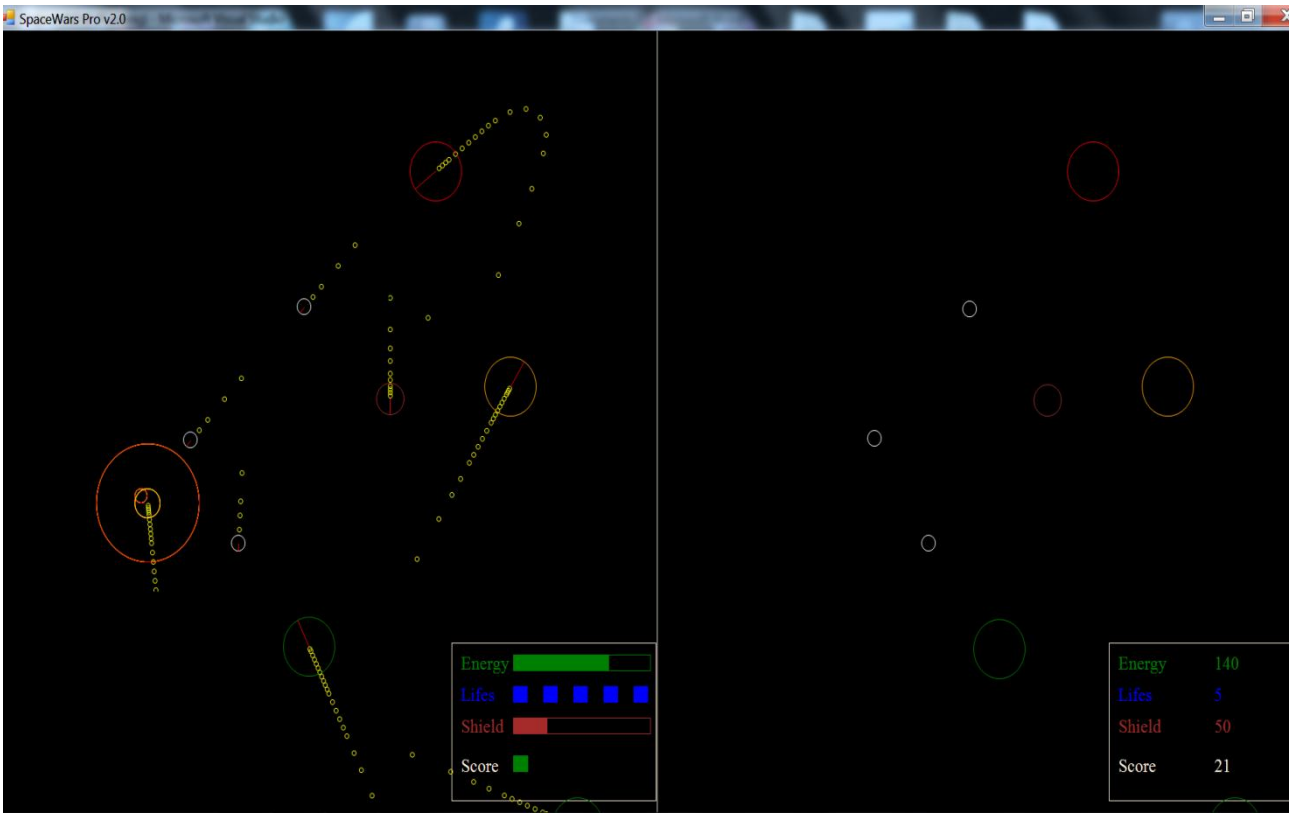
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Departamento de Engenharia Informática



MEI - 2011/2012

RS – Reutilização de Software

SpaceWars – The Game

Trabalho realizado por:

Pedro Miguel Ricardo Geadas 2006131902 pmrg@student.dei.uc.pt

Índice

1 - Introdução

Neste trabalho pretendia-se que fosse implementado uma versão do famoso jogo *SpaceWars*, que simula a batalha entre uma nave dirigida pelo utilizador e naves robot controladas pelo computador, recorrendo para tal a *Software Design Patterns* (SDP's). Com isso se pretendia não só criar o jogo em si, como também permitir uma fácil expansão do mesmo.

Apesar de ser uma tarefa deveras mais complexa, uma vez que é necessário descobrir os objectos pertinentes, organizá-los em classes com a granularidade adequada, definir o interface dessas classes e a sua organização hierárquica, e estabelecer as relações entre as várias classes, é fundamental que se utilizem estas técnicas se quisermos que a nossa aplicação possa sobreviver a médio/longo prazo. Desta forma, e apesar de ao início ser mais complicado e demorado, se quisermos/precisarmos de adicionar novas funcionalidades ao programa desenvolvido, a nossa vida fica muito mais facilitada.

No presente relatório irei abordar os SDP's utilizados, comparando as suas vantagens e desvantagens. No final do mesmo, pode-se encontrar um pequeno manual do utilizador.

1.1 - Visão Geral dos SDP's

Christopher Alexander (Arquitecto): "Cada pattern descreve um problema que ocorre repetidas vezes, e por isso descreve o núcleo de uma solução para esse problema, de forma que essa solução possa ser utilizada repetidas vezes".

Em geral uma Design Pattern (Padrão) tem 4 elementos principais:

•O nome é uma forma de identificar o problema de design.

•O problema descreve quando aplicar a pattern.

•A solução descreve os elementos que constituem o design, o seu relacionamento, responsabilidade e colaboração.

•As consequências são os resultados e as desvantagens da aplicação da pattern.

O que é então um Design Pattern? Os Design patterns não têm a ver com o design de listas ou tabelas de hash e não são também designs para aplicações ou subsistemascompletos. Design patterns são descrições de objectos e classes interrelacionados que são adaptados para resolver um problema de design num contexto particular.

