DI FCT UNL

PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Exemplo de consolidação

Piratas das Caparicas

Piratas das Caparicas

- Desenvolva um programa em Java que implemente o jogo "*Piratas 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.
- 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

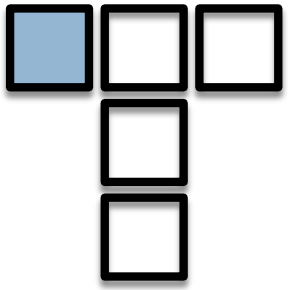
- Na sequência do comando **`nova`**:
  - Sub-comandos para criar os 11 navios:
    - Tipo de navio Linha Coluna Orientação
      - Tipo de navio:
        - **`barca, caravela, nau, fragata, galeao`**
      - Linha – coordenada vertical mais a norte do navio
      - Coluna – coordenada horizontal mais a oeste 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
```

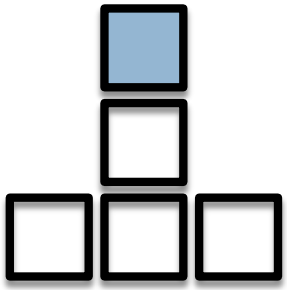
Posicionamento e orientação dos navios

Galeão:

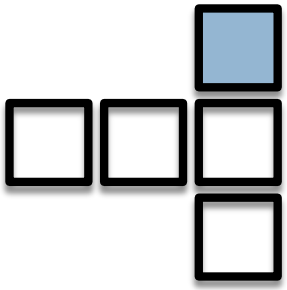
Norte



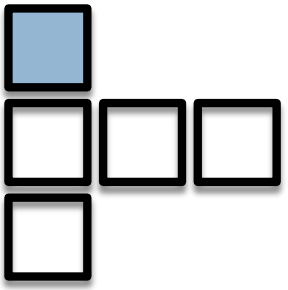
Sul



Este



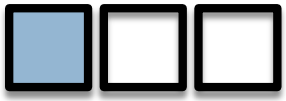
Oeste



Nau:

Norte ou Sul

Este ou Oeste



Posição de referência: linha mais a norte, e nesta, coluna mais a oeste

Operações suportadas pelo jogo

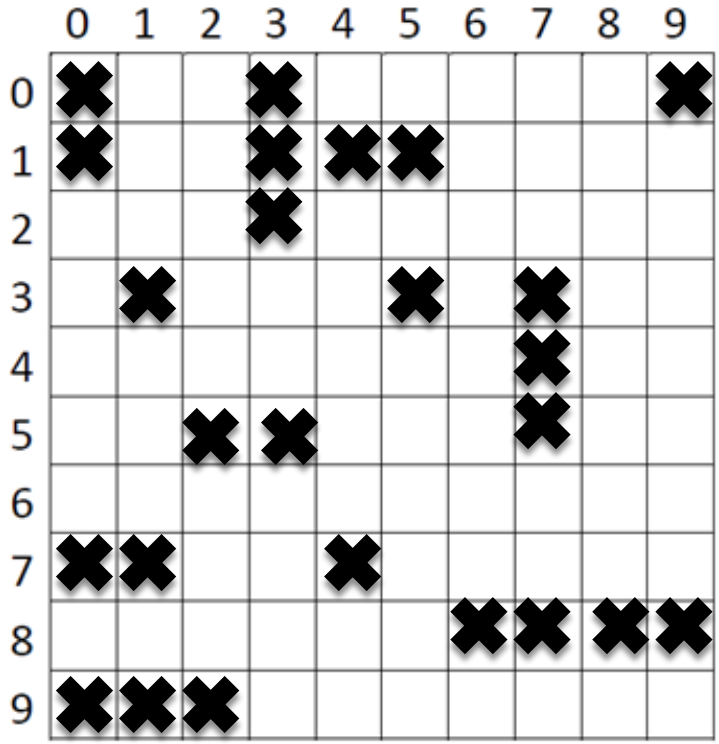
- **`mapa`**
  - Escreve na consola um mapa com a localização dos navios, usando caracteres diferentes para representar se as posições estão ou não ocupadas por algum navio
    - Nos exemplos de interacção, usaremos o carácter '#' para sinalizar as posições ocupadas por algum navio, e o carácter '.' 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

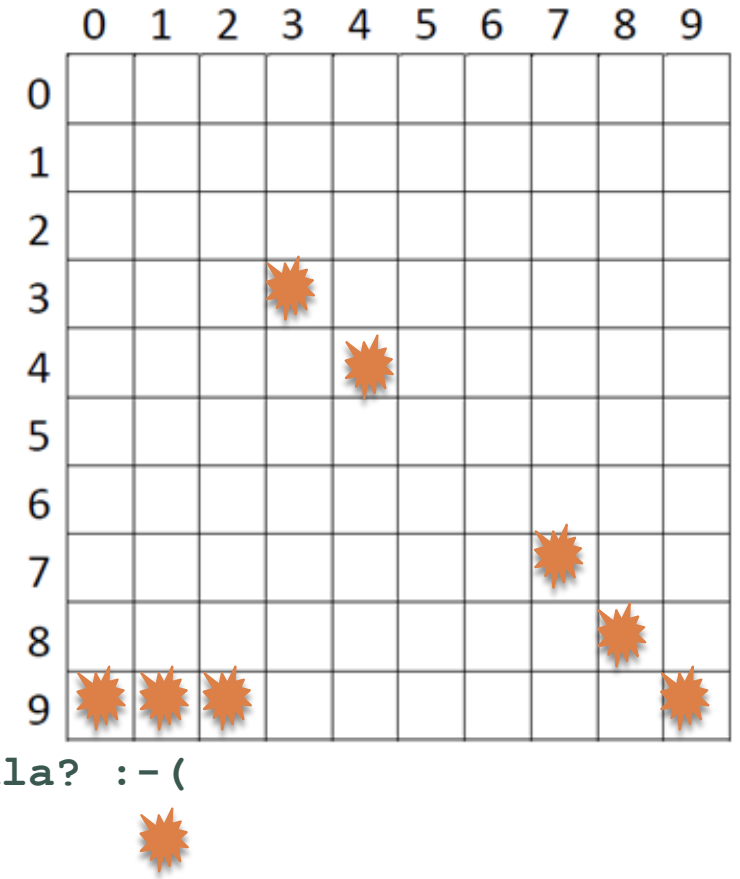
Exemplo de interacção

```
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 jogo através de uma interface bem definida
  - Ou seja, temos:
    - Uma classe com o interpretador de comandos e programa principal (tipicamente chamada **`Main`**)
    - 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

