System Design Document

for

Version: 1.2

Date: 2017-05-29

Author: Philip Nord, Julia Ortheden, Jonas von Essen, Fredrik Viberg

This version overrides all previous versions.

1 Introduction

This document contains a technical description of the construction of the application Not-Enough-Space, a 3D-game for PC located in the atmosphere of the Earth. The application is based on two-minute rounds and constructed for one player.

1.1 Design goals

Some design goals:

- The design must be testable and it should be possible to test just specific parts of the application.
- The application must have an MVC structure with a model without framework dependencies.
- The application will be made for Windows/Linux/Mac, standalone, and single player mode.
- A graphical user interface and in-game HUD will be a requirement.
- The application must have 3D graphics and navigation.
- It must be possible to add new game modes with relative ease.

1.2 Definitions, acronyms and abbreviation

All definitions and terms regarding the Not-Enough-Space game can be found under References.

Round - one round inside the game.
Program - the application itself.
Phase/Part - used interchangeably and refer to the two different phases of the game.
Beam - “Tractor beam”, the classic UFO abduction beam.
jME/jME3 - JMonkey Engine, the framework used to develop the application.
NiftyGUI - A GUI framework integrated in jME.
MVC - Model-View-Controller pattern.

See RAD document for further non-technical definitions etc.

2 System architecture

2.1 Overview

The application is built around a clear MVC-structure. The model package is completely independent from the framework, and contains all data, behaviours and calculations. The model doesn’t care how it’s presented, and communicates with the view and control through an event bus and an interface called PlanetaryInhabitant (PI). The PI contains method declarations that cover all functionality related to in-game 3D entities that the model requires from the framework, without relying on specific implementations. To make the application work, there is a thin control

layer that handles communication between the PIs and the view (the 3D scene, could also be regarded as the jME model). The control package also contains a jME-specific implementation of PI.

To clarify, the model doesn't actually keep track of entity positions. The only reference from the model to the 3D space is the PI, from which positional data can be retrieved and modified.

2.2 Model - Control relation

While the model is independent from the framework, it doesn't contain a clear entry point for control. This is due to the haunting fact that the model can not actually run without *some* framework, since the model doesn't handle 3D calculations or the update loop on its own. The result of this is that the model needs to be controlled in very specific ways, by a thin but wide control layer. The controller needs to call the update method in each model entity individually, and it needs to assign PIs to each new entity created (through EntityCreatedEvents). The framework also needs to supply collision detection events, between *some* of the entities but not all. The reason for this is that the model shouldn't mimic the functionality of jME and just hide it behind an abstraction layer - it should ask only for the help it actually needs. Both the PI interface and the event package are created through the eyes of the model, leading to true framework independency.

2.3 Administration

Controlling which and whether a game mode is active, exiting the application, and pausing the game are all administrative tasks. These are not in any way handled by the model, and as such are left to the framework implementation. This is currently handled by a StateManager, which controls and synchronizes AppStates that handle NiftyGUI screens and game initialization. The different states of the game are menu, in-game and paused. There is currently only one game mode, but the code supports adding more modes. It should be noted, however, that most entities are created for the current game mode, and would only be reusable in a very similar mode.

2.4 Dependency analysis

Not Enough Space is decomposed into the following packages (See figure 1).

