¬Expected Magic

System Design Document

Version 1.0

Agrell Robert, Larborn Sofia, Runvik Arvid, Tomasson Rasmus

2017

Software Engineering
Chalmers University of Technology
Sweden

Contents

1 Introduction

1.1 Design Goals

This document describes the construction of the ¬*Expected Magic* music game application specified in the requirements and analysis document.

- Immediate visual- and auditory response during gameplay.
- Easily maintained extendable modular design.
- A testable domain model.

1.2 Definitions, Acronyms and Abbreviations

The following list explains some of the terms used:

- ¬*Expected Magic* (Unexpected Magic) - The application name.
- *Voice* - Derived from the musical term, every player plays their voice of the song.
- *Note* - an object containing information about a note.
- *Pianoroll* - derived from the name of the music storage medium. Similar to the original music storage medium consisting of a long strip of paper with notes marked on it [1], and the note editing element called "pianoroll" in Digital Audio Workspaces like *FL Studio* [2], the pianoroll in the ¬Expected Magic game can be described as a "strip" of coordinates with note entities placed on it, and is displayed during gameplay as the camera moves over it.
- `.uxm` - a file format created specifically for the ¬Expected Magic project, for defining a song in terms of musical information as well as metadata.
- *libGDX* - An external library for game development.
- *Ashley* - A *libGDX* lightweight entity framework for designing Entity-Component System architecture.
- Entity-Component System - A software architectural pattern.

2 System Architecture

The application is designed in accordance with the *Entity-Component System* pattern, and uses the *libGDX* framework. It runs on a single desktop computer. The application consists of six top level packages: "model", unaware of *libGDX*, "gameEngine", using model to run the game with *libGDX*, "Observers", a package to facilitate the transfer of information between classes and packages, "main", a package containing the main class of the game, "utils", a

package containing general purpose code that could be useful to other projects, and "gdxUtils", a package containing *libGDX*-related general purpose code. The "model" package contains classes defining how a game round should run and for counting score, and a "song" package responsible for song handling. The "gameEngine" package holds all *libGDX*-related code that drives the game. (For more detail, see "3.1 The gameEngine package").