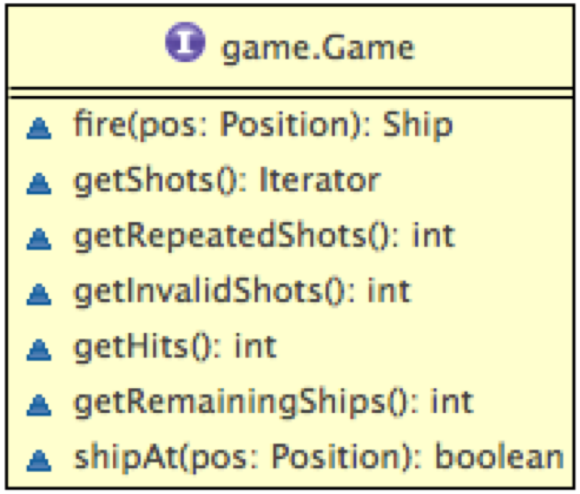
- `Ship fire(Position pos)`
  - Dispara para a posição **`pos`**. Caso o tiro resulte no afundamento de um navio, esse navio é devolvido como resultado da operação. Caso contrário, a operação devolve **`null`**.
- `Iterator<Position> getShots()`
  - Devolve um iterador com as posições atingidas por tiros. 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 getRemainingShips()`
  - Devolve o número de navios por afundar.

A interface Game

```
public interface Game {
   Ship fire(Position pos);
   Iterator<Position> getShots();
   int getRepeatedShots();
   int getInvalidShots();
   int getHits();
   int getRemainingShips();
}
```



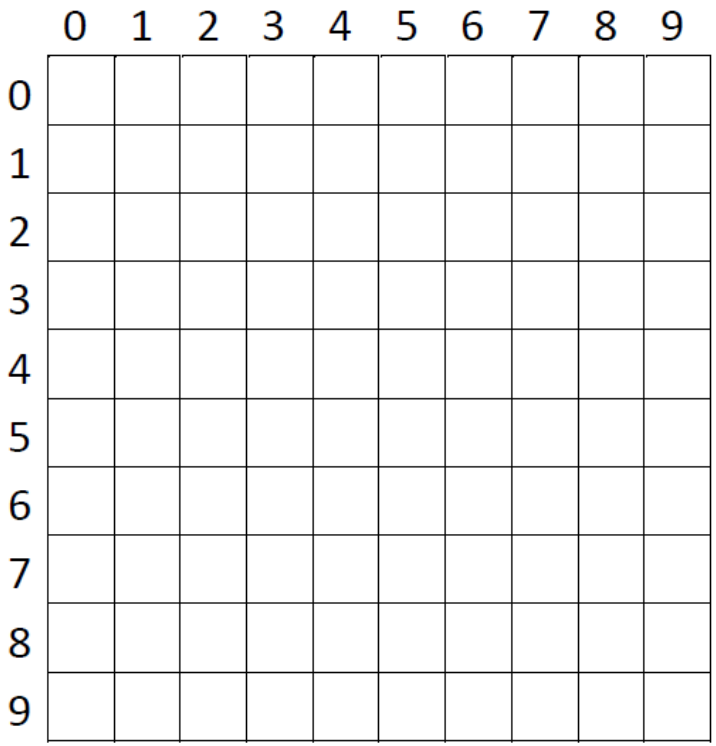
- Temos de definir
  - o iterador
  - a interface Ship
  - a interface Position
- Porque não temos acesso à frota?
  - Normalmente, quem ataca não pode ter acesso à localização da frota
- Mas como é que a frota aparece, então?
  - Ao construir um objecto Game (neste caso: uma instância de GameClass que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

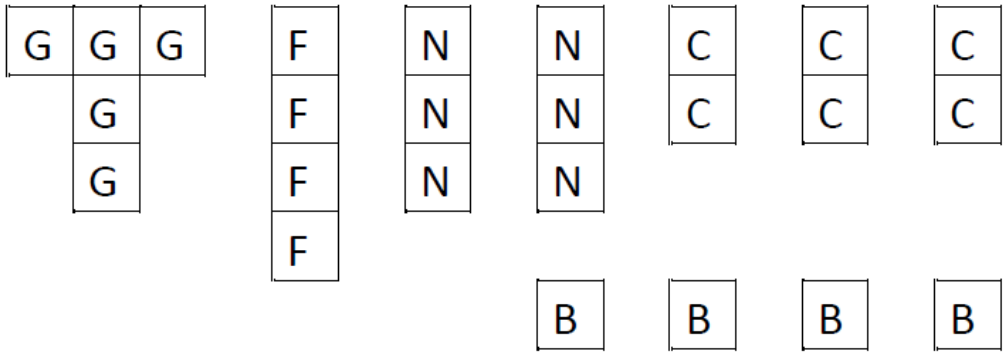
A interface `Ship`

A interface `Ship`

- Esta interface deve poder representar **qualquer um** dos navios da frota
  - Actuais
  - 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`

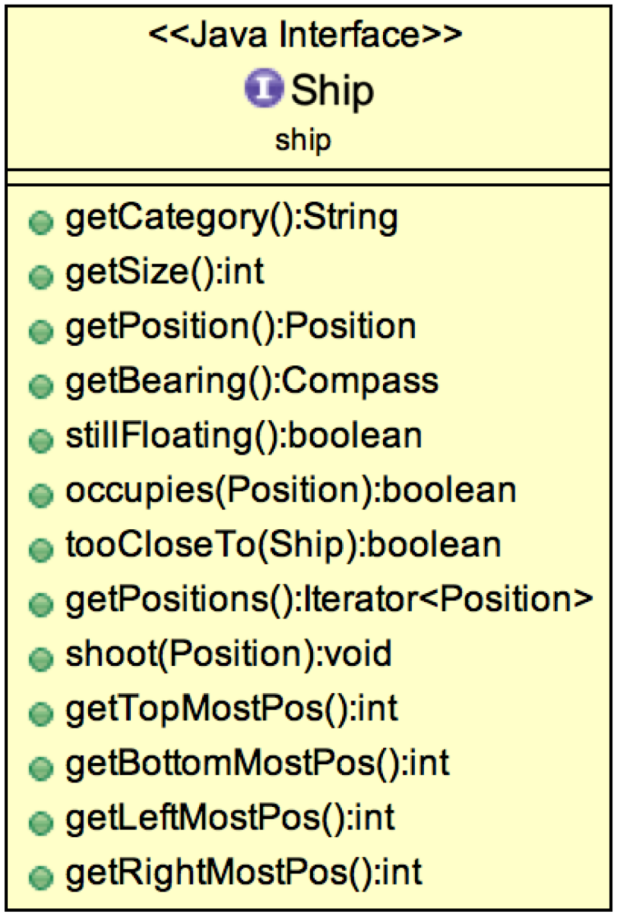
- **`boolean`** `stillFloating()`
  - Operação que devolve **`true`** se o navio estiver a flutuar, ou **`false`** se o navio já foi afundado.
- **`int`** `getTopMostPos()`
  - Ordenada mais a norte ocupada pelo navio
- **`int`** `getBottomMostPos()`
  - Ordenada mais a sul ocupada pelo navio
- **`int`** `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.
    PRE: **`other != this`**
- `Iterator<Position> getPositions()`
  - Devolve um iterador sobre as posições ocupadas pelo navio
- **`void`** `shoot(Position pos)`
  - dispara sobre a posição **`pos`**.

