***Functional Design Specification***

Project: «SeaBattle»

Участники:

**Рыбалкин Роман** – *программист, архитектор, администратор*

**Капустин Сергей** – *программист, архитектор*

© Production of Company «**KMF**» (*Keep Moving Forward*)

**СОДЕРЖАНИЕ**

[**1.** **Структура приложения – игровые модули** 2](#_Toc366944332)

[**2.** **Основные игровые модули** 3](#_Toc366944333)

[**2.1** **Игровые сущности (Game Objects Module)** 3](#_Toc366944334)

[**2.2** **Логический модуль приложения (Logical Module)** 5](#_Toc366944335)

[**2.3** **Модуль для работы с графикой (Graphics Module)** 5](#_Toc366944336)

[**2.4** **Управление и запуск игровых процессов (Game Manager Module)** 9](#_Toc366944337)

[**3.** **Дополнительные игровые модули** 14](#_Toc366944338)

[**3.1** **Модуль взаимодействия с внешней системой –Bluetooth модуль** 14](#_Toc366944339)

[**3.2** **Модуль игровых настроек (Game Settings Module)** 15](#_Toc366944340)

1. **Структура приложения – игровые модули**



**Рис. 1. Основные модули приложения и их связи**

Приложение состоит из нескольких модулей, каждый из которых отвечает за свою часть. На рисунке 1 изображена схема взаимодействия всех модулей приложения.

Модулем является обособленная единица приложения, выполняющая определенный набор функций. Для взаимодействия с внешними системами используется *Facade* модуля – класс, имеющий набор функций для взаимодействия с модулем.

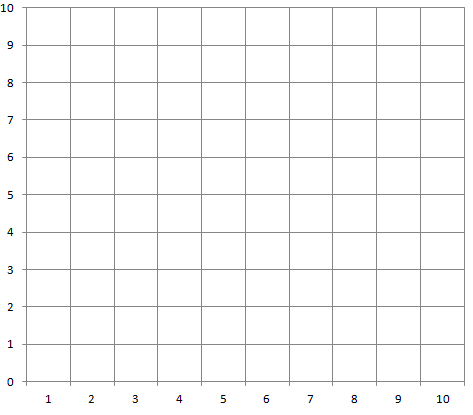
Приложение состоит из нескольких основных модулей, таких как *Logical Module, Game Objects Module, Graphics Module, External System Module, Game Manager Module* и нескольких дополнительных модулей - *Game Settings, Information Module, Localize Module, File Manager, Logger*.

1. **Основные игровые модули**
   1. ***Игровые сущности (Game Objects Module)***

На рисунке 2 представлены основные игровые сущности приложения. Единицей игрового поля является «морская капля» - “*SeaDrop*”. Этот объект имеет X и Y координаты, а так же флаги – “free” и “broken”, означающие, что клетка свободная/занятая и целая/разбитая соответственно. Эти объекты предназначены для создания объекта моря «*BattleSea*», который агрегирует «морские капли» в единое игровое поле. В программном представлении агрегация представлена двумерным массивом, у которого по горизонтали индексы соответствуют координате Y, а по вертикали – координате X. Агрегирующим объектом для «моря битвы» является объект «*Fleet*» (Флот) - который содержит объект «*SeaBattle*» и объекты моря, такие, как корабль «*Ship*».

****

**Рис. 2. Игровые сущности**

****

**Рис. 3. Шаблон игрового поля**

Чтобы добавить новый объект, необходимо реализовать интерфейс *ISeaObject.* После этого этот объект может быть добавлен в «морское поле битвы» *BattleSea.* Так же может быть создан новый объект «флот», для этого нужно реализовать интерфейс *IFleet,* имеющий один единственный метод *doShot(int, int),* принимающий на вход координаты выстрела и возвращающий результат типа *ShotResult.*

Стандартное поле битвы представляет собой набор «морских капель» размером 10х10 (рис. 3). Каждая ячейка этого поля является объектом «*SeaDrop*» со своими координатами. Стандартное поле битвы – объект «флот» содержит следующий набор объектов:

* 100 объектов «*SeaDrop*»
* 1 объект «*BattleSea*»
* 10 объектов кораблей «*Ship*»
  1. ***Логический модуль приложения (Logical Module)***

Логический модуль приложения содержит стратегии для автоматического распределения сущностей на игровом поле и стратегии обстрела вражеского поля. Каждая стратегия создания игрового поля должна реализовать интерфейс *IFleetCreator* и реализовать единственный метод *createFleet()* возвращающий созданный объект *IFleet*. Для создания стратегии обстрела игрового поля соперника необходимо реализовать интерфейс *IShootingStrategy*, имеющий единственный метод *getCoordinatesForShor(ShotResult lastResult)* – принимающий в качестве аргумента результат предыдущего выстрела и возвращающий координаты нового выстрела в виде объекта *Shot*. Так же, используя этот интерфейс, остается возможность для конфигурирования любого вида «обстрела» противника, в том числе и внезапными появлениями странных явлений (стихийные события, чудовища и т.п.).