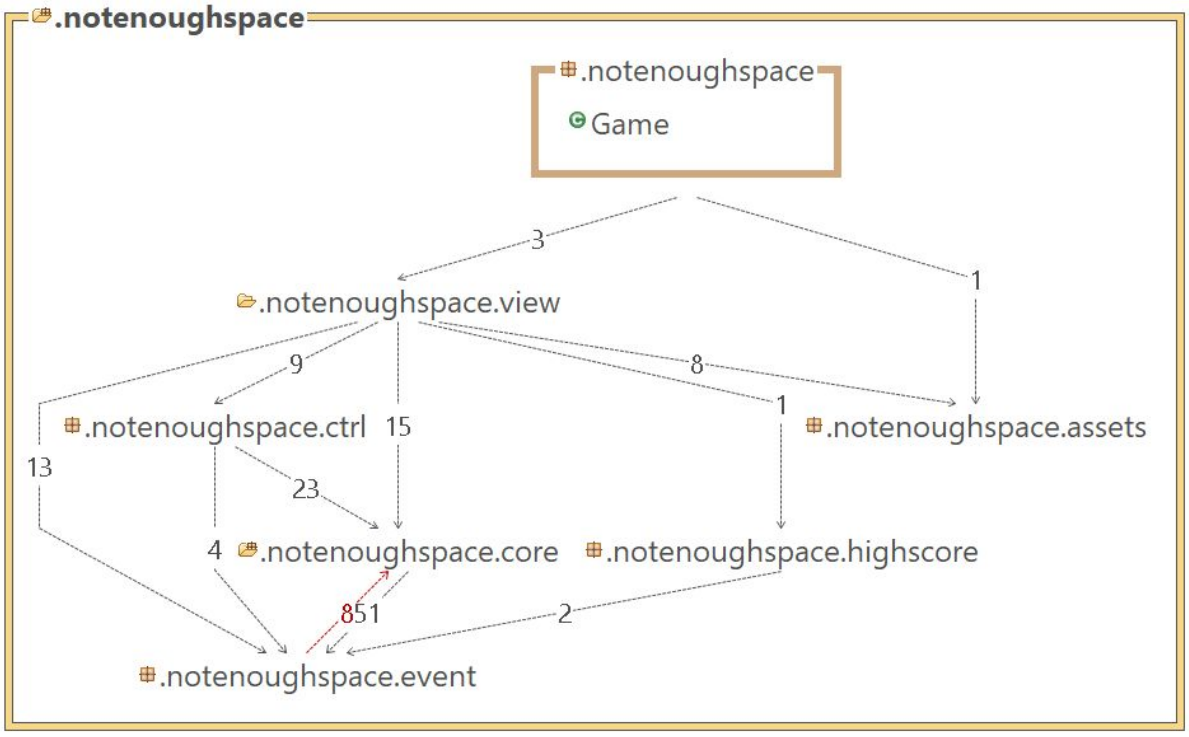


Figure 1: Package diagram for Not Enough Space including the dependencies in between the packages. The circular dependency between core and event is considered harmless, as the events need to pass on data or references from the core. As such, the event package could be considered part of the core.

3. Subsystem decomposition

The package decomposition is mainly centered around the MVC structure, with the core, view and ctrl packages representing the model, view and controller respectively. Additional packages represents certain services, such as asset loading or highscore managing.
See Appendix for subpackage UML class diagrams.

3.1 Core

The core package contains all of the game logic implemented by the application. This package is independent of framework and the rest of the application, relying solely on the event package and the *PlanetaryInhabitant* interface in order to manipulate objects in a 3D-space. The main class of the Core package is *Level*, which is the default game mode. The *Level* spawns a *Ship* and a *Planet*, while also creating an *EntitySpawner* associated with the *Planet.* The *EntitySpawner* makes sure that the planet stays populated with cows, junk, satellites and powerups. The update loop in the framework needs to call the *update()* method in *Level*, but this method only updates the timer and *EntitySpawner*, not the entities. When the timer is up, or the

ship runs out of health, the game is over and a *GameOverEvent* is fired.

3.2 Control

The control package includes all of the framework-specific controllers used to hook the entities in the core package to the update loop. It also includes the concrete implementation of the *PlanetaryInhabitant, JMEInhabitant,* which is a wrapper of the *Spatial*-class used by the framework. Each control calls the *update()* method in its corresponding *Entity*, as well as specific collision related methods. The control is also responsible for handling animations on the *Spatial* in the scene, and controlling any audio related to the entity. Animations and audio are not represented in the core package.

The control package is responsible for checking collision, utilizing framework functionality. Doing real mesh collision checking seemed like the more elegant option, but in hindsight it could be considered too realistic for such a cartoon-themed game. A simpler solution would have been preferable, such as using distances to calculate collision inside the core package. This would have saved a lot of time, and resulted in a more independent model as well as a better looking product.

3.3 Event

The event package contains all of the possible game events and an event bus singleton of the Google Guava *EventBus*. *EventBus* lets any instance subscribe to certain event types posted to the bus. The core package posts events to the bus whenever it needs the framework to do someting (such as creating a spatial with an *EntityCreatedEvent*, or ending the game with a *GameOverEvent*). The core package also listens to some of the events, but the main purpose of the event bus is to communicate with the framework.

3.4 View

The view package consists of both the state package and the scene package, corresponding to the different levels of views included in the game.

The state subpackage consists of the framework specific app states and a state manager used to customize state handling logic in our program. The two states consists of *Menu* and *Round* which both share the same Nifty-object for handling GUI and HUD images and buttons.

The scene package contains the major class of the view-package, *SpatialHandler*, which connects the core and control packages with the scenegraph of jME. This is mainly done through *EntityCreatedEvents* and *EntityRemovedEvents*. *EntityCreatedEvents* are responded to by creating a *Spatial*, attaching a *Control*, assigning the *Entity* to the controller, and assigning the spatial (encapsulated in a *PlanetaryInhabitant)* to the entity. *EntityRemovedEvents* remove

the spatial from the scene, detaches the control and cleans up any other artifacts, such as audio or effects.