2 - Estrutura da aplicação

2.1 – Protótipo 1

Inicialmente foi desenvolvido um protótipo do mesmo jogo, mas sem recorrer a SDP's. Segue então uma imagem do diagrama de classes inicialmente criado:

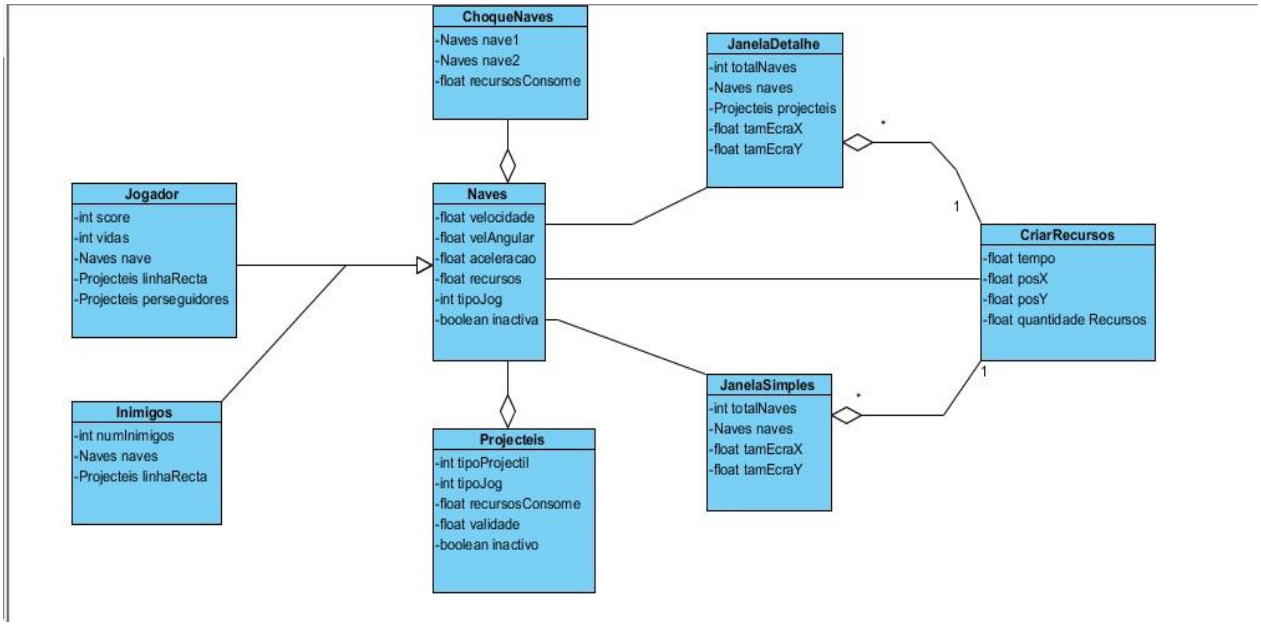


Figura 1 : *Diagrama de Classes do Protótipo 1*

No diagrama acima é possível constatar vários aspectos que fazem parte do jogo:

- A existência de diferentes tipos de Jogador (Jogador e Inimigo), em que ambos possuem uma nave, herdando vários métodos e atributos comuns a ambos desta, e redefinindo aqueles que são específicos na sua classe concreta;
- Cada nave possui vários Projécteis e pode ter um determinado número de Choques, que lhes consome Energia;
- Cada nave tem dois tipos de Janelas de Detalhe, simples e detalhada, que possivelmente irão corresponder a dois diferentes métodos de desenho, respectivamente;
- Existiria também uma Classe encarregue de criar recursos.

Obviamente que este esboço nada tem a ver com a versão final, e serviu apenas como base para o desenvolvimento das ideias fundamentais do jogo. Nesta fase de desenvolvimento, ainda não era pedida a utilização de SDP's.

2.2 – Protótipo 2

Estava claro que a versão acima, além de muito incompleta, estava bastante mal estruturada. Faltavam tratar muitos aspectos que se viriam a mostrar fundamentais para o bom desenvolvimento e funcionamento do jogo. Ao longo do desenvolvimento dessa versão inicial vários problemas foram detectados e corrigidos, e foram então inseridos alguns SDP's que tinham como objectivo colmatar esses problemas, bem como fornecer um apoio e ajuda fundamentais a uma possível expansão futura ao mesmo. Como tal, segue então o Diagrama de Classes do Protótipo 2:

