma posição
- O robô tenta ir a uma casa e encontra um obstáculo. Para isso, faz log da mensagem do tipo `Mensageiro.OBSTACULO`. Note que, com obstáculos, o robô não entra na casa (ou seja, não gera um log `Mensageiro.BUSCA`), uma vez que o obstáculo o impede de fazê-lo.

- O robô, após encontrar o bloco, anda uma posição no caminho de volta à área de armazenagem. Para isso, faz log da mensagem do tipo `Mensageiro.RETORNO`. Note que essa mensagem não é gerada toda vez que ele faz o backtracking mas, em vez disso, apenas quando se dirige de volta à área de armazenagem trazendo um bloco

- O serviço de armazenagem guarda o bloco em uma posição na área de armazenagem. Gerada toda vez que o robô chega de volta ao local de onde começou a busca, carregando um bloco, e desejando armazená-lo na área de armazenamento. Para isso, faz log da mensagem do tipo `Mensageiro.ARMAZENAGEM`. Mais sobre o serviço de armazenagem será detalhado no que segue.

Além disso, o log deve ser atualizado quando:

- O robô corre a sala toda e não acha 4 blocos. Gerada quando não há mais blocos a serem buscados, ou porque não havia nenhum bloco na sala, ou então porque o bloco está inacessível, dependendo de como estão dispostos os obstáculos. Para isso, usa o método `msgNaoAchou()` de seu mensageiro

- O robô recolheu os 4 blocos para a área de armazenagem. Para isso, usa o método `msgFim()` de seu mensageiro

Ao chegar em uma casa onde exista um bloco, o robô deve coletá-lo, removendo-o desta casa. Para isso ele entra na casa, gerando o log apropriado (de `BUSCA` e `CAPTURA`), e substituindo a marca do bloco (`BLOCO_PRESENTE`) pela de "casa visitada" (`MARCA_PRESENTE`). Ele então deve voltar imediatamente à área de armazenagem, ou seja, sua posição inicial em (0,2), seguindo de volta seus passos (o robô somente carrega um bloco por vez).

Ao voltar à posição de onde começou a busca (ao lado da área de armazenagem), carregando um bloco, o robô deve descarregar o bloco dentro da área. Para tal, não há a necessidade de movimentar o robô dentro da área. Você pode supor que há um serviço automático de armazenagem, a quem o robô entrega o bloco.

Esse serviço de armazenagem, por sua vez, guarda os blocos nas seguintes posições (tomadas em ordem): (0,0), (1,0), (0,1) e (1,1). Dessa forma, o trajeto do robô compreende duas fases: uma busca por blocos, que ocorre na sala (fora da área de armazenagem) e seu armazenamento final (dentro da área). Vale lembrar que esse serviço de armazenagem é quem abastece o log com a mensagem do tipo `Mensageiro.ARMAZENAGEM`.

Nesse projeto, a área de armazenagem compreende a área delimitada por (0,0) e (1,1), sendo que o robô inicia sua busca a partir da posição (0,2). Ao finalizar a busca, o robô deve adicionar esse fato ao log (via `msgFim()`). Uma mensagem de erro também deve ser incluída caso os 4 blocos não possam ser encontrados (via `msgNaoAchou()`).

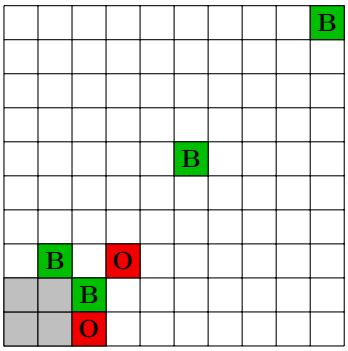
Para esse projeto, você receberá algumas classes e interfaces já compiladas (.class), e outras que você terá que implementar. Essas são:

- IRobo: Interface que contém a assinatura dos métodos que definem um robô e seus movimentos. Ela é apresentada em um .class apenas (você não terá seu código-fonte)