The *SpatialHandler* is a very hard coded class, with methods consisting of large if-else blocks. This is definitely bad practice, since adding new entities to the game requires adding them to the if-else list in *SpatialHandler*, and creating a control class (even if its only function is passing on the *update()* method). For a larger application, this process should be more or less automated, possibly with relevant parameters in the entities and generalized controls. For a small-scope application, the *SpatialHandler* gets the job done well enough.

3.5 Assets

The asset package is a service package for streamlining the asset loading process throughout the program. It uses a singleton pattern where *AssetLoaderFactory* will hand out the static instances of certain asset loader interfaces, from which framework specific asset objects can be requested using String IDs. As seen in figure 1, this package is mostly used by the view, as the assets of the game consist of audio/visual objects such as textures, 3D-models and sound effects. For a larger application, storing stray asset data such as scaling and initial rotation inside an AssetLoader class, as well as using String IDs, can be considered bad practice. Storing all asset data in their respective files would result in easier managing, but it also requires full control of the contents and loading of these files. For an application with a smaller scope, like this one, constructing such a system is unnecessary.

4. Persistent data management

The Assets package makes use of the jME framework to store and load asset files. The files used are 3D-models for the jME view and their associated textures and materials, as well as GUI-related images. NiftyGUI uses .xml-files to load the different screens.

High scores are stored in a .dat-file using standard Java IO functionality. These are loaded on application launch, and stored every time a new high score is added to prevent data loss. The scores are, however, stored directly as an *ArrayList* object containing instances of a *Score* class, which means that the saved data is dependent on the current *Score* version and loading it might not work in the future. Improving this by storing the data as text or plain bytes could be considered important, but it’s beyond the scope of the application.

5. Access control and security

No different roles exist.

6. References

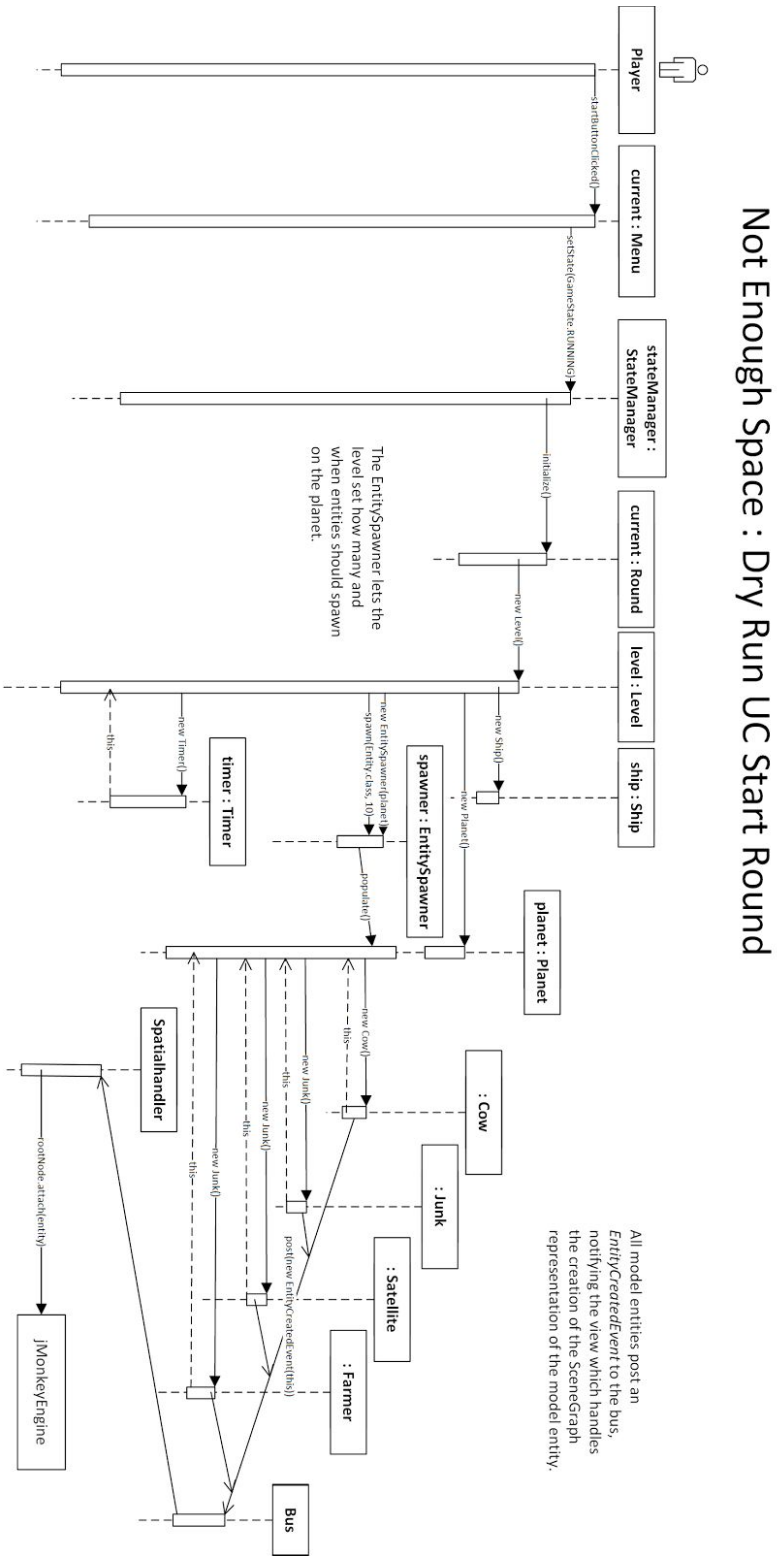
jMonkeyEngine (2016). *jMonkeyEngine Documentation.* Accessed from: https://jmonkeyengine.github.io/wiki/ at 2017-05-26