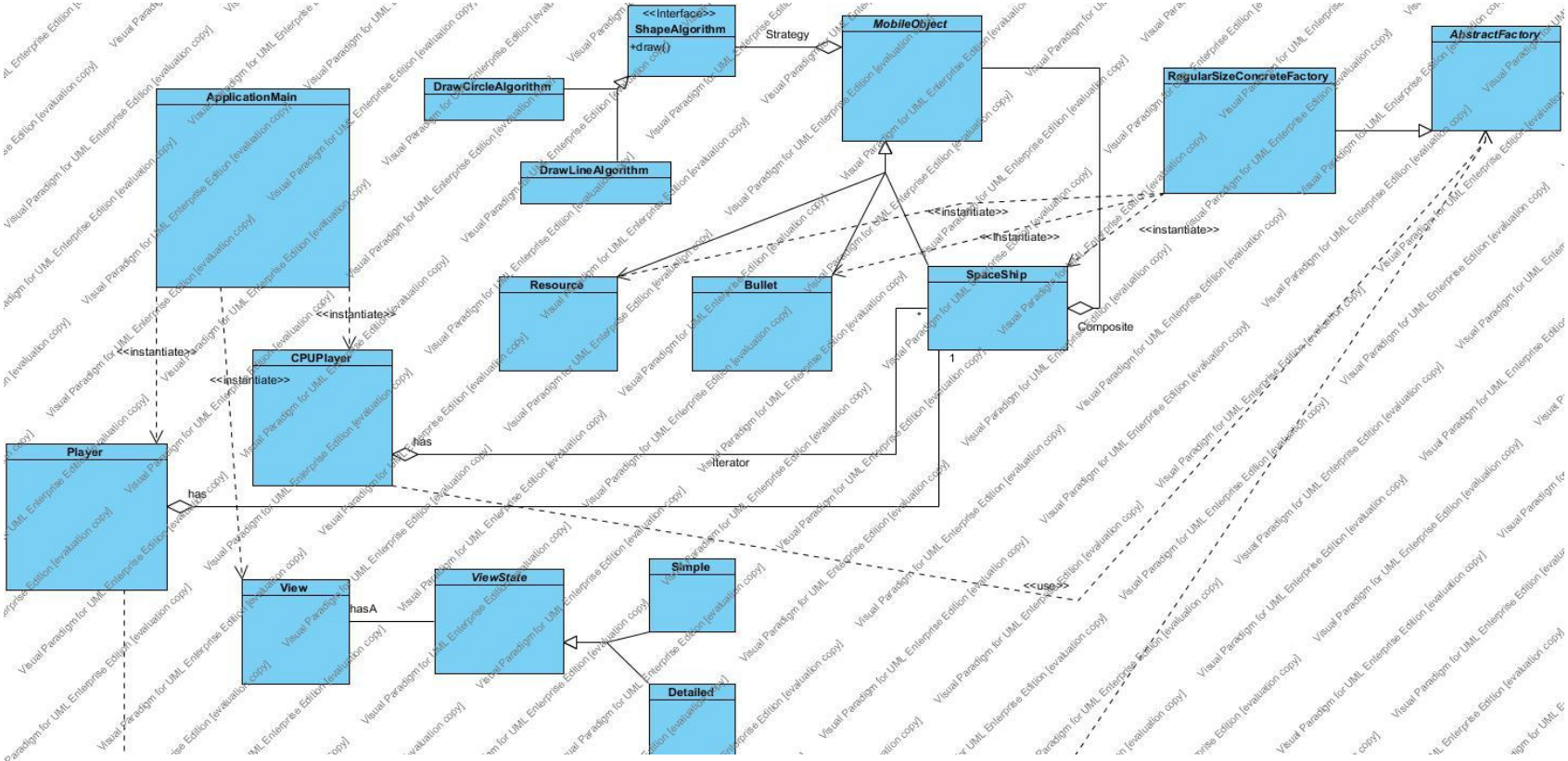


Figura 1 - *Diagrama de Classes do Protótipo 2*

Como se pode facilmente constatar, este diagrama está já bem mais completo que o esboço inicial. De notar que todos os termos que apareçam a ***Sublinhado***, correspondem a SDP's que irei falar e descrever posteriormente neste relatório, de uma maneira mais pormenorizada. Segue então a descrição do Diagrama acima apresentado:

- A classe **ApplicationMain,** responsável por instanciar o **Player** e o **CPUPlayer,** bem como a classe responsável por tratar das diferentes **Views *(State)***;
- Cada **Player/CPUPlayer *(Singleton)*** tem uma ou várias **SpaceShip's,** respectivamente;
- Cada **SpaceShip** tem um determinado número de outros **MobileObject's**, quer sejam **Resource's** ou **Bullet's *(Composite)***;
- Cada **MobileObject** tem uma ***Strategy*** correspondente, **ShapeAlgorithm**, que define um algoritmo, e que será a maneira como este será desenhado;
- De forma a criar os objectos existentes no jogo, existe uma única ***Abstract Factory***, que será passada por parâmetro a cada jogador, de maneira a que estes possam inicializar os seus objectos internos.

Como referi acima, este diagrama está bem mais completo que o esboço inicial e já contêm alguns *Design Patterns.* Apesar de não ir entrar ainda em grande detalhe, basta olhar para o diagrama e alguns aspectos importantes nos saltam imediatamente à vista:

- Caso se pretenda adicionar outro tipo de **Shape** aos nossos objectos, basta adicionar um novo algoritmo;
- Caso se pretenda adicionar outro tipo de **MobileObject** ao jogo, a nossa Factory irá tratar da instanciação do mesmo;
- O mesmo é válido para as **Views**: caso se pretenda outro tipo de View, pode-se adicionar um novo estado, correspondendo a essa View.

Apesar de mais completo e de já conter algumas SDP's, aquando da implementação foram detectados alguns problemas com esta estrutura, e alguns dos requisitos do jogo não podiam ser correctamente tratados. Deste modo, algumas alterações tiveram de ser feitas e alguns dos SDP's mencionados foram retirados e substituídos por outros. Não obstante, a estrutura que a aplicação tinha agora estava bem mais perto

da estrutura final do que o esboço inicial, e era agora também visível a utilidade da utilização de SDP's.