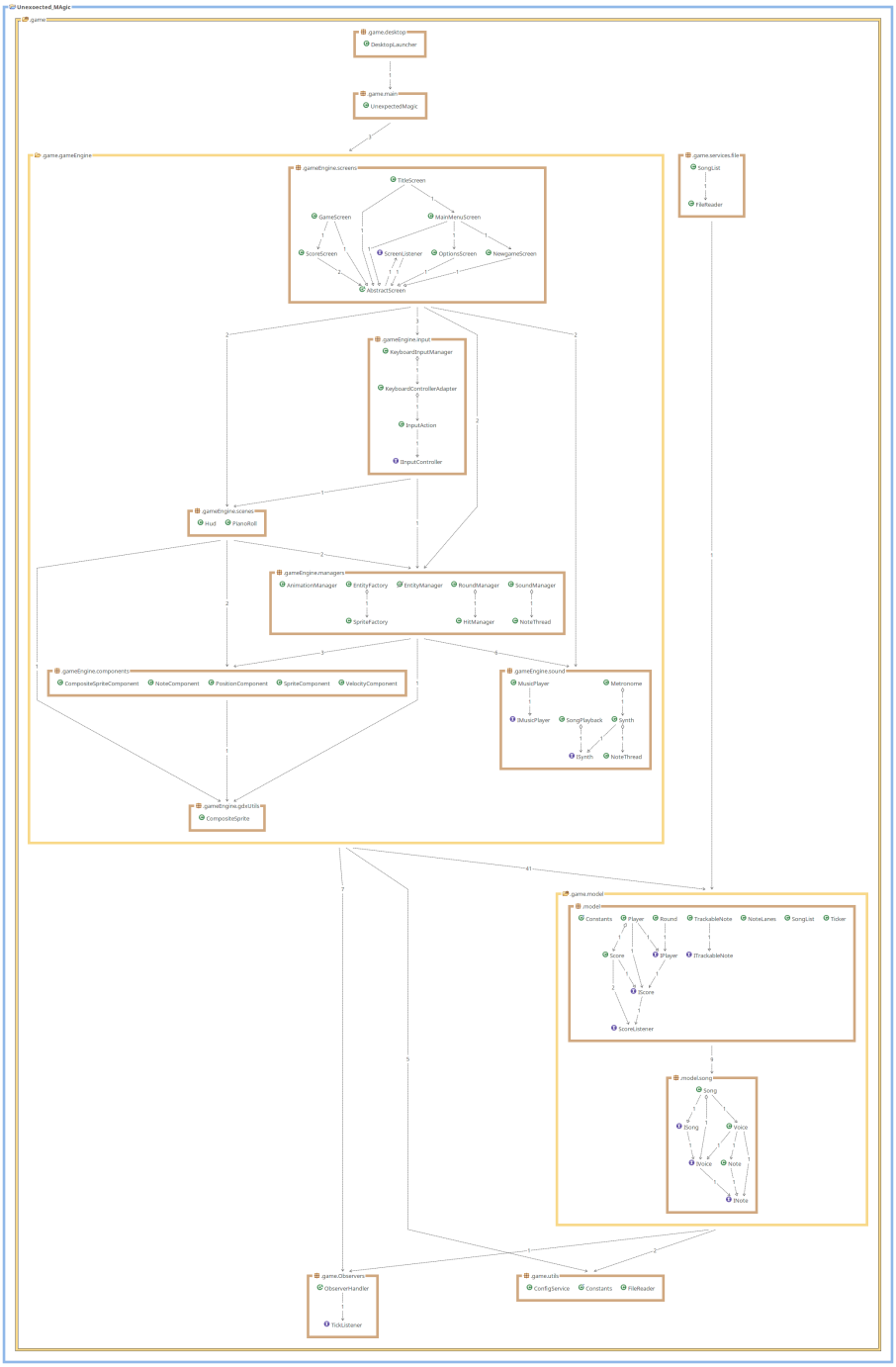


Figure 1: The packages (the arrows show dependencies)

- *Desktop* contains the DesktopLauncher class, which starts the game.
- *Main* contains the UnexpectedMagic class, the game's main class.
- *GameEngine* contains *libGDX*-related functionality and runs the game.
- *Screens* contains the various screens the game will display to the user.
- *Input* translates key presses to commands for the program.
- *Scenes* contains classes defining specific views that are part of a screen.
- *Managers* contains classes managing certain aspects of the program.
- *Components* contains components that can be grouped to create entities.
- *Sound* contains classes whose purpose is to play sounds.
- *Model* defines how the game should be played.
- *Song* model a song.
- *Observers* facilitates communication between classes and packages.
- *Utils* contains general purpose code.
- *GdxUtils* contains *libGDX*-related general purpose code.

2.1 General Observations

2.1.1 The `.uxm` Format

The `.uxm` file format is specifically designed for the ¬*Expected Magic* application. Its purpose is to contain data regarding a song, in a format that can easily be parsed by the game while also being easy for humans to read and write. The data consists of the song title, time signature, tempo and the voices with their notes. Below follows an overview of the syntax:

```
"song title";   %A string that accepts spaces.
4/4;            %Time signature. Numerator can be any int > 0,
                % denominator must be one that also is a
                  power of two.
76;             %BPM (beats per minute). Any int > 0.

%Voice 1
F#5:1/8,C#6:1/8,B5:1/8,F#5:1/8,A#5:3/16,A#5:3/16,B5:1/4;

%Voice 2
E3:1/8,B3:1/8,E4:1/8,B3:1/8,F#4:1/8,B3:1/8,E4:1/8,B3:1/8;
```

Lines starting with '%' are ignored. Whitespace is ignored. Title, time signature and tempo are required. Any number of voices > 0 is accepted. The notes in a voice are separated by either ',' and/or '|', which are interchangeable, and a voice is ended with a ';'. A note is written as: `<pitch><octave>:<note value>`.