J. Hohmuth, M. Karing (2012). *Nifty GUI: The missing Manual.* Accessed from: https://vorboss.dl.sourceforge.net/project/nifty-gui/nifty-gui/1.3.2/nifty-gui-the-manual-1.3.2.pdf at 2017-05-26

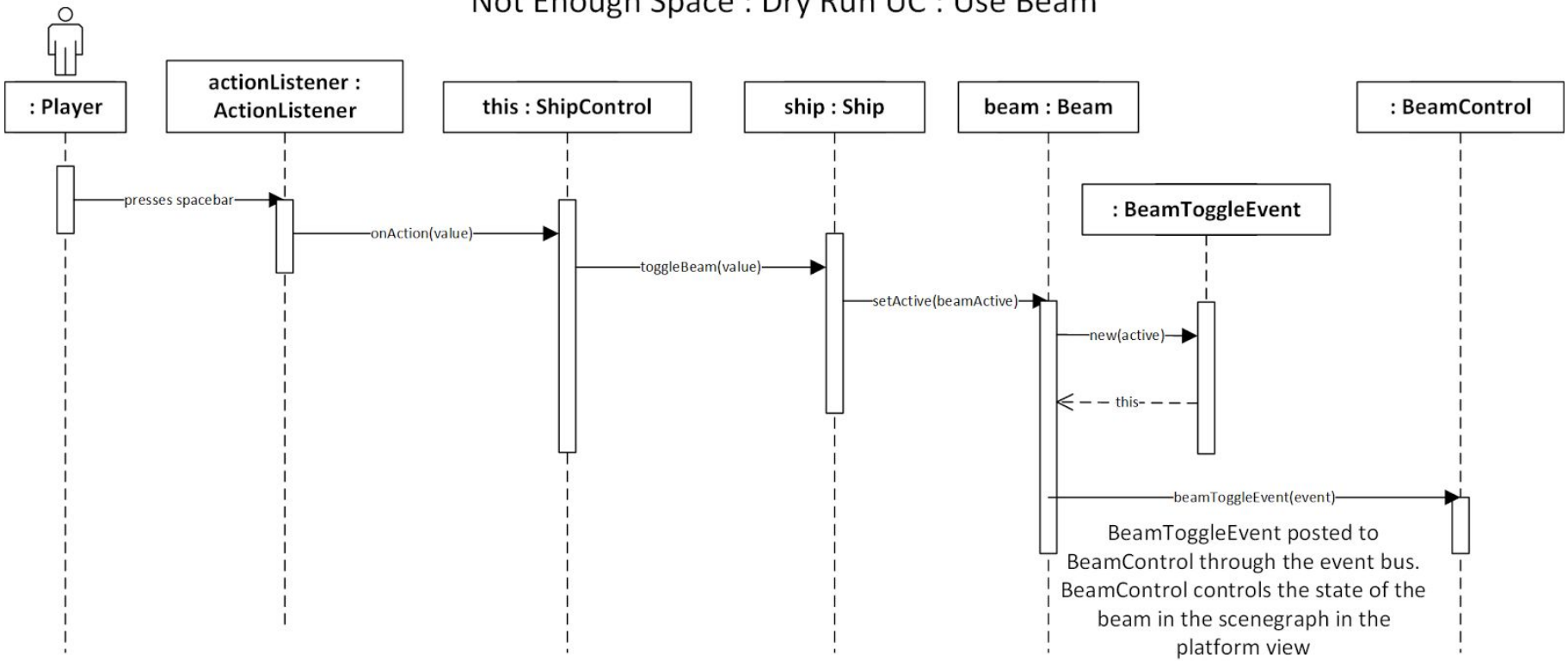
Wikipedia (2017). *Scene graph.* Accessed from: https://en.wikipedia.org/wiki/Scene_graph at 2017-05-26