2.3 – Versão Final

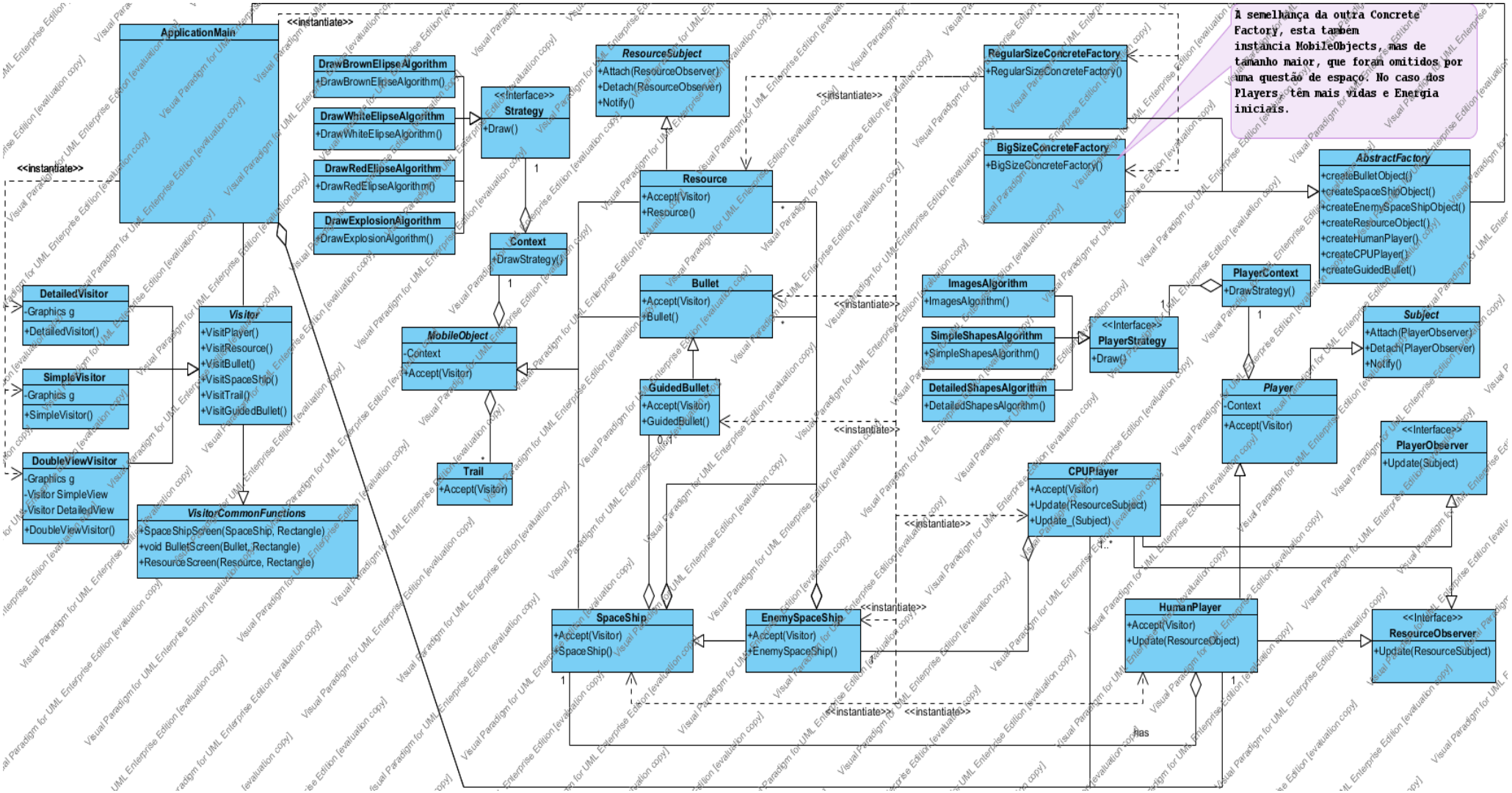


Figura 3 – *Diagrama de Classes Final*

Como se pode visualizar, o número de classes cresceu exponencialmente. Passei de **8 classes iniciais** para **16 classes** na segunda fase, acabando com **34 classes** (**mais 7** que foram omitidas por falta de espaço, correspondentes à segunda **ConcreteFactory**), perfazendo um total de **41 classes no total.**

É um facto que, apesar de normalmente serem implementadas melhores e mais aperfeiçoadas técnicas e algoritmos, quanto mais avança o processo de desenvolvimento de uma aplicação, aumenta também drasticamente o número de classes. Se utilizarmos SDP's é garantido que tal aconteça. Mas claro que este esforço extra é depois compensado, uma vez que se pretendermos adicionar novas funcionalidades, apenas necessitamos usualmente de modificar uma ou duas classes em específico, e não todas elas, poupando imenso tempo e dores de cabeça. E como é sabido, tempo é dinheiro, e o tempo é precioso!

Passo de seguida a detalhar o que se pode ver no diagrama acima:

- A **ApplicationMain** é agora apenas responsável por:
  - Instanciar a ***Abstract Factory***, que irá tratar da instanciação de todas as entidades que participam no jogo, **MobileObjects** e **Players**, e não apenas dos MobileObjects;
  - Instanciar os diferentes **Concrete *Visitors,*** que irão tratar do desenho dos MobileObjects – de notar que no Protótipo 2, a ideia era usar o ***State*** para efectuar tal tarefa. No entanto, dada a natureza do problema e visto que todos os **MobileObject** e **Players** tinham uma função **Draw** diferente (mas todos tinham), optei por retirar de cada objecto essa função e colocar numa classe à parte ***(Visitor*** - Esta opção e este SDP serão abordados mais à frente);
  - Nenhum objecto tem nem define agora internamente a maneira como irá ser desenhado, é o ***Visitor*** que o faz. É por isso que todos os **MobileObjects** e também os **Players**, ou seja, todos os objectos que irão ser desenhados no ecrã, têm que definir o método **Accept(Visitor)**, que irá servir exactamente para isso;
- A ***Abstract Factory*** declara os métodos que criam objectos e que têm de ser implementados pelas **Concrete Factorys** criadas: **RegularSizeFactory** e **BigSizeFactory**. Pelo nome se percebe que a primeira cria objectos de tamanho regular, enquanto a segunda cria objectos de tamanho maior. Em relação aos **Players**, a diferença reside no facto do número de vidas dos primeiros ser menor. Estes são apenas exemplos da utilidade da ***Abstract Factory***. (No diagrama de classes, os objectos criados pela **BigSizeFactory {**BigResource, BigBullet, BigPlayer, etc..**}** foram omitidos por falta de espaço, e também porque as diferenças em relação aos da **RegularSizeFactory** não são muitas);

