PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aulas 12/13

Exemplo de consolidação

Pooratas das Caparicas

Pooratas das Caparicas

- Desenvolva um programa em Java que implemente o jogo "*Pooratas das Caparicas*"
  - Essencialmente, trata-se de um jogo tipo **batalha naval**
  - Em vez de porta-aviões, cruzadores e submarinhos, teremos veleiros de diferentes tipos
  - No futuro, podemos querer construir um jogo mais sofisticado com base nesta versão inicial
    - Novos tipos de navios
    - Mapas com formatos diferentes
    - Terra e mar, fortalezas, movimento de navios...
    - Interacção com o utilizador mais "bonitinha"

Descrição do jogo

- O objectivo deste jogo é afundar a frota inimiga, que se encontra distribuída num mapa
  - grelha quadrada de 10x10 posições, normalmente.
- Pode assumir que cada posição na grelha pode ser identificada pelo par (linha, coluna).
- Cada frota conta com um Galeão (representado por um T), uma Fragata (navio de 4 mastros), duas Naus (navios de 3 mastros), três Caravelas (navios de 2 mastros) e quatro Barcas (navios de 1 mastro).
- Os navios têm de caber completamente na grelha, podendo a sua posição variar. Por exemplo, o T do Galeão pode estar "deitado" para qualquer dos lados, ou invertido, mas não na diagonal.
- Por outro lado, sabe-se também que os navios não se tocam, nem sequer na diagonal.
- Os combates são feitos com rajadas de 3 tiros

Operações suportadas pelo jogo

- **`nova`**
  - O comando **`nova`** permite construir uma frota (deitando fora outra que já exista, se for caso disso)
    - Ou seja, o jogo recomeça!
  - Cada frota deve ter 11 navios:
    - 1 galeão
    - 1 fragata
    - 2 naus
    - 3 caravelas
    - 4 barcas
  - Durante a construção da frota, pode acontecer que se tentem colocar navios em posições inválidas. Esses navios devem ser ignorados, sendo escrita uma mensagem de erro adequada
  - Apenas entram para a frota navios "aceitáveis". Quando a frota fica completa, o programa deve escrever uma mensagem que indica que foi criada uma frota com os tais 11 navios
  - A ordem da criação de navios, dentro da frota, não é importante

Operações suportadas pelo jogo

- Na sequência do comando **`nova`**:
  - Sub-comandos para criar os 11 navios:
    - Tipo de navio Linha Coluna Orientação
      - Tipo de navio:
        - **`barca, caravela, nau, fragata, galeao`**
      - Linha – coordenada vertical mais a norte do navio
      - Coluna – coordenada horizontal mais a leste no mapa
      - Orientação (Norte, Sul, Este, Oeste)
        - **`n, s, o, e`**
  - Exemplo (galeão criado na linha 0, coluna 3, virado para oeste):

```
galeao 0 3 o
```

Operações suportadas pelo jogo

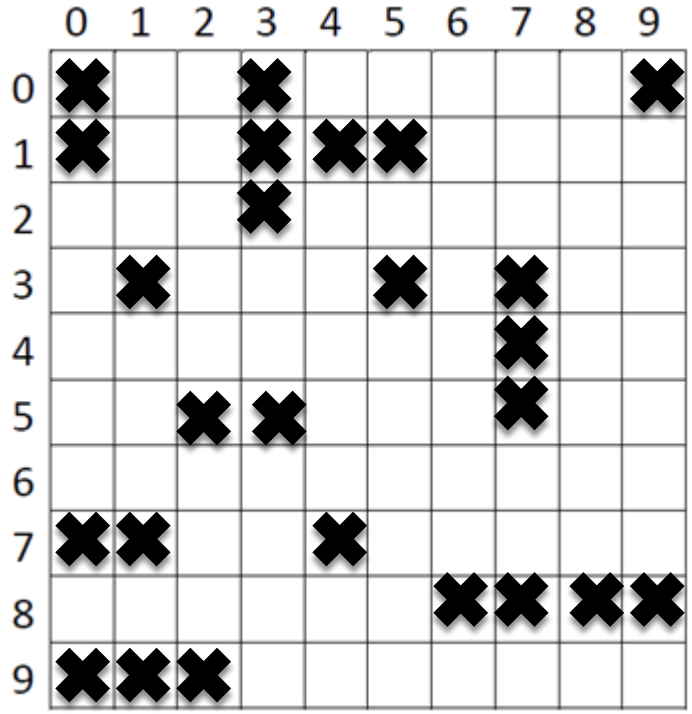
- **`mapa`**
  - Escreve na consola um mapa com a localização dos navios, usando caracteres diferentes para representar se as posições estão ou não ocupadas por algum navio
    - Nos exemplos de interacção, usaremos o carácter '#' para sinalizar as posições ocupadas por algum navio, e o carácter '.' para sinalizar as restantes posições
- **`rajada`**
  - Recebe as coordenadas de 3 pontos atingidos por uma rajada de tiros
    - A coordenada de cada ponto é representa por dois inteiros, correspondendo a uma linha e uma coluna

Operações suportadas pelo jogo

- **`ver`**
  - Escreve na consola um mapa com a localização dos tiros dados nas sucessivas rajadas
    - Nos exemplos de interacção, usaremos o carácter 'X' para sinalizar as posições atingidas por algum tiro, e o carácter '.' para sinalizar as restantes posições
- **`desisto`**
  - Termina a execução do programa
    - O programa deve retornar uma mensagem de despedida e terminar a execução

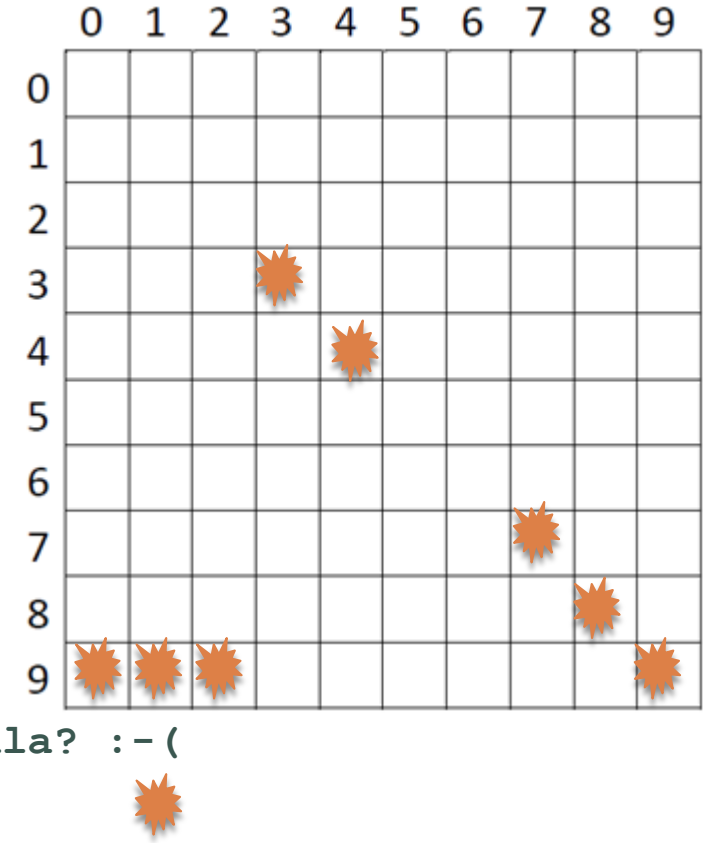
Exemplo de interacção


```
nova
caravela 0 0 s
galeao 0 3 o
barca 0 9 n
fragata 8 6 e
nau 9 0 e
caravela 5 2 o
nau 3 7 n
barca 3 1 s
barca 3 5 e
caravela 7 0 e
barca 7 4 o
11 navios adicionados com sucesso!
```



Exemplo de interacção


```
mapa
#..#.....#
#..###....
...#......
.#...#.#..
.......#..
..##...#..
..........
##..#.....
......####
###.......
rajada 9 9 8 8 7 7
Hits: 1 Inv: 0 Rep: 0 Restam 11 navios.
rajada 9 0 9 1 9 2
Mas... mas... Naus nao sao a prova de bala? :-(
Hits: 4 Inv: 0 Rep: 0 Restam 10 navios.
rajada 11 1 4 4 3 3
Hits: 4 Inv: 1 Rep: 0 Restam 10 navios.
```



Exemplo de interacção


```
ver
..........
..........
..........
...x......
....x.....
..........
..........
.......x..
........x.
xxx......x
desisto
Bons ventos!
```

Comecemos pelo interpretador de comandos

O que deve estar disponível para o interpretador de comandos?

Como fazer **MAL**...

- Misture o interpretador de comandos com as classes e interfaces que representam a parte lógica do jogo
- Garanta que o código de interpretador de comandos ocupa várias páginas, tudo na função `main()`
- Faça o seu interpretador de comandos depender do maior número de detalhes possível da implementação
- Evite usar interfaces para encapsular a parte lógica da aplicação
- Implemente detalhes sobre a interacção do jogo com o jogador nas classes lógicas
- Divirta-se, horas a fio, a remover bugs provocados por más escolhas iniciais...

Ou então...