2.2 Composition Over Inheritance

When it comes to organizing the game objects, inheritance is a straightforward way to let an object use and override the behavior of a base class. [3]. In Java, only single inheritance is permitted. [4] This eliminates the risk of ending up with the multiple inheritance problem sometimes referred to as "Fork-Join Inheritance" or "Diamond of Death", where a class is derived from two or more classes that share a common ancestor.[5] However, only being able to inherit from one class can give rise to other problems.

In the case of ¬*Expected Magic*, the main game objects are the notes, and plans for further development include different kinds of notes as well as other on-screen entities like player representations and enemies. For example; the `Note` object, `Player` object and `Enemy` object could all be derived from a class game object. What then, if a hostile note was to be added? Would it inherit from note object or enemy object? Since organizing game objects in inheritance hierarchies is proven to be a tedious work likely to end up in limiting rigidity and conflicts, the ¬*Expected Magic* project instead follows the principle of "Composition Over Inheritance". This principle is applied through the use of an Entity-Component System called *Ashley*.

2.3 Entity-Component System

The traditional way of developing games has been to represent entities with objects, which inherit from appropriate superclasses. A common design pattern for use with this is *MVC* (Model-View-Controller). Therefore, this combination of methods was initially considered when planning the implementation. However, relatively quickly, it was noted to be a disadvantage to have separate modules for view and control as well as to structure the parts in Inheritance Hierarchies, as this both of these systems restricted flexibility and made the code difficult to understand and organize.

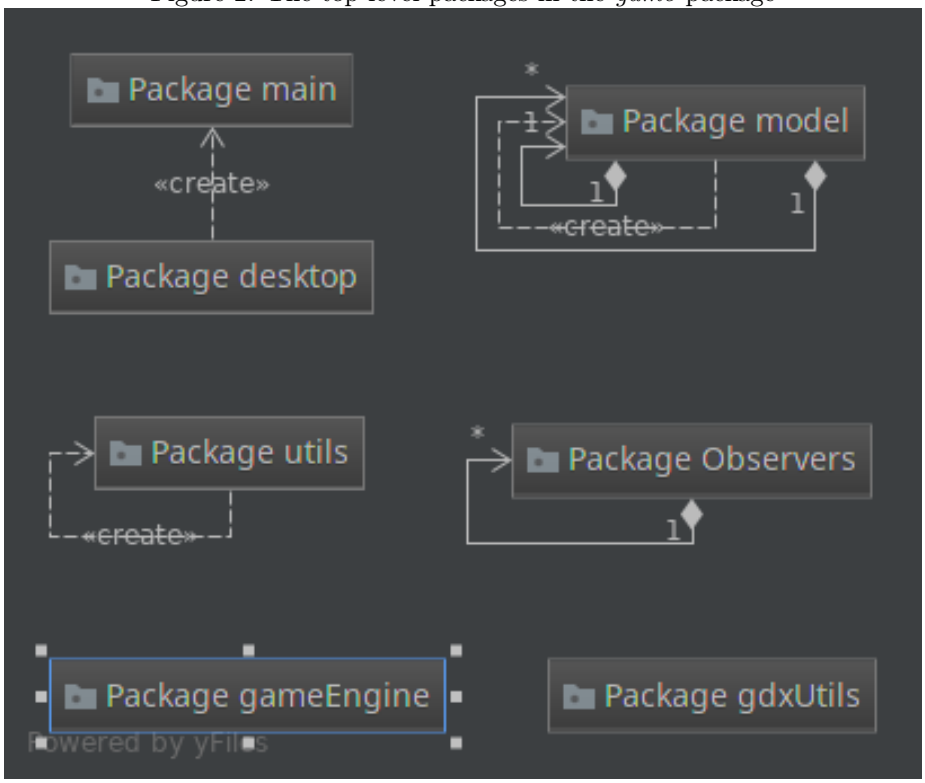
The *MVC* pattern is about splitting the application into a Model that manages the problem domain, a View that presents the model for the user and a Controller that communicates with Model and View. A strength of *MVC* is that the structure enables painless modification of views without affecting the model.[6] However, this flexibility does not give significant benefit to the implementation of ¬*Expected Magic*, as the problem domain's abstract representation and the presented concrete model in this case are very closely related. The kind of separation provided by *MVC* was considered unnecessary and likely to become tedious due to the fact that the user interface is very similar to the model