- Cada **Player**, **HumanPlayer** e **CPUPlayer**, apenas têm uma **SpaceShip (EnemySpaceShip** no caso do CPUPlayer)**,** no entanto existem **vários CPUPlayers**: inicialmente a ideia era ter um CPUPlayer que controlasse várias SpaceShips, mas para simplificar foi tomada esta opção;
- Cada **MobileObject** continua a ter uma ***Strategy*** correspondente, designada agora por **Strategy**, que define vários algoritmos e que será a forma como este será desenhado; A alteração a que procedi aqui foi colocar este SDP de acordo com o que está no livro e adicionei a classe **Context.** Deste modo, cada MobileObject tem um Context respectivo, e é aqui que está definida a chamada à função **DrawAlgorithm** bem como o **set** e **get** da Strategy em questão. Assim sendo, a classe MobileObject apenas precisa de conhecer o Context, e deixa este encarregue por mexer na Strategy propriamente dita;
- Cada **MobileObject** pode ou não deixar um rasto, dependendo do nível de detalhe. Assim sendo, foi criada a classe **Trail** – quantos mais Trails forem definidos pelo utilizador, maior será o rasto deixado pelos MobileObjects;
- É visível no diagrama que cada **SpaceShip** pode ter várias **Bullet's** e vários **Resources** e ainda **GuidedBullet's** (estas últimas ainda não tinham sido considerados no protótipo anterior); foi adicionada também a classe **EnemySpaceShip,** uma vez que estas não podem ter **GuidedBullet's** e havia a necessidade de distinguir o tipo de naves respeitantes a cada tipo de **Player**;
- Cada **Player** tem de ter conhecimento pelo menos da posição dos outros para tratar das colisões entre SpaceShips e Bullets; para atingir esse objectivo, foi utilizado o SDP ***Observer:*** Cada vez que a **SpaceShip** de algum **Player** muda de posição, esta notifica os outros Players. É por isso que a classe Player deriva da classe **Subject**, uma vez que irá ser objecto de uma observação; Por outro lado, apenas os **CPUPlayers** precisam de implementar a interface **PlayerObserver**, pois só estes é que precisam de observar os outros jogadores: nós, HumanPlayers, apenas notificamos que a nossa posição foi alterada (Notify()) , e deixamos que o CPU faça o Update() respectivo;
- Para tratar da captura de **Resources**, a mesma técnica e ***design pattern, Observer,*** foi utilizada: o Resource ao cair notifica todos os observadores que a sua posição mudou, e estes, HumanPlayer e CpuPlayers, fazem o Update respectivo; (**HumanPlayer** e **CPUPlayer** implementam assim a interface **ResourceObserver,** enquanto que a classe **Resource** deriva da classe **ResourceSubject**);
- Para desenhar os atributos de cada Player importantes para o jogo, como número de vidas, energia, escudo e pontuação, e à semelhança do que acontecia com os MobileObjects, estes também pode ter vários tipos de detalhe e foi utilizada a mesma SDP – ***Strategy***.

- O ***Composite*** foi deixado de fora, pois a sua complexidade para o objectivo pretendido, era desnecessária: o ***Iterator*** é suficiente para implementar o pretendido, e é mais fácil de implementar;

Agora que já foi analisado o diagrama e resumidamente os aspectos fundamentais da aplicação, irei então falar sobre as vantagens das *Software Design Patterns* utilizadas e que já foram mencionadas ao longo deste relatório.

3 – *Software Design Patterns* Utilizados

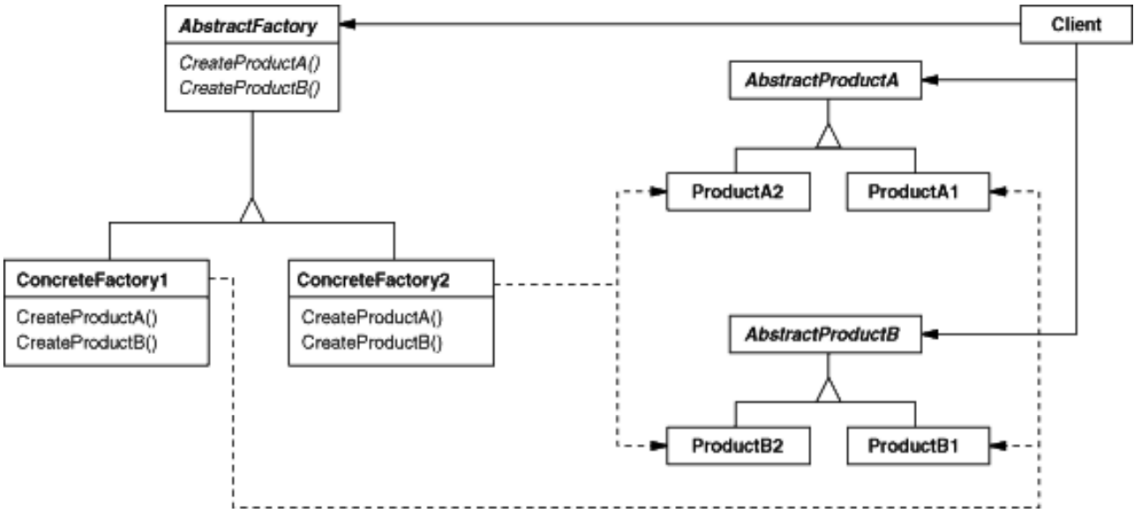
3.1 – Abstract Factory

Como já foi referido, a ***Abstract Factory*** declara os métodos que criam objectos na aplicação, deixando a sua implementação a cargo das **Concrete Factorys**. Neste caso, criei duas Concrete Factorys, **RegularSizeFactory** e **BigSizeFactory**, mas mais podem ser criadas, conforme as necessidades. De seguida, irei apresentar uma breve descrição deste *Design Pattern* bem como algumas das suas vantagens e limitações.

Objectivo

Esta *Pattern* tem como objectivo definir um interface para a criação de famílias de objectos relacionados ou dependentes, sem especificar as suas classes concretas.

Estrutura



Vantagens

1. **Isola classes concretas:** A Abstract Factory ajuda a controlar as classes de objectos que a aplicação cria, uma vez que encapsula a responsabilidade e o processo de criação, isolando assim os clientes da implementação dos mesmos. Os nomes das classes de produtos são isolados na implementação da respectiva Concrete Factory, pois não aparecem no código do cliente.

2. **Torna simples a troca de famílias de produtos:** A Concrete Factory usada, apenas aparece uma vez no código, isto é, quando é instanciada. Deste modo para trocar a Factory usada basta ir a essa linha respectiva e trocar, não sendo necessárias mais alterações.

3. **Promove consistência entre produtos:** Quando os objectos de uma família de produtos são projectados para trabalhar juntos, é importante que a aplicação use objectos de uma única família de cada vez. AbstractFactory torna isso fácil de aplicar.

