

**Mastermind**

Relatório Final

**Métodos Formais em Engenharia de Software**

1º semestre, 4º ano do Mestrado Integrado em Engenharia

Informática e da Computação

14 de dezembro de 2015

Elementos do Grupo:

Pedro José Leal de Sousa - [ei12179@fe.up.pt](mailto:ei12179@fe.up.pt) - 201205016

Mário Filipe Araújo Ferreira - [ei12049@fe.up.pt](mailto:ei12049@fe.up.pt) - 201203906

Luís Alberto Moreira Pinto – ei12108@fe.up.pt – 201204982

Introdução

O presente trabalho foi realizado no âmbito da unidade curricular de Métodos Formais em Engenharia de Software do 4º ano do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto.

Neste relatório é descrito um modelo formal, construído usando a linguagem VDM++, que implementa as regras do jogo de tabuleiro mundialmente famoso Matermind. O intuito deste projeto é o de desenvolver, testar e documentar um modelo formal executável de um sistema de *software* de alta integridade usando a linguagem acima referida e a ferramenta Overture. No final do trabalho é pretendido que tenha sido criada nos alunos a capacidade de modelar formalmente sistemas de software em VDM++, e de demonstrar a consistência do mesmo.

A estrutura deste relatório pode ser descrita da seguinte forma:

- Primeiramente serão descritas as regras do jogo Mastermind e enumerados os requisitos e restrições que foram identificados como necessários para a implementação deste modelo. Esta enumeração será acompanhada de uma descrição da implementação em VDM++ das restrições e requisitos referidos anteriormente.

- Será retratada a forma de escrever e ler dados gravados em disco e enunciada a forma como foi implementada na ferramenta Overture.

- Estará representado o modelo UML do projeto, sendo este constituído por um modelo de casos de uso e diagramas de classes acompanhados de breves descrições dos mesmos.

- De seguida será transcrito todo o código do trabalho, estando o mesmo devidamente dividido e comentado.

- Todos os testes serão descritos e será demonstrada a sua implementação em VDM++. Iremos também completar uma tabela que relata cada teste e identifica o requisito com o qual se relaciona.

- Por fim será descrito o processo de geração de código Java a partir da ferramenta Overture e serão retiradas conclusões de todo o processo de desenvolvimento do trabalho em questão.

Descrição das regras do Jogo

Mastermind

Mastermind é um jogo para dois jogadores aonde o principal objetivo é que o “descodificador” decifre o código de quatro peças gerado pelo outro jogador.

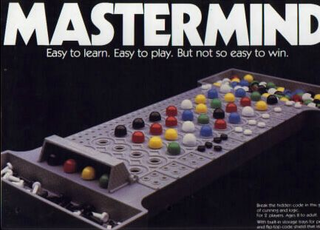


Fig1: Representação do tabuleiro de jogo original

O jogo decorre num tabuleiro retangular com uma capa numa das extremidades do mesmo que esconde quatro orifícios e várias linhas (doze, dez, oito ou seis) igualmente de quatro buracos cada. Ao lado de cada uma destas linhas estão dispostos 4 orifícios de tamanho menor. Para colocar nos buracos maiores são usados pinos de 6 cores diferentes e nos de menor tamanho são colocados pinos de duas cores, pretos ou brancos.

Os dois jogadores devem definir previamente quantos jogos irão ser jogados, este número de jogos deve ser par, e qual dos jogadores se define como *codemaker* enquanto o outro como *codebreaker*.

O *codemaker* começa por definir o código a decifrar na área coberta pela capa, visível para ele mas não para o outro jog ador. De seguida o *codebreaker* tenta adivinhar o código definido em termos de cor e ordem dos pinos, tendo ao seu dispor o número de tentativas definido antes de o jogo começar.

- Cada tentativa é feita através da colocação de 4 pinos numa da linha de orifícios mais afastada da capa do código que esteja livre.

A cada tentativa o *codemaker* responde com a colocação dos pinos na zona lateral das tentativas que representam se o código do *codebreaker* está correta ou errada:

- Por cada peça da tentativa que esteja no lugar correto e tenha a cor correspondente ao código definido é colocada um pino preto.