and that the game is run according to the sequence pattern Game Loop [7] in which the game logic processes the game's different entities in each iteration.

In order to solve this, the principle of Composition Over Inheritance was tried out, which proved more suitable for the ¬*Expected Magic* application. This principle is applied in ¬*Expected Magic* by structuring the application's various objects according to and managing them using the software architecture pattern Entity-Component System. This is done using *Ashley*, a Java Entity System supported by *libGDX*.[8]

3 Subsystems Decomposition

Figure 2: The top level packages in the *game* package



3.1 The *gameEngine* package

The *gameEngine* package contains the following packages:

- components

- managers
- scenes
- screens
- sound
- input

3.1.1 components

Contains component classes that implements the *libGDX Ashley* interface "`Component`". The sole purpose of a component is to hold a single kind of data, a `PositionComponent` for example, could hold a position, which might be implemented as `float` values for x and y. Instances of components can be grouped together to form an "`Entity`". The components used in the ¬*Expected Magic* game are as follows:

- `PositionComponent` - Values for x and y position.
- `CompositeSpriteComponent` - A drawable composite of *libGDX* "Sprites", images that can be drawn on the screen.

3.1.2 managers

Contains classes that manage different aspects of the program The managers used in the ¬*Expected Magic* game are as follows:

- `EntityFactory` - Constructs and returns entities.
- `SpriteFactory` - Construcs `CompositeSpriteCompontent`s.
- `RoundManager` - Handles a game round. Manages a `Round` and a `Ticker`.
- `HitManager` - Handles hit logic by telling the `Synth` what note to play and determining if a player should get score.
- `AnimationManager` - Handles animating an animation sheet.

3.1.3 scenes

Contains classes that define scenes that can be rendered. The scenes used in the ¬*Expected Magic* game are as follows:

- `Hud` - Defines the heads-up display that overlays the game visuals with labels.
- `PianoRoll` - Defines the area showing the pianoroll with the falling notes.

3.1.4 screens

Contains the screen classes controlling and drawing the different screens. All ¬*Expected Magic* screens are derived from the *libGDX* `ScreenAdapter` class which implements the *libGDX* interface `Screen`. The screens used in the ¬*Expected Magic* game are as follows:

- `AbstractScreen` - Screen superclass. Contains shared functionality, for example camera, viewport, resizing, etc.
- `MainMenuScreen` - The main menu screen, presenting the user with options such as playing, changing preferences, etc.
- `NewgameScreen` - Screen that is shown when the user chooses to start a new game from main menu. Presents options for selecting song, players, etc.
- `GameScreen` - Screen that handles the in-game activity. On this screen the pianoroll and heads-up scenes are drawn.
- `OptionsScreen` - Screen that shows options that the user can change. Can be accessed from main menu.
- `ScoreScreen` - Screen displaying player scores after a song is played.
- `TitleScreen` - Title and credits screen that is displayed when launching the application.

3.1.5 sound

- `ISynth` - Interface for the `Synth` class.
- `Synth` - Handles playing notes using the `javax.sound.midi` synthesizer.
- `NoteThread` - Plays a specified note for a specified time.
- `SongPlayback` - Plays any voices not assigned to a player during gameplay.
- `Metronome` - Plays constant beat according to current song.
- `IMusicPlayer` - Interface for the `MusicPlayer` class.
- `MusicPlayer` - Handles playing of sound files for menu music.

