

FCT/UNL Mestrado Integrado em Engenharia Informática

Programação Orientada pelos Objectos, 2018/2019

Teste 2

5 de Junho, 2019

Instruções:

- Antes de começar a resolver, leia o enunciado do princípio até ao fim.
 - As interfaces e classes dos grupos 1 e 2 têm mais métodos do que os que deverá implementar na resolução deste teste. Em cada grupo, tenha o cuidado de ver com muita atenção quais os métodos que deve implementar, para não desperdiçar o seu tempo a implementar métodos que não lhe são pedidos.
 - Disponibilizamos a descrição sumária de todos os métodos, incluindo os que não tem de implementar, para que os possa usar na sua resolução.
 - Pode usar caneta ou lápis.
 - Não é permitido consultar quaisquer elementos para além deste enunciado.
 - Por favor, responda a grupos diferentes em folhas separadas. Verifique que todas as suas folhas de resposta estão identificadas.
 - **Este teste tem a duração máxima de 120 minutos.**
-

Nos grupos 1 e 2 terá que **implementar parcialmente** algumas classes necessárias à construção de um programa para ser usado pela Autoridade Nacional de Protecção Civil.

- No primeiro grupo implementaremos uma classe que representa os meios técnicos (veículos) de uma corporação de bombeiros.
- No segundo grupo implementaremos uma classe que representa a informação sobre as várias corporações de bombeiros, a nível nacional e regional, os meios técnicos (veículos) ao seu dispor, e as missões em que esses meios estão envolvidos.
- No terceiro grupo implementaremos um método adicional de uma classe externa que apresenta na consola informação guardada nas restantes classes.
- No quarto grupo implementaremos um teste unitário e acrescentaremos asserções a uma classe.

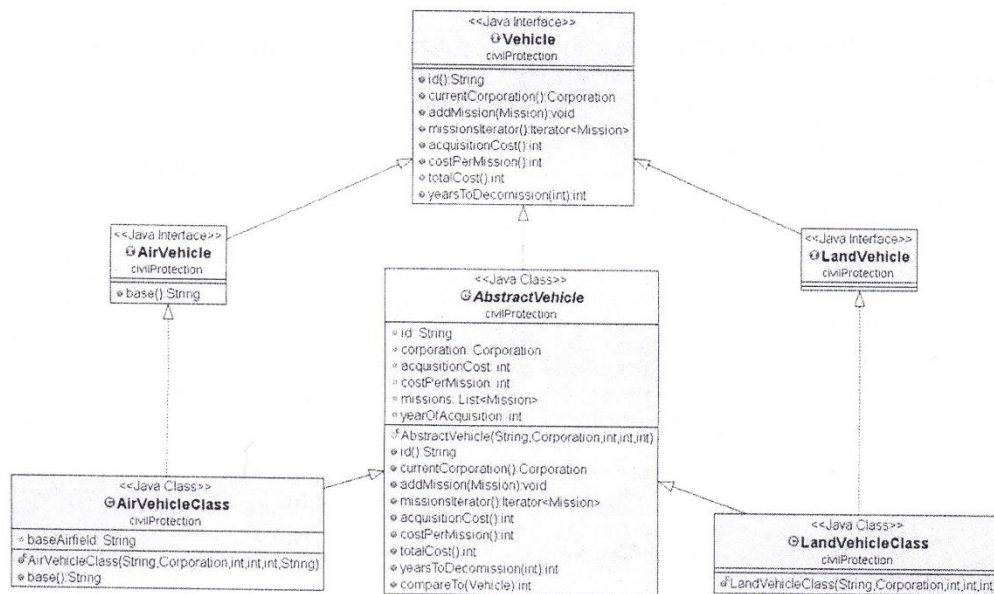


Figura 1 - Veículos terrestres e aéreos

A interface Vehicle especializa a interface Comparable e representa um veículo usado por uma corporação de bombeiros. Neste momento existem veículos aéreos, representados pela interface AirVehicle, e veículos terrestres, representados pela interface LandVehicle. Parte significativa da implementação destes veículos é feita numa classe abstracta AbstractVehicle, que é especializada por uma classe AirVehicleClass e uma classe LandVehicleClass. Na interface Vehicle:

- id devolve uma String com o identificador de um veículo (identificador único).
- currentCorporation devolve a corporação de bombeiros a que o veículo pertence.
- addMission acrescenta uma nova missão à lista de missões realizadas por um veículo. Essas missões são registadas por ordem de inserção no sistema.
- missionsIterator devolve um iterador para a lista de missões realizadas pelo veículo, por ordem de inserção no sistema.
- acquisitionCost devolve o custo de aquisição do veículo.
- costPerMission devolve o custo associado à realização de cada missão pelo veículo.
- totalCost devolve o custo total deste veículo, ou seja, o que resulta da sua aquisição e utilização em missões.
- yearsToDecommission recebe como argumento o ano actual, usando essa informação para calcular quantos anos de vida útil o veículo tem, antes de a sua utilização ser descontinuada. Assume-se que, ao fim de um determinado conjunto de anos, o tempo de vida útil de um veículo esgota-se.

A classe abstracta AbstractVehicle implementa a interfaces Vehicle. O construtor desta classe recebe o identificador do veículo, a corporação, o custo de aquisição, o ano de aquisição e o custo por missão. O método compareTo compara os veículos, usando como critério o custo total de operação. Este método é declarado na interface Comparable.

A interface AirVehicle especializa a interface Vehicle, acrescentando o método base, que devolve o nome da base aérea a partir de onde este veículo opera. A classe AirVehicleClass implementa essa interface. O construtor recebe os mesmos argumentos que a classe abstracta, seguidos de um argumento adicional com o nome da base aérea.

A interface LandVehicle especializa a interface Vehicle, sendo uma interface etiqueta. A classe LandVehicleClass implementa essa interface. O construtor recebe os mesmos argumentos que o construtor da classe abstracta.

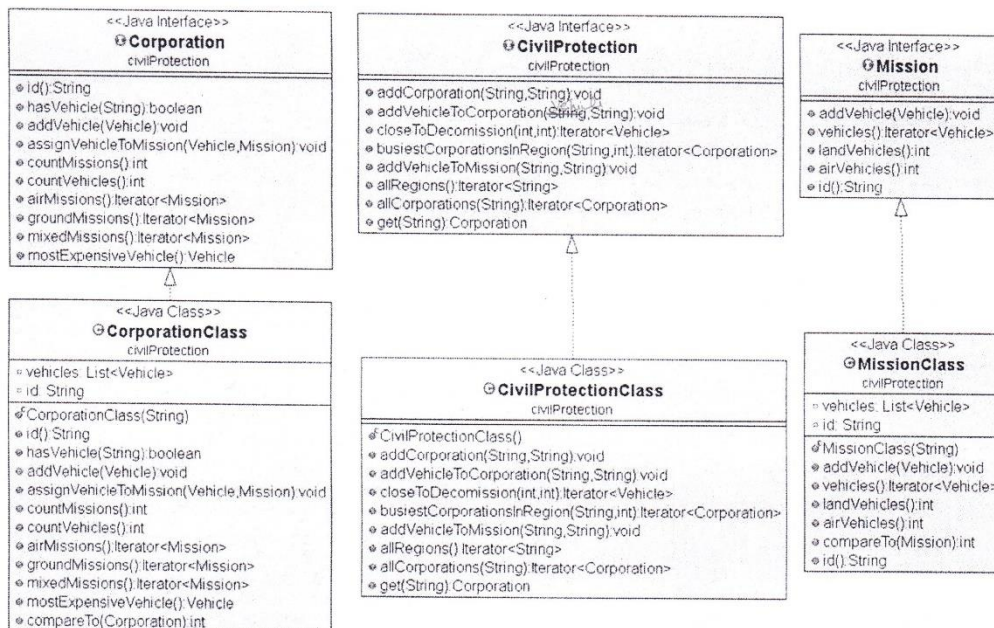


Figura 2 - Corporações, Protecção Civil e Missões

A figura 2 representa as classes e interfaces mais directamente associadas aos grupos 1 e 2. A interface Corporation especializa a interface Comparable, representa uma corporação de bombeiros (identificador único) e tem os seguintes métodos:

- `id` devolve uma `String` com o identificador da corporação de bombeiros (identificador único).
- `hasVehicle` devolve `true` se a corporação tiver o veículo cujo identificador único é passado como argumento, ou `false`, caso contrário.
- `addVehicle` adiciona o veículo passado como argumento à corporação de bombeiros. Se o veículo já existir na corporação de bombeiros, este método lança a excepção `VehicleAlreadyExistsException`.
- `assignVehicleToMission` atribui uma nova missão ao veículo. Como viu na descrição da interface `Vehicle`, cada veículo guarda informação sobre todas as missões em que participa. O método recebe como argumentos um veículo e uma missão. A operação lança:
 - a excepção `VehicleDoesNotExistException` se o veículo não existir na corporação;
 - a excepção `MissionDoesNotExistException`, se a missão não existir;
 - a excepção `MissionAlreadyHasVehicleException`, se o veículo já estava atribuído a esta missão.
- `countMissions` devolve o número total de missões distintas em que esta corporação participou.
- `countVehicles` devolve o número total de veículos que esta corporação tem.
- `airMissions` devolve um iterador para todas as missões desempenhadas por veículos aéreos desta corporação, por uma ordem qualquer, mas sem missões repetidas. Se não existir nenhuma missão, o iterador é devolvido à mesma, neste caso permitindo iterar uma colecção vazia.
- `groundMissions` devolve um iterador para todas as missões desempenhadas por veículos terrestres desta corporação, por uma ordem qualquer, mas sem missões repetidas. Se não existir nenhuma missão, o iterador é devolvido à mesma, neste caso permitindo iterar uma colecção vazia.
- `mixedMissions` devolve um iterador para todas as missões desempenhadas por veículos desta corporação, por uma ordem qualquer, mas sem missões repetidas. Se não existir nenhuma missão, o iterador é devolvido à mesma, neste caso permitindo iterar uma colecção vazia.
- `mostExpensiveVehicle` devolve o veículo desta corporação de bombeiros com maior custo total associado. Este método pode lançar a excepção `NoVehiclesException`, no caso de não existir nenhum veículo nesta corporação de bombeiros.

A classe `CorporationClass` implementa a interface `Corporation`. O seu construtor recebe o identificador único da corporação de bombeiros. Quando é criada, uma corporação não tem quaisquer veículos. Além disso, implementa também o método `compareTo` que ordena, por ordem decrescente, corporações pelo número de missões em que a corporação participou, em caso de empate por ordem crescente de veículos e, se o empate ainda subsistir, por ordem lexicográfica crescente.

A interface `CivilProtection` representa o repositório de dados da Protecção Civil e tem os seguintes métodos:

- `addCorporation` adiciona uma corporação a uma região, dados os identificadores (únicos) de região e de corporação de bombeiros (note que neste caso o identificador da corporação já era único a nível nacional, pelo que continua a ser único dentro da região). Se a região ainda não existir, é criada uma nova região. Se já existir, a nova corporação de bombeiros é adicionada às corporações da região já existente. Cada região pode ter uma ou mais corporações de bombeiros. Este método pode lançar a excepção `CorporationAlreadyExistsException`.
- `addVehicleToCorporation` adiciona um veículo a uma corporação de bombeiros. Recebe o identificador do veículo e o identificador da corporação de bombeiros. Este método pode lançar a excepção `CorporationDoesNotExistException`, se a corporação não existir, `VehicleDoesNotExistException`, se o veículo não existir, e `VehicleAlreadyExistsException`, se o veículo já existir e já for parte da colecção de veículos daquela corporação.
- `closeToDecomission` recebe o ano actual e um limiar máximo (`threshold`) de anos a considerar, e devolve um iterador para uma lista de veículos que estão a menos que o limiar máximo de chegar à idade em que devem ser abatidos. Por exemplo, se um veículo tiver 8 anos de uma vida útil estimada de 10 anos e esta operação for invocada com um limiar de 2 ou mais anos, o veículo é apresentado nesta lista. Se o limiar estipulado fosse 1, o veículo não seria listado, por estar a mais de um ano de atingir o seu tempo de vida útil. A ordem pela qual os veículos filtrados serão visitados com o iterador devolvido por este método é indiferente.
- `busiestCorporationsInRegion` devolve um iterador para corporações de bombeiros de uma determinada região. O método recebe dois parâmetros: o identificador da região e o número máximo de corporações a listar. O iterador permite visitar, por ordem decrescente, as corporações ordenadas pelo número de missões em que a corporação participou, em caso de empate por ordem crescente de veículos e, se o empate ainda subsistir, por ordem lexicográfica crescente. O método pode lançar a excepção `RegionDoesNotExistException`, se a região indicada não existir.
- `addVehicleToMission` recebe como argumentos o identificador de missão e o identificador de veículo, adicionando o veículo à missão. Se a missão ainda não existir, a execução deste método deve criar a missão, atribuindo-lhe em seguida o primeiro veículo. O método pode lançar a excepção `VehicleDoesNotExistException`, se o veículo não existir, ou a excepção `MissionAlreadyHasVehicleException`, se o veículo já fizer parte da missão.
- `allRegions` devolve um iterador para visitar os nomes de todas as regiões, por ordem lexicográfica.
- `allCorporations` devolve um iterador para visitar todas as corporações de uma região, ordenadas pelo critério estabelecido pelo método `compareTo`, implementado em `CorporationClass`.
- `get` devolve a corporação, dado o seu identificador. Se a corporação não existir, lança a excepção `CorporationDoesNotExistException`.

