Batalha Naval

Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Grupo T3G3:
José Bateira - 201000575
Pedro Cunha - 200505567

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

December 2012

Resumo

Este trabalho consiste na elaboração, documentação e teste de um modelo formal executável usando uma variante de VDM (Vienna Development Tools), o VDM++.

Como tema para o projecto foi escolhido o jogo clássico "Batalha Naval". São demonstradas métricas, procedimentos e informações de testes ao longo do relatório, que servem para ilustrar que este modelo de desenvolvimento de código, apesar de não estar imune a erros (é impossível um teste exaustivo de todas as situações), o número destes é substancialmente reduzido face a um desenvolvimento de código considerado "normal", substanciado pela abrangência dos testes (100%).

Índice

1 Introdução

Na ciência da computação e engenharia de software, métodos formais são técnicas baseadas em formalismos matemáticos para a especificação, desenvolvimento e verificação dos sistemas de software e hardware. O seu uso para o desenvolvimento de software e hardware é motivado pela expectativa de que, como em outras disciplinas de engenharia, podem contribuir para a confiabilidade e robustez de um projeto executando análises matemáticas apropriadas. Entretanto, o alto custo do uso de métodos formais significa que eles são geralmente apenas usados no desenvolvimento de sistemas de alta-integridade, no qual há alta probabilidade das falhas conduzirem para a perda da vida ou sério prejuízo (como o caso do Ariane 5 que tentou meter um número de 64-bits em 16-bits). Neste caso, o objectivo era usar estes métodos para construir o jogo Batalha Naval sem falhas do género do Ariane 5, confirmando estes métodos com testes, abrangendo todas as linhas de código.

2 Requisitos e Principais Restrições

Requisitos

1. Cada jogador tem dois tabuleiros 10x10;
2. Cada jogador tem 10 barcos;
3. Só dá para 2 jogadores;
4. Os jogadores não sabem o tabuleiro um do outro;
5. Cada jogador dispara um tiro por turno;
6. Cada jogador alvo reporta se foi atingido um navio ou não;
7. O jogador activo marca num tabuleiro diferente o resultado do tiro com um marcador diferente;
8. O jogador cujo barco foi atingido marca no seu tabuleiro que a casa onde está o barco foi de facto atingida;
9. Existem quatro barcos com tamanho dois, três barcos com tamanho três, dois barcos de tamanho quatro e um barco de tamanho cinco;
10. Um barco é considerado como ter afundado se as casas que ocupa tiverem sido atingidas;
11. Um barco não pode ocupar a mesma casa que outro barco;
12. Um barco não pode ser colocado na diagonal;
13. Ganha o jogador que primeiro afundar todos os barcos do adversário.

Principais Restrições

1. O número de casas num tabuleiro não pode ser maior que 100 (10*10);
2. Todas as coordenadas (Casas e Barcos) têm de estar dentro do tabuleiro;
3. Uma casa ocupada por um barco não pode ser ocupada por outro barco;
4. Um jogador só ganha se todos os barcos do adversário estiverem afundados;
5. Só é possivel atingir uma casa que não tenha sido ainda atingida;
6. O número de barcos de um jogador não é maior que 10.

3 Especificação em VDM++ das Restrições

Restrições em VDM++

1. ```
   card(houses) = BOARD_SIZE * BOARD_SIZE;
   ```
2. ```
   coords(X) >= 1 and coords(X) <= Board‘BOARD_SIZE and
   coords(Y) >= 1 and coords(Y) <= Board‘BOARD_SIZE;
   ```
3. ```
   forall c in set coords &
       let h in set board.houses be st h.x = c(1) and h.y = c(2) in
       (
          not h.hasShip
       );
   ```
4. ```
   forall s in set elems ships & s.isDown();
   ```
5. ```
   if not h.hasShip or h.hit then return MISS;
   ```
6. ```
   (len ships) <= len sizes;
   ```