APPENDIX

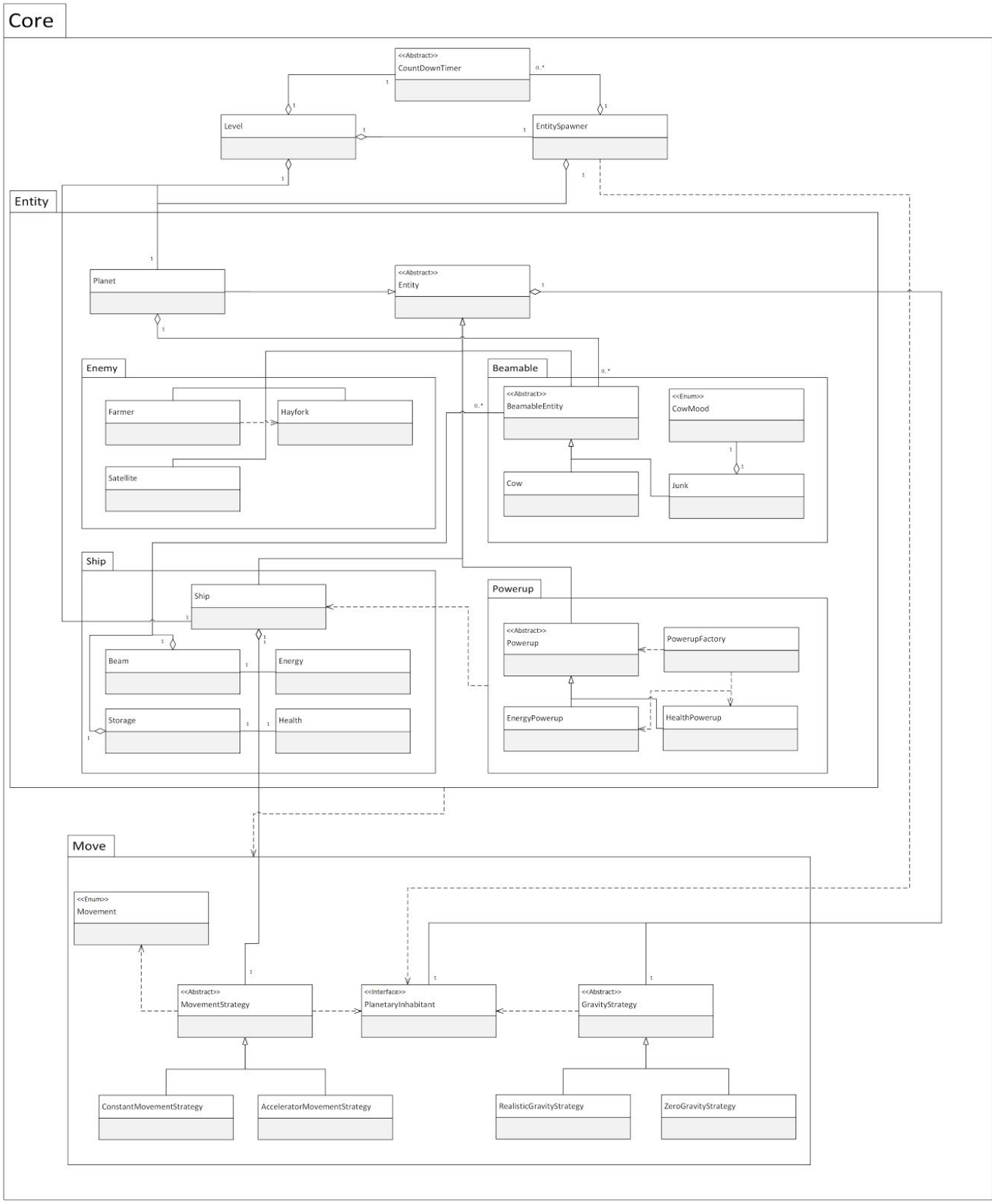
Sequence diagrams