На рисунке 4 показаны описанные интерфейсы и их реализации для логического модуля. Взаимодействия с логическим модулем осуществляются через методы класса *LogicFacade*. При создании объекта флота может использоваться как автоматическая, так и ручная стратегия создания. В случае ручной стратегии объект флота формируется на основании пришедшего дескриптора «флота» – *FleetDescriptor,* который имеет описание, как самого объекта «флота», так и агрегирует дескрипторы морских объектов – кораблей, чудовищ и т.д. Для получения координат выстрела используется стратегия обстрела игрового поля противника, которая может быть получена через метод *getStandartShootingStrategy*. Далее, с добавлением новых объектов «флота» или новых стратегий обстрела противника будут добавляться и методы для их создания в фасад логического модуля.

* 1. ***Модуль для работы с графикой (Graphics Module)***

Основной чертой дизайна данного приложения является полная автономность графики и независимость от нее основного ядра приложения. Взаимодействие графического модуля с ядром приложения осуществляется через прослойку в виде *Game Manager Module –* этот модуль будет описан далее. Таким образом, архитектура приложения подчиняется паттерну **MVC**: Model – ядро приложения, View – графический модуль приложения, Controller – прослойка в виде *Game Manager* модуля. Для упрощения работы с графикой в приложения используется движок *AndEngine*.

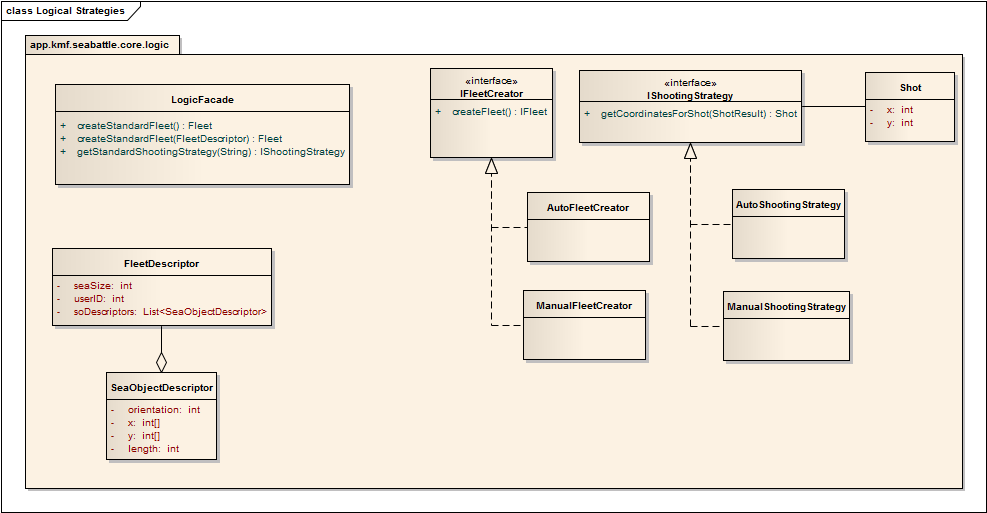


Рис. 4. Структура логического модуля приложения



Рис. 5. Игровые сцены и их взаимосвязи

Графический модуль построен по принципу «многоэкранного приложения». Под «экраном» подразумевается отдельная сцена – меню, настройки, игровое окно и т.п. Каждый «экран» - сцена, являющаяся автономной сущностью, определяющей, какие ресурсы необходимы для нее, какие действия будут происходить при взаимодействии пользователя с «экраном» и т.п. На рисунке 5 представлены основные игровые сцены приложения и их взаимосвязи.

 На следующем рисунке представлены основные *Use Cases* приложения.

На рисунке 7 представлена диаграмма классов и их связей для загрузки и работы с графическими объектами. При старте приложения будут загружены все необходимые ресурсы, в это время пользователю будет показан *Progress Bar*.

Рис. 6. Use Cases

****

Рис. 7. Основные классы и их взаимосвязи графического модуля

Все необходимые сущности собраны в классе *Textures –* он загружает и инициализирует текстуры. Этот класс содержит атлас объектов текстур. Размер атласа должен быть кратен степени двойки – другие размеры OpenGL не понимает, например 1024х512 и все текстуры должны «умещаться» на этом атласе, не пересекаясь друг с другом. Для добавления новой текстуры, которая будет использоваться в сцене, необходимо добавить загрузчик и методы для получения объекта текстуры в этот класс. Класс *Textures* используется *SceneController*’ами. Объект *SceneController* используется *SceneManager*’ом и имеет методы для инициализации и «замораживания» управляемой им сцены. Каждая сцена создается с помощью своего контроллера по методу loadScene(). Так же каждый контроллер сцены имплементирует интерфейс *IOnSceneTouchListener*, что позволяет обрабатывать нажатия на сцену.