A classe `CivilProtectionClass` tem um construtor sem argumentos que inicializa as variáveis de instância de modo a que as colecções a criar fiquem todas vazias.

A interface `Mission` especializa a interface `Comparable` e representa uma missão, com os seguintes métodos:

- `addVehicle` adiciona um veículo à missão, caso ainda não exista. Caso o veículo já esteja nessa missão, o método lança a excepção `MissionAlreadyHasVehicleException`.
- `vehicles` devolve um iterador para todos os veículos que fazem parte de uma missão.
- `landVehicles` devolve o número de veículos terrestres na missão.
- `airVehicles` devolve o número de veículos aéreos de uma missão.
- `id` devolve o identificador da missão.

A classe `MissionClass` tem ainda um construtor que recebe o identificador único da missão e cria uma missão, que começa por não ter quaisquer veículos associados. Os veículos têm de ser associados posteriormente. Finalmente, esta classe também implementa o método `compareTo`, permite ordenar as missões por ordem decrescente de veículos usados.

Grupo 1 – Colecção pequena (`CorporationClass`)

A classe `CorporationClass` destina-se à gestão da base de dados dos veículos de uma corporação de bombeiros (na ordem das dezenas, no máximo), pelo que temos uma lista denominada `vehicles` para guardar todos os veículos, sejam eles terrestres ou aéreos. Tendo em conta que neste grupo não pode adicionar variáveis de instância e que apenas tem acesso às variáveis de instância `vehicles` e `id`, implemente os seguintes métodos.

- O construtor, de modo a que, ao ser criada, a lista de veículos da corporação de bombeiros esteja vazia. Apresente também, acima da declaração do construtor, a declaração das duas variáveis de instância (`vehicle` e `id`).
- `public void addVehicle(Vehicle vehicle) throws VehicleAlreadyExistsException`
- `public Iterator<Mission> airMissions()`
- `public Vehicle mostExpensiveVehicle() throws NoVehiclesException`

Grupo 2 – Coleção grande (CivilProtection)

Pretende-se implementar uma classe para a gestão dos meios da protecção civil. A implementação da classe deve ter em conta a eficiência das pesquisas a nível geral pelo identificador de uma corporação, pesquisas a nível geral pelo identificador de um veículo, a listagem ordenada por região de indicadores das corporações pertencentes a essa região e a eficiência do método `busiestCorporationsInRegion`.

- Defina as variáveis de instância e escolha as estruturas de dados que achar mais adequadas de acordo com os requisitos de eficiência mencionados acima.
- Defina o construtor de modo a que, ao ser criado, não existam regiões, corporações e veículos.
- `public void addCorporation(String regionId, String corporationId)
throws CorporationAlreadyExistsException`
- `public void addVehicleToMission(String missionId, String vehicleId)
throws VehicleDoesNotExistException, MissionAlreadyHasVehicleException`
- `public Iterator<Vehicle> closeToDecommission(int currentYear, int threshold)`
- `public Iterator<Corporation> busiestCorporationsInRegion(String regionId, int howMany)
throws RegionDoesNotExistException`

Grupo 3 – Programa principal

- Suponha que está a fazer um método `printSummary` no seu programa principal que deve escrever na consola uma listagem ordenada por região com a seguinte informação:
 - Nome da região
 - Nome da corporação, número de missões da corporação, número de veículos da corporação.