A interface `Ship`

```
package poo;

public interface Ship {
  String getCategory();
  int getSize();
  Position getPosition();
  Compass getBearing();
  boolean stillFloating();
  boolean occupies(Position pos);
  boolean tooCloseTo(Ship other);
  Iterator<Position> getPositions();
  void shoot(Position pos);
  int getTopMostPos();
  int getBottomMostPos();
  int getLeftMostPos();
  int getRightMostPos();
}
```



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`

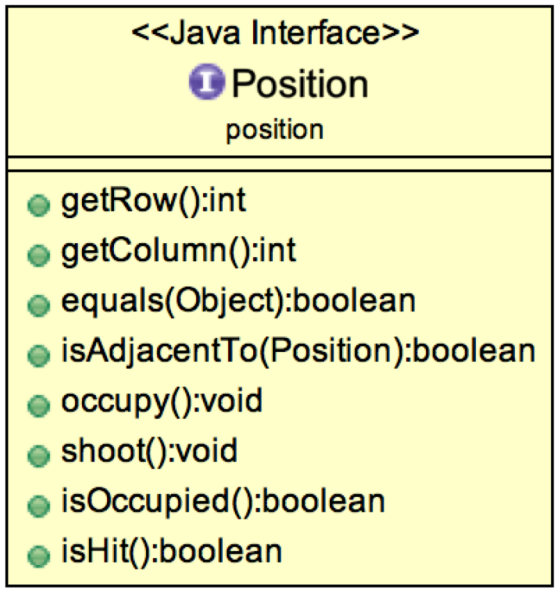
- `int getRow()`
  - Devolve a linha da posição
- `int getColumn()`
  - Devolve a coluna da posição
- `boolean equals(Object other)`
  - Devolve `true` se as coordenadas forem semelhantes, `false` caso contrário
- `boolean isAdjacentTo(Position other)`
  - Devolve `true` se a posição `other` for adjacente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

```
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 `Iterator`

A interface `Iterator<E>`

- **`void`** `init()`
  - Vai para o início da colecção de objectos
- **`boolean`** `hasNext()`
  - Devolve **`true`** se existirem mais objectos a visitar, ou **`false`**, caso contrário
- `E next()`
  - Devolve o próximo objecto

    PRE: `hasNext()`

O enumerado Compass

O enumerado Compass

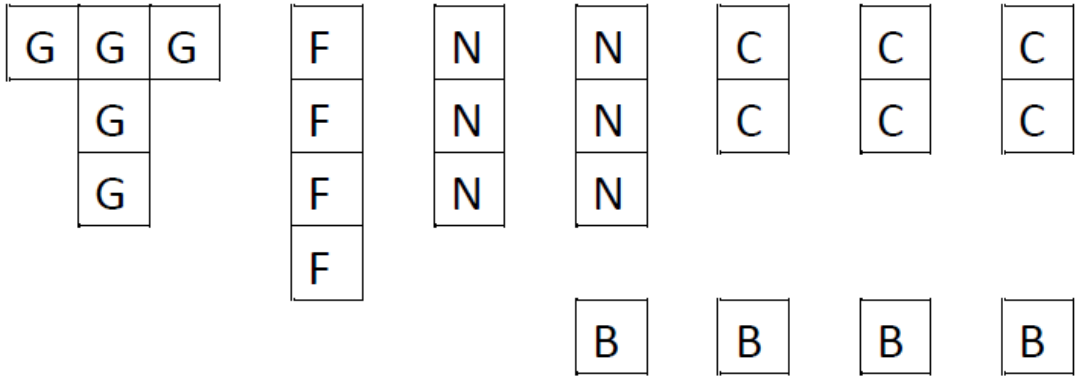
```
public enum Compass {
  NORTH ('n'),
  SOUTH ('s'),
  EAST ('e'),
  WEST ('o');

  private final char c;

