System Design Document for NUCLEUS

Version: 1.0

Date: 160529

Authors: Erik Jansson, Herman Carlström, Miranda Bånnsgård, Andreas Erlandsson, Hannes Lagerroth

This version overrides all previous versions.

1 Introduction

The mobile and desktop application Nucleus is a level-completion-based game in which the player strives to build a given molecule by catching flying protons and neutrons. The player rotates said molecule in order to intercept the incoming particles. Due to the game's simple and intuitive nature it should be designed in a way to maximize ease of modification for developers in order to keep the game interesting.

1.1 High level design goals

The design should be strictly according to the MVC design paradigm. This is particularly important due to the fact that the application will utilize the cross-platform libGDX game engine. The model should therefore exist independently from the View and Controller-based classes which use the libGDX library in order to offer enough flexibility to easily be redesigned using another game engine.

The modules should have loose couplings to each other, but a strong internal coupling. The classes, functions and code should be self-explanatory in their nomenclature.

Model classes will be designed in compliance with dependency injection principles in order to maintain easily testable code using mock classes. The design should follow the Dependency Inversion Principle, with a continuous programming towards interfaces instead of classes.

The dependency injection together with extensive use of interfaces will allow for testing of the model with JUnit. Over 90% coverage of the model is preferred.

1.2 Definitions, acronyms and abbreviations

libGDX - Game development framework which enables games to be played on several platforms with the same core model.

Nucleon - An umbrella term for the subatomic elements, protons and neutrons.

Gluon point - Attachment point that holds on nucleons. The aim of the game is to fill these gluon points with said nucleons.

Molecule - The molecule is a rotated by the player and made to intercept flying protons and neutrons. The molecule is made up of a number of gluon points, which catch the flying nucleons upon colliding with them. Each different molecule constitutes its own level.

MVC - A software architectural pattern that divides the software application into three interconnected parts (Model, View, and Controller), with the goal of separating internal representations of information from the ways that information is presented to or accepted from the user.

IDE - "Integrated Development Environment" is a software program which provides with a working environment/facility for software development.

Android studio - the official IDE for Android platform development.

Gradle - Build tool for dependencies, well integrated into the Android Studio IDE.

Jacoco - Jacoco is a Java Code Coverage tool which allows one to find out which parts of the code is executed.

2 System design

2.1 Overview

The application will be designed accordingly to the MVC-design pattern, with usage of the libGDX game development framework. A *GameScreen* class implementing libGDX's *Screen* interface will be used to call a timed update to the model classes as well as render the model data. The clock "ticks" will ensure a smooth and standardised flow of in-game time. A

controller will be able to change model data while other controllers will handle the less complex menu-, level-, win-, lose- and pause-screens.

The application dependencies are handled with Gradle simply due to the fact that Gradle is well integrated into the Android development environment.

The application handles reading both visual files and sound files as well as writing to the system in order to locally keep track of the player's progress.

2.1.1 MVC

The design follows the Model-View-Controller pattern and the majority of the application is divided into three major modules; *model*, *view* and *controller* (see Appendix Figure 1). The *model* module consists of the game logic and all that is needed for the game itself to function. The *view* module is responsible for the visuals of the application. The *view* module observes the model and renders it's current state. At each game tick the view checks the model again, and renders it. The *controller* module handles all interaction with the user, and alters the model accordingly. The modules contain submodules, which have been organized according to package-by-feature.

Three other separate subsystems are utilized by the main application and will be discussed in section 2.2.4.

2.1.2 Design Patterns

The application uses a number of design patterns and follow acknowledged design principles.

- The **Singleton** design pattern is used for the *NMusicPlayer* in the *libGDXMusic* package and for the in-game controller *GameInputHandler*. The singleton prevents instantiation through a private constructor, and instead returns the instance of the class that already exists. If no instance exists, it creates a new one and returns it. The

singleton is used to prevent the classes (of which only one instance is necessary) from being instantiated several different time, and in the case of the *NMusicPlayer* to keep track of which song is actually playing. For the *GameInputHandler*, we only want one to exist, and alternate it between states.