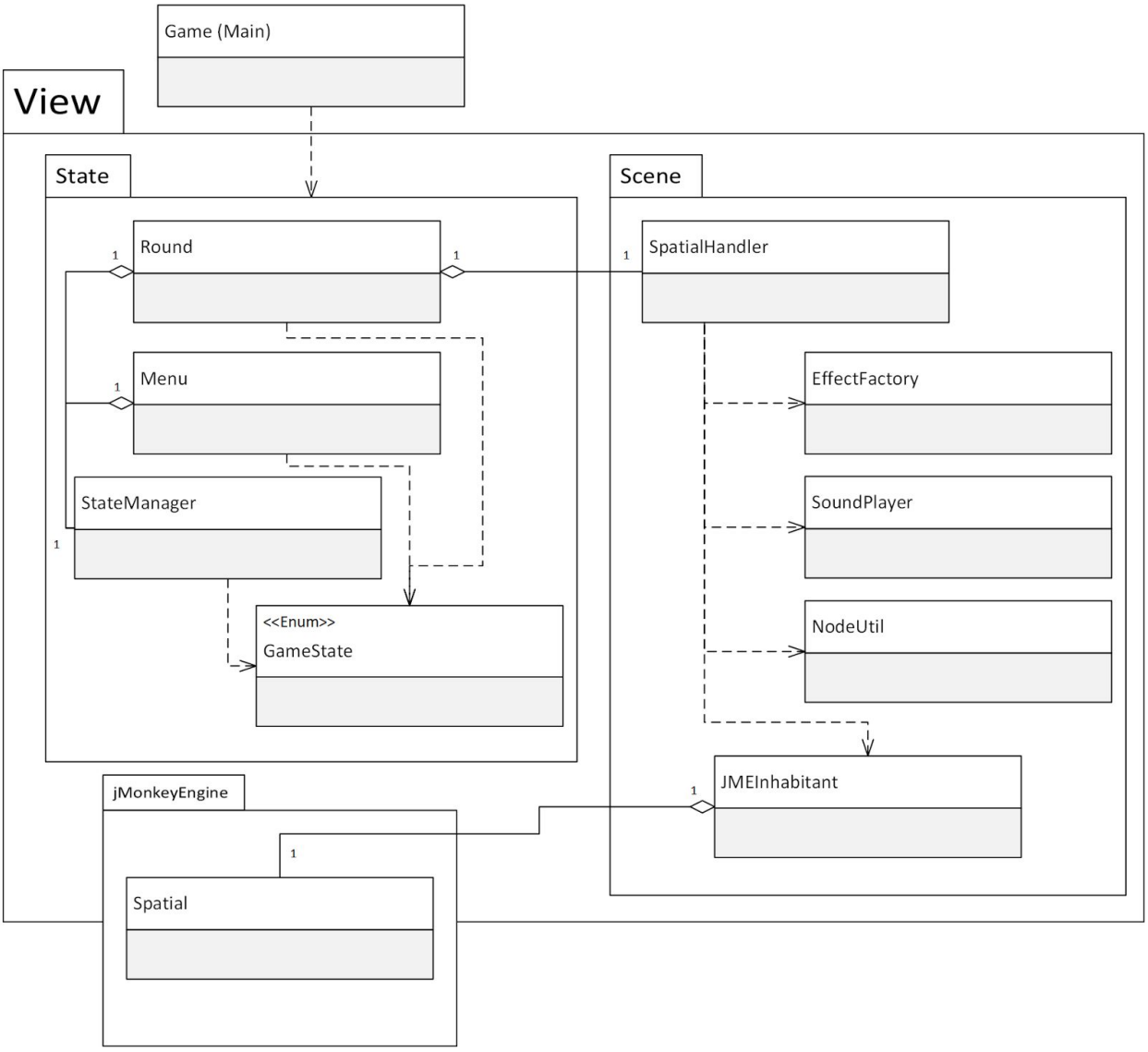