Чтобы игра корректно отображалась на устройствах с разными размерами экрана, есть несколько способов:

* Создать постоянные ширину и высоту для объекта камеры и openGL автоматически будет растягивать (сжимать) картинку. Минус – на экранах с другой пропорцией оставшаяся часть экрана будет закрашиваться черной заливкой.
* Использовать метод для автоматической корректировки размера:

private float getScreenResolutionRatio() {  
    DisplayMetrics dm = new DisplayMetrics();  
    getWindowManager().getDefaultDisplay().getMetrics(dm);  
    return ((float) dm.widthPixels) / ((float) dm.heightPixels);  
  }

this.mCamera = new Camera(0, 0, CAMERA\_WIDTH, CAMERA\_HEIGHT);  
final EngineOptions options = new EngineOptions(true,  
        ScreenOrientation.LANDSCAPE, new RatioResolutionPolicy(  
            getScreenResolutionRatio()), this.mCamera);  
    return new Engine(options);

* Писать инициализацию для различных размеров экранов…

Последовательность запуска игры следующая:

1. Происходит инициализация главной активности *AEGameActivity* – класс, наследующий абстрактный класс движка *AndEngine* *BaseGameActivity* и реализует 4 основных метода:

* public void onLoadComplete()
* public Engine onLoadEngine()
* public void onLoadResources()
* public Scene onLoadScene()

Вызываются эти методы в следующем порядке: onLoadEngine->onLoadResources->onLoadScene->onLoadComplete. Метод *onLoadEngine* загружает сам движок и камера приложения. В методе *onLoadResources* выполняется загрузка всех используемых в приложении ресурсов – в этом методе может быть вызвано действие показа индикатора прогресса загрузки. Во время вызова метода *onLoadScene* будут созданы все необходимые объекты сцен с использованием *SceneManager’ов*. В завершении вызывается метод *onLoadComplete*.

1. После загрузки движка по умолчанию устанавливается сцена с главным меню. Менеджер сцен при установке текущей сцены устанавливает ее контроллер в качестве слушателя нажатий пользователя на экран. Каждый контроллер сцены хранит у себя экземпляр класса менеджера и в любой момент может передать управление другой сцене.
2. Пользователь может навигироваться по меню, просмотреть статистику, после нажатия на кнопку «Новая игра» открывается сцена с выбором игрового режима. Пользователь может выбрать игру с *Android*, другим игроком на одном устройстве или на другом через Bluetooth. Следующим этапом пользователю будет показана сцена создания игрового поля – здесь имеется возможность ручного или автоматического добавления кораблей с возможностью ручной корректировки.
3. Все игровые сущности создаются и сохраняются в игровой карте *MapGameObjects*, которая имеет пиксельный размер ячеек (аналог *SeaDrop*) и их количество. Для стандартной игры карта имеет размер = 10х10, и величина ячейки = 50px – 50px. Игровые объекты «занимают» свои ячейки и хранятся в игровой карте. Каждая ячейка имеет x и y координату, что позволяет игровому модулю взаимодействовать с логическим модулем системы.
4. Во время игры нажатия будут обрабатываться текущей сценой, а точнее ее менеджером, и в зависимости от результата нажатия вызываются соответствующие методы у *Game Manager Module*. В зависимости от ответа менеджера игры текущая сцена может изменять свое состояние.
   1. ***Управление и запуск игровых процессов (Game Manager Module)***

Поскольку приложение построено с применением принципа архитектуры MVC – необходим модуль, осуществляющий функцию *Controller*. Этим модулем является *Game Manager* модуль, осуществляющий взаимодействие между ядром приложения и графической частью.

Для взаимодействия графического модуля с ядром приложения используется одноименный класс Game Manager, содержащий основные методы, предоставляющие возможность взаимодействовать с основными модулями ядра. Ключевым моментом является то, что «общение» с управляющим модулем производится через JSON объекты, которые трансформируются в объект *String*. JSON объекты передаются в качестве аргументов методам управляющего модуля и ответ методов происходит в виде JSON объекта, обернутого в строку. Управляющий модуль разбирает пришедший объект в JSON и работает с ним. Для отправки ответа так же собирается JSON объект и трансформируется в строку, которая отправляется в качестве ответа. Такой формат взаимодействия продиктован необходимостью управляющего модуля взаимодействовать с внешними системами. Для работы с JSON объектами управляющий модуль содержит отдельные методы.