4 Diagrama Conceptual em UML

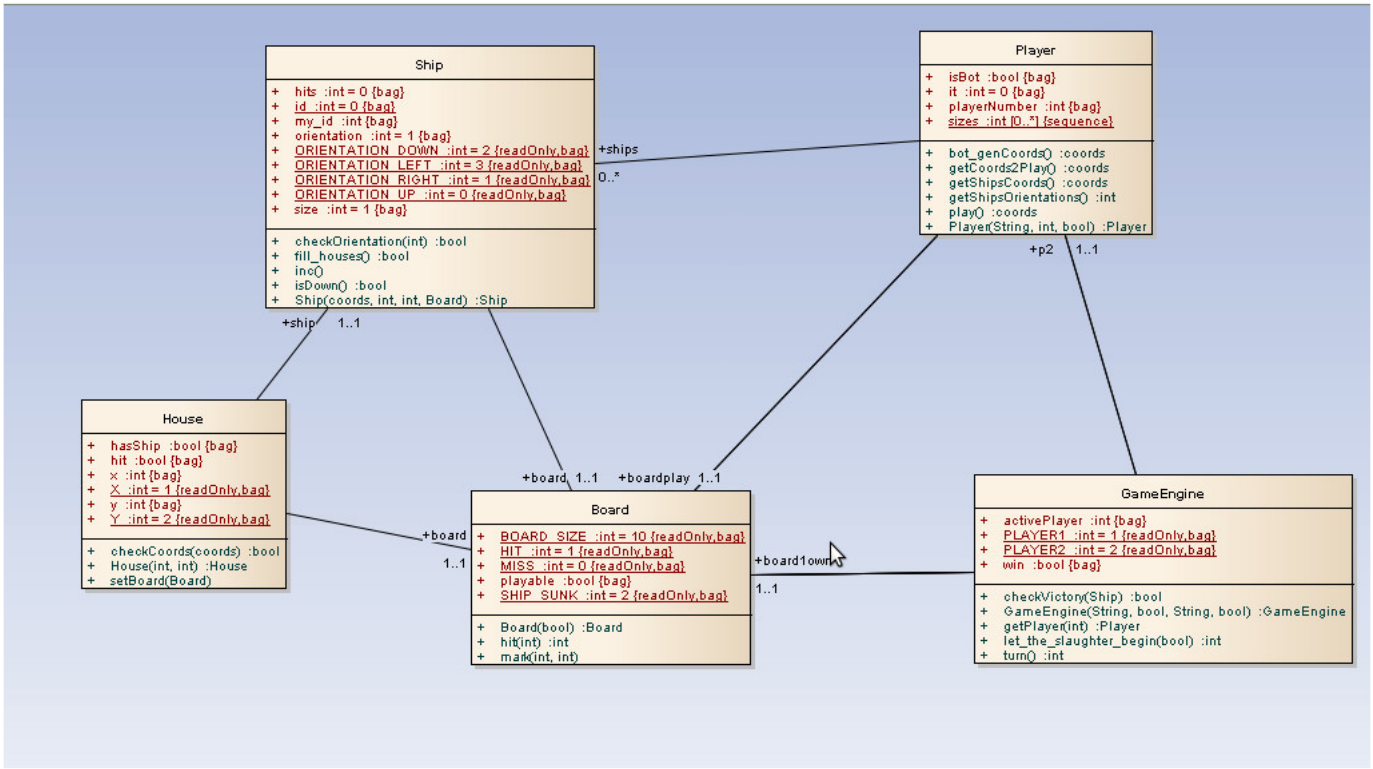

Figura 1: Diagrama Conceptual em UML

5 Classes e script de teste

Classes

1. GameEngine

   ```
   GameEngine : VDMUtils`String * bool * VDMUtils`String * bool ==> GameEngine
   ```

2. Board

   ```
   Board: bool ==> Board
   ```

3. Player

   ```
   Player : VDMUtils`String * int * bool ==> Player
   ```

4. House

   ```
   House : int * int ==> House
   ```

5. Ship

   ```
   Ship: House`coords * int * int * Board ==> Ship
   ```

6. CLI - Command Line Interface

Visto que o projecto foi desenvolvido no Overture, não foi desenvolvido um script de teste.

6 Matriz de Rastreabilidade dos testes com os requisitos

Requesito — Teste	testBoardHousesNumber	testCheckCoords	testNoOverlapedShips	testFullGame1	testFullGame2
1				X	X
2	X	X	X	X	X
3	X	X	X	X	X
4				X	X
5				X	X
6				X	X
7				X	X
8				X	X
9		X	X	X	X
10				X	X
11			X	X	X
12		X	X	X	X
13				X	X

7 Descrição das Classes

7.1 GameEngine

7.1.1 Descrição

Esta classe é o ponto de entrada no jogo. É a que cria os jogadores, atribui tabuleiros, gere turnos e verifica as condições de vitória.