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

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

Implementações

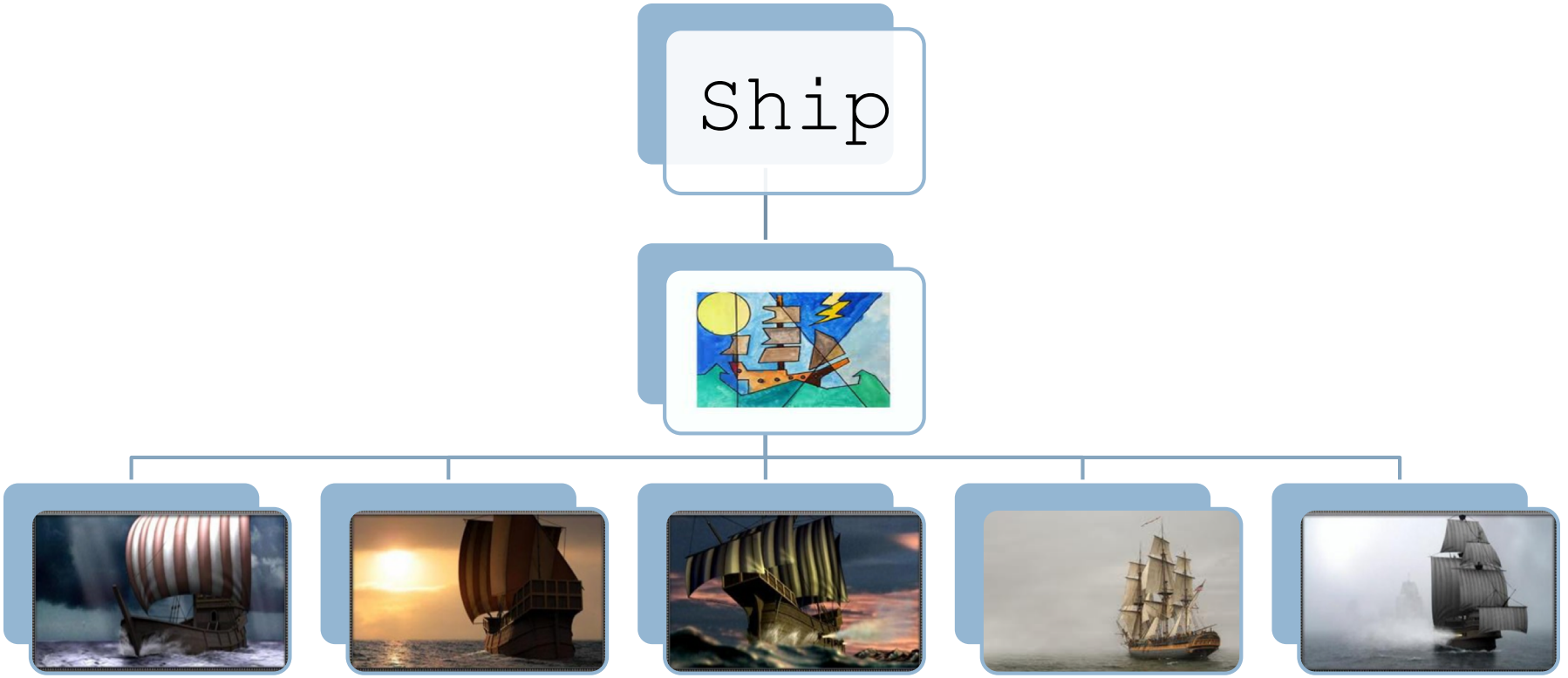
Implementação de `Ship`

Comecemos pelas "docas"

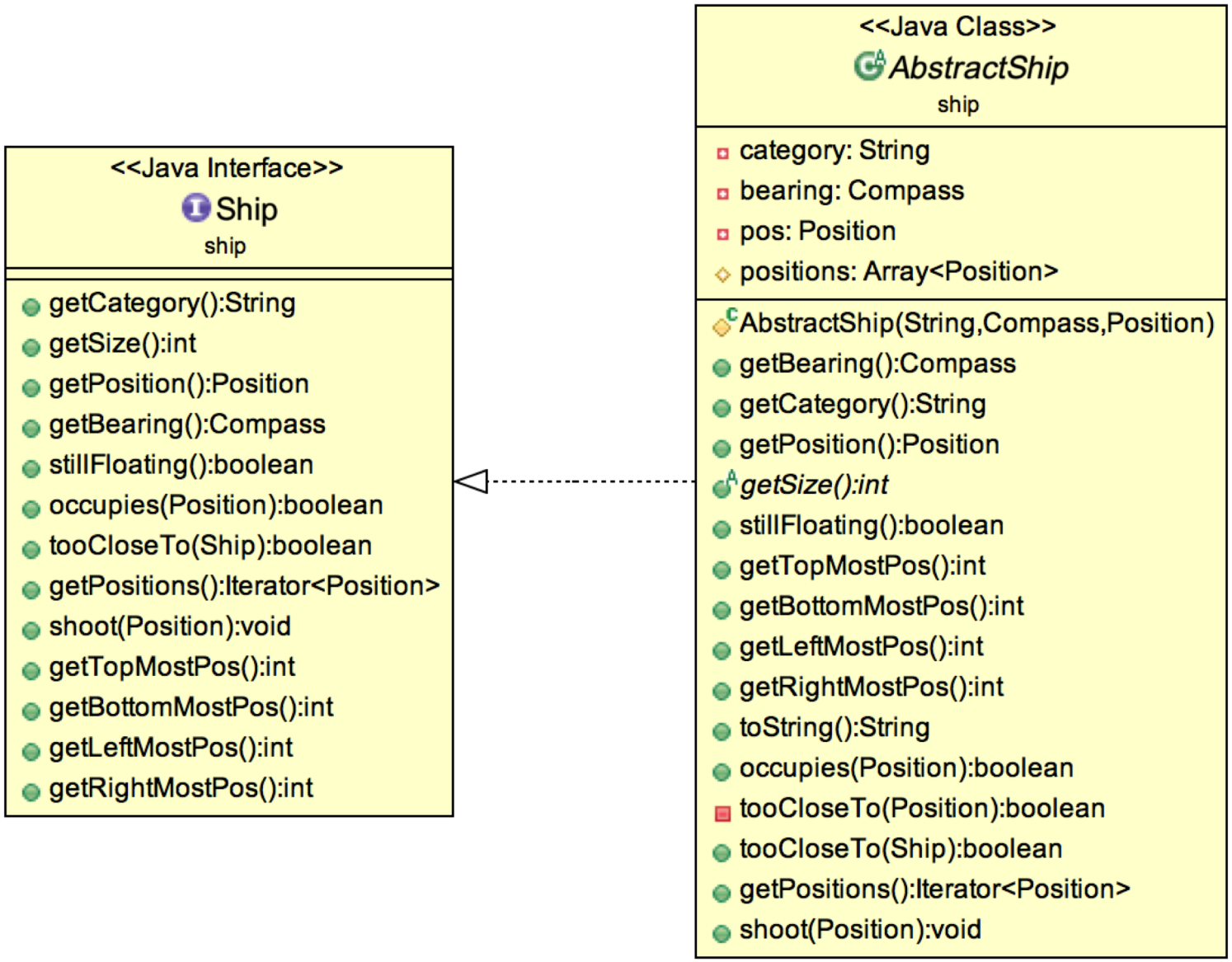
- Temos de representar os vários navios
- Um navio tem:
  - Dimensão (e formato)
  - Posição
  - Orientação
  - Pode ainda estar a flutuar, ou não



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 Array<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() {
   Iterator<Position> it = positions.iterator();
      while (it.hasNext())
         if (!it.next().isHit())
            return true;
      return false;
}

public int getTopMostPos() {
   Iterator<Position> it = positions.iterator();
   int top = it.next().getRow();
   while (it.hasNext()) {
      Position pos = it.next();
      if (pos.getRow() < top)
         top = pos.getRow();
   }
   return top;
}
```

Classe abstracta `AbstractShip`

```
public int getBottomMostPos() {
    Iterator<Position> it = positions.iterator();
    int bottom = it.next().getRow();
    while (it.hasNext()) {
        Position pos = it.next();
        if (pos.getRow() > bottom)
            bottom = pos.getRow();
    }
    return bottom;
}

public int getLeftMostPos() {
    Iterator<Position> it = positions.iterator();
    int left = it.next().getColumn();
    while (it.hasNext()) {
        Position pos = it.next();
        if (pos.getColumn() < left)
            left = pos.getColumn();
    }
    return left;
}