- A classe com o programa principal interage com o estado do jogo através de uma interface bem definida
  - Ou seja, temos:
    - Uma classe com o interpretador de comandos e programa principal
    - Uma interface a modelar a interacção entre o interpretador de comandos e a parte lógica do jogo
    - Uma classe a implementar o jogo, propriamente dito
      - Mais uma série de outras classes e interfaces que sejam necessárias para a parte lógica do jogo

A interface Game

A interface **Game**

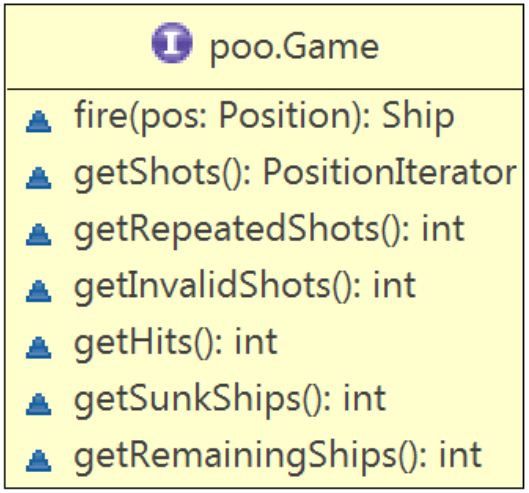
- `Ship fire(Position pos)`
  - Dispara para a posição **pos**. Caso o tiro resulte no afundamento de um navio, esse navio é devolvido como resultado da operação. Caso contrário, a operação devolve **`null`**.
- `PositionIterator getShots()`
  - Devolve um iterador com as posições atingidas por tiros. Nada se sabe sobre o resultado dos tiros em cada uma das posições.
- **`int`** `getRepeatedShots()`
  - Devolve o número de tiros repetidos. Note que o primeiro tiro a acertar numa posição não conta, mas os tiros que posteriormente sejam feitos a essa posição serão contabilizados.

A interface `Game`

- `int getInvalidShots()`
  - Devolve o número de tiros fora do mapa.
- `int getHits()`
  - Devolve o número de tiros no alvo, não contando com repetições.
- `int getSunkShips()`
  - Devolve o número de navios afundados.
- `int getRemainingShips()`
  - Devolve o número de navios por afundar.

A interface Game


```
public interface Game {
   Ship fire(Position pos);
   PositionIterator getShots();
   int getRepeatedShots();
   int getInvalidShots();
   int getHits();
   int getSunkShips();
   int getRemainingShips();
}
```



- Temos de definir
  - o iterador
  - a interface Ship
  - a interface Position
- Porque não temos acesso à frota?
  - Normalmente, quem ataca não pode aceder à localização da frota
- Mas como é que ela aparece, então?
  - Ao construir um objecto da classe GameClass, que implementa Game, podemos passar uma frota construída

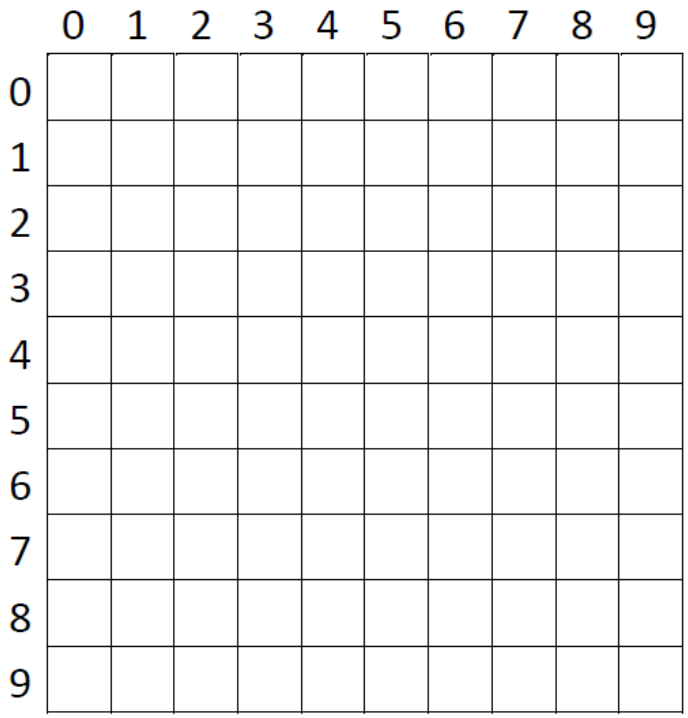
A interface `Ship`

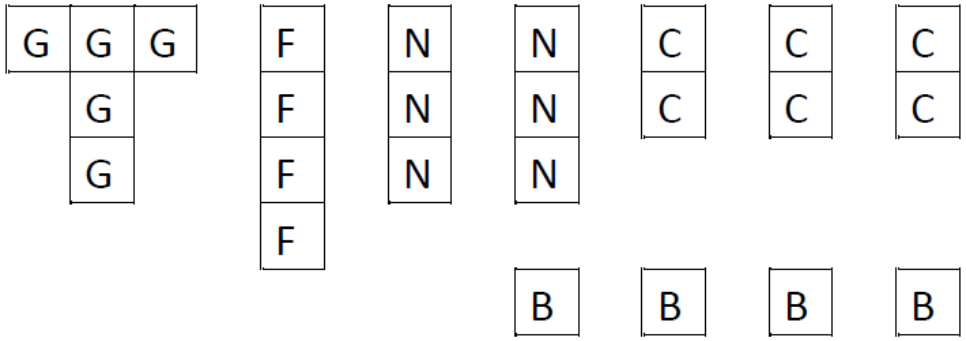
A interface `Ship`

- Esta interface deve poder representar **qualquer um** dos navios da frota
  - Actuais, e outros que venham a ser inventados numa nova versão do jogo, se for caso disso

A interface `Ship`

Que tipos de navios existem, à partida?





G – Galeão
F – Fragata
N – Nau
C – Caravela
B – Barca

O que podemos fazer aos navios?

- (Criar o navio, com uma determinada posição e orientação)
- Consultar a sua categoria
- Consultar o seu número de mastros (corresponde ao número de posições ocupadas no mapa)
- Consultar a sua orientação
- Consultar se ainda flutua, ou se foi afundado
- Testar se ocupa, ou não, uma determinada posição

O que podemos fazer aos navios?

- Testar se se encontra, ou não, demasiado próximo de uma determinada posição
- Obter as posições que o navio ocupa
- Dar um tiro ao navio
- Adicionalmente, é útil poder consultar, ao tentar testar se é possível adicionar um navio a um mapa
  - A abcissa mais à esquerda ocupada pelo navio
  - A abcissa mais à direita ocupada pelo navio
  - A ordenada mais acima ocupada pelo navio
  - A ordenada mais abaixo ocupada pelo navio

A interface `Ship`

- `String getCategory()`
  - Devolve a categoria do navio.
- `int getSize()`
  - Devolve a dimensão do navio, ou seja, o número de posições que ele ocupa no mapa.
- `Position getPosition()`
  - Obter a posição do navio. Esta posição representa o ponto mais acima, e, entre as mais acima, a que estiver mais à esquerda, do navio.
- `Compass getBearing()`
  - Obter a orientação do navio. Um navio pode estar virado para norte, sul, este ou oeste

A interface `Ship`

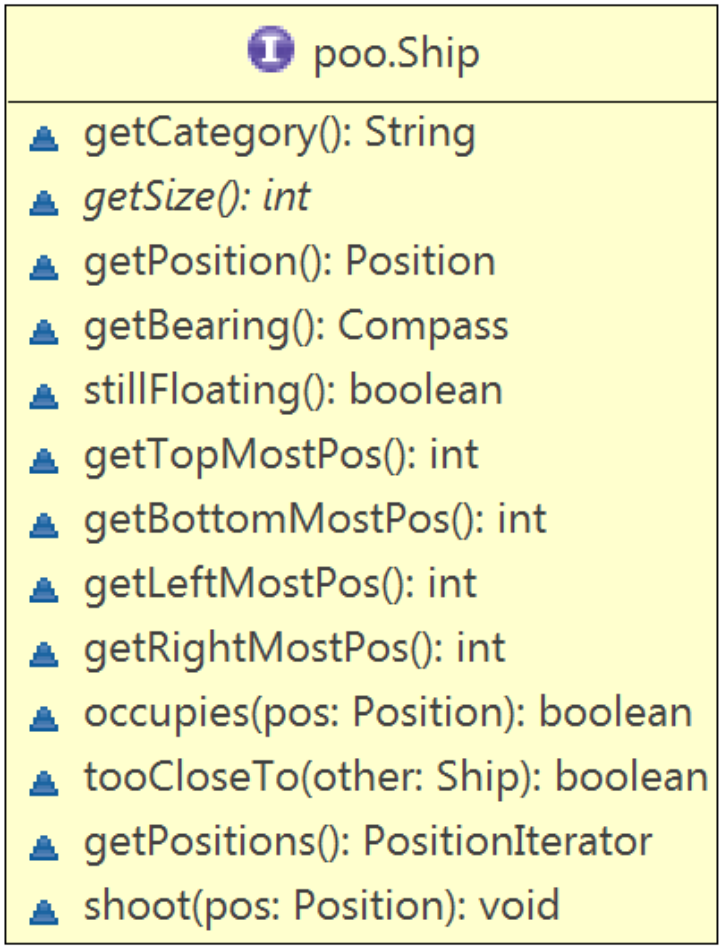
- `boolean stillFloating()`
  - Operação que devolve `true` se o navio estiver a flutuar, ou `false` se o navio já foi afundado.
- `int getTopMostPos()`
  - Ordenada mais a norte ocupada pelo navio
- `int getBottomMostPos()`
  - Ordenada mais a sul ocupada pelo navio
- `int getLeftMostPos()`
  - Abcissa mais a oeste ocupada pelo navio
- `int getRightMostPos()`
  - Abcissa mais a este ocupada pelo navio

A interface `Ship`

- `boolean occupies(Position pos)`
  - Devolve `true` se o navio ocupa a posição `pos`, `false`, caso contrário.
- `boolean tooCloseTo(Ship other)`
  - Devolve `true` se os dois navios estiverem perto demais, ou seja, se os navios se estiverem a tocar.
- `PositionIterator getPositions()`
  - Devolve a lista de posições ocupadas pelo navio.
- `void shoot(Position pos)`
  - dispara sobre a posição `pos`.

