Projeto 5

Árvores de Robôs

Unidade Curricular de
Laboratório de Programação

2021/2022

Objetivos

- Trabalhar recursividade usando objetos;
- Manipulação de árvores;

Antes de Começar

- Fazer *download* do ficheiro `studentsProject5.zip` acessível na página de LabP;
  Este ficheiro contém:
  - Uma classe `RunProject5.java` com a qual pode experimentar jogar o jogo;
  - Os esqueletos das classes que deve implementar;
  - Ficheiros de *input* para a classe `RunProject5.java`;
  - Ficheiros de testes para os métodos que terá que implementar;
  - Uma pasta com ficheiros de *output* que pode utilizar para comparar os resultados obtidos pela sua solução com os resultados esperados.
- No Eclipse,
  - crie um projeto Java e "arraste" para a pasta `src` os ficheiros `RunProject5.java, Robot.java, Family.java, Game.java, GameEngine.java e GameMsg.java.`
  - arraste os ficheiros de *input* para a raiz do projeto;
  - finalmente, importe os testes JUnit da pasta `Testes` tal como descrito na página 14 do guião "Introdução ao IDE Eclipse".

Algumas informações úteis

De modo a poder realizar este projeto, poderá consultar informação:

- Tutorial sobre recursão da disciplina de LABP
- Exercícios de AED sobre árvores.
- Slides de AED sobre árvores.

Enunciado

Neste projeto vamos simular um jogo que se passa numa realidade distópica onde famílias de robôs se disputam para saber quem terá a maior influência. Vamos fazer isso organizando em árvores as diversas famílias de robôs que lutam pelo poder. Lembrem-se que no jogo dos tronos (dos robôs) só pode existir um vencedor.
Robôs são divididos em três tipos: Os Gigachads, progenitores da família, capazes de criar outros robôs; Os engenheiros, capazes de se juntar a outro robô e se tornar um Gigachad; e os robôs protótipos, robôs básicos sem consciência.
Cada família de robôs é uma árvore, onde um nó da árvore vai representar um robô daquela família e a raiz irá representar o criador daquela família. Robôs podem criar outros robôs, lutar entre si e se juntar a outros robôs. Para aumentar sua influência, famílias podem decidir destruir seus próprios protótipos ou doá-los para outras famílias. O jogo termina quando restar somente uma família que ainda tem robos ativos.

O que fazer então?

O ficheiro `studentsProject5.zip` contém, entre outros elementos, a estrutura base das classes necessárias ao jogo – `Robot, Family e Game` –, que o aluno terá que desenvolver. Contém também um enumerado `Robot.Type`, que define os valores `ENGINEER, PROTOTYPE e GIGACHAD`, um enumerado `GameEngine.Action`, que define as ações que podem ser realizadas no jogo, nomeadamente `CREATE, ATTACK, MERGE, OFFER, DESTROY`, e ficheiros que permitem experimentar e testar as classes a desenvolver. O material disponibilizado compreende também a pasta `expectedOutputs` contendo os ficheiros de *output* esperados, correspondentes aos ficheiros de *input* dados.

O jogo pode ser jogado entre 2 e 9 famílias. O objetivo do jogo é ser a última família com robôs funcionando ao final do turno.

Árvores são estruturas de dados **recursivas** e devem ser implementadas usando recursão para suas operações.

Regras do jogo:

Geração do Id:

1) Cada família deve ter um robô que começou a família. Esse robô deve ser inicializado com um id único de 0 até 9.
2) Cada robô criado tem um id próprio e único. Esse id é determinado pelo id do criador concatenado com o número de descendentes que aquele pai já teve. Exemplo: Se o id do robô criador é `1`, o id dos três primeiros robôs que ele criar devem ser `10`, `11` e `12`. Se o robô `12` for retirado da família, o próximo robô criado por `1` deve ser o `13`.

Estado do Robô

1) Um robô que acabou de ser criado deve ser considerado como ativo.
2) Robôs podem vir a ser desativados durante o decorrer do jogo, mas devem permanecer na árvore da família. Assim que foram desativados, devem ter de ser marcados como não funcionais e não podem mais executar ou ser alvo de ações.
3) Robôs que forem removidos de suas famílias devem ser removidos da árvore da família, mas, caso sejam transferidos para outra família, continuam funcionando normalmente na sua nova família. Somente robôs protótipos podem ser removidos.