public int getRightMostPos() { ... }
```

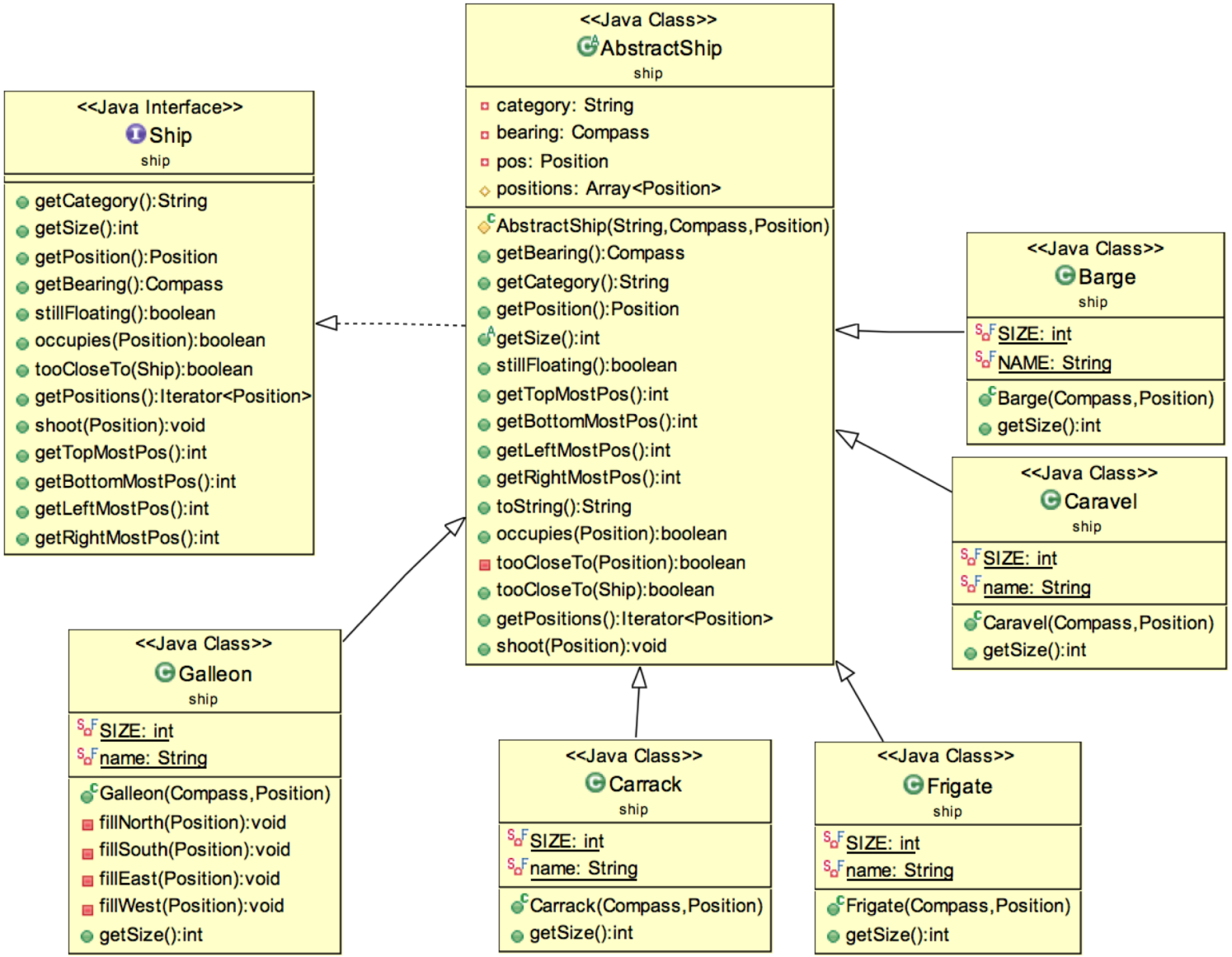
Classe abstracta `AbstractShip`

```
public boolean occupies(Position pos) {
    Iterator<Position> it = positions.iterator();
    while (it.hasNext())
        if (it.next().equals(pos))
            return true;
    return false;
}

private boolean tooCloseTo(Position pos) {
    Iterator<Position> it = positions.iterator();
    while (it.hasNext())
        if (it.next().isAdjacentTo(pos))
            return true;
    return false;
}
```

Classe abstracta `AbstractShip`

```
public boolean tooCloseTo(Ship other) {
   Iterator<Position> otherPos = other.getPositions();
   while (otherPos.hasNext())
      if (tooCloseTo(otherPos.next()))
         return true;
   return false;
}

public Iterator<Position> getPositions() {
   return positions.iterator();
}

public void shoot(Position pos) {
   Iterator<Position> it = positions.iterator();
   while (it.hasNext()) {
      Position p = it.next();
         if (p.equals(pos))
            p.shoot();
   }
}
```

Hierarquia de navios

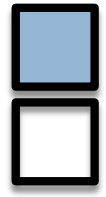


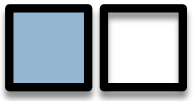
Classe Barge (Barca)

```
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 ArrayClass<Position>(SIZE);
     positions.insertLast(new PositionClass(pos.getRow(),
                                            pos.getColumn()));
  }

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

Classe `Caravel` (Caravela)

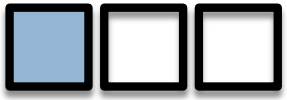
```
public class Caravel extends AbstractShip {
  private static final int SIZE = 2;
  private static final String NAME = "Caravela";

  public Caravel(Compass bearing, Position pos) {
     super(Caravel.name, bearing, pos);
     positions = new ArrayClass<Position>(SIZE);

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

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

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





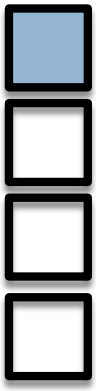
A classe Carrack (Nau)

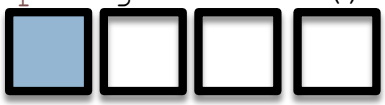
```
public class Carrack extends AbstractShip {
  private static final int SIZE = 3;
  private static final String NAME = "Nau";

  public Carrack(Compass bearing, Position pos) {
    super(NAME , bearing, pos);
    positions = new ArrayClass<Position>(SIZE);

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

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

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



A classe `Frigate` (Fragata)

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

  public Carrack(Compass bearing, Position pos) {
    super(Carrack.name, bearing, pos);
    positions = new ArrayClass<Position>(SIZE);

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

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

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





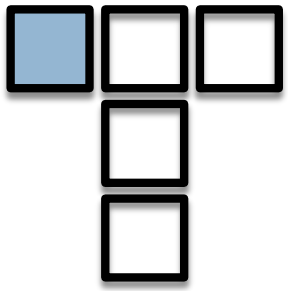
A classe `Galleon` (Galeão)