A interface Ship


```
package poo;
/**
 * Representa todos os navios deste jogo
 * @author Miguel Goulão
 *
 */
public interface Ship {
  String getCategory();
  int getSize();
  Position getPosition();
  Compass getBearing();
  boolean stillFloating();
  int getTopMostPos();
  int getBottomMostPos();
  int getLeftMostPos();
  int getRightMostPos();
  boolean occupies(Position pos);
  boolean tooCloseTo(Ship other);
  PositionIterator getPositions();
  void shoot(Position pos);
}
```



A interface `Position`

A interface `Position`

- Permite manipular posições do mapa do jogo
  - Podemos obter as suas coordenadas (abcissas e ordenadas)
  - Podemos comparar com outra posição, para determinar se são equivalentes
  - Podemos testar se uma posição é adjacente a outra
  - Podemos ocupar uma posição
  - Podemos testar se uma posição está ou não ocupada
  - Podemos disparar sobre uma posição
  - Podemos testar se já disparámos sobre uma posição

A interface `Position`

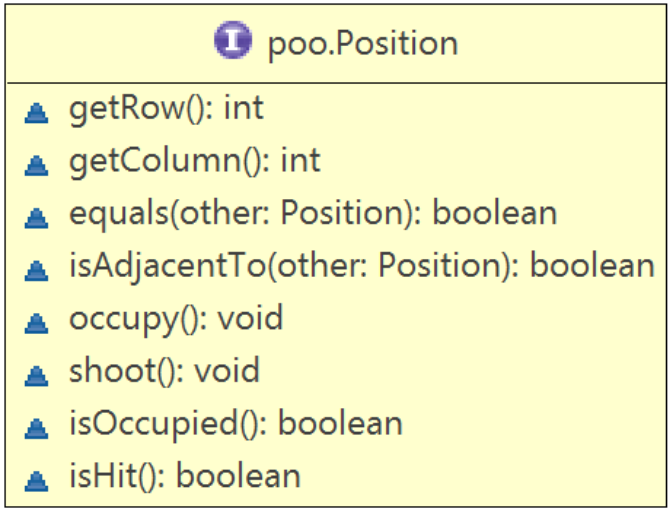
- `int getRow()`
  - Devolve a linha da posição
- `int getColumn()`
  - Devolve a coluna da posição
- `boolean equals(Position other)`
  - Devolve `true` se as coordenadas forem semelhantes, `false` caso contrário
- `boolean isAdjacentTo(Position other)`
  - Devolve `true` se a posição `other` for adjacente à posição do objecto

A interface `Position`

- **`void`** `occupy()`
  - Faz com que a posição passe a estar ocupada por um navio
- **`void`** `shoot()`
  - Dispara sobre a posição
- **`boolean`** `isOccupied()`
  - Devolve **`true`** se esta posição é ocupada por algum navio, **`false`** caso contrário
- **`boolean`** `isHit()`
  - Devolve **`true`** se esta posição já foi atingida, **`false`** caso contrário

A interface `Position`


```
/**
 * Representa uma posição no mapa que pode ser, ou não
 * atacada e que pode ter, ou não, um navio a ocupa-la.
 * @author Miguel Goulão
 */
public interface Position {
  int getRow();
  int getColumn();
  boolean equals(Position other);
  boolean isAdjacentTo(Position other);
  void occupy();
  void shoot();
  boolean isOccupied();
  boolean isHit();
}
```



A interface `PositionIterator`

A interface `PositionIterador`

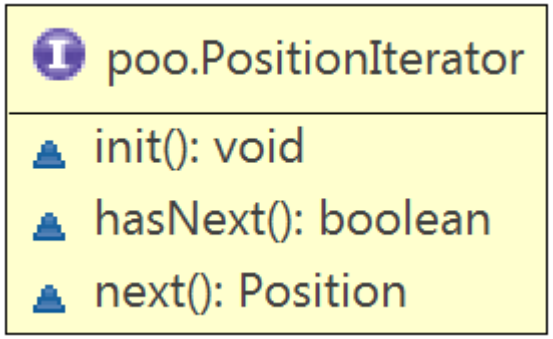
- Permite manipular um iterador de posições
  - Podemos ir para o início da lista de posições
  - Podemos verificar se há mais posições a visitar
  - Podemos visitar a próxima posição

A interface `PositionIterator`

- `void init()`
  - Vai para o início da colecção de posições
- `boolean hasNext()`
  - Devolve `true` se existirem mais posições a visitar, ou `false`, caso contrário
- `Position next()`
  - Devolve a próxima posição se existir, ou `null`, caso contrário

A interface `PositionIterator`


```
/**
 * A interface <code>PositionIterator</code> permite visitar todos os
 * elementos de uma colecção de posições, representadas como objectos
 * que implementam a interface <code>Position</code>.
 * @author Miguel Goulão
 */
public interface PositionIterator {
  void init();
  boolean hasNext();
  Position next();
}
```



O enumerado Compass

O enumerado Compass

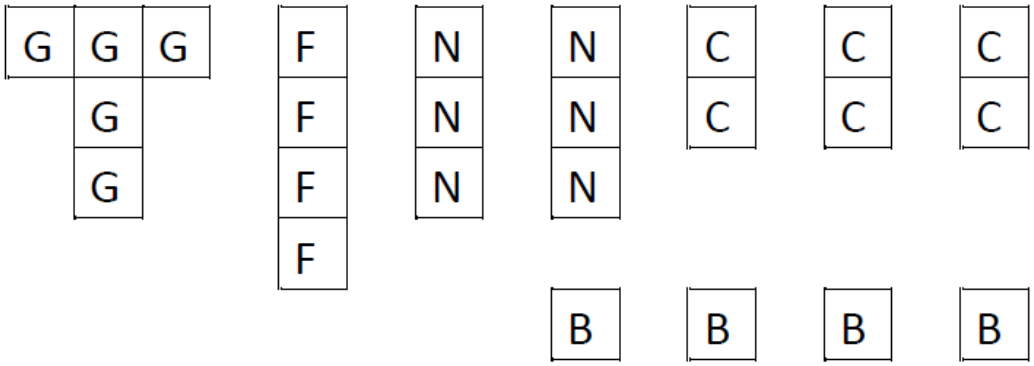

```
public enum Compass {
   NORTH ('n'),
   SOUTH ('s'),
   EAST ('e'),
   WEST ('o');

   private final char c;

   Compass(char c) {
      this.c = c;
   }

   public String toString() {
      return ""+c;
   }
}
```

Implementações

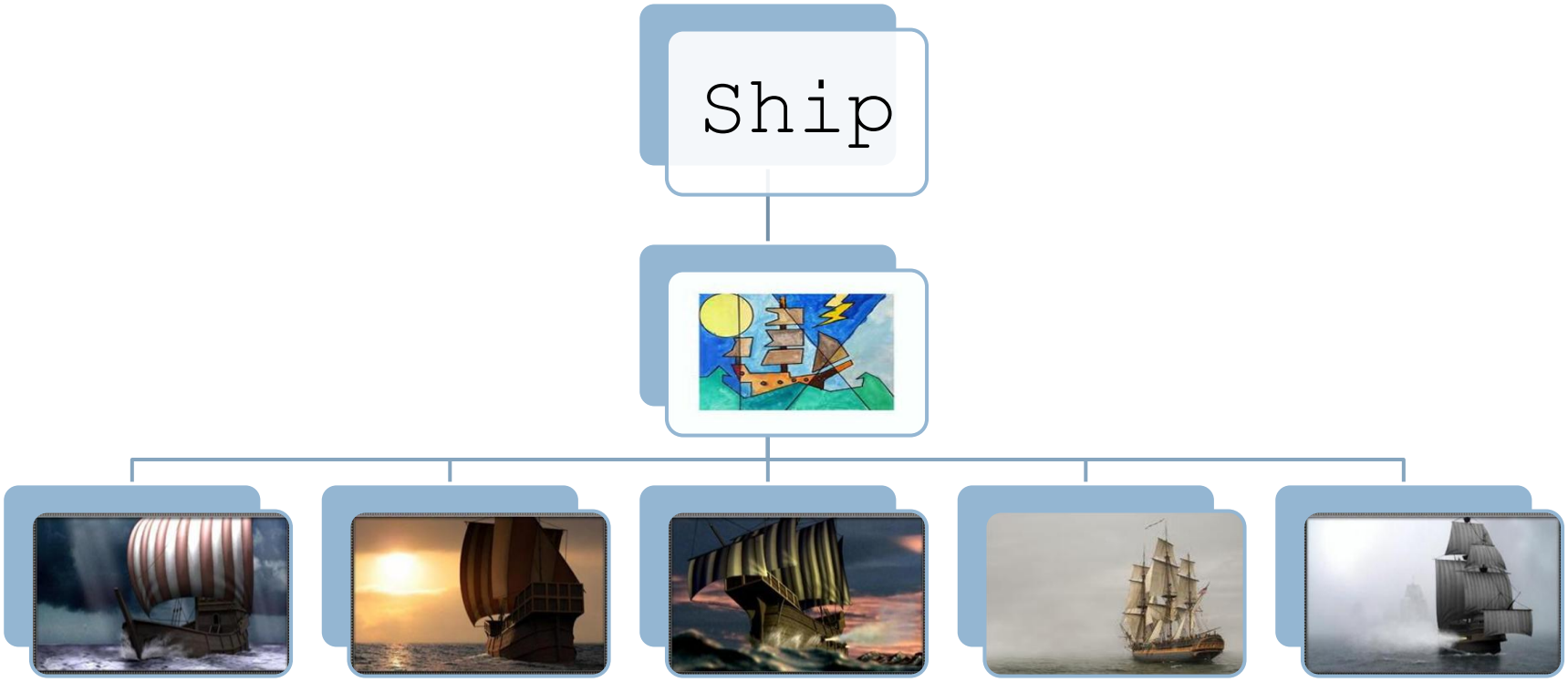
Implementação de `Ship`

Comecemos pelas "docas"