Criação de Robôs e organização da árvore:

1) Cada robô pode criar no máximo 10 robôs.
2) Somente robôs de tipo `GIGACHAD` podem criar robôs dos tipos `ENGINEER` ou `PROTOTYPE`.
3) Robôs do tipo `ENGINEER` não podem criar robôs, mas podem ter descendentes através de doações (mais detalhes no tópico 5 de Ações)
4) Um robô criado deve ter uma influência igual a metade do robô que o criou (arredondado para baixo). Criar um robô é uma atividade desgastante, por isso, depois da criação, deve-se remover 2 pontos de influência do robô criador.
5) O mínimo de influência que um robô pode ter é 1. Um `GIGACHAD` com influência 1 ainda pode criar robôs. Neste caso, tanto o criador quanto o robô criado devem ter influência 1.

6) Os robôs criados por um robô `r` devem ser filhos de `r` na árvore e devem ser organizados, da esquerda para a direita, por ordem crescente do seu id próprio.

Cada família tem um chefe. Inicialmente, o chefe deve ser o seu criador. Se por qualquer motivo o chefe da família for desativado, deve-se então procurar por um novo chefe de família. Para isso deve-se utilizar uma busca em profundidade prefixa na árvore da família, partindo do seu criador. Somente `ENGINEER` e `GIGACHAD` podem ser chefes de família.

A busca em profundidade prefixa utilizada para buscar o próximo chefe de família segue o seguinte algoritmo (começando do criador da família, que é a raiz da árvore corrente):

1) Verifica se o robô corrente pode ser chefe.
2) Em caso positivo, retorna o robô corrente.
3) Em caso negativo, executa novamente as etapas 1, 2 e 3 do algoritmo para as subárvores da árvore corrente que tem como raiz o robô corrente, começando do primeiro filho (mais à esquerda).
4) Caso nenhum robô possa ser chefe da família, não deve retornar nada.

Ações:

1) Em cada turno do jogo, uma família pode realizar uma das seguintes ações: criar, atacar, agrupar, oferecer protótipo e destruir protótipo.
2) Criar (`CREATE`): As regras para o `CREATE` estão descritas na alínea anterior.
3) Agrupar (`MERGE`): A família pode selecionar um de seus robôs para se agrupar a um robô de outra família. Somente podem se agrupar engenheiros com protótipos. Quando eles se juntam, o protótipo deve ser removido de sua família e o engenheiro deve mudar de tipo para `GIGACHAD`. A influência do agora `GIGACHAD` deve então ser acrescida da quantidade de influência que o protótipo tinha.
4) Atacar (`ATTACK`): Qualquer robô pode atacar um outro robô de outra família. Em caso de combate, o robô com a maior influência ganha. Em caso de empate, o robô que iniciou o combate perde. O robô que for derrotado deve ser desativado. Nada acontece com o robô vitorioso.
5) Oferecer protótipo (`OFFER`): Uma família pode decidir oferecer um protótipo para uma outra família. Se o fizer, a influência do chefe da família que ofereceu o protótipo deve ser acrescida em duas vezes a influência daquele protótipo. O protótipo doado deve ser o mais próximo do criador da família usando uma busca em largura. O protótipo será adicionado como filho do chefe da família que o recebeu e seu id deve ser alterado para um id válido (enquanto filho do chefe de família que o recebeu). Não deve ser criado um protótipo. O protótipo deve ser removido da família original.
    a. Exemplo: Se a família Asimov (Figura 1: Família Asimov) resolver oferecer um protótipo para a família Musk, o protótipo `11` de Asimov deve ser o doado e `1` vai ter sua influência acrescida de 18 pontos. Numa outra família onde o chefe da família fosse o robô com id `022`, que nunca criou nenhum robô, o protótipo será adicionado como seu descendente com o id `0220`. O robô `022` não deveria perder pontos de influência.
6) Destruir protótipo (`DESTROY`): Como forma de evitar que o protótipo seja usado por outra família, uma família pode decidir remover um protótipo. Só protótipos dentro da própria família podem ser removidos.