- Já os pinos brancos indicam a existência de uma peça de cor correta mas numa posição errónea.

Uma vez fornecido este *feedback* o *codebreaker* tem direito a uma nova tentativa até adivinhar o código ou atingir o número máximo. Apenas o *codemaker* ganha pontos em cada jogo:

- Se o *codebreaker* conseguir descobrir o código secreto são atribuídos pontos ao codemaker num número igual à quantidade de tentativas realizadas;

- Se o *codebreaker* não conseguir descobrir o código secreto são atribuídos pontos ao codemaker num número igual à quantidade de tentativas realizadas mais um;

No final do número de partidas previamente acordado o jogador com maior pontuação é o vencedor.

Lista de requisitos do sistema de software

Através da análise da descrição do jogo foi possível identificar variados requisitos necessários implementar para que o modelo formal fosse coerente e correspondesse ao objetivo pretendido e definido no início do projeto. Alguns destes objetivos serão enunciados de seguida:

1. O sistema responder o número de peças na posição correta e com a cor correta e o número de peças com a cor certa mas no sítio errado de cada tentativa;
2. Retornar o número de tentativas que o *codebreaker* usou para acertar na sequência correta;
3. Guardar a informação de um campeonato, com mais que uma equipa, e calcular a soma do número de tentativas que a equipa vencedora precisou, em todos os jogos, para ganhar o torneio.

Depois de esmiuçados estes objetivos e a análise supracitada foi elaborada uma lista mais especificada de requisitos adicionais:

1. Em cada partida, deve ser possível o *codemaker* criar o código chave.
2. A cada tentativa, num determinado jogo, deve ser possível saber quantas peças estão com a cor certa no lugar certo;
3. A cada tentativa, num determinado, deve ser possível saber quantas peças estão com a cor certa no lugar errado;
4. Após cada tentativa, deve ser possível verificar se a solução foi encontrada (tentativa = código secreto);
5. Após uma partida deverá ser possível verificar o número de tentativas que foram necessárias para adivinhar a solução;
6. No final de uma partida deve ser trocado o papel dos dois jogadores (*codemaker* para *codebreaker* e vice versa)
7. No final de uma partida deve ser atribuída a pontuação obtida pelo codemaker. Essa pontuação respeita as regras enunciadas previamente e está relacionada com o número de tentativas que o adversário gastou para adivinhar o código secreto.
8. Num jogo, com duas equipas, existem sempre duas partidas, em que cada equipa tenta quebrar o código definido pela adversária;
9. Relativamente ao modo torneio, dado um jogo, com duas equipas, deve ser possível a cada equipa definir o código a ser quebrado pela equipa adversária;
10. Deve ser possível criar um campeonato, onde participem várias equipas;
11. Deve ser possível adicionar equipas ao campeonato;
12. As equipas que participam num determinado jogo de um determinado campeonato, têm que estar inscritas no campeonato;
13. Um campeonato tem que ter no mínimo 2 equipas;
14. Deve ser possível gravar informações sobre um determinado torneio num ficheiro em disco.
15. Deve ser possível adicionar jogos aos campeonatos, e os perdedores devem ser eliminados.

Restrições essenciais para a integridade do jogo

Na sequência desta elaboração de requisitos foram identificadas restrições que se apresentaram como cruciais para a correção do programa e estão intimamente ligadas com a lógica do jogo.

1. Depois de o código secreto ter sido decifrado não é possível efetuar mais nenhuma jogada;
2. Só é possível submeter uma tentativa se o jogo ainda não terminou;
3. As sequências de tentativas e códigos secretos tem de ser compostas por peças de um conjunto de cores definidas (Blue, Green, Red, Orange, Yellow, Pink);
4. Cada jogo aceita tentativas e soluções com um número de elementos definidos aquando da criação do jogo;
5. Cada jogo aceita um número de tentativas definido aquando da criação do jogo;
6. O número de peças com a cor certa no sítio certo não pode ser superior ao tamanho da solução do tabuleiro;
7. O número de peças com a cor certa, mas no sítio errado não pode ser superior ao tamanho da solução do tabuleiro;
8. O número de equipas num campeonato tem de ser um número par maior ou igual a 2;
9. As equipas que participam nos jogos de um campeonato têm que estar inscritas nesse campeonato;

Modelo UML

Todo as funcionalidades do presente programa podem ser descritas através de um modelo de casos de uso aonde estão retratados todos os atores identificáveis ao longo da implementação do projeto. Todos os atores estão acompanhados dos seus casos de uso principais.