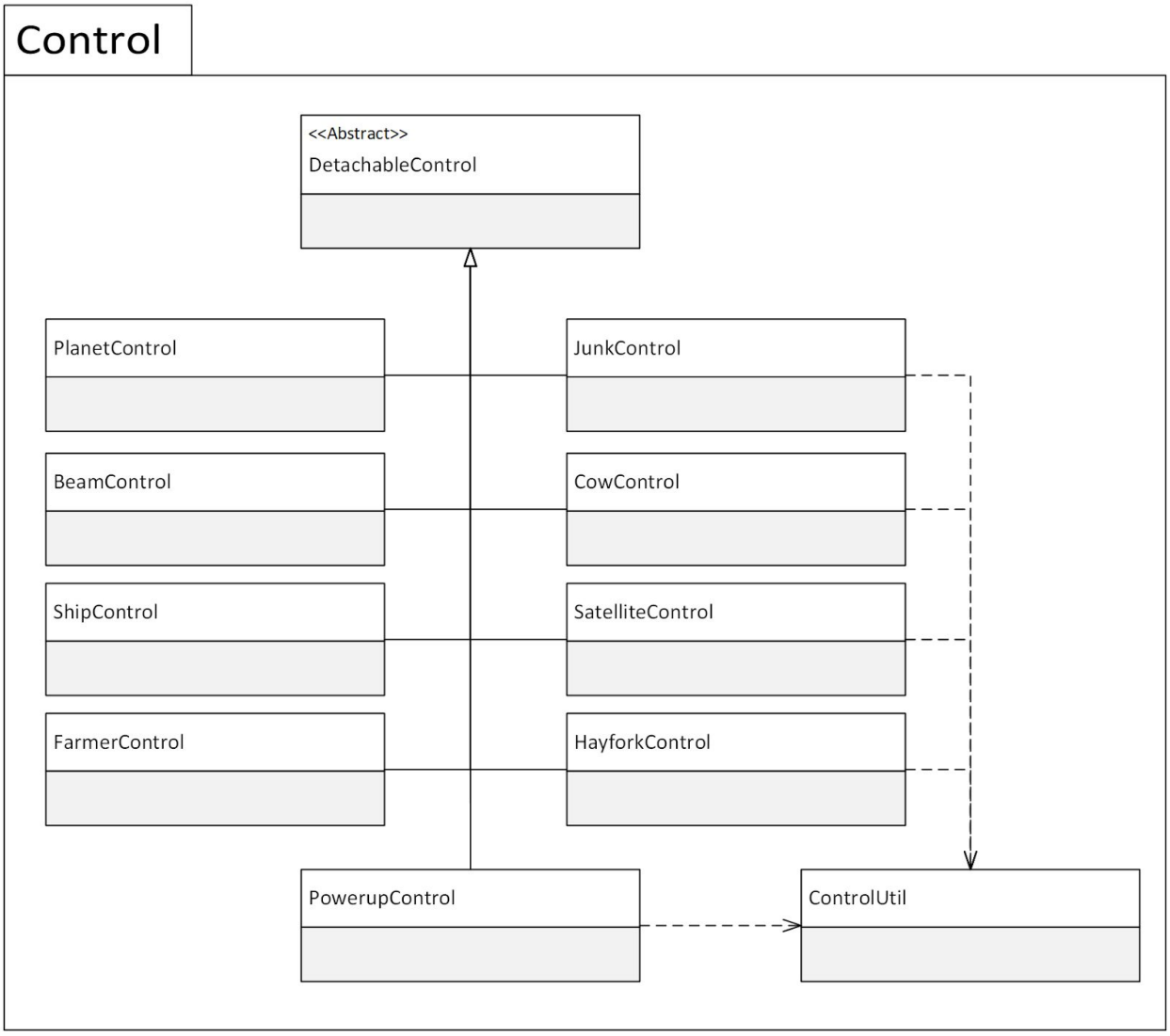
Not Enough Space : Dry Run UC : Use Beam