Desafio

1) A cada 3 turnos, ocorre um evento chamado **desafio**. Quando ocorre este evento, deve ser verificado, através de uma busca em largura, se há um `GIGACHAD` ou `ENGINEER` com influência maior ou igual ao chefe da família. Em caso de empate nas influências, deve ser retornado o último elemento encontrado pela busca em largura. Caso um robô desafiante exista, este robô irá desafiar o chefe da sua família para um duelo. Ganha o duelo quem tiver a maior influência. O ganhador deste duelo passa a ser o chefe da família. O derrotado deve ser desativado. Em caso de empate, o atual chefe da família é vitorioso.

Começando o jogo.

1) O jogo deve receber como parâmetro o nome das famílias e as influências dos chefes das famílias. O id dos chefes de família deve ser gerado automaticamente contando de 0 para a primeira família, até o máximo de 9 para a última. Cada família deve começar inicialmente com um único `GIGACHAD` (chefe da família e criador). Como parte da inicialização, esse chefe deverá criar (nesta ordem), um robô engenheiro e dois protótipos.

Para ilustrar as regras, vamos usar a família Asimov abaixo como exemplo:

A criação da família Asimov seguiu a seguinte sequência de ações em seus turnos (por simplificação, vamos considerar que outras famílias não vão realizar ações para modificar a família Asimov):

0) Inicialização da família com nome “Asimov”, sendo `1` seu criador, com influência `20`.
    a. Criação do engenheiro `10` com influência 10
    b. Criação do protótipo `11` e `12` com influências 9 e 8, respectivamente.
    c. O Gigachad `1` fica com influência 14 ao fim da criação dos 3 robôs.
1) `1` criou um robô do tipo engenheiro (`13`).
2) `10` juntou-se a um robô protótipo com influência `1` e se torna um `GIGACHAD`.
3) `10` criou um robô do tipo engenheiro (`100`)
4) `10` criou um robô do tipo protótipo (`101`)
5) `13` se juntou a um robô protótipo com influência `11`.
6) `13` criou um robô do tipo protótipo (`130`).

Ao final do turno 6 vai acontecer um desafio entre o robô `1` (atual chefe da família) e o robô `13`.

Os robôs `1`, `10` e `13` são do tipo `GIGACHAD`, o robô `100` é do tipo engenheiro e os robôs `11`, `12`, `101` e `130` são protótipos. As suas influências (ao final da criação de `130`) estão indicadas entre parêntesis retos.

Na família Asimov, a escala de sucessão a lista dos possíveis próximos chefes de família é, na ordem de prioridade: `10`, `100` e `13`

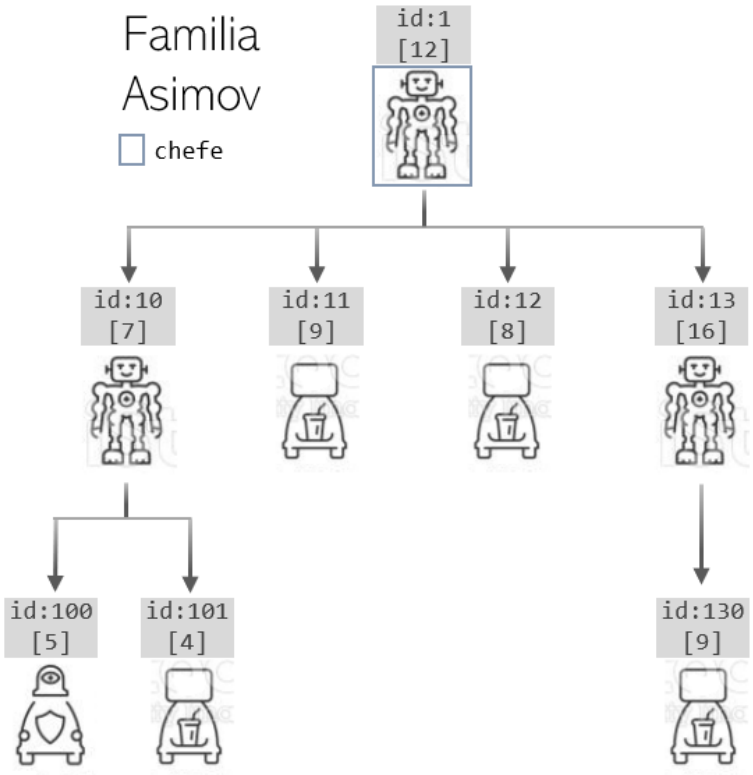

Figura 1: Família Asimov

Implementação

Descrevem-se de seguida as classes `Robot`, `Family` e `Game`.