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

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

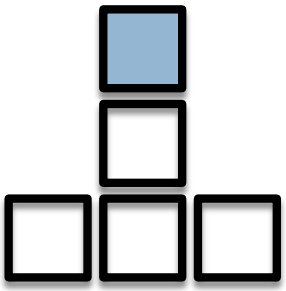
A classe `Galleon` (Galeão)

```
private void fillNorth(Position pos) {
  for (int i = 0; i < 3; i++)
    positions.insertLast(new PositionClass(pos.getRow(), pos.getColumn()+i));
  positions.insertLast(new PositionClass(pos.getRow()+1, pos.getColumn()+1));
  positions.insertLast(new PositionClass(pos.getRow()+2, pos.getColumn()+1));
}

private void fillSouth(Position pos) {
  for (int i = 0; i < 2; i++)
    positions.insertLast(new PositionClass(pos.getRow()+i, pos.getColumn()));
  for (int j = 2; j < 5; j++)
    positions.insertLast(new PositionClass(pos.getRow()+2, pos.getColumn()+j-3));
}
```

Norte



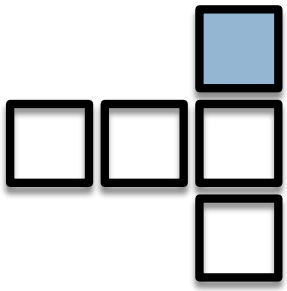
Sul



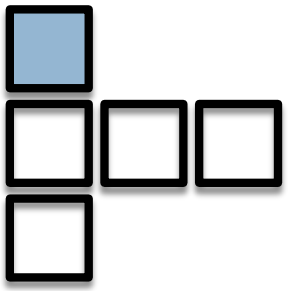
A classe `Galleon` (Galeão)

```
private void fillEast(Position pos) {
  positions.insertLast(new PositionClass(pos.getRow(), pos.getColumn()));
  for (int i = 1; i < 4; i++)
    positions.insertLast(new PositionClass(pos.getRow()+1, pos.getColumn()+i-3));
  positions.insertLast(new PositionClass(pos.getRow()+2, pos.getColumn()));
}

private void fillWest(Position pos) {
  positions.insertLast(new PositionClass(pos.getRow(), pos.getColumn()));