- The **Observer** pattern is used in controlling the different states of the game. Classes observing the *Level* class are notified when the game is paused, won, lost or resumed from a paused state. The Observer pattern is also utilized in order keep track of player progress in the *ProgressTracker* controller. New data is written to local system files when the player conquers a new level. The normal built-in Observer found in the *java.util* library is not used due to the type insecurity associated with simply passing *Object* classes as parameters. In order to enforce stronger typing we instead implemented our own Observer and Observable interfaces located in the *nucleusObservers* package.
- The *GameInputHandler* utilizes the **State** design pattern to modify the behaviour its behaviour. The *GameInputHandler* reacts differently to input depending on what state is currently active.
- The **Dependency Inversion Principle** is used thoroughly in the whole application, with a continuous implementation towards interfaces instead of classes. Allows for easier change of code, and modularity where it is easier to add new functionality or replace modules.
- The **Object Pool** pattern is used in the *Assets* class. *Assets* acts as a collection of textures and music from which the views classes retrieve single instances of the needed objects. This makes for effective use of memory.
- The *LevelBuilder* package, as the name implies, uses the **Builder Object** pattern. The package uses a standardized set of steps to assemble components and build the different levels which can vary greatly in appearance, difficulty and play characteristics, and delivers these to the game.
- Specific functionality for how a .txt file is loaded into the *AssetManager* is required by the *levelBuilder* module. In order to properly handle this we have a **Decorator** class, the Text class in the *AssetManager* package, to alternate the behaviour of the

String class for this single purpose. This allows for our own implementation of the class at specific needs, but does not alter the String class for other usages.

2.2 Software decomposition

The application is launched from a platform-specific launcher class, after which it enters the rest of the platform-independent application. The *NucleusGame* class creates the bare necessities of the game; a *ButtonEventHandler*, loads the necessary game assets using the *AssetHandler* package, and creates the "master" *ApplicationController*. From here the application controls the game flow from Screens to the actual playable levels. The player can navigate through the game's menus (held in the *screen* classes, with popup windows held in the *dialog* package) with the options to toggle sound and rotation direction of the molecule (see Appendix Figure 2).

When the player selects a playable level in the *LevelScreen* the application builds the appropriate level and hands the control over to the *GameInputHandler* and its currently active state. The game's *screen* classes implement the libGDX interface *screen*, which requires the method *void render(float delta)*. The float *delta* is the time elapsed since the last call of the *render* method and therefore functions as the game's internal clock which updates and keeps the level ticking along. The game's current state is notified to listeners in the *ProgressTracker* (which keeps track of player wins and progress), as well the views and dialogs responsible for showing pause screens, congratulatory winning screens and lose screens. The rendering of visual elements in the game, such as the flying nucleons or the rotating molecule is handled by separate classes in the *viewables* packages.

2.2.1 Model

The *model* module consists of five game engine-independent submodules categorized by the functionality and logical grouping of the classes included, and each submodule has its own

area of responsibility. (See Appendix Figure 3 for a detailed UML representation of the model classes)

- model.collisions - handles the collision between two *ICollidable* objects (an interface that both *GluonPoint* and a *Nucleon*) and contains the *Vector* class which is used throughout the application as the holder for positions and velocities in a 2D-environment.
- model.level - the "top-most" module containing the game logic concerning the level and the *nucleonGun*.
- model.molecule - contains the entire molecule and gluon point game logics.
- model.nucleusObservers - contains the specially developed *IObserver* and *IObservable* interfaces
- Model.particles - contains the game logic for the nucleons.

2.2.2 View

The view package consists of both graphical and non-graphical views. The subpackage *libGDXGraphics* contains three subsubpackages; one for the dialogue windows, one for the screens and one for the viewables (the graphic assets which are rendered during gameplay). These three subpackages are responsible for visual aspects of the GUI.

The other subpackage in Views is libGDXMusic. It is a "non-graphic view", which represents the music player and is responsible for the different sounds and songs played when interacting with different graphical views.

2.2.3 Controller

The controller packages consists of all the classes that delegate user input. This inlucdes listeners for buttons. *NucleusGame* extends the *Game* class of libGDX, and is responsible for starting and quitting the application. All input from buttons goes through *ButtonEventHandler***,** which in turn delegates all actions to *ApplicationController*, the main

controller of user input. This class handles everything that is not directly related to playing the game, such as music, menus, and the starting of levels. For direct of the game, *GameInputHandler*, implementing a *NInputProcessor*, handles all game inputs. GameInputHandler in turn holds two states, a *NormalPlayState* and an *InvertedPlayState* for normal and inverted play respectively.

2.2.4 Decomposition into subsystems

We have three separate subsystems, two of which have been separated from the three main modules.

The subsystem *levelBuilder* and its sister-subsystem a*ssetHandler* is the basic solid foundation on which the application is built. The a*ssetHandler* loads and holds onto graphic assets, sound assets and perhaps most importantly, the level data text files.

The long-term success of popular mobile applications such as Candy Crush and Angry Birds can arguably largely be attributed to their seemingly never-ending number of short, simple levels. In an attempt to follow in the footsteps of these games, an easily expandable level system was strived for. The only things needed to add another level to the game is a new graphic asset for the molecule to be built and the raw text data for the corresponding level. A new level can therefore be added in under 20 minutes and with no “hard coding”. The *assetHandle*r automatically loads these level-specific assets and sends these to the levelBuilder that constructs each level with the appropriate game components as per the data in the text file loaded by the *assetHandler*. The non-level specific graphic assets that are used throughout the game and in menus are loaded at startup to minimize in-game loading times, a technique called pooling. Due to how the level data is read from files within the game’s asset folder, the above system could easily be adapted to load the levels from a central database.