3.1.6 input

- `KeyboardInputManager` - Handles input from the keyboard. Sends information to `KeyboardboardControllerAdapter`.
- `KeyboardboardControllerAdapter` - Translates input from `KeyboardInputManager` into an action and sends information to `InputAction`.

- `IInputController` - Interface for an `InputController`, in this case `InputAction`.
- `InputAction` - implements `IInputController`, last layer in the input handling system, sends instructions to the `RoundManager`.

3.2 The *model* package

Defines how the game should be run. The model classes used by the program are as follows:

- `IPlayer` - Interface that defines a player object.
- `Player` - Representation of a player.
- `IScore` - Interface for score calculation.
- `Score` - Class that calculates score.
- `ITrackableNote` - Interface for trackable notes.
- `TrackableNote` - A note with non-static fields.
- `ScoreListener` - Interface for classes that wish to listen to Score.
- `Round` - Class that represents a round.
- `Ticker` - Class that translates time into ticks.
- `SongList` - Keeps track of the songs the game can play.

3.2.1 The *song* package

This package contains classes defining a song, they are as follows:

- `ISong` - Song interface.
- `IVoice` - Voice interface.
- `INote` - Note interface.
- `Song` - Song class, made up of several voices.
- `Voice` - Class that represents a voice, contains notes.
- `Note` - A specific note.

3.3 The *Observers* package

This package contains classes that facilitate observation of classes by other classes, it contains the following classes:

- `ObserverHandler` - Class for managing observers.
- `TickListener` - Interface for classes that wish to listen for game ticks.

3.4 The *utils* package

Provides tools for file reading, input interpreting, etc. The utility classes used in ¬*Expected Magic* are as follows:

- `Action` - Enum used by `ConfigService`.
- `ConfigService` - Used for keeping track of input configuration.
- `Constants` - Holds global constants such as viewport dimansions.
- `FileReader` - Reads and parses `.uxm` files to a format used by the model.

3.5 The *gdxUtils* package

Package for helper classes that are dependent on *libGDX*. The *libGDX*-dependent helper classes used in ¬*Expected Magic* are as follows:

- `CompositeSprite` - A composite sprite made up of multiple *libGDX* `Sprite`s.

3.6 The *Main* package

simply contains the desktop launcher class

- `UnexpectedMagic` - The desktop launcher

4 Persistent Data Management

¬*Expected Magic* does not currently have much persistent storage, but it does read files of a format called `.uxm`. These files are used in the creation of the `Song` objects that contain all information about the songs that are available for playing in the game. Plans for further development include a persistent high-score list.

References

[1] U. Nankipu, "Piano roll." [Online]. Retrieved from: http://www.bbc.co.uk/ahistoryoftheworld/objects/fl0uHkdfQkOX31MkZwKJqQ, 2014.

[2] F. Studio, "Piano roll." [Online]. Retrieved from: https://www.image-line.com/support/FLHelp/html/pianoroll.htm.

[3] T. J. Tutorials, "Inheritance." [Online]. Retrieved from: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html, 2015.

[4] T. J. Tutorials, "Multiple inheritance of state, implementation, and type." [Online]. Retrieved from: https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html, 2015.

[5] B. N. J. P. V. Eddy Truyen, Wouter Joosen, "A generalization and solution to the common ancestor dilemma problem in delegation-based object systems." [Online]. Retrieved from: https://people.cs.kuleuven.be/~eddy.truyen/PUBLICATIONS/DAW2004.pdf, 2004. Chapter 2. The common ancestor dilemma.

[6] A. Inc., "Model-view-controller." [Online]. Retrieved from: https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html, 2012.

[7] R. Nystrom, "Game loop." [Online]. Retrieved from: http://gameprogrammingpatterns.com/game-loop.html, 2014.

[8] libgdx, "Ashley." [Online]. Retrieved from: https://github.com/libgdx/ashley, 2017.