  for (int i = 1; i < 4; i++)
    positions.insertLast(new PositionClass(pos.getRow()+1, pos.getColumn()+i-1));
  positions.insertLast(new PositionClass(pos.getRow() + 2, pos.getColumn()));
}

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

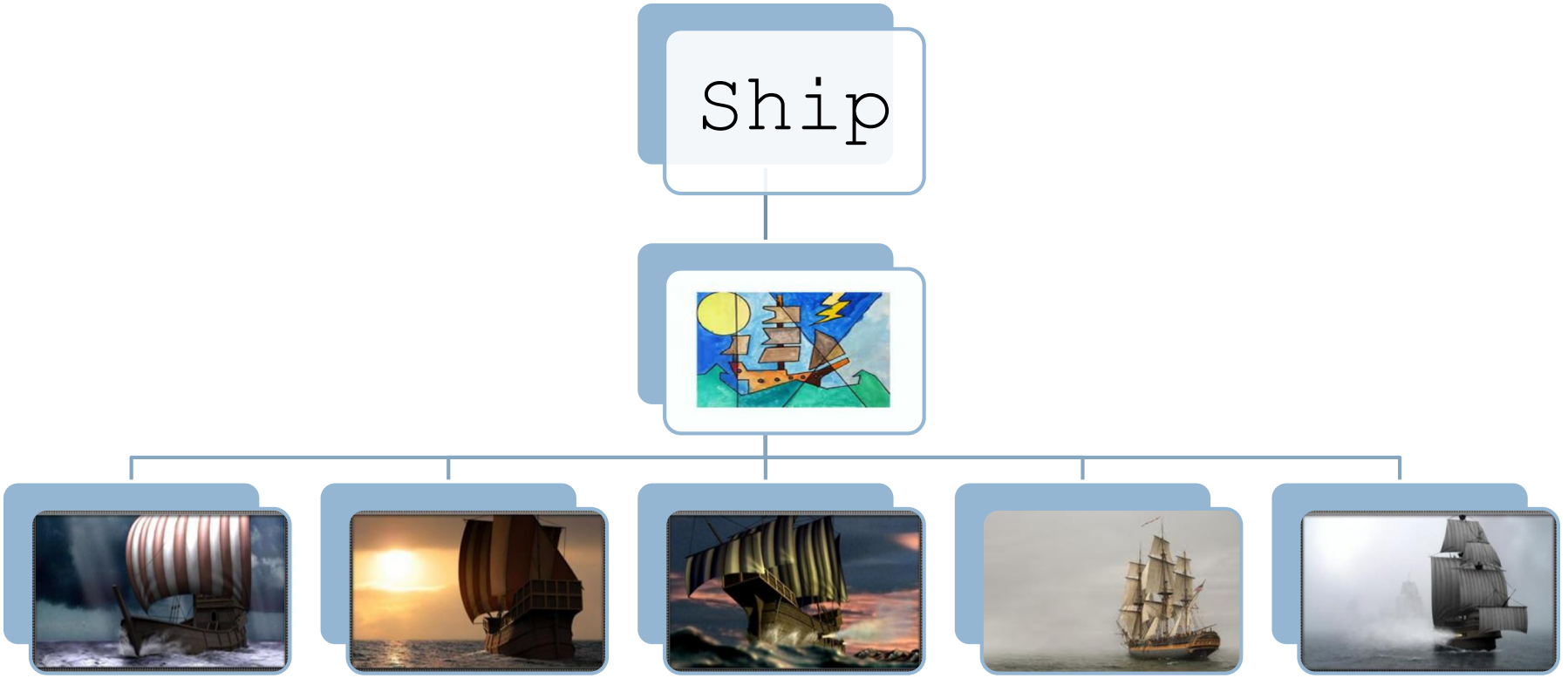
Este



Oeste



A hierarquia de navios



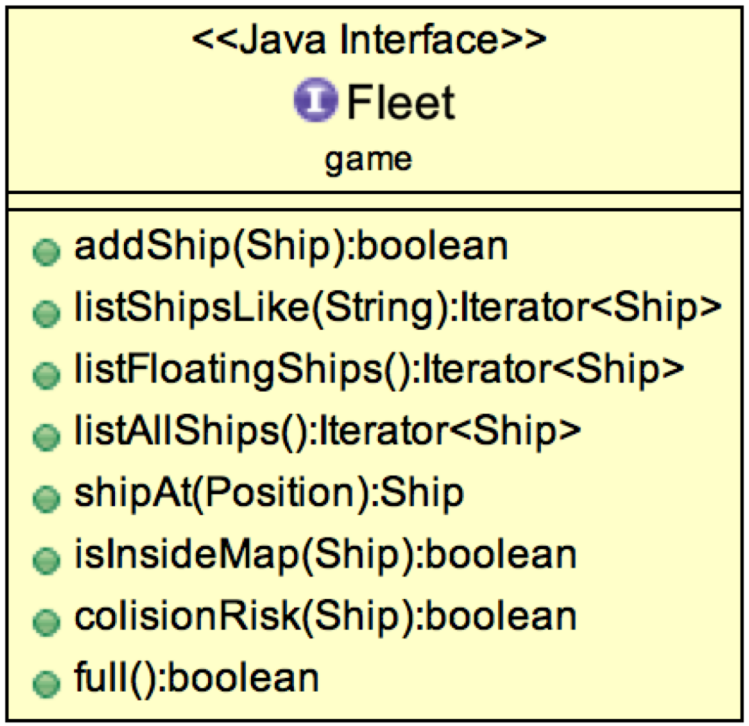
Frota de navios

Frota de navios (interface Fleet)

- `boolean addShip(Ship s)`
  - Acrescenta o navio s à frota
- `Iterator<Ship> listShipsLike(String category)`
  - Cria e devolve um iterador de navios da categoria category existentes na frota.
- `Iterator<Ship> listFloatingShips()`
  - Cria e devolve um iterador de navios existentes na frota que ainda não foram afundados.
- `Iterator<Ship> 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.
- `boolean insideMap(Ship s)`
  - Devolve true caso o navio s estiver dentro do mapa, ou false, caso contrário.
- `boolean colisionRisk(Ship s)`
  - Devolve true caso o navio s estiver adjacente a outro navio do mapa, ou false, caso contrário.
- `boolean full()`
  - Devolve true caso a frota esteja completa (11 navios), ou false, caso contrário.

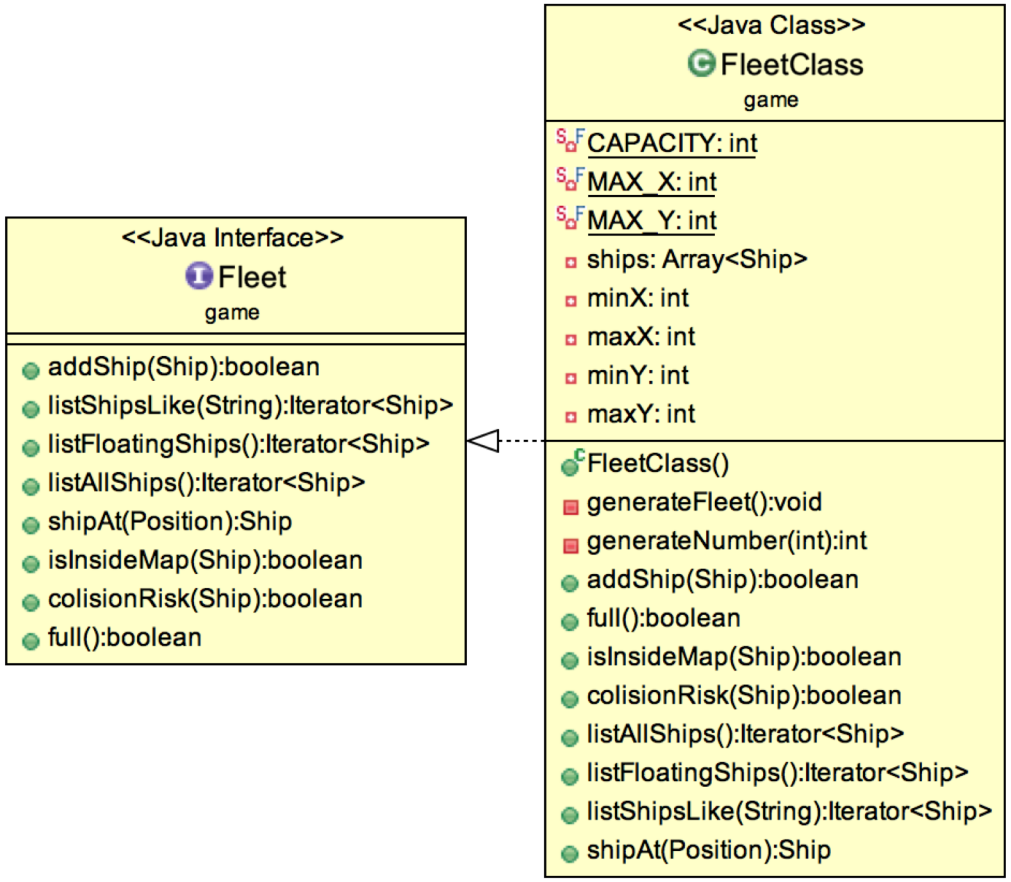
A interface `Fleet`

