Hatch Run

Para o projeto final optamos por fazer uma versão da aplicação *Subway Surfers*. Nesta versão (que segue um tema alusivo a *Hatches*) o utilizador muda de *lane* orientando o telemóvel (giroscópio).

O objetivo do jogo é atingir o *score* mais alto possível. Os pontos são ganhos através das seguintes formas:

- cada milisegundo vale pontos;
- cada moeda apanhada também corresponde a pontos;
- existirá também um *power-up* que duplica os pontos ganhos durante um período de tempo;

Pretendemos implementar apenas a versão Android com possibilidade para conectar às redes sociais para publicar o *score*.

Autoras: Beatriz Soares Mendes, up201604253
Joana Sofia Mendes Ramos, up201605017

Design Patterns

Próprios à API *Libgdx* e utilizados por nós

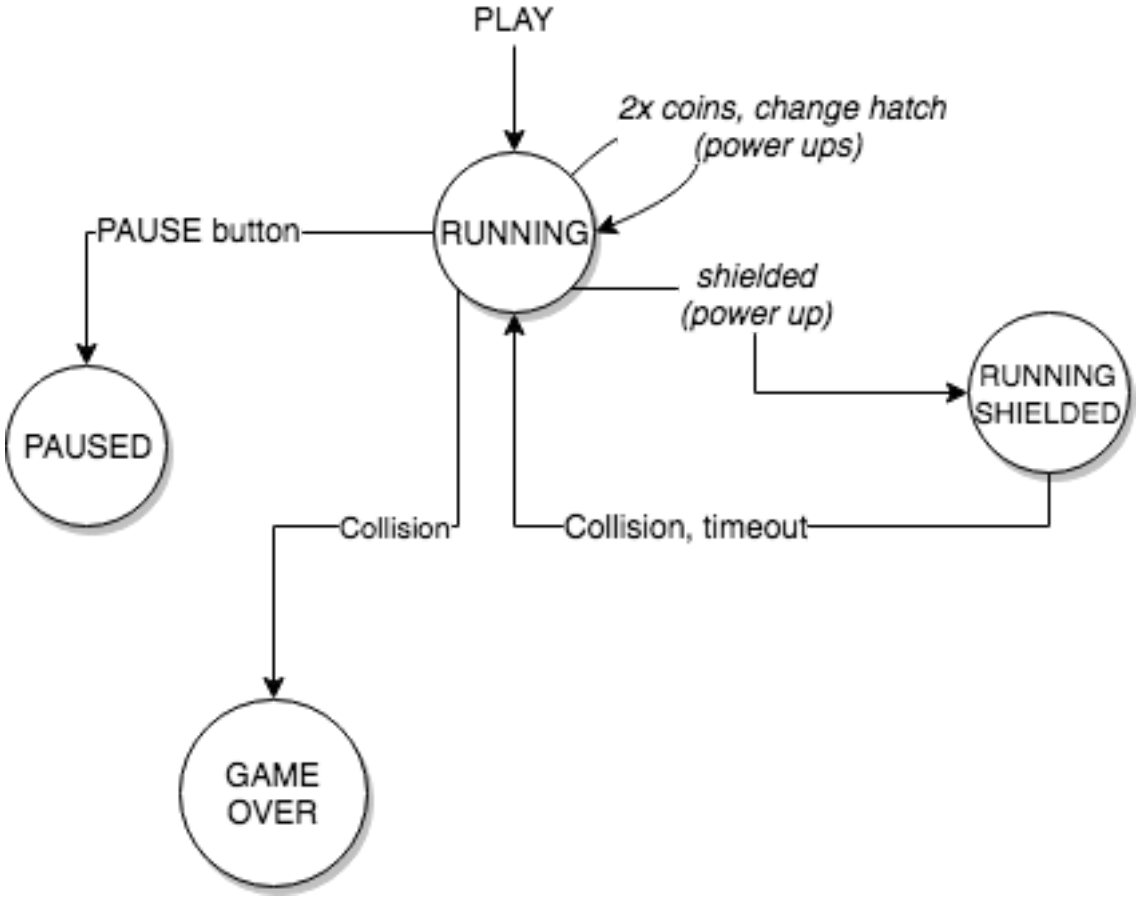
- *Observer Pattern*: Utilizada na classe do tipo *Screen Adapter*, que implementa a interface *Screen*: esta classe chama autonomamente o método *render* quando necessário. Cada *screen* criado é um observador. Utilizada de forma a facilitar a gestão de vários *screens* abstraindo-nos da forma como eles são atualizados.
- *Game Loop*: A aplicação é criada, *renderizada* e atualizada de forma abstrata, recorrendo a *Game*, *Screen* e *Listeners*.