The last subsystem is the *collisions* package which contains the basic vector mathematics and collision detection used throughout the model and on which the game mechanics are based.

The *collisions* package is general enough to be used in any two-dimensional application model in which collisions are detected by radial proximity.

2.2.5 Layering

The layering follows a strict hierarchical design, where a module in the chain has no knowledge of parents. According to the same logic the module on the top of the hierarchy has knowledge of all its children. From this follows that the Controller has knowledge about the Model and the View, but the Model has no knowledge of either.

2.2.6 Dependency analysis

There are no circular dependencies between the packages and none inside, with the exception of the *IObservable* and *IObserver* interfaces in the model package. This is a deliberate design decision due to the tight logical coupling of these two interfaces (there is no point in have an observer with nothing to observe for example).

2.2.7 Tests

A separate package outside the application is used for unit testing the game's model classes. The testing is done using the assertions of the JUnit testing library. In compliance with dependency injection and the superior code quality and testability that follows, very few objects are created within the model classes. Any classes that an object will need is usually passed to it in the constructor typed as an interface. This makes it possible to create dummy mock-classes with very predictable and simple behaviour to pass into a class that is to be tested. The class can then be tested in a fully isolated and controlled virtual "lab" environment with minimal dependency on other complex classes.

JaCoCo is used to control the test coverage of the model.

2.3 Concurrency issues

As the application runs on a single thread, there are no concurrency issues.

2.4 Persistent data management

In the current version of the application the game keeps track of which levels the player has completed with the use of libGDX's *Preferences* class. This platform-independent solution creates a ~/.prefs directory in which it stores player progress. The *Preferences* class is an elegant solution to save simple local data using what essentially works as a HashMap with a key, "progress", and the value being the highest completed level. This solution could be expanded to hold such things as a player highscore, achievements or other details in the next iterations of development.

2.5 Boundary conditions

Assets are currently not being disposed of in an efficient manner. This is not a problem for this simple three level version of Nucleus but must be revised as the application grows in size and complexity. The consequences of this dodgy asset handling is that loading and playing a multitude of levels in one play session would clog up system memory, not least in mobile phones.

3 References

APPENDIX

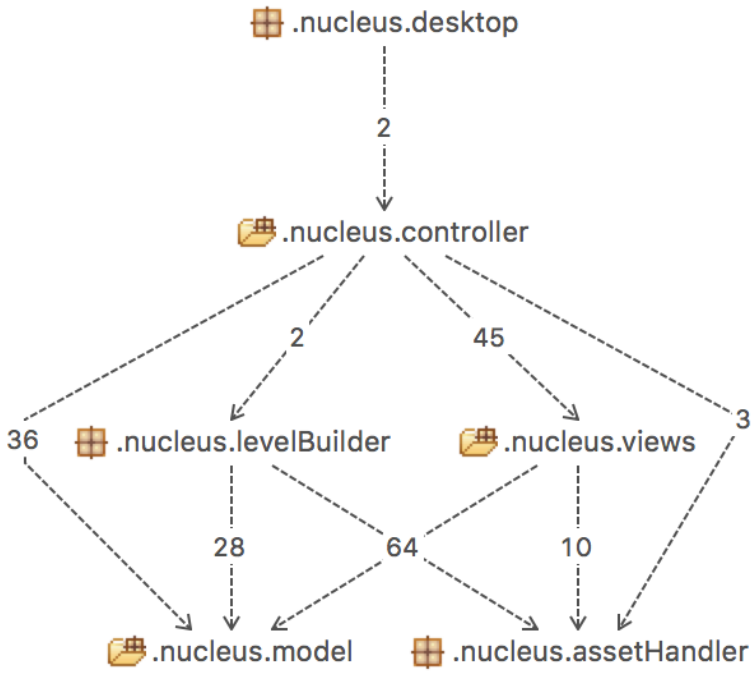

Figure 1. An overview of the packages in the nucleus application taken from a STAN analysis

