isec Engenharia

Programação Avançada

2019/2020

Planet Bound

Relatório da 1ª Fase do Trabalho Prático

Bruno Rodrigues – 2017014129

Turma P1

Índice

1. Opções e decisões de implementação

De modo a manter a coesão do código e facilidade de legibilidade, o programa está dividido em 7 *packages*, todos inseridos no mesmo *package* "base", pt.isec.br.TP_PA19_20:

PACKAGES	**UTILIZAÇÃO**
LOGIC.DATA	Guarda os packages que gerem todos os dados do jogo.
LOGIC.DATA.PLANET	Guarda as classes que lidam com os planetas.
LOGIC.DATA.PLANET.ALIEN	Guarda as classes que lidam com os aliens. Colocada dentro do package do planeta porque o alien é inerente aos planetas.
LOGIC.DATA.SHIP	Guarda as classes das naves e do drone.
LOGIC.DATA.STATES	Guarda todas as classes que gerem a máquina de estados.
UI	Guarda classes de interação com o utilizador.

O modelo (*logic*) e a vista (*ui*) estão separados como descrito no enunciado, em que a vista depende do modelo.

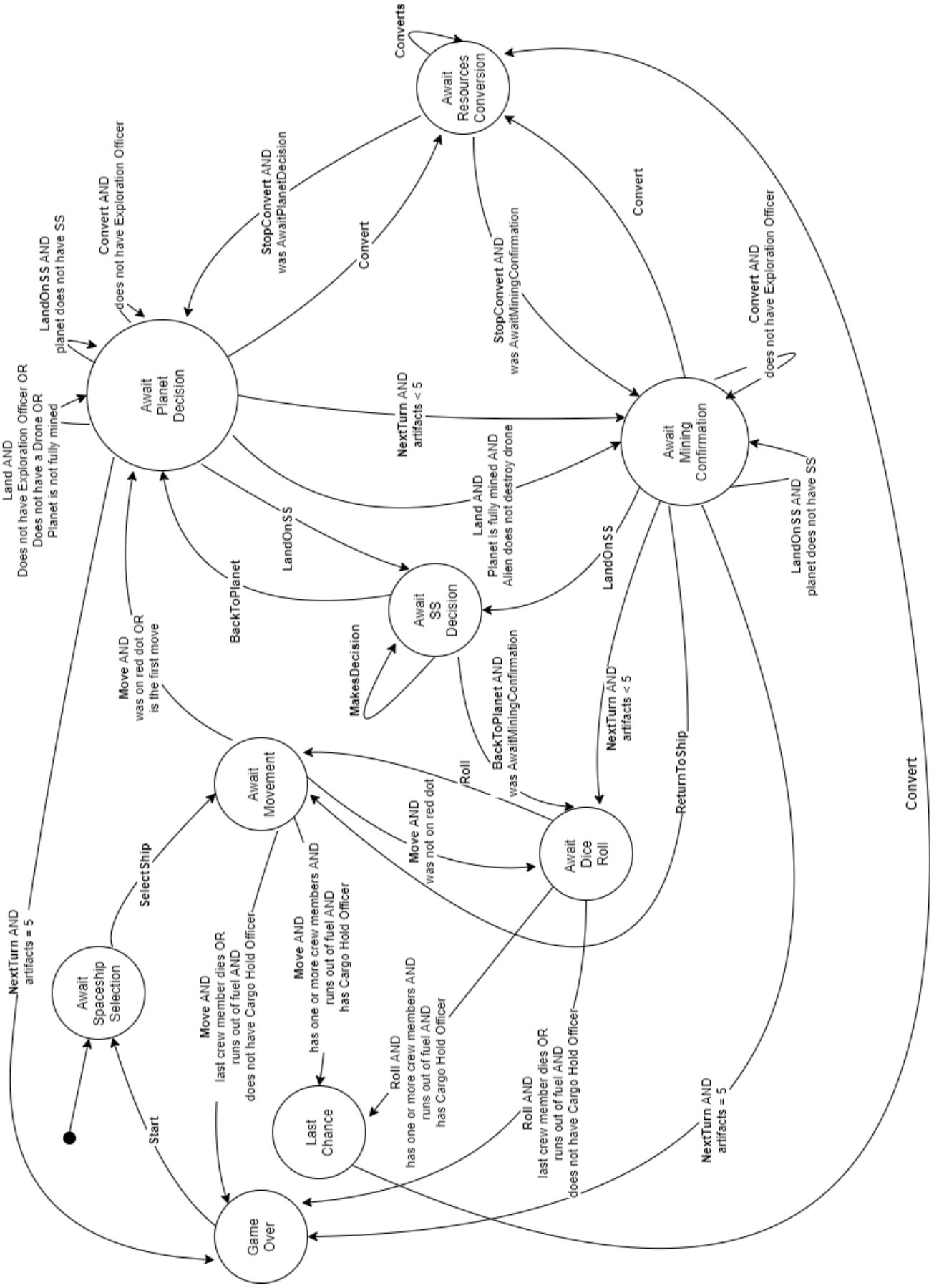
As decisões foram tomadas tendo em conta primeiro o enunciado e em seguida o ficheiro das regras do jogo. Dentro das regras do jogo foi dado prioridade ao que estava escrito e pela relevância do sítio onde estava escrito e em seguida às imagens com informação onde houvesse inconsistências.