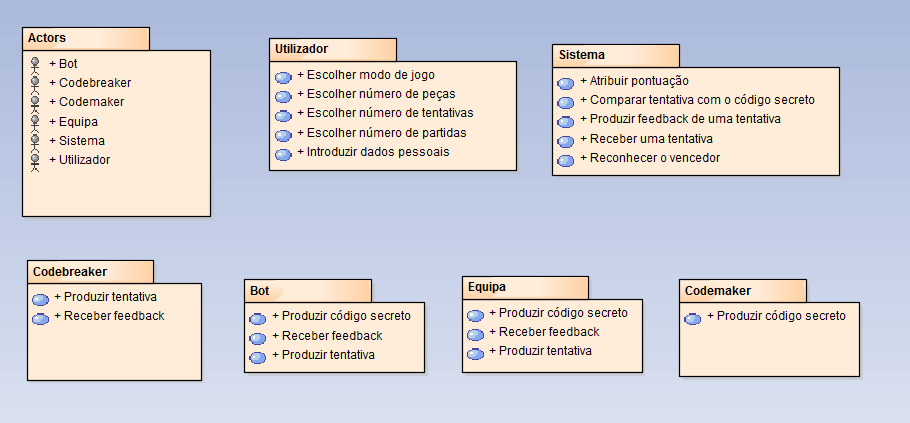


Fig2: Diagrama de casos de uso do software

Casos de uso principais

* **Esolher número de partidas:** como utilizador quero definir o número de partidas a jogar contra o outro jogador, sela ele outro utilizador, um bot ou uma equipa.
* **Esolher número de peças:** como utilizador quero definir o número de peças do código secreto e de cada tentativa.
* **Esolher número de tentativas:** como utilizador quero definir o número máximo de tentativas que o *codebreaker* dispõe para decifrar o código em cada partida.
* **Produzir o código secreto:** como *codemaker* quero definir a combinação secreta de peças a decifrar pelo outro jogador.
* **Produzir tentativa:** como *codebreaker* quero produzir uma nova tentativa para tentar decifrar o código secreto previamente definido.
* **Produzir feedback:** como sistema quero produzir o *feedback* relativamente a uma tentativa recebida de forma a retornar o número de peças de cor correta no sítio certo e o número de peças de cor correta no local errado.

Diagrama de classes

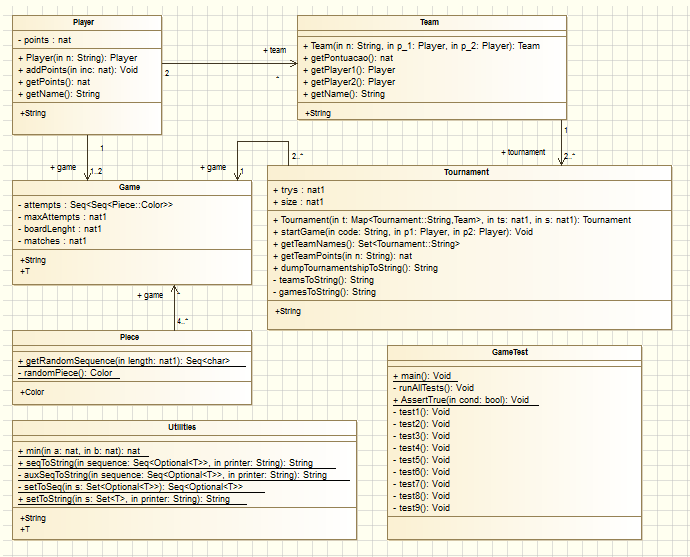


Fig3: Diagrama de classe do sistema

**Classe Player:**

* Esta classe representa um jogador e contém todos os seus dados pessoais bem como os pontos acumulados por si ao longo de um jogo. Todas as suas operações são relativas à atualização dos seus dados de forma a manter este elemento atualizado durante a execução do programa.