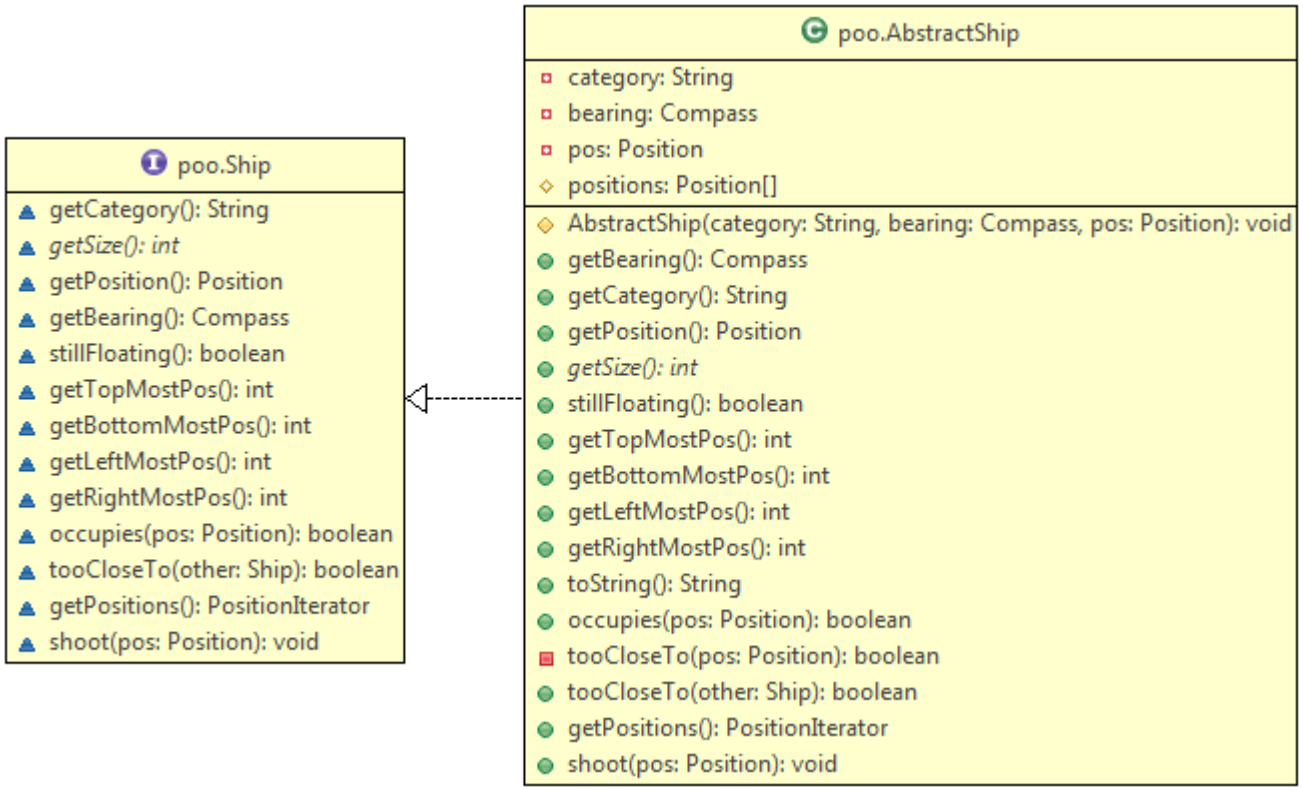
- Temos de representar os vários navios
- Um navio tem:
  - Dimensão (e formato)
  - Posição
  - Orientação
  - Pode ainda estar a flutuar, ou não



A hierarquia de navios



Classe abstracta `AbstractShip`



Classe abstracta **AbstractShip**

```
public abstract class AbstractShip implements Ship {
  private String category;
  private Compass bearing;
  private Position pos;
  protected Position[] positions;

  protected AbstractShip (String category, Compass bearing, Position pos) {
    this.category = category;
    this.bearing = bearing;
    this.pos = pos;
    // Concrete classes must define the positions vector!
  }

  public Compass getBearing() {
    return bearing;
  }

  public String getCategory() {
    return category;
  }
```

Classe abstracta **AbstractShip**

```
  public Position getPosition() {
    return pos;
  }

  public abstract int getSize();

  public boolean stillFloating() {
    for(int i = 0; i < getSize(); i++)
      if (!positions[i].isHit())
        return true;
    return false;
  }

  public int getTopMostPos() {
    int top = positions[0].getRow();
    for (int i = 1; i < getSize(); i++)
      if (positions[i].getRow() < top)
        top = positions[i].getRow();
    return top;
}
```

Classe abstracta `AbstractShip`

```
public int getBottomMostPos() {
  int bottom = positions[0].getRow();
  for (int i = 1; i < getSize(); i++)
    if (positions[i].getRow() > bottom)
      bottom = positions[i].getRow();
  return bottom;
}

public int getLeftMostPos() {
  int left = positions[0].getColumn();
  for (int i = 1; i < getSize(); i++)
    if (positions[i].getColumn() < left)
      left = positions[i].getColumn();
  return left;
}

public int getRightMostPos() {
  int right = positions[0].getColumn();
  for (int i = 1; i < getSize(); i++)
    if (positions[i].getColumn() > right)
      right = positions[i].getColumn();
  return right;
}
```

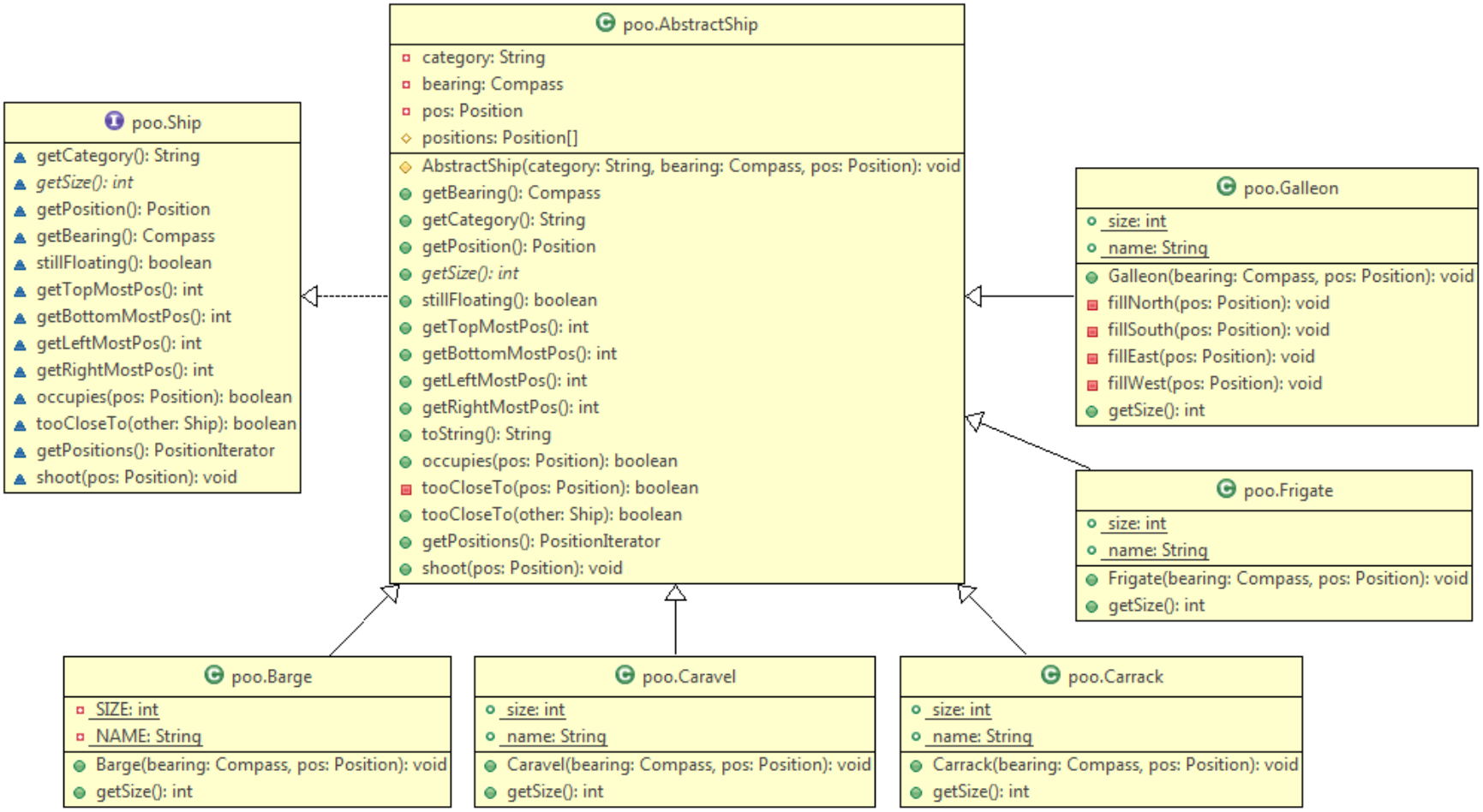
Classe abstracta **AbstractShip**

```
public String toString() {
  return "[" + category + " " + bearing + " " + pos + "]";
}

public boolean occupies(Position pos) {
  for (int i = 0; i < this.getSize(); i++)
    if (positions[i].equals(pos))
      return true;
  return false;
}

private boolean tooCloseTo(Position pos) {
  for (int i = 0; i < this.getSize(); i++)
    if (positions[i].isAdjacentTo(pos))
      return true;
  return false;
}
```