```
public interface Fleet {
  boolean addShip(Ship s);
  Iterator<Ship> listShipsLike(String category);
  Iterator<Ship> listFloatingShips();
  Iterator<Ship> listAllShips();
  Ship shipAt(Position pos);
  boolean isInsideMap(Ship s);
  boolean colisionRisk(Ship s);
  boolean full();
}
```



Classe FleetClass

```
public class FleetClass implements Fleet {
  private static final int CAPACITY = 11;
  private static final int MAX_X = 9;
  private static final int MAX_Y = 9;

  private int minX;
  private int maxX;
  private int minY;
  private int maxY;
  private Array<Ship> ships;


  public FleetClass() {
     ships = new ArrayClass<Ship>(CAPAC
     this.minX = 0;
     this.maxX = MAX_X;
     this.minY = 0;
     this.maxY = MAX_Y;
  }
```



Classe FleetClass


```
public boolean addShip(Ship s) {
   boolean result = false;
   if (!full() && isInsideMap(s) && !colisionRisk(s)) {
      ships.insertLast(s);
      result = true;
   }
   return result;
}


public boolean full() {
   return ships.size() == CAPACITY;
}

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

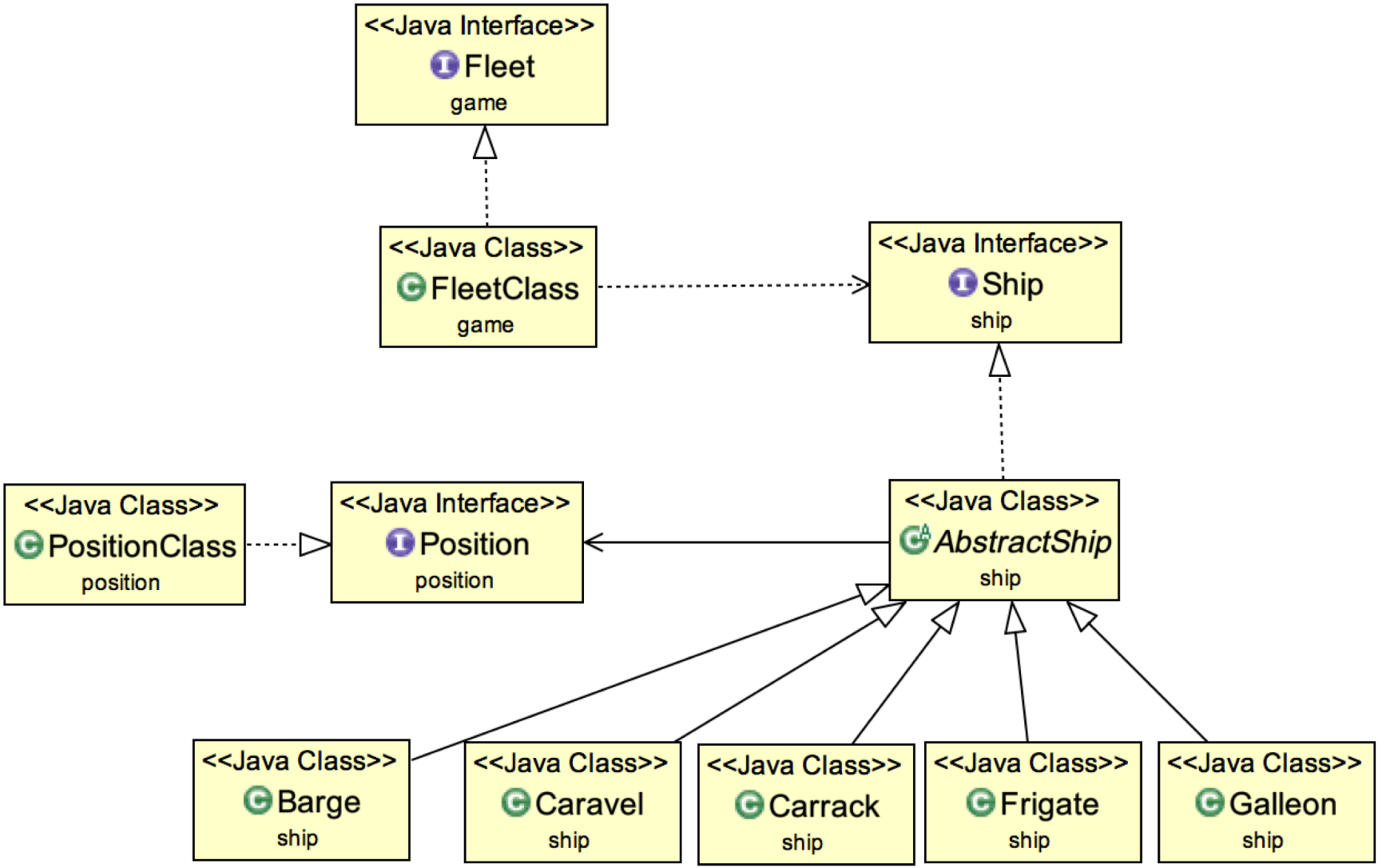
Classe `FleetClass`


```
public boolean colisionRisk(Ship s) {
   Iterator<Ship> it = ships.iterator();
   while (it.hasNext())
      if (it.next().tooCloseTo(s))
         return true;
   return false;
}

public Iterator<Ship> listAllShips() {
   return ships.iterator();
}

public Iterator<Ship> listFloatingShips() {
   return new FloatingShipIteratorClass(ships);
}

public Iterator<Ship> listShipsLike(String category) {
   return new CategoryShipIteratorClass(ships, category);
}
```

Classe `FleetClass`


```
public Ship shipAt(Position pos) {
   Iterator<Ship> it = ships.iterator();
   while (it.hasNext()) {
      Ship ship = it.next();
         if (ship.occupies(pos))
            return ship;
   }
   return null;
}
```

Exemplo completo da frota



A classe `GameClass`

A classe GameClass

```
public class GameClass implements Game {
  private static final int MAX = 100;

  private Array<Position> shots;
  private Fleet fleet;
  private int countInvalidShots;
  private int countRepeatedShots;
  private int countHits;
  private int countSinks;

  public GameClass(Fleet fleet) {
     shots = new ArrayClass<Position>(MAX);
     countInvalidShots = 0;
     countRepeatedShots = 0;
     this.fleet = fleet;
}
```

```
public GameClass() {
  shots = new ArrayClass<Position>(MAX);
  countInvalidShots = 0;
  countRepeatedShots = 0;
  fleet = generateFleet();
}
```

A classe GameClass

```
public Ship fire(Position pos) {
  if (!validateShot(pos))
     countInvalidShots++;
  else { // valid shot!
     if (repeatedShot(pos))
        countRepeatedShots++;
     else {
        shots.insertLast(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() >= minX
             && pos.getRow() <= maxX
             && pos.getColumn() >= minY
             && pos.getColumn() <= maxY);
}

private boolean repeatedShot(Position pos) {
   Iterator<Position> it = shots.iterator();
   while (it.hasNext())
      if (it.next().equals(pos))
         return true;
   return false;
}

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

A classe GameClass

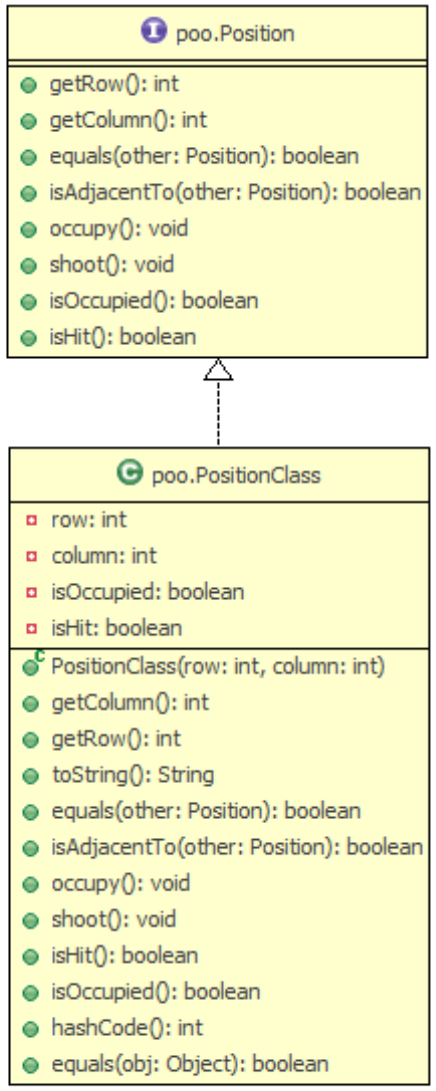
```
public int getHits() {
    return this.countHits;
}

public int getRemainingShips() {
    Iterator<Ship> it = fleet.listFloatingShips();
    int result = 0;
    while (it.hasNext()) {
        result++;
        it.next();
    }
    return result;
}
```

A classe GameClass

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

public Iterator<Position> getShots() {
    return shots.iterator();
}

public boolean shipAt(Position pos) {
    return fleet.shipAt(pos) != null;
}
```

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 = " + getRow() + " Coluna = " + getColumn());
}

// 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; }
}
```