System design document for Frank the Tank

Table of Contents

Version	1.1
Date	2012-05-20
Author	Johan Brook, Jesper Persson, John Barber Unenge, Chris Nordqvist

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The system should have a clear separation between model data, logic, and the view. The model should act as standalone classes which can be initialized and tested without or with very little graphical representation. The design should use logical and clear bindings in between the different parts, as well as a stable system for the communication between the model and the view.

As a result of the loose coupling between the model data, rendering and the actual visual representation, the system should allow graphical elements such as sprites and tile sheets to be switched before launch without recompiling. It would allow users to use their own images for sprites.

1.2 Definitions, acronyms and abbreviations

- HUD - Heads Up Display. On-screen indicators for health points and weapon status.
- Java - a platform independent object-oriented language.
- JRE - Java Runtime Environment. Software needed to build and run a Java application.
- MVC - Model View Controller. A design pattern where data (model), logic (controller) and visual representation (view) is clearly separated in order to make use of reusable components and loose coupling.
- Entity - a moving object in the game world.
- Collidable object - simply an object in the game world which other objects collides with.
- Wave - a new “round” or batch of enemies.
- Items - as of now, medical packages and ammo crates.
- JSON - JavaScript Object Notation, a lightweight replacement for XML.

2 System design

2.1 Overview

The system will use a form of MVC pattern, with the model completely separated from the view, since they are only communicating through the *Event Bus*. The controllers are responsible for logic and setting up references.

The application entry point is the `Main.java` class. An AppController is initialized from there models and more relevant controllers are initialized and set up. The game logic (the “game loop”) is running on a separate thread along with the AI controller. GameModel is owning all objects on screen and is responsible for checking collisions and moving the player. The NavigationController, which is listening on key events from the user’s keyboard, is sending appropriate events back to the game model, which is positioning the player according to the keys.

Several components uses a *game loop* in order to update their state. A callback method (often *update*) is called repeatedly from a desired refresh rate (for now at 60 FPS) where the component is refreshing its logical state based on a delta time – the time since the last update.

2.1.1 The Event Bus

Instead of having a spaghetti system of property listeners, we are using an Event Bus which any class may register and listen to. Thanks to the bus, we are able to create a flexible and open communications system. The game is taking place in the model, and the view is merely listening to

different events triggered from the model.

2.1.2 Object Factory

The model objects are not initialized themselves directly in the model. An object factory with static methods is used for creating new objects dynamically based on data from the JSON files (see 2.1.2). Because of that, code related to the initialization of model objects may be altered in once place instead of all over the domain model.

2.2 Software decomposition

The system is composed in different packages (not a complete list):

- `controller (controller classes)`
- `model (model classes)`
- `graphics (view and rendering classes)`
- `main (application entry point)`
- `event (event handling)`
- `audio (audio controllers and libraries)`
- `tools (util and factory classes)`

2.2.1 Models

The model components include both abstract and concrete real world objects, such as Player, Enemy, MachineGun, and more. The package is testable and runnable without a view. A GameModel is used for holding all game world objects and entities, and responsible for updating the state for everyone.

2.2.2 Controllers

There are several controllers: the ViewController, AppController, NavigationController, SoundController and AIController. They are responsible for different parts of the game, and are communicating either with concrete references or via events.

2.2.3 Views

The view is completely separate from the model, and only listens to events from the latter. It's running in a separate thread and utilizes OpenGL rendering for optimal performance.

2.2.4 Layering

N/A.

2.2.5 Dependency analysis

Library dependencies:

- JSON, a library used to parse and use JSON to retrieve game data.
- Apache Commons IO, the JSON library depends on this library.
- Jogamp, a library containing JOGL, JOAL and gluegen. These are used for rendering and audio controlling in the application.
- SoundSystem, Paulscode. Used for sound.

2.3 Concurrency issues

As our application uses multiple threads; one for the update logic, one for user input, one for rendering and one for sound we've had some concurrency issues. The first one being in the game model and the AI controller. The problem occurred when trying to add objects to a List while another thread is iterating over that same list. The solution to this problem was to use a List that supported add objects while iterating over it. The other concurrency issue arose when loading textures into OpenGL. What we tried to do was loading the resources when a certain event was triggered by the game model.

This would not work as there was no OpenGL context current on that thread. We managed to find that even if we did succeed to load the resources in one thread, it would still be a headache to transfer it to the thread running the rendering. The solution to this problem was to let the event change the name of the current resource and thus trigger loading if that resource has not been used before.