Foi decidido não permitir conversões de recursos na Space Station, sendo só possível na nave, pelo que é descrito na fase "Convert Resources" do livro de regras, e não pelo que é dito mais acima na descrição fase "Space Travel". Esta decisão foi tomada com base na premissa que "Convert Resources" está mais relacionada com as permissões de conversão do que "Space Travel".

De modo semelhante, a fase "Space Travel" indica que para comprar um novo Drone, é necessário 3 de cada recurso, enquanto que na "Convert Resources" são necessários 2. Foi dado de novo prioridade ao descrito na fase "Convert Resources" pelo mesmo motivo descrito anteriormente.

Não foi implementada a fase 6 *Expend Fuel* porque os gastos de combustível vão sendo feitos à medida que cada ação é efetuada, o que deixa esta fase redundante.

2. Diagrama da Máquina de Estados



2.1. Explicação do Diagrama

ESTADO	EXPLICAÇÃO
AWAIT SPACESHIP SELECTION	Escolha da nave inicial.
AWAIT MOVEMENT	Cálculo de para onde é o próximo movimento (não tem interação).
AWAIT PLANET DECISION	Escolha da ação a realizar enquanto orbita um planeta.
AWAIT MINING CONFIRMATION	Decisão do que fazer depois da mineração de um planeta.
AWAIT RESOURCES CONVERSION	Decisão do que fazer após a conversão de recursos.
AWAIT SS DECISION	Escolha de conversões exclusivas à estação espacial (precisa de estar num planeta com estação espacial para aceder).
AWAIT DICE ROLL	Mostra o evento que irá acontecer antes da chegada a um novo planeta (a única interação é uma confirmação).
GAME OVER	Jogo termina com vitória ou derrota. Dá a possibilidade ao jogador de jogar de novo ou terminar.

O programa começa diretamente na escolha de uma nave. Daí, o fluxo do programa é linear, apesar de poder ter pequenos "desvios". Depois de escolher a nave, o fluxo é sempre de chegar a um planeta, sair do planeta, sofrer um evento e repetir.

Estando num planeta é possível minerá-lo, converter recursos, continuar a navegação ou aterrar numa estação espacial caso esta exista.

Converter recursos e aterrar numa estação espacial são possíveis em vários estados, porque não faria sentido não poder converter um recurso antes de prosseguir navegação, e se podemos converter um recurso também devemos poder aterrar na nave espacial, pois uma das razões para converter recursos pode ser para ter um recurso diferente para poder comprar algo na estação espacial. Assim sendo, é possível realizar estas ações nos estados "*Await Planet Decision*", "*Await Mining Confirmation*" e "*Await SS Decision*".

Ver: 4.1. Funcionalidades extra

Não há uma passagem direta de "*Await Planet Decision*" para "*Await Dice Roll*", porque nunca há uma passagem direta. Mesmo que a única coisa que o jogador faça quando chega a um planeta seja continuar a navegação, o estado seguinte é "*Await Mining Confirmation"* para poder ainda assim converter recursos ou aterrar numa estação espacial. De seguida passa para "*Await Movement"* que aí sim remete o programa para "*Await Dice Roll*".

Não foram utilizados outros padrões de programação.