**Classe Team**

* Esta classe é similar à anterior uma vez que representa um jogador mas coletivo. Cada equipa é constituída por dois jogadores e as suas operações são igualmente utilizadas para atualizar os seus dados pessoais ao longo de um jogo.

**Classe Tournament**

* Classe representativa de um torneio, constituído por várias equipas e que guarda os dados relativos a toda a execução do jogo. As suas operações são relativas a retornos dos dados da classe e de atualização dos mesmos.

**Classe Piece**

* Como o nome indica esta classe retrata uma peça do jogo, com o atributo cor. As operações da classe são utilizadas no decorrer de cada jogo para gerar novos códigos aleatórios e no algoritmo do bot para gerar novas tentativas de adivinhar o código secreto.

**Classe Game**

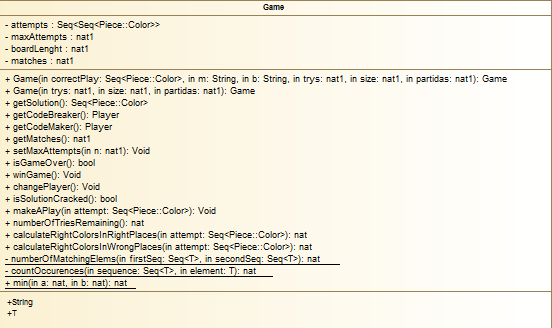


Fig4: Diagrama da classe Game

* Engloba toda a lógica de jogo e todos os dados para que o mesmo se execute corretamente. As operações desta classe são essenciais para a integridade do jogo e implementam a lógica e regras do jogo, bem como os requisitos do sistema, como referido anteriormente.

**Classe GameTest**

* Inclui todos os testes necessários para verificar a correção do sistema, estando relacionada com todas as outras classes do programa.

**Classe Utilities**

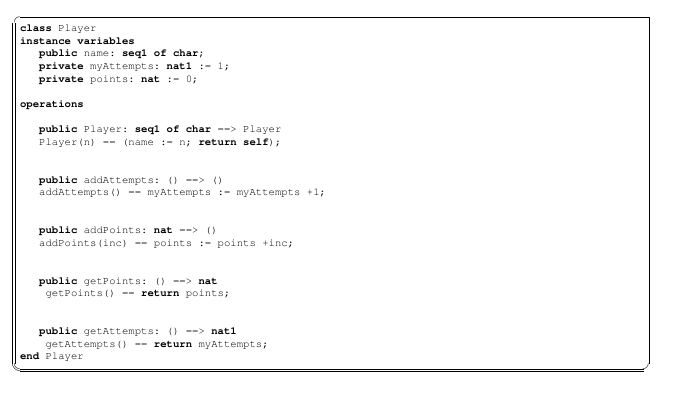
* Métodos úteis ao longo de todo o projeto. As operações desta classe são passiveis de serem utilizadas dentro de todas as outras classes

Por uma questão de legibilidade do diagrama as relações das duas últimas classes não foram representadas uma vez que estas estão relacionadas com todas as outras.