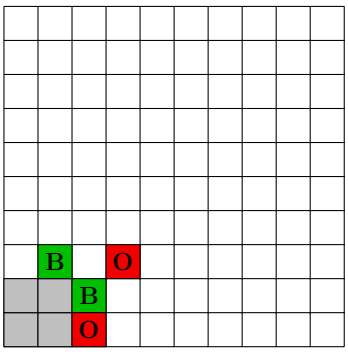
- ISala: Interface que contém a assinatura dos métodos que definem a sala onde a busca será executada. Ela é apresentada em um .class apenas (você não terá seu código-fonte)
- Mensageiro: Classe com os métodos a serem utilizados para gerenciamento do log de mensagens. Não passe nenhuma mensagem além das previstas nesse codumento. A classe é apresentada em um .class apenas (você não terá seu código-fonte)
- **Sala**: Classe que define a sala onde será feita a busca. Implementa a interface ISala (cabe a você fazer essa implementação)
- **Robo**: Classe Robo que implementa o robô e a busca que ele faz. Implementa a interface IRobo (cabe a você fazer essa implementação)
- **TestaRobo**: Classe que testa a busca desenvolvida. Serve como exemplo de como o robô deve ser executado. Use para seus testes apenas

A descrição detalhada de cada classe (javadoc), bem como a implementação das classes fornecidas, você encontra no arquivo ep1.zip, disponibilizado no site do edisciplinas da USP. Esse arquivo contém 2 diretórios:

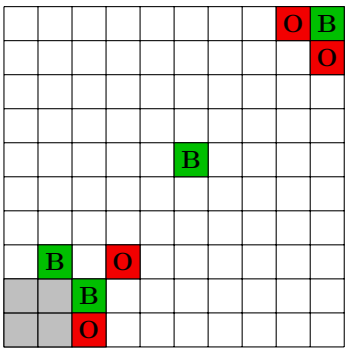
- robo: com as classes e interfaces java a serem utilizadas. Além disso, o diretório também contém três exemplo de saída (Exemplo_ok.txt, Exemplo_falha1.txt e Exemplo_falha2.txt), que ilustram como seria a busca feita nas seguintes condições:
  - Para o exemplo em Exemplo_ok.txt, a seguinte configuração de blocos e obstáculos:



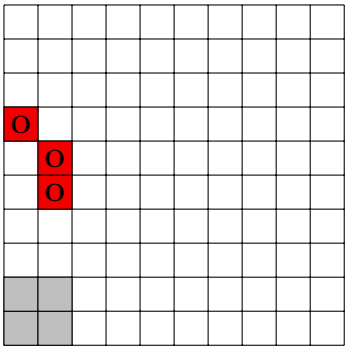
  - Para o exemplo em Exemplo_falha1.txt, a seguinte configuração de blocos e obstáculos (note que não foram espalhados 4 blocos, por isso a falha):



– Para o exemplo em Exemplo_falha2.txt, a seguinte configuração de blocos e obstáculos (note que um dos blocos está inacessível, por isso a falha):



– Para o exemplo em Exemplo_falha3.txt, a seguinte configuração de blocos e obstáculos (o objetivo aqui é ilustrar as mensagens no backtracking. O erro é obviamente por não haver blocos a serem buscados):



- documentacao: com a documentação gerada pelo javadoc (inicie em index.html).
- TestaRobo.java: um arquivo para seus testes. Esse arquivo simula como os testes do sistema serão feitos.

2.1 Entrada

A entrada ao sistema se dará como exemplificado na classe `TestaRobo`. Por meio desta, você deve então criar o robô (e sua sala correspondente), via uma chamada ao construtor `Robo()` (note que os testes serão feitos com o **construtor padrão de** `Robo`, ou seja, qualquer outro construtor será ignorado).

Ao criar o robô, o sistema deve também criar a sala (implementação de `ISala`) onde este andará, bem como o mensageiro (objeto da classe `Mensageiro`) responsável pelo gerenciamento do log dos movimentos do robô.

Uma vez criado o robô, o usuário do sistema pode então povoar a sala com blocos e obstáculos, via chamadas ao método `adicionaBloco` do robô. **Atenção**: é imprescindível que seu robô não adicione nenhum bloco ou obstáculo durante sua criação. O abastecimento destes se dará **unica e exclusivamente** pelo usuário do sistema, via chamadas a `adicionaBloco`. A não observância disso pode tornar o sistema inútil ao usuário, refletindo fortemente na nota.