3. Classes

3.1. Descrição

CLASSE	DESCRIÇÃO
ALIEN (+4 DERIVADAS: UMA POR TIPO)	Cria um alien e guarda as suas características de ataque e morte, assim como a sua posição durante a mineração de um planeta.
PLANET (+4 DERIVADAS: UMA POR TIPO)	Cria um planeta e guarda as suas características: o número de recursos que tem, as vezes que foi minerado, se tem uma estação espacial e o seu alien associado.
SHIP (+2 DERIVADAS: UMA POR TIPO)	Cria uma nave e guarda as suas características.
DATAGAME	Guarda todas as informações relativas ao jogo (nave, oficiais, planeta, etc). É onde estão as funções que fazem o jogo decorrer e que são invocadas pelos estados, como por exemplo *whereTo*() chamado pela função *move*() do estado *AwaitMovement* para decidir para que estado vai a seguir consoante o próximo sítio é um evento ou um planeta.
ISTATES	Interface da máquina de estados.
STATEADAPTER	Ponte entre a interface IStates e os estados.
GAMETOSTATES	Ponte entre a *UI* e a lógica do jogo
AWAIT(…)	Estado referente ao momento do jogo onde se encontra o jogador. Cada classe deste tipo tem *Overrides* de funções que lhe dizem respeito. Exemplo: função *roll*() com *override* no estado *Await Dice Roll* para fazer o evento seguinte. É o único *override* desta função nos estados.
TEXT	Interface por linha de comandos
COLORS	Classe abstrata para alterar as cores do texto que aparece na linha de comandos com recurso a *ANSI Escape Codes*

3.2. Relacionamento entre Classes

Para o diagrama de relacionamento entre classes, ver o **Anexo A**.

A classe:

- *Text* do package *ui* depende da classe *GameToStates* da package *logic.states*
- *GameToStates* depende da interface *IStates* do mesmo *package* e da classe *DataGame* do *package logic.data*
- *DataGame* depende da classe *Ship* do *package logic.data.ship*, da classe *Planet* do *package logic.data.planet*, e do *IStates* do *package logic.states*
- *Planet* depende da classe *Alien* do *package logic.data.planet.alien*
- *Ship* depende da classe *Drone* do mesmo *package*
- Todos os estados são dependentes da classe *StateAdapter* que por sua vez é dependente da interface IStates, todos localizados no *package logic.states*
- As classes *Military* e *Mining* são dependentes de *Ship* dado que herdam da última.
- Do mesmo modo que o ponto anterior, *BlackPlanet, BluePlanet, GreenPlanet* e *RedPlanet* dependem de *Planet.*
- Do mesmo modo que o ponto anterior, *BlackAlien, BlueAlien, GreenAlien* e *RedAlien* dependem de *Alien.*

4. Implementação de funcionalidades

Funcionalidade	
Implementação de uma máquina de estados	Implementado
Diferenciação entre vista e modelo	Implementado
Utilização de estados polimórficos	Implementado
Fábrica de objetos	Implementado
O jogo inicia com a escolha do tipo de nave	Implementado
O jogo termina no final do turno em que o último artefacto é encontrado	Implementado
Quando o drone se encontra numa célula adjacente ao alienígena inicia-se automaticamente um ataque	Implementado
Na fase space travel, probabilidades de planeta branco 7/10 e planeta vermelho 3/10	Implementado
Em cada setor o jogador pode avançar ou explorar recursos do planeta caso tenha o oficial e drone	Implementado
Probabilidade 1/8 de efetuar movimentação através de *wormhole*	Implementado
O tipo de planeta é determinado de forma aleatória e uniforme	Implementado
Eventos modificados e selecionar um evento específico	Implementado
Fases específicas do jogo	Implementado

4.1. Funcionalidades extra

Extra ao que é pedido no enunciado, foi decidido dar oportunidade ao jogador de fazer conversões em mais momentos e não só quando está em órbita do planeta, mas também quando já saiu dele e está à espera de um evento e antes de começar a minerar o planeta. Esta decisão prende-se com o facto de se o jogador está na nave e não está a acontecer nada nesse momento, não faz sentido que ele não possa converter recursos nessa situação.

Para além disso foi também implementado um sistema de pontuação com as seguintes ponderações:

ATRIBUTO	PONTUAÇÃO
Weapon system	1 ponto por célula
Shield system	1 ponto por célula
Fuel system	2 pontos por célula
Artifacts	10 pontos por artefacto
Recursos	2 pontos por recurso

As pontuações podem depois ser guardadas num ficheiro de nome “highscores.txt” e vistas no início e fim da execução do programa.