Robot

Representa um nó da árvore. As instâncias da classe `Robot` representam os robôs que pertencem às famílias. Uma instância de `Robot` deve ter pelo menos o seu id, sua influência, seu tipo, se está desativado e uma lista de `Robot` criados por ele (a lista dos seus filhos). Os métodos públicos que oferece permitem identificar aquele robô e obter informações sobre suas relações. Devem ser fornecidos **pelo menos** os seguintes:

- `public Robot(String id, int influence, Type type)`
  - Constrói um novo robô com seu id, o seu nível de influência e tipo.
- `public boolean isHeadOfFamily()`
  - Devolve se o robô corrente é ou não chefe de família.
- `public boolean isWorking()`
  - Retorna se o robô corrente está ativo.
- `public boolean fabricate(Type type)`
  - Cria um robô do tipo determinado no parâmetro, filho do robô corrente. O novo robô criado deve seguir as regras de criação de robôs. Atualiza a influência do robô corrente. Retorna `false` se o tipo do robô corrente não for `GIGACHAD` ou se tiver tido um total de 10 descendentes anteriormente.

- `public boolean hasDefeated(Robot challenger)`
    - Se o robô corrente tiver uma influência maior do que `challenger`, retorna `true`. Caso contrário, `false`.
- `public void deactivate()`
    - Desativa o robô corrente. Um robô desativado não deve poder realizar ou ser alvo de nenhuma ação e nem ser chefe de família.
- `public String getId()`
    - Retorna o id do robô corrente.
- `public Robot.Type getType(){`
    - Retorna o tipo do robô corrente.
- `public int getInfluence(){`
    - Retorna a influência do robô corrente

Family

As instâncias da classe `Family` representam a árvore da família. Uma instância da classe `Family` deve ter pelo menos o seu nome, o robô que foi o criador da família (a raiz da árvore) e quem é o atual chefe da família. Deve oferecer pelo menos o seguinte construtor e métodos públicos:

- `public Family (String name, Robot owner)`
    - Constrói uma nova família com um nome e o robô que deu origem a família;
- `public Robot findById(String id)`
    - Recebe um id e retorna um robô que tenha aquele id que pertença a esta família. Caso o robô não pertença, deve retornar `null`.
- `public boolean isExtinct()`
    - Verifica se esta família tem todos os membros desativados. Em caso positivo, retorna `true`, caso contrário, `false`.
- `public Robot searchForHeir()`
    - Procura por um novo chefe para esta família, de acordo com o ponto 6 das regras de criação de robôs e organização da árvore. O novo chefe deve necessariamente ser um `ENGINEER` ou `GIGACHAD` e estar ativo. Se for possível encontrar um, retorna-o. Caso contrário, retorna `null`;
- `public boolean contains(Robot b)`
    - Verifica se existe o robô dentro desta família. Retorna `true` em caso positivo e `false` caso contrário.
- `public Robot findInfluencer()`
    - Realiza uma busca em largura e retorna o `GIGACHAD` ou `ENGINEER` com maior influência nesta família. Em caso de empate, deve retornar o último elemento encontrado na busca.
- `public boolean challengerVictory(Robot challenger)`
    - Assume-se que `challenger` é do tipo `ENGINEER` ou `GIGACHAD`. Recebe como parâmetro um robô `challenger` que irá desafiar o chefe da família. Em caso de vitória, se torna o novo chefe desta família e o antigo chefe deve ser desativado. Se o desafiante for vitorioso, deve retornar `true`, caso contrário, `false`.
- `public String getName()`
    - Retorna o nome desta família.

- `public Robot getHeadFamily()`
  - Retorna o chefe desta família.
- `public Robot getOwner()`
  - Retorna o criador desta família.