Ни рисунке 8 представлена диаграмма классов управляющего модуля. Помимо основного класса имеется класс ExternalClientManager, наследующий класс GameManager. Этот класс предназначен для взаимодействия приложения с внешней системой, например через Bluetooth модуль (будет описан ниже). Важным моментом является система оповещений графического модуля о действиях внешней системы или системы Android. Этот момент проявляется в случае режима игры с Android и через Bluetooth модуль – так как в этом случае графический модуль не вызывает активности для 2-го игрока, и как следствие, не получает информации в ответ. Эта информация будет получена через объект листнера, который будет передаваться и использоваться управляющем модулем.



Рис. 8. Структура управляющего модуля приложения

Ниже представлены диаграммы взаимодействия модулей приложения для 3-х разных режимов работы приложения:

* 2 игрока на одном устройстве (рис. 9)
* игрок против Android (рис. 10)
* игрок против игрока на другом устройстве через Bluetooth (рис. 11)



Рис. 9. Режим работы приложения – 2 игрока на одном устройстве

****

Рис. 10. Режим работы приложения – игрока против Android

****

Рис. 11. Режим работы приложения – 2 игрока через Bluetooth

1. **Дополнительные игровые модули**
   1. ***Модуль взаимодействия с внешней системой –Bluetooth модуль***

 В приложении поддерживается режим работы с внешней системой через канал данных, например с использованием Bluetooth адаптера. На рисунке 12 представлена структура Bluetooth модуля приложения.

Фасадом модуля является интерфейс *BluetoothManager*, который содержит набор необходимых методов для взаимодействия 2-х систем. Реализация для Android системы описывается классом *AndroidBluetoothManager*. Все «тяжеловесные» операции, такие как ожидание подключения, поиск досутпных устройств, передача данных вынесены в отдельные потоки, чтобы работа модуля не вызывала подвисаний игрового интерфейса.

Рис. 12. Структура Bluetooth модуля приложения

Принцип работы модуля следующий:

1. Создается объект BluetoothAdapter, позволяющий взаимодействовать с Bluetooth модулем телефона: включать/выключать модуль, запускать поиск устройств и другие операции.
2. В случае игры через Bluetooth используется 2 устройства, одно из которых выступает сервером, 2-е – клиентом. Задача сервера – открыть соединение и ожидать подключение со стороны другого устройства. Задача же клиента по найденному или полученной MAC адресу сервера установить с ним подключение. Таким образом, сервер будет иметь функцию «Создателя игры», а клиент – функцию «Подключиться к …». После установки соединения между сервером и клиентом 2 устройства могут обмениваться данными по общему каналу.
3. Обмен данными между устройствами производится через объекты *InputStream* и *OutputStream*, которые могут быть получены после установки соединения у объекта *BluetoothSocket*. Передача данных производится через массив байтов byte[]. Таким образом, основной формат общения через объекты String остается актуальным, поскольку объект String преобразуется в массив байтов методом *toBytes()*.
4. Получение данных осуществляется в отдельном потоке в зацикленном состоянии. При этом, для передачи данных в основную активность используется объект Handler. Пример создание этого объекта:

android.os.Handler h = new Handler() {

public void handleMessage(android.os.Message msg) {

switch (msg.what) {

case RECIEVE\_MESSAGE: // если приняли сообщение в Handler

byte[] readBuf = (byte[]) msg.obj;

String strIncom = new String(readBuf, 0, msg.arg1);

sb.append(strIncom); // формируем строку

int endOfLineIndex = sb.indexOf("\r\n"); // определяем символы конца строки

if (endOfLineIndex > 0) { // если встречаем конец строки,

String sbprint = sb.substring(0, endOfLineIndex); // то извлекаем строку

sb.delete(0, sb.length()); // и очищаем sb

txtArduino.setText("Ответ " + sbprint);

}

break;

}

};

};

И пример использования данного объекта при приему данных от другого устройства:

public void run() {

byte[] buffer = new byte[256]; // buffer store for the stream

int bytes; // bytes returned from read()

// Keep listening to the InputStream until an exception occurs

while (true) {

try {

// Read from the InputStream

bytes = mmInStream.read(buffer); // Получаем кол-во байт и само собщение в байтовый массив "buffer"

h.obtainMessage(RECIEVE\_MESSAGE, bytes, -1, buffer).sendToTarget(); // Отправляем в очередь сообщений Handler

} catch (IOException e) {

break;

}

}

}

* 1. ***Модуль игровых настроек (Game Settings Module)***