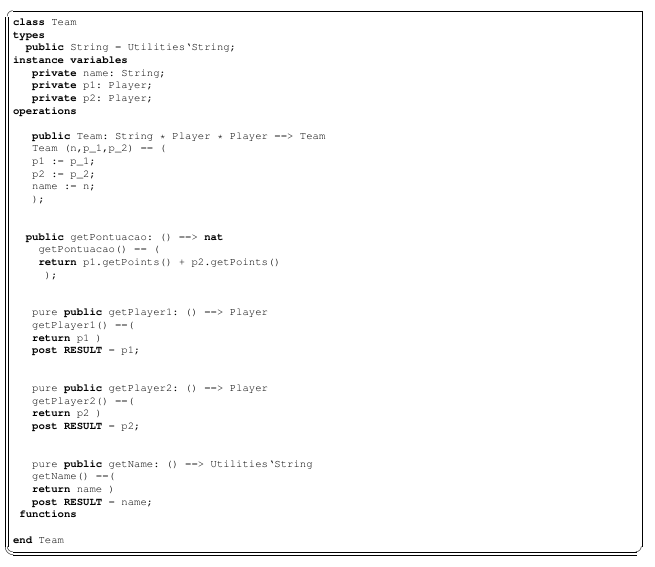
Formal VDM++ model

Neste tópico serão expostas todas as classes do programa, aonde é implementada a lógica do jogo e estão representados todos os requisitos e restrições referidos anteriormente. Todo o código será acompanhado de comentários que explicitem a sua função.

Classe Player

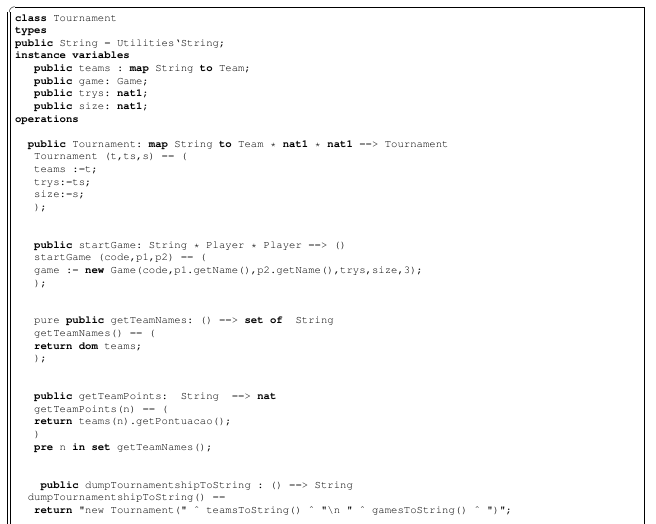


Classe Team



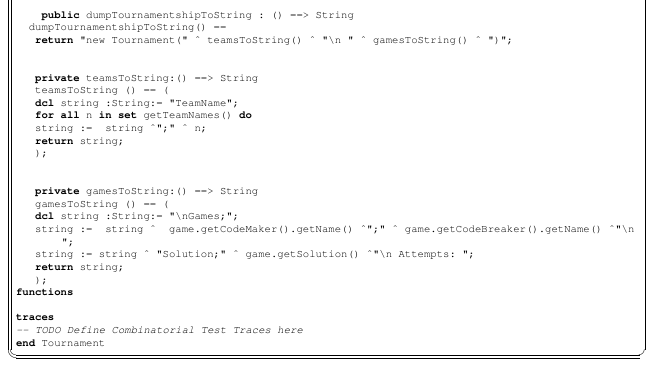
As operações desta classe usam pós-condições para verificar que o elemento retornado é o correto.

Classe Tournament

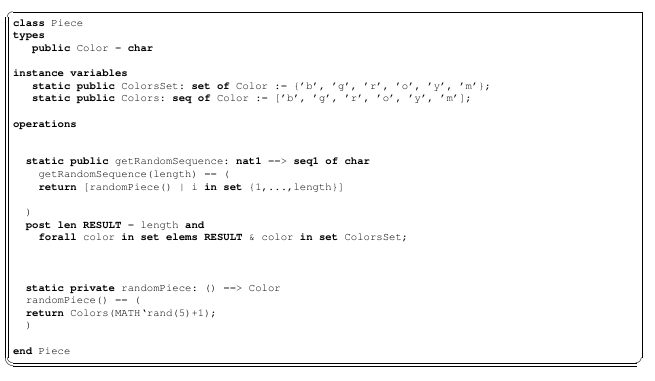


A pré-condição da operação anterior certifica-se que a equipa que é passada como argumento está contida no mapa de equipas do torneio.

As operações apresentadas de seguida foram construídas de forma a permitir que os dados de um torneio fossem gravados em disco num ficheiro de texto.