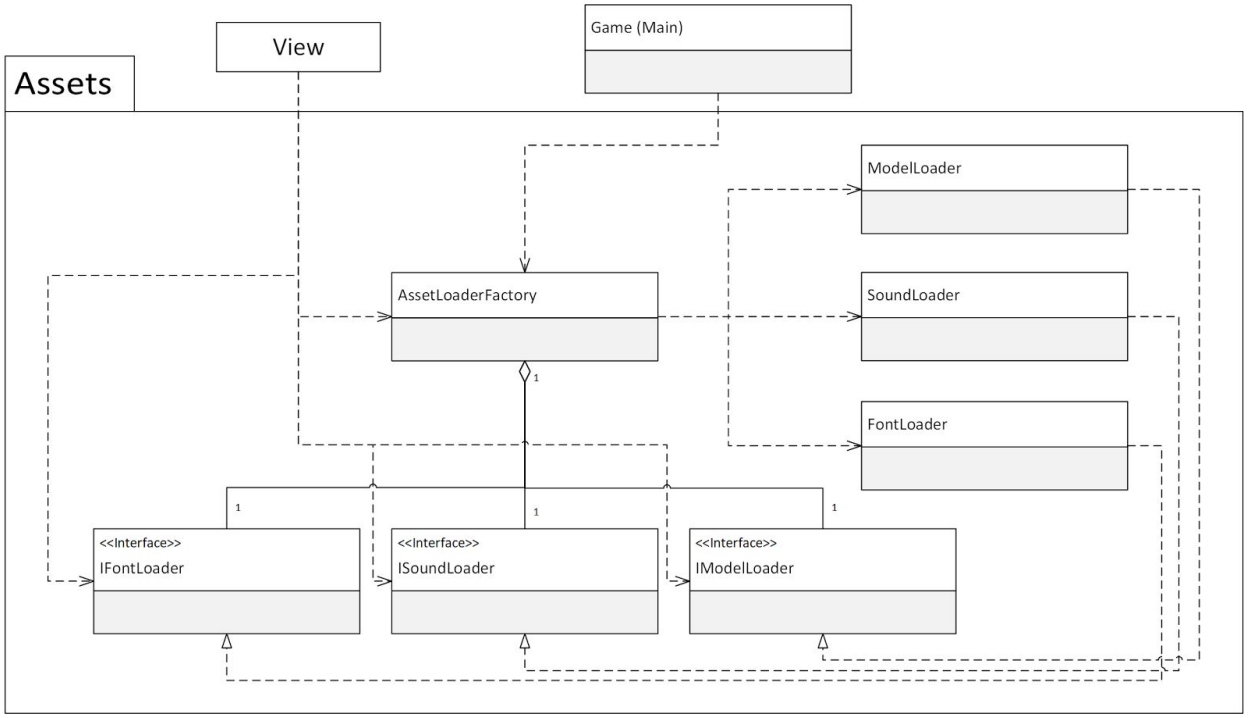


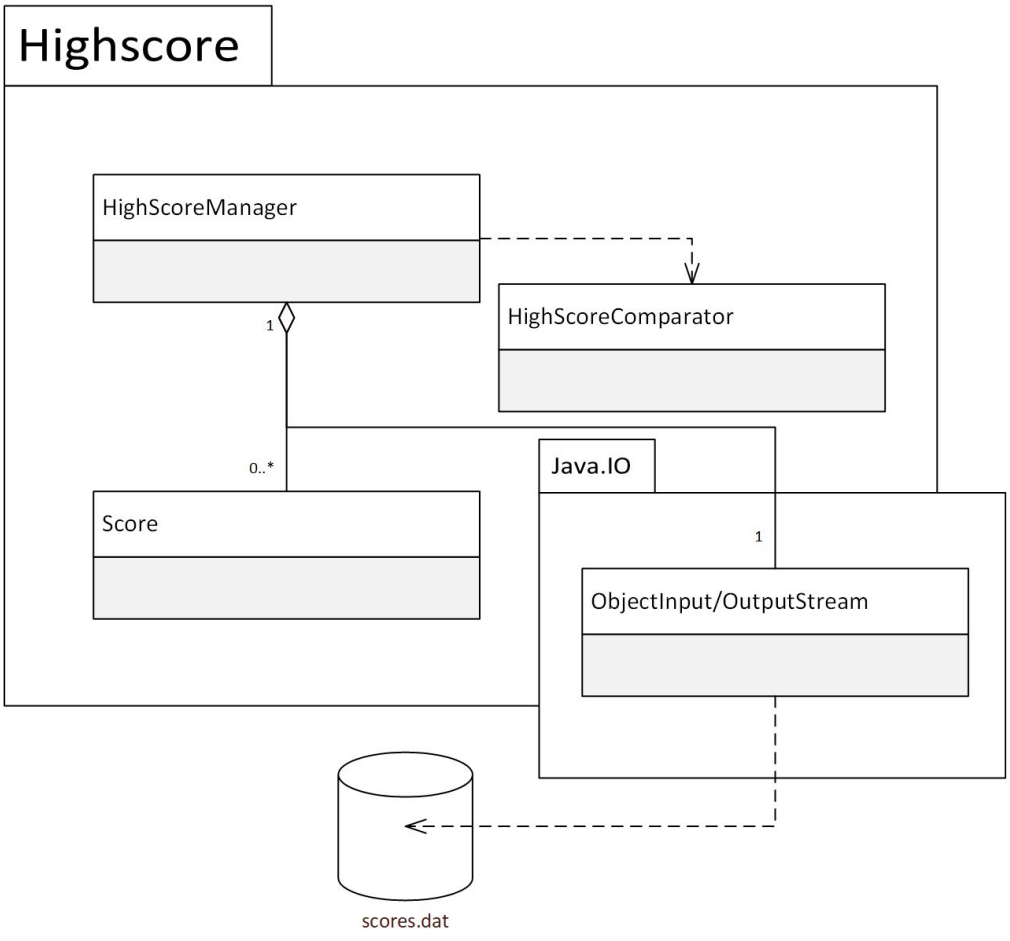
Subpackage Class Diagrams




Game (Main)
View
State
Round
Menu
StateManager
<<Enum>>
GameState
Scene
SpatialHandler
EffectFactory
SoundPlayer
NodeUtil
JMEInhabitant
jMonkeyEngine
Spatial
1


Control
<<Abstract>>
DetachableControl
PlanetControl
JunkControl
BeamControl
CowControl
ShipControl
SatelliteControl
FarmerControl
HayforkControl
PowerupControl
ControlUtil


Assets
View
Game (Main)
ModelLoader
AssetLoaderFactory
SoundLoader
FontLoader
1
1
1
1
<<Interface>>
IFontLoader
<<Interface>>
ISoundLoader
<<Interface>>
IModelLoader


Highscore
HighScoreManager
HighScoreComparator
1
0..*
Java.IO
Score
1
ObjectInput/OutputStream
scores.dat