Game

As instâncias da classe `Game` representam jogos onde o objetivo é ser a última família com robôs ativos. O jogo termina quando há somente uma família ativa. Uma instância da classe `Game` deve ter pelo menos uma lista das famílias que participam do jogo, a família que esteja atualmente jogando e qual o turno atual do jogo. Num dado turno, cada uma das famílias que participam no jogo vai jogar, à vez, executando uma ação possível.

São os seguintes os métodos públicos da classe `Game`. É pressuposto que nenhum dos parâmetros seja `null`:

- `public Game(String[] families, int[] influences)`
  - Assumindo que `families.length > 1` e `families.length < 10` e `families.length == influences.length`, constrói um novo jogo recebendo como primeiro parâmetro os nomes das famílias e como segundo as influências dos seus robôs criadores.
- `public boolean merge(Robot bot, Family family, Robot prot)`
  - Supondo `getPlaying().contains(bot)` e `family.contains(otherBot)`, recebe os robôs que se vão se juntar, engenheiro e protótipo, e a família do protótipo. Verifica se `bot` é um `ENGINEER` e `prot` é um `PROTOTYPE`. Em caso positivo, juntar os robôs. Devolve `true` caso o merge tenha sido feito com sucesso e `false` caso contrário.
- `public boolean createRobot(Robot bot, Robot.Type type)`
  - Supondo que `getPlaying().contains(bot)`, tenta criar na família que está jogando um novo filho de `bot`, com o tipo indicado. Retorna `true` se o robô foi criado e `false` caso contrário.
- `public boolean attack(Robot botAtk, Family fDef, Robot botDef)`
  - Supondo que `getPlaying().contains(botAtk)` e `fDef.contains(botDef)`, recebe os robôs que se vão atacar, atacante e defensor, e a família do defensor. Devolve `true` caso o robô atacante tenha conseguido derrotar o robô defensor e `false` caso contrário.
- `public boolean offerPrototype(Family fReceive)`
  - Recebe como parâmetro a família que vai receber o protótipo. A família que está jogando deve oferecer o protótipo. Devolve `true` caso a oferta tenha sido executada com sucesso e `false` caso contrário.
- `public boolean destroyPrototype(Robot robot)`
  - Supondo que `getPlaying().contains(robot)`, recebe como parâmetro o robô que deve ser destruído. Devolve `true` caso o robô seja um protótipo e foi destruído e `false` caso contrário.

- `public boolean isFinished()`
    - Determina se o jogo foi encerrado. Ele deve ser encerrado quando só restar uma família com robôs não desativados.
- `public Family getPlaying()`
    - Devolve a família que está jogando neste momento.
- `public List<Family> getFamilies()`
    - Devolve uma lista com as famílias que estão participando do jogo.
- `public void updatePlaying()`
    - Utiliza-se da lista de famílias para atualizar a família que está jogando. Cada família tem o seu turno e a ordem deve seguir a inserção na lista de famílias. Se for executado enquanto a última família da lista está jogando, deve atualizar a família que está jogando para a primeira da lista.
- `public Family findFamilyByName(String familyName)`
    - Recebe como parâmetro um nome e procura dentre as famílias que estão jogando alguma que tenha este nome. Se essa família existir, retorna ela. Caso contrário, retorna `null`.
- `public int getTurn()`
    - Retorna o turno atual do jogo.
- `public void nextTurn()`
    - Passa para o próximo turno.

Antes de entregar

Antes de entregar, certifique-se da correção da formatação e inclua comentários *javadoc* para todos os métodos.

Inclua nas classes `Game`, `Family` e `Robot` uma linha contendo a *tag* `@author` onde deve constar o seu nome e número de aluno.

O que entregar

Deve criar o ficheiro `P5fcxxxxx.zip`, onde `xxxxx` é o seu número de aluno, contendo os ficheiros:

`Game.java`, `Family.java` e `Robot.java`

Importante

O facto das vossas classes passarem nos testes fornecidos não significa que a classe esteja 100% correta. Há testes e verificações que não são feitas de propósito, de modo a incentivar os alunos a irem à procura de pontos de eventuais falhas no código. Além disso, os pesos para a avaliação dados a cada um dos testes pode ser diferente. **Devem ter particular atenção com os @requires adequados para o projeto.**