Dentro de cada região, a ordem de apresentação das corporações deve ser a estabelecida pelo método `compareTo`. Não necessita de implementar o método `compareTo`.

Por favor, implemente o método da classe `Main`:

```
private static void printSummary(CivilCorporation Cp)
```

Grupo 4 – Testes unitários e asserções

- Implemente uma operação de teste unitário (**Junit**) ao método `yearsToDecommission`, declarado pela interface `Vehicle` e implementado na classe `AbstractVehicleClass`. O seu teste deve verificar vários casos interessantes: o que acontece quando o veículo está dentro de período de vida útil e fora do limiar (`threshold`), o que acontece quando o veículo está dentro do período de vida útil e dentro do limiar (e.g. se o limiar for 3 anos e a viatura estiver a 3 ou menos anos de atingir o fim do período de vida útil), o que acontece quando a viatura atingiu exactamente o período de vida útil e o que acontece quando a idade da viatura já excede a idade limite da sua vida útil. No seu teste deve usar mais do que um veículo e mais do que um tipo de veículo (terrestre, ou aéreo), mas não necessita de testar todos os tipos de veículos com todas as situações acima descritas.
- Considere o método `addVehicleToCorporation` definido na classe `CivilProtectionClass`:

```
public void addVehicleToCorporation(String vehicleId Vehicle vehicle, String corporationId)  
throws CorporationDoesNotExistException, VehicleDoesNotExistException,  
VehicleAlreadyExistsException;
```

Implemente, usando asserções quando adequado, o método `addVehicleToCorporation`. O seu método deve avaliar e tratar, de modo adequado, pelo menos:

- 3 pré-condições – a corporação e o veículo têm de existir, mas o veículo não pode pertencer à corporação
- 2 pós-condições – o veículo passou a fazer parte do conjunto de veículos da corporação e o número de veículos da corporação passou a contar com exactamente mais uma unidade.

Algumas das interfaces abaixo reproduzidas poderão ser úteis na resolução deste teste:

List<E> - add(E) boolean - add(int, E) void - addAll(int, Collection<? extends E>) boolean - addAll(Collection<? extends E>) boolean - clear() void - contains(Object) boolean - containsAll(Collection<?>) boolean - equals(Object) boolean - get(int) E - hashCode() int - indexOf(Object) int - isEmpty() boolean - iterator() Iterator<E> - lastIndexOf(Object) int - listIterator() ListIterator<E> - listIterator(int) ListIterator<E> - remove(int) E - remove(Object) boolean - removeAll(Collection<?>) boolean - retainAll(Collection<?>) boolean - set(int, E) E - size() int - subList(int, int) List<E> - toArray() Object[] - toArray(T[]) <T> T[]	Map<K, V> - clear() void - containsKey(Object) boolean - containsValue(Object) boolean - entrySet() Set<Entry<K, V>> - equals(Object) boolean - get(Object) V - hashCode() int - isEmpty() boolean - keySet() Set<K> - put(K, V) V - putAll(Map<? extends K, ? extends V>) void - remove(Object) V - size() int - values() Collection<V>
Set<E> - add(E) boolean - addAll(Collection<? extends E>) boolean - clear() void - contains(Object) boolean - containsAll(Collection<?>) boolean - equals(Object) boolean - hashCode() int - isEmpty() boolean - iterator() Iterator<E> - remove(Object) boolean - removeAll(Collection<?>) boolean - retainAll(Collection<?>) boolean - size() int - toArray() Object[] - toArray(T[]) <T> T[]	SortedMap<K, V> - comparator() Comparator<? super K> - entrySet() Set<Entry<K, V>> - firstKey() K - headMap(K) SortedMap<K, V> - keySet() Set<K> - lastKey() K - subMap(K, K) SortedMap<K, V> - tailMap(K) SortedMap<K, V> - values() Collection<V> SortedSet<E> - comparator() Comparator<? super E> - first() E - headSet(E) SortedSet<E> - last() E - subSet(E, E) SortedSet<E> - tailSet(E) SortedSet<E>