Desvantagens

1. **Não é fácil suportar de novos tipos de produtos:** Produzir novos tipos de produtos não é fácil. Isto porque a interface da AbstractFactory fixa o conjunto de produtos que podem ser criados. O suporte de novos tipos de produtos requer que se estenda a interface da Factory, que envolve a mudança da própria AbstractFactory e todas as suas subclasses.

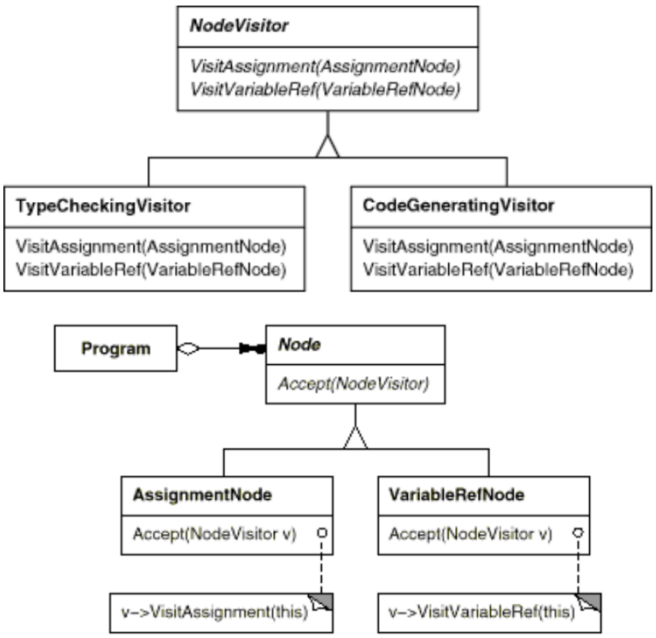
3.2 – Visitor

Inicialmente todos os **MobileObjects** e **Player** tinham um método **Draw,** responsável pelo desenho dos mesmos. De modo a suportar a operação de mudança de detalhe pretendida, tinha que criar 3 métodos diferentes em cada classe: SimpleView, DetailedView, e DoubleView. Para desenhar o método respectivo em run-time, era necessário condições "if" para determinar que método chamar. Para resolver esse problema, foi uilizado o **Visitor**.

Objectivo:

Representar uma operação a ser executada sobre os elementos de uma estrutura de objectos. Visitor permite que se defina uma nova operação, sem alterar as classes dos elementos sobre os quais opera.

Estrutura



A estrutura do **Visitor** é a que se pode ver em cima. Irei então comparar a mesma com a que foi usada no projecto. Existe uma classe Abstracta Visitor, que equivale à NodeVisitor acima. Nessa classe são declarados os métodos que todos os Visitors Concretos têm de implementar. Depois, analogamente à classe Node, existe no meu projecto a classe MobileObject e Player, onde previamente existia a função Draw como já foi dito. Essa função passa agora para o Visitor Concreto, passando as classes abrangidas a implementar apenas um método Accept(Visitor v), onde cada classe chama o seu método respectivo, e se "envia" por parâmetro para o Visitor em questão. (Por exemplo, a classe SpaceShip irá chamar v->VisitSpaceShip(this), enquanto a classe Player irá chamar v->VisitPlayer(this)).

Vantagens

1. **Visitor torna fácil a adição de novas operações:** Este SDP torna mais fácil a adição de novas operações, que dependem normalmente de componentes de objectos complexos. Podemos definir uma nova operação sobre um objecto, adicionando simplesmente um novo Visitor Concreto. Em contraste, e como já foi referido, ao espalhar funcionalidades (métodos) ao longo muitas classes, é necessário mudar o método em todas as classes, para definir uma nova operação. Esta foi uma das principais razões pelas quais decidi implementar o Visitor no meu projecto, como já foi explicado acima.

2. **O Visitor reúne operações relacionadas e separa as não relacionadas:** Comportamentos relacionados não estão espalhados pelas diferentes classes que definem a estrutura do objecto; é localizada num Visitor. Esta vantagem também é bem perceptível no meu projecto, uma vez que as funções que passaram para o Visitor estavam todas elas relacionadas, pois como também já foi dito, foi a função Draw de cada classe que passou para o mesmo. Comportamentos específicos de cada classe são particionados em suas subclasses de Concrete Visitor. Isso simplifica tanto as classes que definem os elementos bem como os algoritmos definidos nos Visitors. Conforme necessário, cada Concrete Visitor pode ter diferentes parâmetros privados, conforme a classe respectiva que vai ser visitada. Faço também uso deste aspecto no meu projecto.

3. **Acumulação de Estados.** Os Visitors podem acumular estados, enquanto visitam cada elemento da estrutura do objecto. Sem um Visitor, estes estados seriam passados como argumentos extra para as operações que realizam a travessia, ou tinham que aparecer como variáveis globais. Na minha implementação, também recorro a esta vantagem do Visitor, para trocar, por exemplo, a Strategy respectiva a cada MobileObject no caso de estes serem atingidos.

Utilizo uma variável que me guarda o estado anterior do Objecto, e, depois de desenhar a explosão, volto a colocar o estado antigo do Objecto no mesmo.

Desvantagens

1. **Adicionar ConcreteElements novos é difícil. (novos tipos de Player ou MobileObject, no âmbito deste projecto)** O Visitor torna difícil adicionar novas subclasses de Element. Cada nova ConcreteElement dá origem a uma nova operação abstracta na interface Visitor e uma implementação correspondente em todas as classes ConcreteVisitor.

2. **Quebrar o encapsulamento.** O SDP Visitor assume que a interface de cada ConcreteElement é suficiente flexível para permitir que os Visitors façam o seu trabalho. Como resultado, o Pattern muitas vezes obriga a fornecer operações públicas para aceder ao estado interno de um elemento, o que pode comprometer seu encapsulamento.