Classe abstracta **AbstractShip**

```
  public boolean tooCloseTo(Ship other) {
    PositionIterator otherPos = other.getPositions();
    otherPos.init();
    while (otherPos.hasNext())
      if (tooCloseTo(otherPos.next()))
        return true;
    return false;
  }

  public PositionIterator getPositions() {
    return new PositionIteratorClass(positions, this.getSize());
  }

  public void shoot(Position pos) {
    for (int i = 0; i < this.getSize(); i++)
      if (positions[i].equals(pos))
        positions[i].shoot();
  }
}
```

Hierarquia de navios



Classe Barge (Barca)

```
public class Barge extends AbstractShip {
  private static final int SIZE = 1;
  private static final String NAME = "Barca";

  /**
   * @param bearing - orientação do navio
   * @param pos - posição do ponto mais acima e
   * mais à esquerda deste navio
   */
  public Barge(Compass bearing, Position pos) {
    super(NAME, bearing, pos);
    positions = new Position[SIZE];
    positions[0] = new PositionClass(pos.getRow(), pos.getColumn());
  }

  public int getSize() {
    return SIZE;
  }
}
```

Classe Caravel (Caravela)

```
public class Caravel extends AbstractShip {
  public static final int SIZE = 2;
  public static final String name = "Caravela";

  public Caravel(Compass bearing, Position pos) {
    super(Caravel.name, bearing, pos);
    positions = new Position[SIZE];

    if (bearing == Compass.NORTH || bearing == Compass.SOUTH)
      for (int r = 0; r < SIZE; r++)
        positions[r] = new PositionClass(pos.getRow()+r, pos.getColumn());

    else if (bearing == Compass.EAST || bearing == Compass.WEST)
      for (int c = 0; c < SIZE; c++)
        positions[c] = new PositionClass(pos.getRow(), pos.getColumn()+c);
  }

  public int getSize() {
    return Caravel.SIZE;
  }
}
```

A classe Carrack (Nau)

```
public class Carrack extends AbstractShip {
  public static final int SIZE = 3;
  public static final String name = "Nau";

  public Carrack(Compass bearing, Position pos) {
    super(Carrack.name, bearing, pos);
    positions = new Position[SIZE];

    if (bearing == Compass.NORTH || bearing == Compass.SOUTH)
      for (int r = 0; r < SIZE; r++)
        positions[r] = new PositionClass(pos.getRow()+r, pos.getColumn());

    else if (bearing == Compass.EAST || bearing == Compass.WEST)
      for (int c = 0; c < SIZE; c++)
        positions[c] = new PositionClass(pos.getRow(), pos.getColumn()+c);
  }

  public int getSize() {
    return Carrack.SIZE;
  }
}
```

A classe **Frigate** (Fragata)

```
public class Frigate extends AbstractShip {
  public static final int SIZE = 4;
  public static final String name = "Fragata";

  public Frigate(Compass bearing, Position pos) {
    super(Frigate.name, bearing, pos);
    positions = new Position[SIZE];
    if (bearing == Compass.NORTH || bearing == Compass.SOUTH)
      for (int r = 0; r < SIZE; r++)
        positions[r] = new PositionClass(pos.getRow()+r, pos.getColumn());

    else if (bearing == Compass.EAST || bearing == Compass.WEST)
      for (int c = 0; c < SIZE; c++)
        positions[c] = new PositionClass(pos.getRow(), pos.getColumn()+c);
  }

  public int getSize() {
    return Frigate.SIZE;
  }
}
```

A classe Galleon (Galeão)

```
public class Galleon extends AbstractShip {
  public static final int SIZE = 5;
  public static final String name = "Galeao";

  public Galleon(Compass bearing, Position pos) {
    super(Galleon.name, bearing, pos);
    this.positions = new Position[SIZE];
    switch (bearing) {
      case NORTH:
        fillNorth(pos);
        break;
      case EAST:
        fillEast(pos);
        break;
      case SOUTH:
        fillSouth(pos);
        break;
      case WEST:
        fillWest(pos);
        break;
    }
  }
```

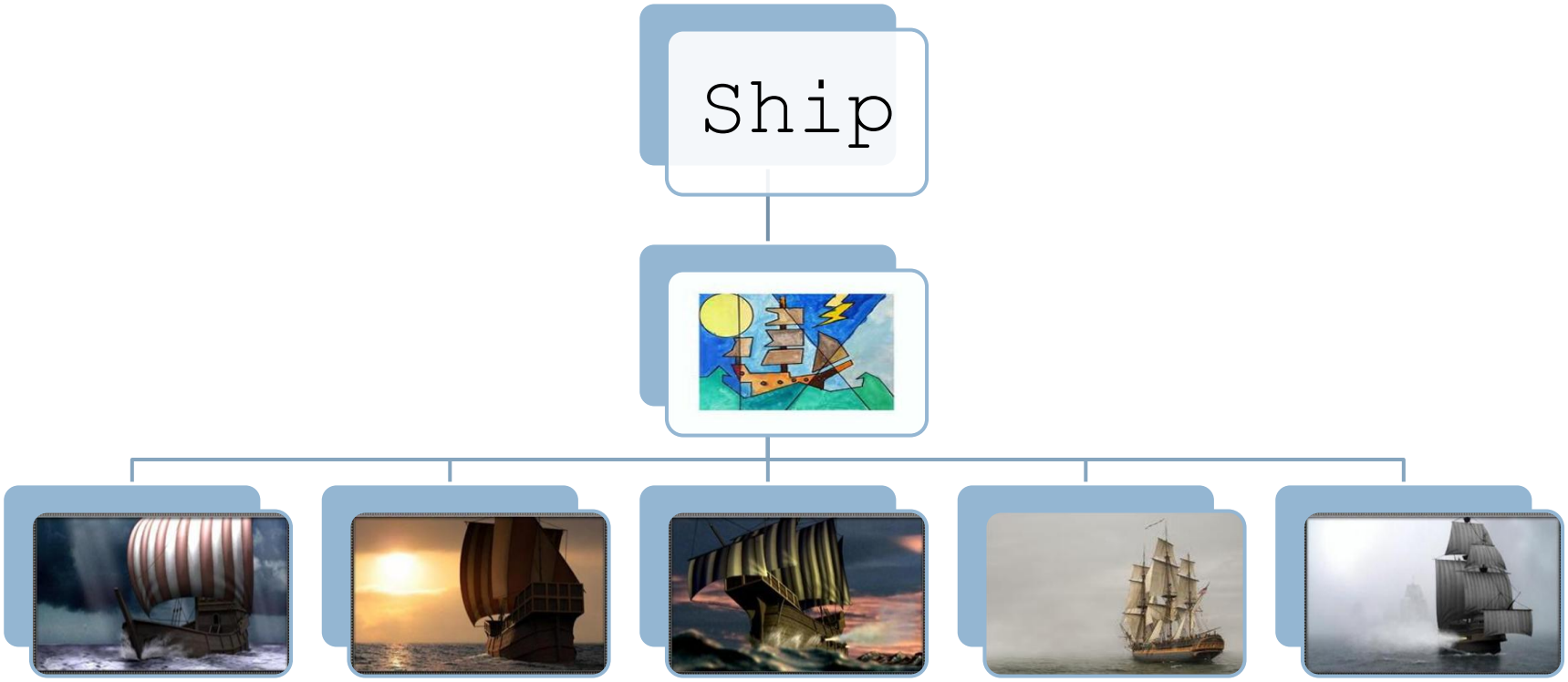
A classe `Galleon` (Galeão)

```
private void fillNorth(Position pos) {
  for (int i = 0; i < 3; i++)
    positions[i] = new PositionClass(pos.getRow(), pos.getColumn()+i);

  positions[3] = new PositionClass(pos.getRow()+1, pos.getColumn()+1);
  positions[4] = new PositionClass(pos.getRow()+2, pos.getColumn()+1);
}

private void fillSouth(Position pos) {
  for (int i = 0; i < 2; i++)
    positions[i] = new PositionClass(pos.getRow()+i, pos.getColumn());

  for (int j = 2; j < 5; j++)
    positions[j] = new PositionClass(pos.getRow()+2, pos.getColumn()+j-3);
}
```