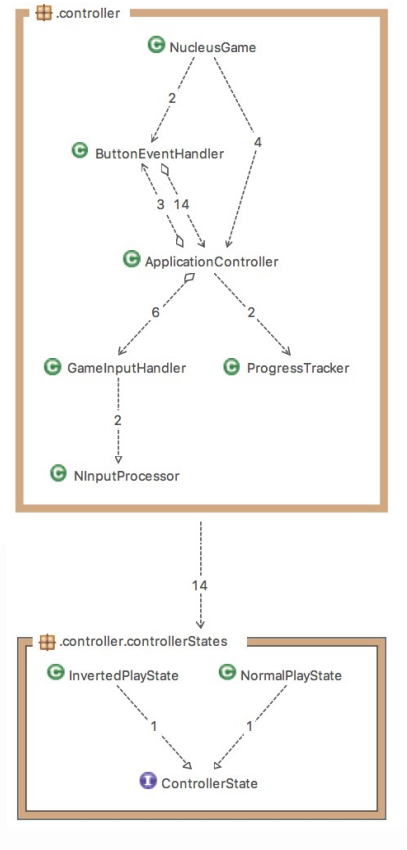

Figure 2. A detailed view of classes in the controller module

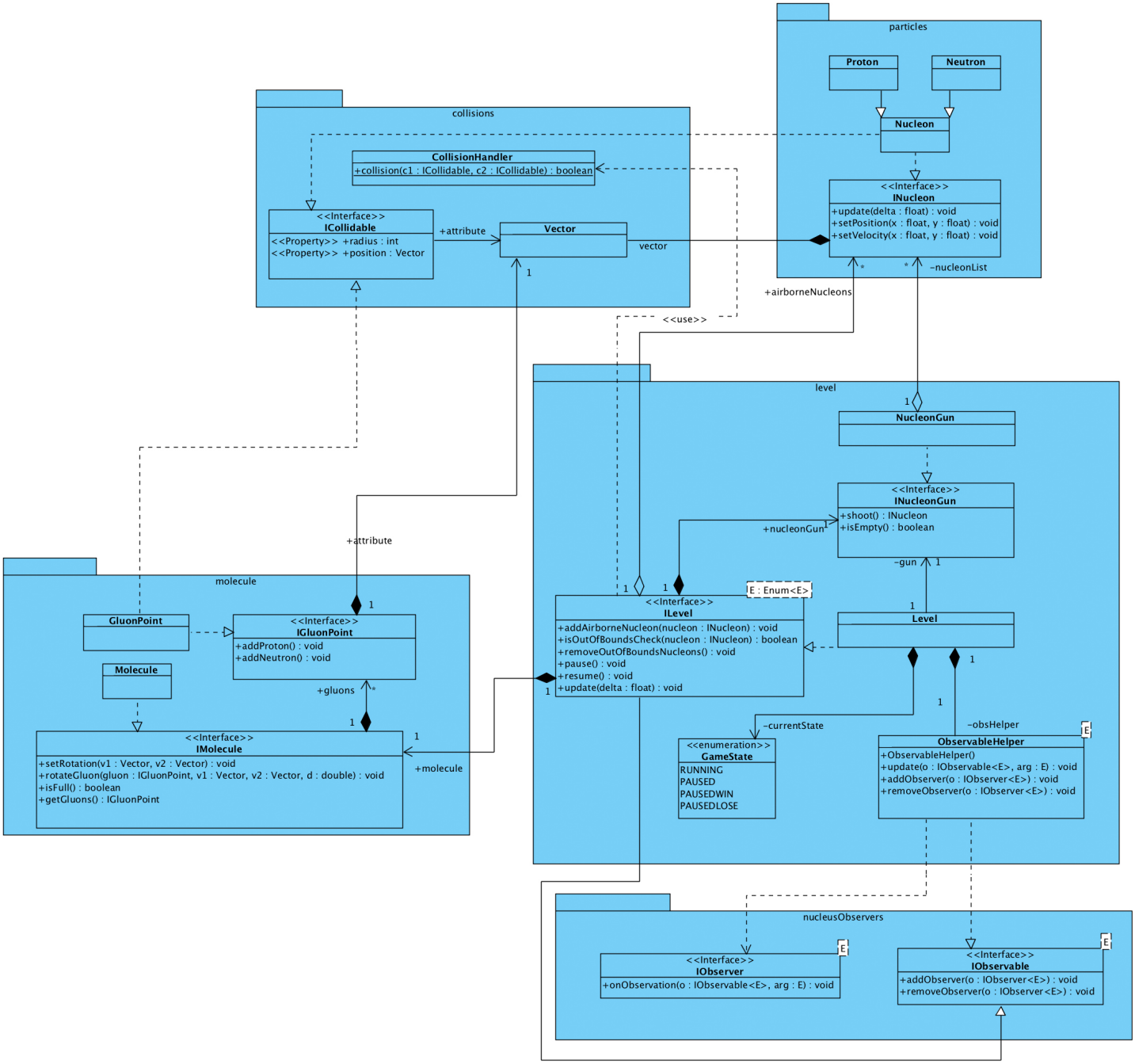

Figure 3. A detailed UML class diagram of the model subpackages and modules