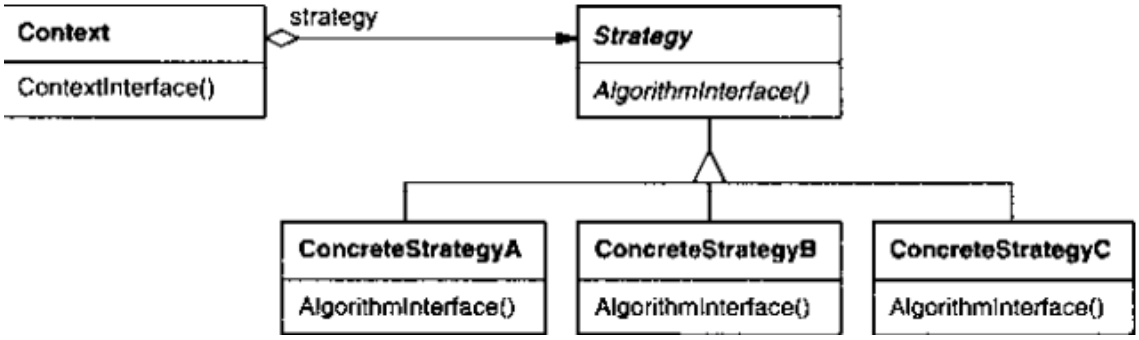
3.4 – Strategy

Definir um método de desenho próprio e único para cada objecto, é bastante limitado. Por outro lado, se definirmos em cada objecto vários métodos de desenho, apesar de mais flexível, ainda é um pouco limitado, pois para adicionar um novo tipo de desenho, temos que alterar todas as classes que queremos desenhar e é ineficiente, pois teríamos de ter condições “if” para verificar qual o desenho a ser feito, consoante o método escolhido. Para resolver este problema, resolvi usar a Strategy. Irei então descrever melhor o que é e como foi utilizada esta Pattern no projecto.

Objectivo

Definir uma família de algoritmos, encapsular cada um, e torná-los intercambiáveis. A Strategy permite que o algoritmo varie, independentemente dos clientes que o utilizam.

Estrutura



A estrutura usada neste projecto é idêntica à que está apresentada acima. A classe Strategy, define um método que terá de ser implementado por todas as Strategys Concretas, neste caso o Draw. Assim sendo, podemos então definir os algoritmos pretendidos. No meu caso, defini algoritmos para desenhar elipses, linhas, explosões, quadrados, etc. Para adicionar uma nova forma ou desenho à aplicação, basta criar uma nova Strategy, e fazer o Set desta quando for preferível. No presente projecto, e para demonstrar também o poder e usabilidade desta Pattern, é possível mudar a forma e cor da SpaceShip do HumanPlayer, em run-time. Para tal, só preciso de aceder ao Context actual, e passar-lhe a Strategy respectiva.

Vantagens

1 **Strategy elimina declarações condicionais**. A pattern Strategy oferece uma alternativa para instruções condicionais, para a selecção do comportamento desejado. Como já mencionei acima, a Strategy elimina estas declarações condicionais.

2 **A escolha de implementações.** Strategies podem fornecer implementações diferentes do mesmo comportamento. O cliente pode escolher entre as estratégias com diferentes trade-offs de tempo e/ou espaço.

Desvantagens

1 **Os clientes devem estar cientes de estratégias diferentes.** A pattern tem uma potencial desvantagem no facto de que um cliente deve entender as diferenças entre Strategies, para que possa seleccionar a apropriada. Deste modo, os Clientes podem ser expostos a problemas de implementação. Neste projecto, as Strategies não são muito complexas, por isso este drawback é suplantado pelas vantagens da utilização do pattern.

2 **Sobrecarga de comunicação entre a Estratégia e Contexto**. A interface Strategy é compartilhada por todas as classes ConcreteStrategy, sejam os algoritmos que implementam triviais ou complexos. Na minha implementação, **este problema não existe.** Como Context e Strategy estão bastante desassociados, não irão ocorrer por parte do Context operações e inicializações de informação desnecessária. O Context, em cada Objecto que faça uso dele, apenas irá inicializar a Strategy pretendida e/ou mudá-la conforme pretendido pelo Utilizador.

3 **Aumento do número de objetos.** Este SDP pode aumentar o número de objectos numa aplicação. No caso deste jogo, este problema não se aplica, pois o número de Strategies existente não é assim tao grande e não descompensa a utilização desta pattern. No entanto, quando isso acontece, é possível reduzir essa sobrecarga através da implementação de estratégias como objectos stateless. A pattern Flyweight descreve esta abordagem com mais detalhe, no entanto não irá ser abordada no presente relatório.

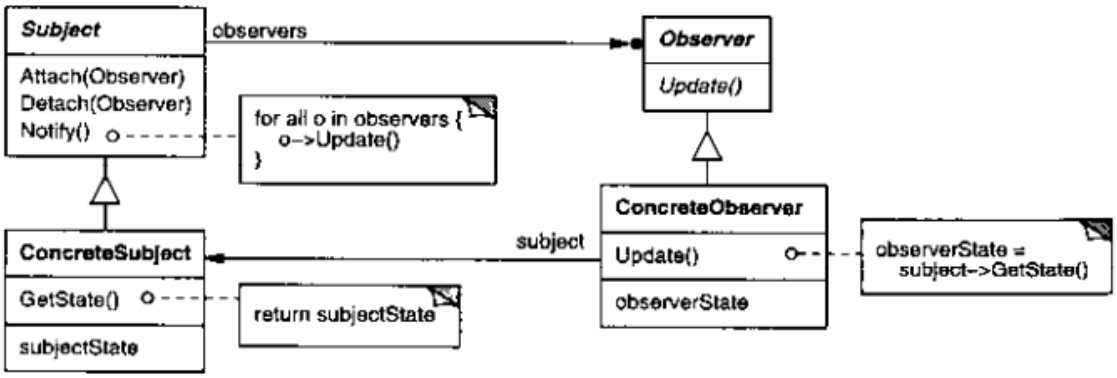
3.5 – Observer

Este SDP foi utilizado neste projecto para tratar da interacção entre os diferentes objectos, mais concretamente do choque entre naves, captura de recursos por estas bem como determinar se uma nave foi ou não atingida. Como já foi também descrito na estrutura da aplicação, cada vez que a **SpaceShip** de algum **Player** muda de posição, os outros Players precisam de ter conhecimento e ser notificados. Deste modo, sempre que existe uma alteração na posição das naves, é testado se algum choque ocorreu ou se algum recurso foi apanhado, entre outras coisas. Vamos ver agora como o Observer atinge o objectivo, fornecendo assim à aplicação o comportamento pretendido.