Classe Piece

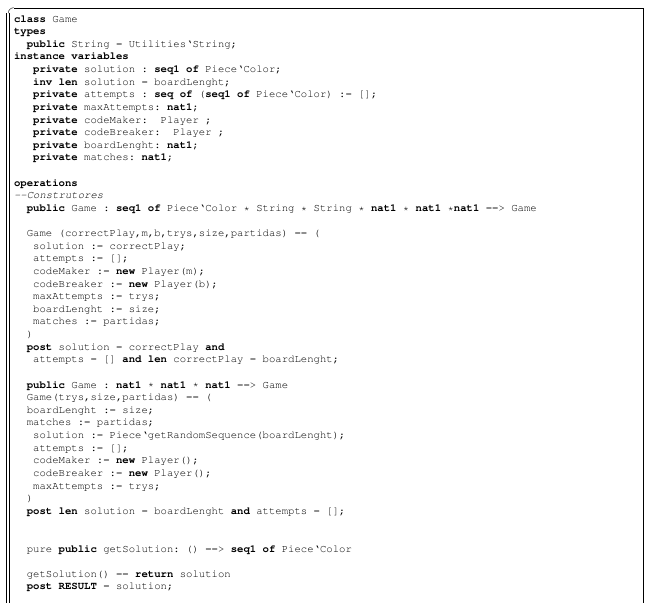


Nesta classe são instanciadas variáveis do tipo *set* e *seq* de forma a explorar as potencialidades da linguagem e uma vez que a sua utilização se revelou útil.

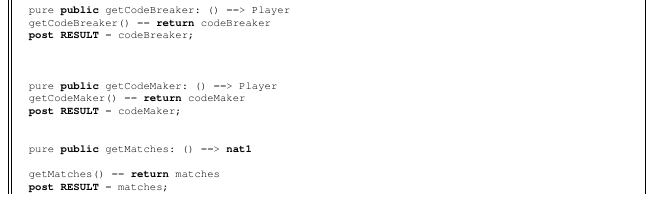
A operação *getRandomSequence(length)* retorna uma sequência aleatória do tamanho que é passado como argumento pela variável length.

Na operação *randomPiece()* é também usada a libraria MATH da linguagem VDM++ para criar um número aleatório dentro do intervalo definido.

Classe Game



Acima podem ser verificadas todas as variáveis essenciais para a execução do jogo que são inicializadas no construtor da classe. Esse mesmo construtor usa a pós-condição para certificar que a sequência de tentativas está vazia e que o tamanho da solução corresponde ao tamanho definido pelo utilizador.



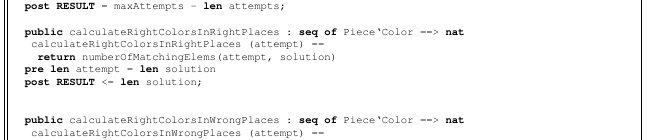
De seguida são apresentadas algumas funções cruciais para a lógica do jogo, entre elas:

*isSolutionCracked:* verifica se a solução está contida no seq de tentativas do utilizador.

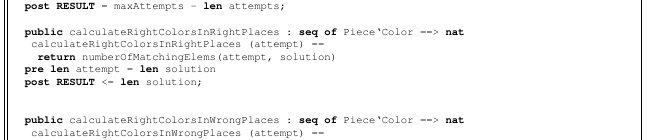
*makeAPlay:* verifica na pré-condição se o jogo ainda não acabou, se tal aconteceu permite que seja efetuada uma nova jogada e na pós-condição reduz o número de tentativas disponíveis.