7.1.2 Invariantes

- `inv activePlayer in set {PLAYER1, PLAYER2};`

7.1.3 Operações

1. GameEngine. Recebe os nomes dos jogadores e dois booleanos que usa referir se são jogadores computador ou não.

   ```
   GameEngine : VDMUtils‘String * bool * VDMUtils‘String * bool ==> GameEngine
   ```

2. Turn. Não tem argumentos e devolve um inteiro correspondente ao próximo jogador a jogar. Retorna 0 se algum dos jogadores ganhar. Responsável pela chamada da função da condiçao de vitória e pelas chamadas de tiro e marcação nos tabuleiros.

   ```
   public turn : () ==> int
   ```

3. CheckVictory. Recebe uma sequência de barcos e verifica se estão todos afundados. Se estiverem retorna true, senão retorna false.

   ```
   public checkVictory: seq of Ship ==> bool
   ```

4. getPlayer. Recebe um inteiro que é o id do jogador e retorna uma referência para o próprio jogador.

   ```
   public getPlayer : int ==> Player
   ```

7.2 Board

7.2.1 Descrição

Esta classe é a que representa um tabuleiro do jogo. Cada tabuleiro cria as suas próprias casas e tem funções para gerir tiros e marcações de tiros.

7.2.2 Invariantes

-

7.2.3 Pré Condições

- `pre House‘checkCoords(coords)`

7.2.4 Operações

1. Board. Recebe um booleano que diz se é o tabuleiro onde o jogador coloca peças ou se é o de marcação.

   ```
   Board: bool ==> Board
   ```

2. Hit. Responsável por marcar uma casa do tabuleiro como tendo sido atingida e de verificar se tem algum barco e, tendo, de o marcar também. Recebe uma sequência de inteiros (coordenadas) e devolve um inteiro consoante o resutlado.

   ```
   public hit : seq of int ==> int
   ```

3. Mark. Consoante o resultado do Hit, vai marcar ou não uma casa noutro tabuleiro como atingida e se tiver barco, idem. Recebe uma coordenada e um inteiro e não devolve nada.

   ```
   public mark : seq of int * int ==> ()
   ```

7.2.5 Pós Condições

- `post card(houses) = BOARD_SIZE * BOARD_SIZE;`
- `post RESULT in set {MISS, HIT, SHIP_SUNK};`

7.3 Player

7.3.1 Descrição

Classe que trata um jogador, desde gerar barcos e guardá-los como gerar tabuleiros para cada jogador a enviar tiros e gerar coordenadas para esses tiros.

7.3.2 Invariantes

- `inv (len ships) <= len sizes;`

7.3.3 Pré Condições

- `pre it <= 100`

7.3.4 Operações

1. Player. Recebe uma string com o seu nome, um inteiro com o seu id e um booleano para saber se é um "bot"ou não. Cria tabuleiros e barcos e coloca os barcos.

   ```
   public Player : VDMUtils`String * int * bool ==> Player
   ```

2. getShipCoords. Retorna uma sequência de coordenadas para os barcos, lidas de um ficheiro.

   ```
   public getShipsCoords: ()  ==> seq of House`coords
   ```

3. getShipOrientions. Retorna as orientações de cada barco, lidas de um ficheiro, de 1 a 4 seguindo os ponteiros do relógio e sendo 1 = direita

```
public getShipsOrientations: () ==> seq of int
```

4. getCoords2Play. Retorna sequência de coordenadas para jogar, lidas de um ficheiro.

```
public getCoords2Play: ()  ==> seq of House‘coords
```

5. Play. Retorna coordenadas da casa alvo.

```
public getCoords2Play: ()  ==> seq of House‘coords
```

6. bot_genCoords. Retorna coordenas da casa alvo geradas sequencialmente caso sejam "bots"

```
public bot_genCoords : () ==> House‘coords
```

7.3.5 Pós Condições

- `post forall orientation in set elems RESULT & Ship‘checkOrientation(orientation);`
- `post forall coord in set elems RESULT & House‘checkCoords(coord);`
- `post House‘checkCoords(RESULT);`

7.4 House

7.4.1 Descrição

Classe que gere toda a informação relevante de uma casa, se foi atingida, se tem barco, a que tabuleiro pertence, se tiver barco qual e o que é uma coordenada (sequencia de inteiros).