Objectivo

Definir uma dependência um-para-muitos entre objectos, para que, quando um objecto muda de estado, todos os seus dependentes sejam notificados e actualizados automaticamente.

Estrutura



Como já foi também referido, este pattern foi utilizado em duas situações distintas: notificar diferentes Players sobre alterações de outros; notificar diferentes Players sobre novos Resources que estejam disponíveis para capturar. Deste modo, ao observarmos o diagrama de classes final, podemos ver que existem 2 Subjects concretos distintos (PlayerSubject e ResourceSubject), bem como 2 Observers concretos distintos também (ResourceObserver e PlayerObserver). Embora cada Subject e cada Observer tenham objectivos diferentes, ambos têm a mesma estrutura, e são semelhantes com a estrutura apresentada acima. No entanto, existem obviamente diferenças, mas Cada Concrete Subject herda os métodos definidos pela classe Abstracta Subject, que permitirá adicionar novos Observers ao objecto em questão, Attach(Observer), remover Observers, Detach(Observer) e notificar os Observers actualmente a monitorar o objecto, Notify(). Por sua vez, cada Observer concreto tem que implementar um método virtual da interface Observer, e de cada vez que recebem uma notificação, efectuar a actualização do seu estado interno. De seguida, irei apresentar as vantagens e desvantagens da utilização desta pattern.

Vantagens

1 **Acoplamento abstracto entre Subject e Observer.** Tudo o que o Subject sabe é que tem uma lista de observadores, cada um em conformidade com a interface simples da classe Observer. O sujeito não sabe a classe concreta de nenhum observador, garantindo assim o mínimo acoplamento entre os diferentes Subjects e os Observers. Uma vez que Subject e Observer não estão intimamente ligados, podem pertencer a diferentes camadas de abstracção num sistema, o que também é uma grande vantagem da utilização deste pattern.

2 **Suporte à comunicação de broadcast.** Ao contrário de um pedido comum, não é necessário especificar um receptor para notificação que um Subject envia. A notificação é transmitida então automaticamente a todos os objectos interessados, que subscreveram o Subject através da operação Attach, como já foi mencionado.

O Subject não sabe da existência ou não de Observers, pois a sua única responsabilidade é notifica-los. Isso dá-nos a liberdade de adicionar e remover observadores a qualquer momento, que era exactamente o pretendido, como pode ser facilmente constatado.

Desvantagens

1 **Updates inesperados.** Esta desvantagem não se aplica no projecto em questão, pois as modificações que possam ocorrer nos Subjects não irão afectar possíveis Observers, nem criar comportamentos inesperados por parte dos mesmos.

Conclusão

Com a realização deste trabalho prático, foram obtidos bastantes conhecimentos sobre *Software Design Patterns*, as suas estruturas, vantagens e desvantagens, entre outros conceitos também eles importantes. A importância da utilização de técnicas de Reutilização de Software toma proporções gigantescas, sobre as quais nunca tinha despendido muito tempo para reflectir. Este trabalho prático ajudou-me a compreender o que são e para que servem os SDP's, bem como a valorizar e incentivar a sua utilização.

Reforçando tudo aquilo que tem vindo a ser dito, é com prazer e entusiasmo que constato a facilidade com que uma aplicação no geral, e a minha em concreto, pode ser estendida quando trabalhamos e utilizamos SDP's. Devo ainda dizer que, se assim não fosse, seria impossível alterar e adicionar funcionalidades, com facilidade, à mesma. E se assim é com uma aplicação desta envergadura, imagino como seria tentar expandir aplicações bem maiores e mais complexas… Tudo menos trivial.

Apesar de estar bastante satisfeito com o resultado final, convém salientar alguns problemas com que me deparei: devido à complexidade inerente de uma abordagem baseada em SDP's e também às diferentes, e complexas, decisões que tive de tomar ao optar pelas SDP's em questão, de entre tantas, muitas ideias que iriam tornar o jogo ainda mais apelativo, mais completo e mais versátil, acabaram por morrer por falta de tempo. O presente trabalho, que foi inicialmente previsto para ser realizado por 3 pessoas, acabou por ser desenvolvido inteiramente por mim, não porque eu assim o quisesse, mas porque teve que ser – os meus colegas de grupo desistiram da cadeira aproximadamente por volta da 3ª meta, deixando-me nesta ingrata situação. No entanto isso não me levou a desistir, antes pelo contrário, ainda me deu mais força para continuar. Não obstante, ainda que o tempo também se tenha tornado curto, e comigo a fazer o trabalho de três pessoas, o resultado final é, considero eu, muito bom.

Para finalizar, resta-me dizer que considero que este trabalho e esta cadeira foram bastante importantes na minha formação académica, bem como na de qualquer outro aluno, uma vez que os conhecimentos adquiridos irão fazer seguramente muita falta um dia mais tarde quando nos encontrarmos no mercado de trabalho

Manual de Utilizador

As teclas que devem ser usadas para jogar são:

"D" ou "RIGHT" – *rodar para a direita*

"A" ou "LEFT" – *rodar para a esquerda*

"W" ou "UP" – *acelerar*

"S" ou "DOWN" – *abrandar*

"Q" – *Velocidade Máxima*

"E" – *Velocidade Mínima*

"J" – *Salto para posição aleatória do mapa*

"G" – *Disparar míssil telecomandado*

"MOUSE_CLICK" – *Disparar mísseis normais*

"F" – *Mudar o tamanho dos mísseis*

"T" – *Mudar a forma da nave*

"C" – *Mudar a cor da nave (na forma circular apenas)*

"L" – *Mudar o nível de detalhe (Simples, Detalhado, Duplo)*