Uma vez povoado o tabuleiro, o usuário pode então colocar o robô para andar na sala, chamando o método `buscaBlocos()` do robô. Ao final, se assim o desejar, ele pode ver em tela o resultado (usando o método `imprimeMensagens()`, da objeto mensageiro no robô).

2.2 Saída

A saída do sistema, e que será usada para avaliação, é o log gerado. Então é imprescindível que este esteja correto. Para exemplos, consulte os arquivos anexos ao EP.

O log é obtido fazendo-se uma chamada ao método `imprimeMensagens()` do `mensageiro` definido no robô (veja exemplo em `TestaRobo.java`), ou então via o método `mensagens()`.

2.3 Material a Ser Entregue

Este trabalho é individual. Cada aluno deverá implementar e submeter via edisciplinas sua solução.

A submissão será feita via um arquivo zip (o nome do arquivo deverá ser o número USP do aluno. Por exemplo 1234567.zip). Este arquivo deverá conter EXATAMENTE os seguintes arquivos: Sala.java e Robo.java.

Observação: no arquivo zip não adicionar (sub-)diretórios ou outros arquivos, apenas esses dois arquivos. Note que estes arquivos são .java (e não .class).

Qualquer tentativa de fraude ou cola implicará em nota zero para todos os envolvidos. Guarde uma cópia do trabalho entregue.

Não modifique nada do que foi entregue (assinaturas, pacotes etc)! Você pode apenas adicionar código. **Não crie novas classes**, pois apenas Sala.java e Robo.java devem ser entregues.

Caso o EP não compile, a nota do trabalho será zero. É importante que você teste seu trabalho executando a classe TestaRobo (note que ela não testa todas as funcionalidades de todas as classes). Todas as classes (exceto TestaRobo) pertencem ao pacote `robo`. **Não mude isso**!

3 Avaliação

Para avaliação, será observada a corretude do programa (ou seja, se faz o que é pedido e **como** é pedido), bem como sua adequação às regras para entrega (nome do zip, arquivos entregues, assinaturas preservadas etc).

É de sua responsabilidade verificar:

- Se o material entregue está de acordo com as especificações
- Se tudo compila e o sistema roda a partir da `TestaRobo`
- Se a entrega realmente ocorreu (ou seja, se o upload foi feito corretamente). Então faça o upload, baixe e teste o que baixou.

Falhas nos itens acima não serão toleradas.
Atrasos não serão tolerados. Então não deixe para a última hora.
Algumas outras observações pertinentes ao trabalho, que influem em sua nota, são:

- Este exercício-programa deve ser elaborado individualmente.
- Não será tolerado plágio, em hipótese alguma.
- Exercícios com erro de sintaxe (ou seja, erros de compilação), receberão nota ZERO

Atenção! A avaliação se dará como nos moldes da classe TestaRobo entregue. Ela contém os passos seguidos para criar o tabuleiro e povoá-lo, bem como executar a busca. O log gerado por seu robô será então usado na avaliação. **É imprescindível que o log esteja correto**, pois ele corresponde à saída do sistema. Vale frisar novamente que, a cada decisão tomada pelo seu robô, ele deve registrá-la no log. Do contrário a avaliação não será possível, gerando uma nota zero.

3.1 Dicas para Verificação de Funcionamento

Além de formato, corretude etc, sua implementação deve se ajustar perfeitamente ao pacote enviado a vocês. Para aumentar a chance disso acontecer, é aconselhável fazer o seguinte:

1. Crie um diretório qualquer. Digamos que seja `meuDir`
2. Baixe novamente o arquivo `.zip` do EP e o descompacte dentro de `meuDir`. Você verá a `TestaRobo.java`, o diretório `robo` e, dentro dele, as interfaces e a classe `Mensageiro` já compiladas, além dos `.java` para você completar.
3. Substitua então as classes `Robo.java` e `Sala.java` pela sua implementação delas
4. Via linha de comando, vá ao diretório `meuDir`
5. Lá compile tudo com `javac TestaRobo.java` (isso irá compilar inclusive as tuas implementações, usando a `TestaRobo.java` fornecida no EP)
6. Rode com `java TestaRobo`, e compare os resultados com os testes fornecidos

Seguindo esses passos, o sistema certamente funcionará nos testes. Quanto a estar totalmente correto, aí são outros quinhentos...