Planeados para serem concebidos e utilizados por nós

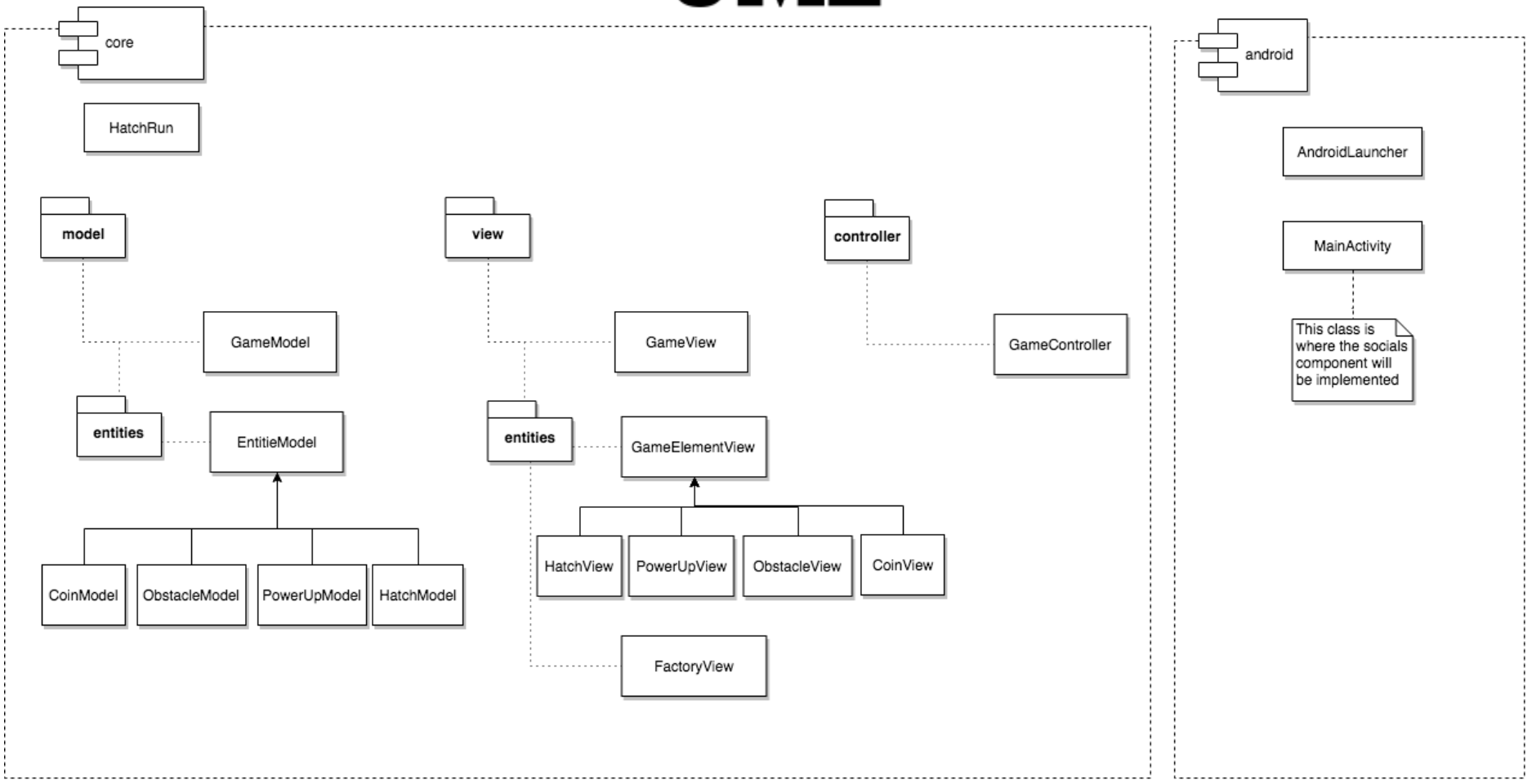
- *Model View Controller*: Escolhido por facilitar os testes à aplicação (aplicados ao módulo *Controller*), assegurar coesão, *low coupling* e facilidade de alterações uma vez que todos os módulos utilizam os mesmos dados. O *Model package* terá as estruturas de dados dos objetos de jogo; O *View package* terás as *views* para esses mesmos objetos (com este *pattern* é também uma vantagem podermos ter várias *views* para os mesmos objetos) e receberá os *inputs* da interação com o utilizador, passando-os ao *Controller* para tratamento; O *Controller package* será responsável pela lógica de jogo, atualizando devidamente o *Model* após tratar o *input* do utilizador (ou a falta dele).
- *Object Pool*: Uma vez que de outra forma objetos como moedas e obstáculos seriam criados e destruídos a uma taxa alta desnecessariamente, este *design pattern* foi escolhido para melhorar a performance a este nível.
- *Factory*: Utilizado para fabricar *views* em tempo de execução concedendo dinamismo à aplicação.

State: Dado que o *Hatch* não tem sempre o mesmo comportamento (é alterado através de *power-ups* e também tendo em conta o tempo que passou desde o início da *run*) é conveniente implementar este *pattern* para que ele possa alterar o seu comportamento consoante o estado em que está.

State Diagram



UML


core
HatchRun
model
GameModel
entities
EntitieModel
CoinModel
ObstacleModel
PowerUpModel
HatchModel
view
GameView
entities
GameElementView
HatchView
PowerUpView
ObstacleView
CoinView
FactoryView
controller
GameController
android
AndroidLauncher
MainActivity
This class is where the socials component will be implemented

App Mock Up


HATCH
RUN
PLAY
PLAY
50 m
SCORE
10
80 pts
PAUSED
RESTART
HOME
HOME
RESTART
PLAY AGAIN
SCORE
120
PLAY AGAIN
Share on Facebook

Test Design

- Testar se o *Hatch* apanha as moedas (colidindo com elas, deverá poder continuar a *run* normalmente, i.e., sem parar ou ser obrigado a mudar de *lane*, e fazê-las desaparecer, tendo o número de moedas correspondente adicionado à HUD);
- Testar se o *Hatch* apanha os *power-ups* (colidindo com eles, deverá poder continuar a *run* normalmente, i.e., sem parar ou ser obrigado a mudar de *lane*, e fazê-los desaparecer, verificando-se alguma mudança a nível da lógica de jogo consoante o *power-up*);
- Testar a atualização do *score* de acordo com as moedas apanhadas e o tempo da *run*.
- Testar se o *Hatch* perde o jogo aquando da colisão com obstáculos;
- Testar a atualização conveniente da HUD;
- Testar se o *Hatch* muda de *lane* de acordo com o giroscópio do utilizador.