7.4.2 Invariantes

- `inv checkCoords([x] ^ [y]);`

7.4.3 Pré Condições

- `pre checkCoords([x1] ^ [y1]);`
- `pre is_Board(b);`

7.4.4 Operações

1. House. Construtora com uma coordenada.

```
public House : int * int ==> House
```

2. setBoard. Referencia a Casa a um tabuleiro.

```
public setBoard : Board ==> ()
```

7.4.5 Funções

1. CheckCoords. Função que verifica se as coordenadas estão dentro do tabuleiro. Retorna true ou false consoante estejam ou não.

   ```
   public static checkCoords : coords -> bool
   ```

7.5 Ship

7.5.1 Descrição

Classe que representa um barco. Tem a informação das orientações (consoante a orientação, as posições que o barco ocupa a partir da posição inicial variam), sabe a que tabuleiro percence e quais são as suas coordenadas.

7.5.2 Invariantes

- `inv checkOrientation(orientation);`
- `inv len coord_init = 2;`
- `inv id >= 0;`
- `inv card(coords) >= 0 and card(coords) <= size;`

7.5.3 Pré Condições

- `pre forall x in set {c(1),c(1)+orientations(o)(1)*(s-1)}, y in set {c(2),c(2)+orienta`
  `House‘checkCoords([x] ^ [y])`
- ```
  pre forall c in set coords &
  let h in set board.houses be st h.x = c(1) and h.y = c(2) in
  (
  not h.hasShip
  )
  ```
- `pre hits < size`

7.5.4 Operações

1. Ship. Cria-se a si próprio com uma coordenada inicial, uma orientação, um tamanho e um tabuleiro.

   ```
   public Ship: House‘coords * int * int * Board ==> Ship
   ```

2. Fill_Houses. Função responsável atribuir a todas as casas com as coordenadas do barco o barco em si.

   ```
   public fill_houses: ()==> bool
   ```

3. Inc. Incrementa o número de tiros de que o barco já foi alvo.

   ```
   public inc: () ==> ()
   ```

4. IsDown. Verifica se o barco foi ao fundo comparando o tamanho com o número de tiros.

   ```
   public isDown : () ==> bool
   ```
   ```

7.5.5 Pós Condições

- `post fill_houses();`
- `post hits <= size;`

7.5.6 Funções

1. CheckOrientation. Verifica se uma dada orientação é válida (ou seja, está entre 1 e 4).

   ```
   public static checkOrientation : int -> bool
   ```

7.6 CLI

7.6.1 Descrição

Classe responsável pela interface gráfica. Lê de ficheiros alguns códigos ASCII e recebe informações sobre o que escrever.

8 Informação Sobre Cobertura dos Teste

8.1 GameEngine

Function or operation	Coverage	Calls
GameEngine	100.0%	3
checkVictory	100.0%	398
getPlayer	100.0%	10
let_the_slaughter_begin	100.0%	2
turn	100.0%	398
GameEngine.vdmpp	100.0%	811

8.2 Board

Function or operation	Coverage	Calls
Board	100.0%	10
hit	100.0%	398
mark	100.0%	398
Board.vdmpp	100.0%	806

8.3 Player

Function or operation	Coverage	Calls
Player	95.4%	5
bot_genCoords	100.0%	199
getCoords2Play	0.0%	0
getShipsCoords	100.0%	2
getShipsOrientations	100.0%	2
play	66.6%	199
Player.vdmpp	86.9%	407

8.4 House

Function or operation	Coverage	Calls
House	100.0%	1000
checkCoords	100.0%	5198
setBoard	100.0%	1000
House.vdmpp	100.0%	7198

8.5 Ship

Function or operation	Coverage	Calls
Ship	100.0%	40
checkOrientation	100.0%	168
fill_houses	100.0%	40
inc	100.0%	118
isDown	100.0%	1250
Ship.vdmpp	100.0%	1616

9 Conclusão

Apesar de não terem sido registados tempos e de não ter sido gerado código noutra linguagem, o código em VDM++ funciona na perfeição e requeriu (à parte da curva de aprendizagem da linguagem) menos esforço do que o normal na especificação dos métodos e regras do jogo. Quando se começou a escrever código já era sabida a estrutura geral do programa (que foi sendo refinada ao longo do tempo). De facto confirmou-se que se deram muitos menos erros seguindo uma metodologia de TDD (Test-driven development).

Ficou realmente demonstrado pelos que o uso de pré e pós condições, aliados às invariantes, tornam o código muito mais seguro contra erros humanos e de aleatórios.