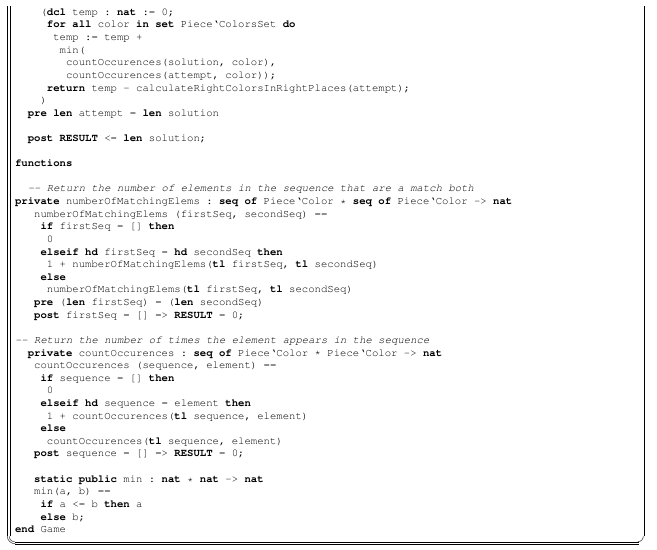
*changePlayer:* permite que dois jogadores disputem várias partidas ao trocar o seu papel.



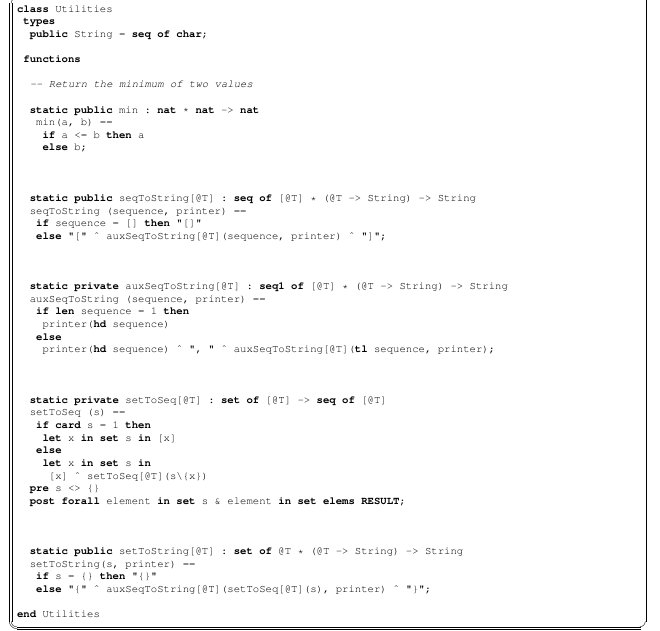


A operação acima referida e a seguinte são usadas na atribuição de um feedback depois de ter sido efetuada uma jogada válida. Através uma operação auxiliar são calculadas as peças colocadas na posição correta na ordem correta e também as peças que apenas têm uma cor correta. Na pós-condição é certificado que a resposta é no máximo do mesmo que a solução.





Classe Utilities



Nesta classe foram definidos operações maioritariamente com a finalidade de executar mudanças de tipo de variável. Estas operações foram usadas na implementação inicial e no projeto final acabaram por ficar um pouco de parte.

Model validation

Conclusões

Através da implementação deste trabalho todos os membros do grupo sentiram que a capacidade de compreensão e desenvolvimento de modelos formais de software foi melhorada em grande escala.

O objetivo principal foi alcançado pois o resultado final representa o modelo que foi enunciado no início do projeto, respeitando todas as restrições e requisitos apresentados ao mesmo tempo que as regras de lógica de jogo são respeitadas e o jogo é fielmente retratado.

Foi possível também compreender a linguagem VDM++ de forma muito mais alargada, e explorar as possibilidades que esta ferramenta, em conjunto com o Overture, proporciona. Toda a interface de implementação e execução de testes permitiu que a modelagem formal do sistema corresse de forma faseada e que cada nova característica fosse imediatamente testada. Tudo isto foi benéfico para que a demonstração da consistência do modelo fosse facilmente construída e executada.

Como potências melhorias do projeto é possível identificar dois casos que o grupo identificou como passíveis de serem aperfeiçoados:

- Melhorar o algoritmo que o bot usa para efetuar as suas jogadas.

- Melhorar a interface gráfica em java de forma a tornar a experiência do utilizador mais agradável.

Divisão do Trabalho:

- Mário Ferreira 33.3%

- Pedro Sousa 33.3%

- Luís Pinto 33.3%