A classe Galleon (Galeão)

```
  private void fillEast(Position pos) {
    positions[0] = new PositionClass(pos.getRow(), pos.getColumn());
    for (int i = 1; i < 4; i++)
      positions[i] = new PositionClass(pos.getRow()+1, pos.getColumn()+i-3);

    positions[4] = new PositionClass(pos.getRow()+2, pos.getColumn());
  }

  private void fillWest(Position pos) {
    positions[0] = new PositionClass(pos.getRow(), pos.getColumn());
    for (int i = 1; i < 4; i++)
      positions[i] = new PositionClass(pos.getRow()+1, pos.getColumn()+i-1);

    positions[4] = new PositionClass(pos.getRow() + 2, pos.getColumn());
  }

  public int getSize() {
    return Galleon.size;
  }
}
```

A hierarquia de navios



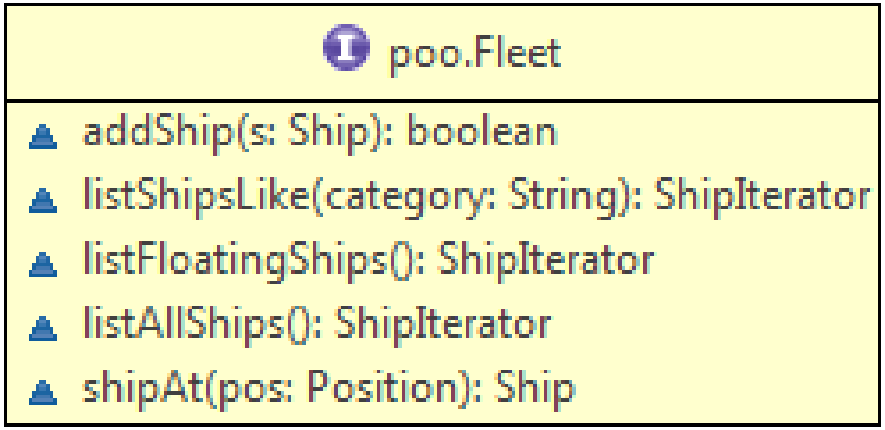
Frota de navios

Frota de navios (interface Fleet)

- **`boolean`** `addShip(Ship s)`
  - Tenta acrescentar o navio s à frota. A operação deve devolver true, em caso de sucesso, ou false, caso contrário.
    - O navio tem de caber dentro do mapa
    - O navio não estiver perto demais, ou mesmo sobreposto, a outros navios.
- `ShipIterator listShipsLike(String category)`
  - Cria e devolve um iterador de navios da categoria category existentes na frota.
- `ShipIterator listFloatingShips()`
  - Cria e devolve um iterador de navios existentes na frota que ainda não foram afundados.
- `ShipIterator listAllShips()`
  - Cria e devolve um iterador de navios existentes na frota.
- `Ship shipAt(Position pos)`
  - Devolve o navio que estiver a ocupar a posição pos, ou null, se não existir nenhum navio nessa posição.

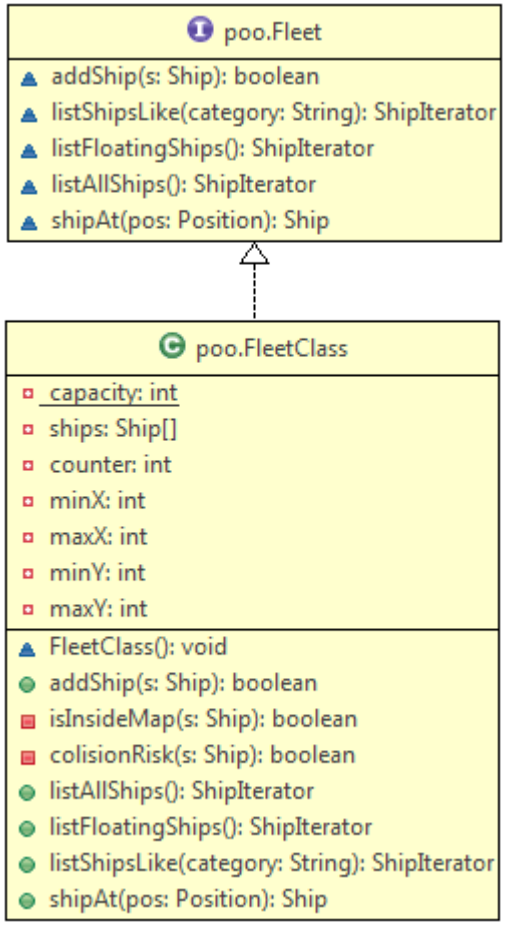
A interface `Fleet`

```
public interface Fleet {
  boolean addShip(Ship s);
  ShipIterator listShipsLike(String category);
  ShipIterator listFloatingShips();
  ShipIterator listAllShips();
  Ship shipAt(Position pos);
}
```



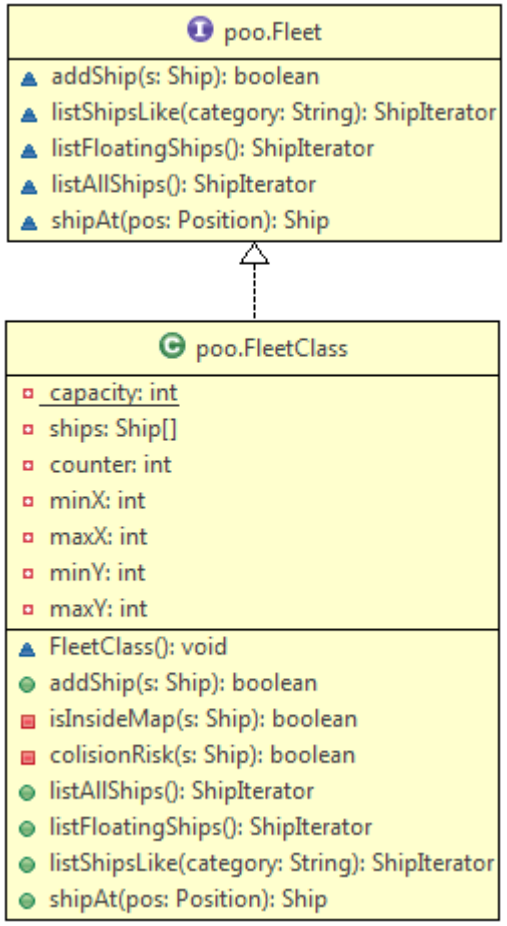
Classe FleetClass

```
public class FleetClass implements Fleet {
  private static int capacity = 11;
  private Ship[] ships;
  private int counter;

  private int minX;
  private int maxX;
  private int minY;
  private int maxY;

  FleetClass() {
    ships = new Ship[capacity];
    counter = 0;
    this.minX = 0;
    this.maxX = 9;
    this.minY = 0;
    this.maxY = 9;
  }
```



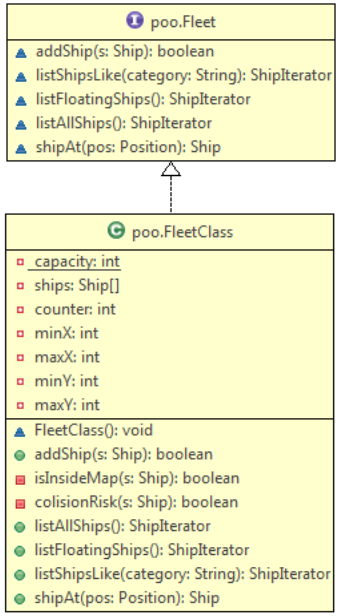
Classe FleetClass

```
public boolean addShip(Ship s) {
  boolean result = false;
  if (    (counter < capacity) && (isInsideMap(s))
       && (!colisionRisk(s))){
    ships[counter++] = s;
    result = true;
  }
  return result;
}

private boolean isInsideMap(Ship s) {
  return (    s.getLeftMostPos() >= minX
           && s.getRightMostPos() <= maxX
           && s.getTopMostPos() >= minY
           && s.getBottomMostPos() <= maxY);
}

private boolean colisionRisk(Ship s) {
  for (int i = 0; i < this.counter; i++)
    if (ships[i].tooCloseTo(s))
       return true;
  return false;
}
```



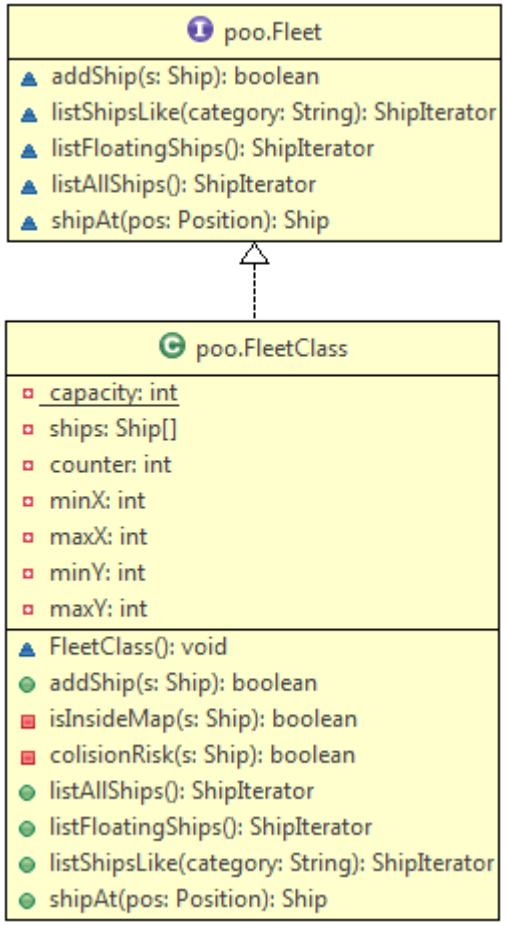
Classe FleetClass

```
public ShipIterator listAllShips() {
  return new ShipIteratorClass(ships, counter);
}

public ShipIterator listFloatingShips() {
  Ship[] floatingShips = new Ship[counter];
  int floatingCounter = 0;
  for (Ship s: ships)
    if (s.stillFloating())
      floatingShips[floatingCounter++] = s;
  return new ShipIteratorClass(floatingShips, floatingCounter);
}

public ShipIterator listShipsLike(String category) {
  Ship[] shipsLike = new Ship[counter];
  int likeCounter = 0;
  for (Ship s: ships)
    if (s.getCategory().equals(category))
      shipsLike[likeCounter++] = s;
  return new ShipIteratorClass(shipsLike, likeCounter);
}
```



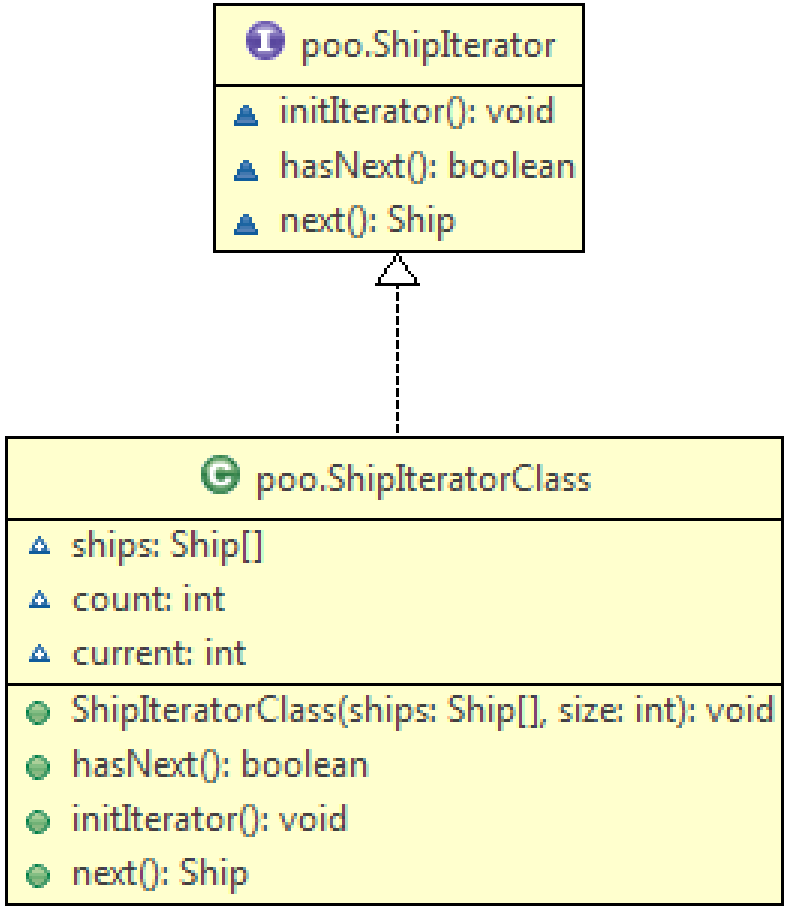
Classe `FleetClass`

```
  public Ship shipAt(Position pos) {
    for (int i = 0; i < counter; i++)
      if (ships[i].occupies(pos))
        return ships[i];
    return null;
  }
}
```



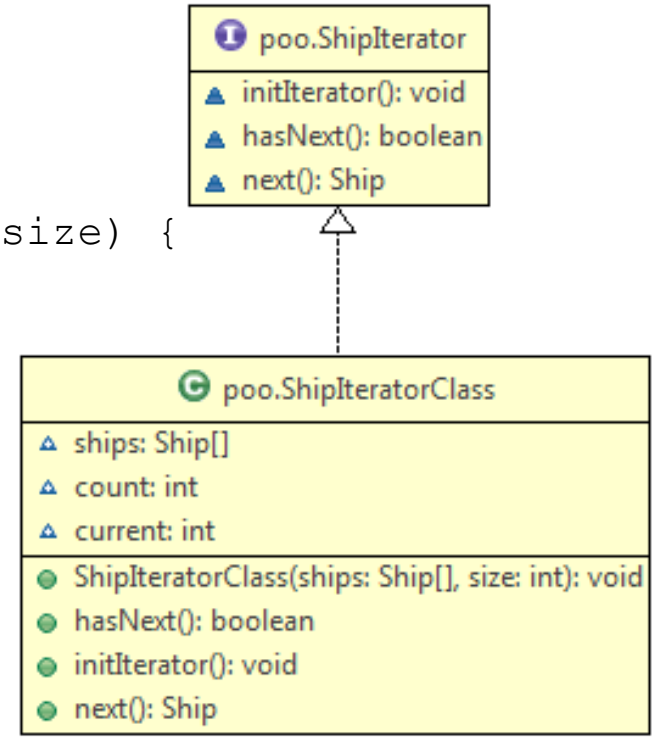
A interface `ShipIterator`


```
public interface ShipIterator {
  void initIterator();
  boolean hasNext();
  Ship next();
}
```



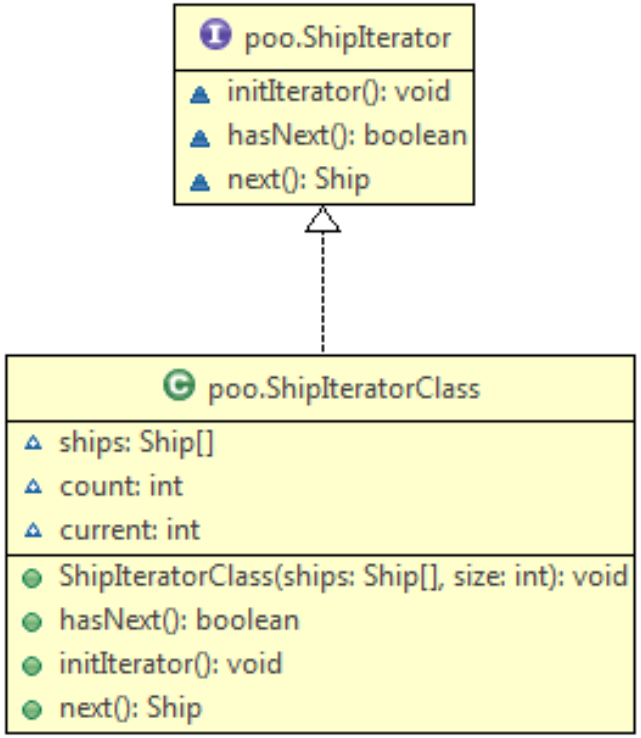
A classe ShipIteratorClass


```
public class ShipIteratorClass implements ShipIterator {
  Ship[] ships;
  int count;
  int current;

  public ShipIteratorClass(Ship[] ships, int size) {
    current = 0;
    this.ships = ships;
    if (ships != null)
      count = size;
    else
      count = 0;
  }
```

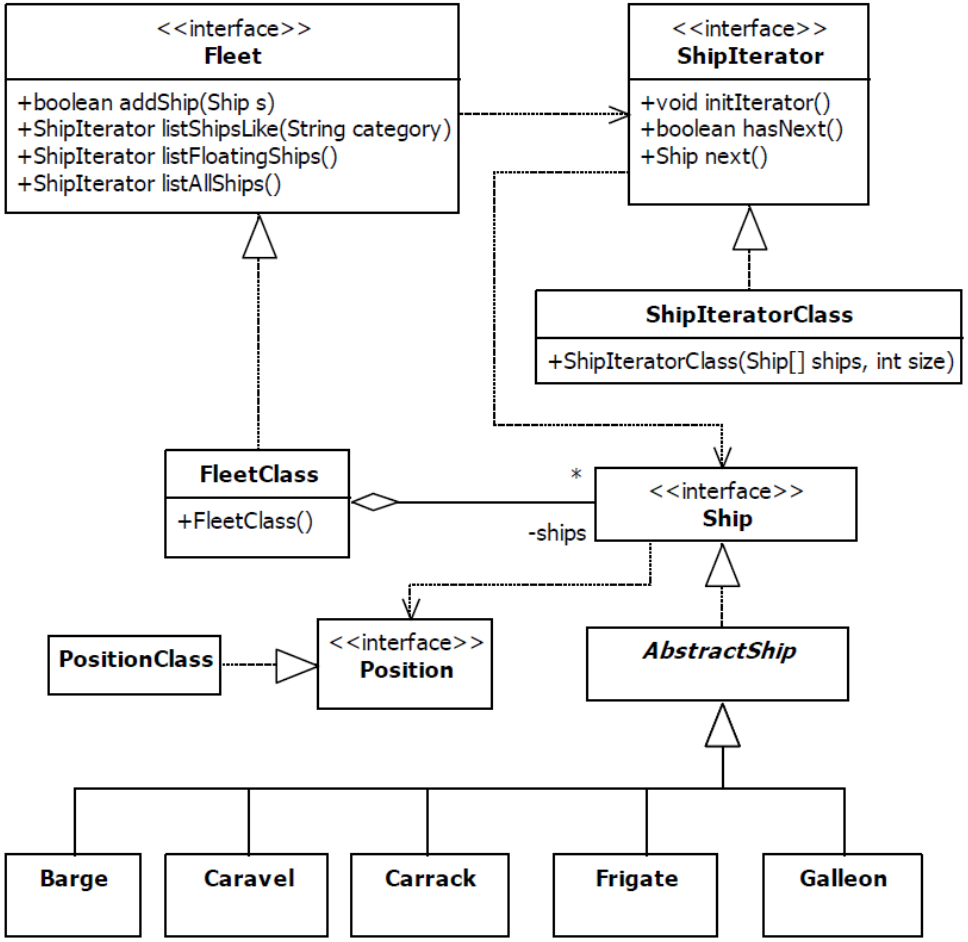


A classe ShipIteratorClass


```
  public void initIterator() {
    current = 0;
  }

  public boolean hasNext() {
    return current < count;
  }

  public Ship next() {
    Ship result = null;
    if (hasNext())
      result = ships[current++];
    return result;
  }
}
```



Exemplo completo da frota



A classe Game

A classe GameClass


```
public class GameClass implements Game {
  private Position[] shots;
  private int counter;
  private Fleet fleet;
  private int countInvalidShots;
  private int countRepeatedShots;
  private int countHits;
  private int countSinks;

  public GameClass(Fleet fleet) {
    shots = new Position[100];
    counter = 0;
    countInvalidShots = 0;
    countRepeatedShots = 0;
    this.fleet = fleet;
  }
```

A classe GameClass


```
public Ship fire(Position pos) {
  if (!validateShot(pos))
    countInvalidShots++;
  else { // valid shot!
    if (repeatedShot(pos))
      countRepeatedShots++;
    else {
      shots[counter++] = pos;
      Ship s = fleet.shipAt(pos);
      if (s != null) {
        s.shoot(pos);
        countHits++;
        if (!s.stillFloating()) {
          countSinks++;
          return s;
        }
      }
    }
  }
  return null;
}
```

A classe GameClass


```
private boolean validateShot(Position pos) {
  return (   pos.getRow() >= 0
          && pos.getRow() <= 9
          && pos.getColumn() >= 0
          && pos.getColumn() <= 9);
}

private boolean repeatedShot(Position pos) {
  for (int i = 0; i < counter; i++)
    if (shots[i].equals(pos))
      return true;
  return false;
}

public int getInvalidShots() {
  return this.countInvalidShots;
}
```

A classe GameClass


```
public int getRepeatedShots() {
  return this.countRepeatedShots;
}

public int getHits() {
  return this.countHits;
}

public int getRemainingShips() {
  ShipIterator it = fleet.listFloatingShips();
  it.initIterator();
  int result = 0;
  while (it.hasNext()) {
    result++;
    it.next();
  }
  return result;
}
```

A classe GameClass


```
  public int getSunkShips() {
    return this.countSinks;
  }

  public PositionIterator getShots() {
    return new PositionIteratorClass(shots, this.counter);
  }
}
```

A classe `PositionIteratorClass`

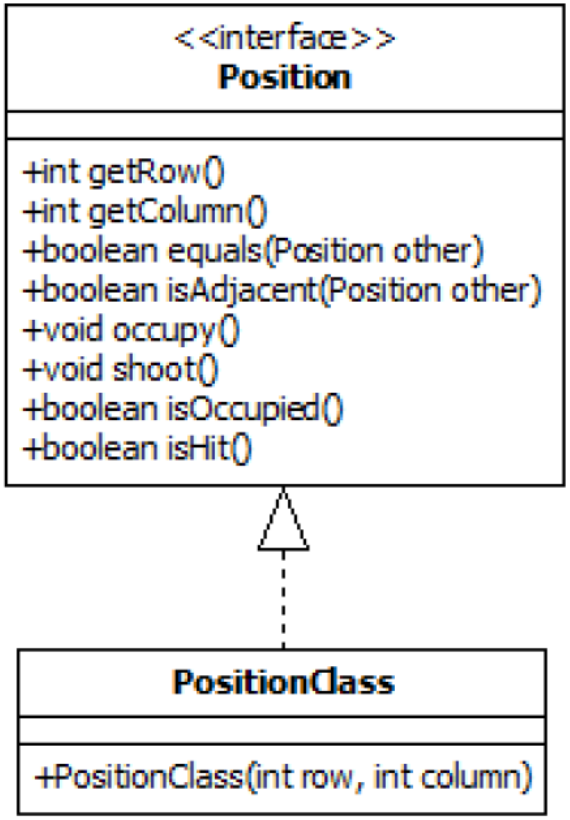
A classe PositionIteratorClass


```
public class PositionIteratorClass implements PositionIterator {
  private Position[] vector;
  private int counter;
  private int current;

  public PositionIteratorClass(Position[] vector, int counter) {
    this.vector = vector;
    this.counter = counter;
    this.current = -1;
  }

  public void init() {
    if (counter > 0)
      current = 0;
    else
      current = -1;
  }
```

A classe PositionIteratorClass


```
  public boolean hasNext() {
    return (current > -1 && current < counter);
  }

  public Position next() {
    if (hasNext())
      return vector[current++];
    else
      return null;
  }
}
```

A classe `PositionClass`



A classe PositionClass


```
public class PositionClass implements Position {
  private int row;
  private int column;
  private boolean isOccupied;
  private boolean isHit;

  public PositionClass (int row, int column) {
    this.row = row;
    this.column = column;
    this.isOccupied = false;
    this.isHit = false;
  }

  public int getColumn() {
    return column;
  }

  public int getRow() {
    return row;
  }
```

A classe PositionClass


```
public String toString() {
  return ("Linha = " + row + " Coluna = " + column);
}

// Assumimos estar apenas interessados na linha e coluna...
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (!(obj instanceof PositionClass))
    return false;
  PositionClass other = (PositionClass) obj;
  if (column != other.column)
    return false;
  if (row != other.row)
    return false;
  return true;
}
```

A classe PositionClass


```
  public boolean isAdjacentTo(Position other) {
    if (other == null)
      return false;

    return (   Math.abs(this.getRow() - other.getRow()) <= 1
            && Math.abs(this.getColumn() - other.getColumn()) <= 1);
  }

  public void occupy() { isOccupied = true; }

  public void shoot() { isHit = true; }

  public boolean isHit() { return isHit; }

  public boolean isOccupied() { return isOccupied; }
}
```

Vamos arrumar ideias...

Mecanismos de suporte à extensibilidade

- Herança
  - Torna as classes entidades extensíveis.
    - O código da superclasse é reutilizado (e reinterpretado) no contexto da subclasse, sem que tenha de ser reescrito.
    - Por vezes é preciso redefinir alguns dos métodos herdados e nesse caso escrevem-se novos métodos na subclasse (não se altera o código dos métodos originais).
- Subtipo (também conhecido como polimorfia de inclusão)
  - Permite a escrita de código abstracto, que funciona tão bem com as classes existentes como com novas classes a criar no futuro.
- Envio de mensagens
  - Incorpora uma análise de casos implícita, que funciona tão bem com as classes existentes como com novas classes a criar no futuro.
- Modularidade
  - Nem todo o código dum programa pode ser abstracto. O código que define o comportamento duma entidade concreta deve ser colocado, todo ele, dentro da classe concreta respectiva.

Algumas heurísticas

- Use correctamente os mecanismos de herança, sub-tipo, envio de mensagens e modularidade
- Procure a extensibilidade
  - Evite testar explicitamente a classe concreta dos objectos
  - Descubra os conceitos abstractos no seu problema
- Apenas o código das classes concretas deve ser concreto
- Cada classe preocupa-se com as suas tarefas, deixando as tarefas das outras classes para as outras classes!

Resumo: O que podemos fazer numa sub-classe?

- Uma sub-classe herda todos os membros públicos e protegidos da sua super-classe, independentemente de estarem, ou não, declaradas no mesmo package.
  - Quando uma sub-classe está no mesmo package que a super-classe, também herda os membros acessíveis às classes do package
  - Podemos usar os membros herdados tal como estão
  - Podemos substituir os membros herdados
  - Podemos esconder os membros herdados
  - Podemos acrescentar novos membros

Resumo: O que podemos fazer numa sub-classe? Membros de dados (variáveis e constantes)

- Os membros de dados herdados podem ser usados directamente, tal como quaisquer outros membros de dados definidos na sub-classe
- É possível declarar um campo na sub-classe com o mesmo nome de um campo na super-classe, escondendo assim o campo da super-classe
  - Isto é possível, de facto, mas não é recomendado!
- É possível declarar novos membros de dados na sub-classe, que não estejam na super-classe

Resumo: O que podemos fazer numa sub-classe? Métodos e construtores

- Os métodos herdados podem ser usados directamente, tal como estão
- Pode-se escrever um novo método de instância numa sub-classe com a mesma assinatura de um método da super-classe, reimplementando assim o método da super-classe na sub-classe
- Pode-se escrever um novo método de classe na sub-classe, com a mesma assinatura de um método de classe da super-classe, escondendo assim o método estático da super-classe
- Podem-se declarar novos métodos na sub-classe que ainda existiam na super-classe.
- Pode-se escrever um construtor na sub-classe que invoque, explicita ou implicitamente, o construtor da super-classe
  - A invocação explícita é feita com a palavra reservada **`super`**

O que significa ser sub-tipo

- Os sub-tipos devem satisfazer o princípio da substituição
  - Regra da assinatura
    - As assinaturas dos têm de ser compatíveis e os sub-tipos têm de disponibilizar todos os métodos dos super-tipos
  - Regra dos métodos
    - As chamadas aos métodos dos sub-tipos têm de provocar um comportamento semelhante ao esperado nos super-tipos
  - Regra das propriedades
    - Os sub-tipos têm de preservar todas as propriedades dos super-tipos