2.4 Persistent data management

The model as well as the view are in need to storing static parameters somewhere, such as the player's speed. It's not desirable to hard-code those values in the source code, so instead we built a parser module which is used for reading external files in the JSON format. The JSON files contain static model values but also serialized view objects, for example the different animations for actors. The data from the external JSON files themselves will of course be cached in a local JSON object inside the module, and will thus not be parsed dynamically every time other modules calls for data from the files.

2.5 Access control and security

N/A.

2.6 Boundary conditions

N/A. Launched as normal desktop application.

3 References

- JSON (JavaScript Object Notation) – http://json.org/

Appendix I

External libraries

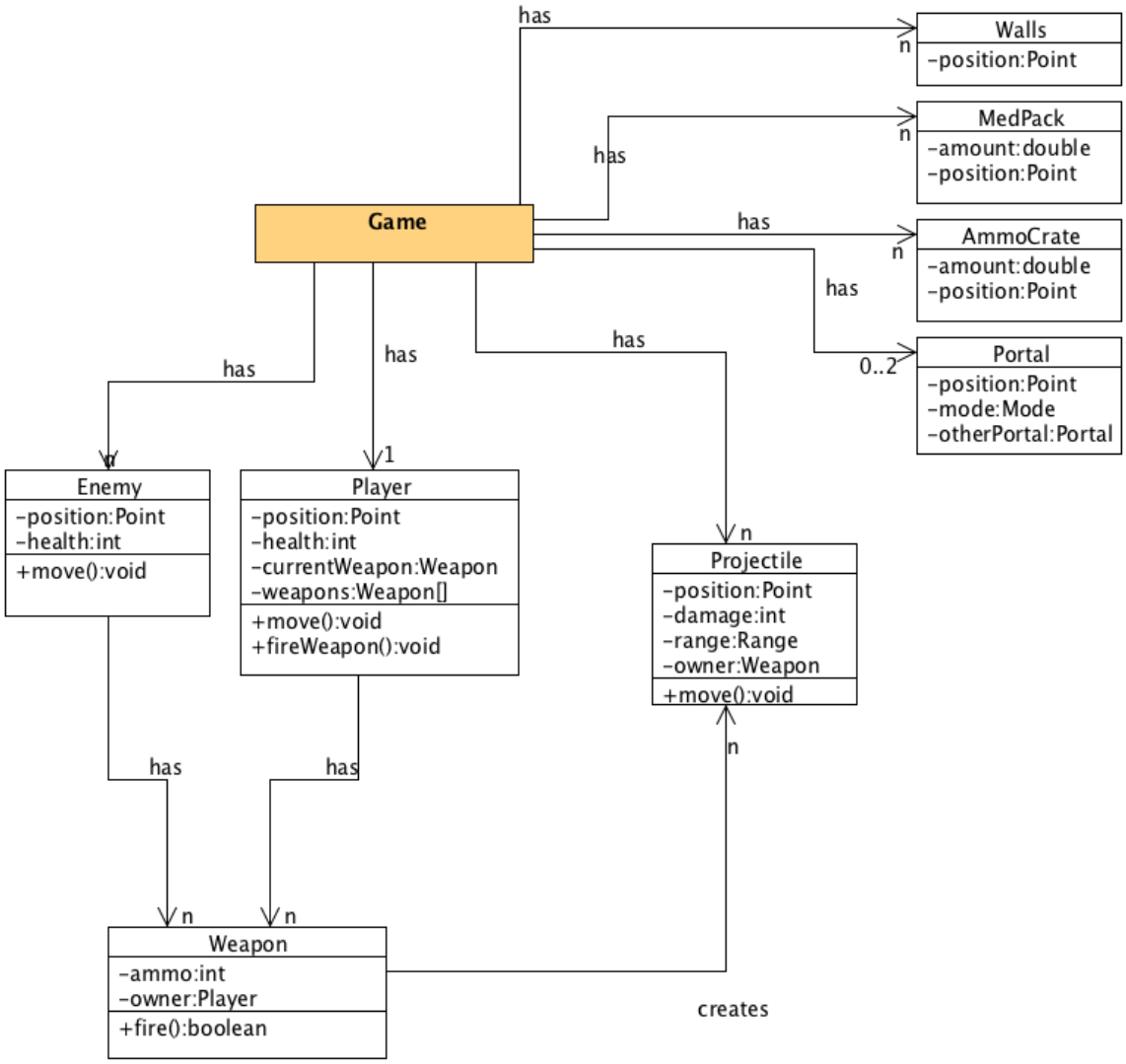
- Java JSON parser: http://www.json.org/java/index.html
- Apache Commons IO: http://commons.apache.org/io/
- Jogamp: http://jogamp.org/deployment/jogamp-current/archive/

Appendix II

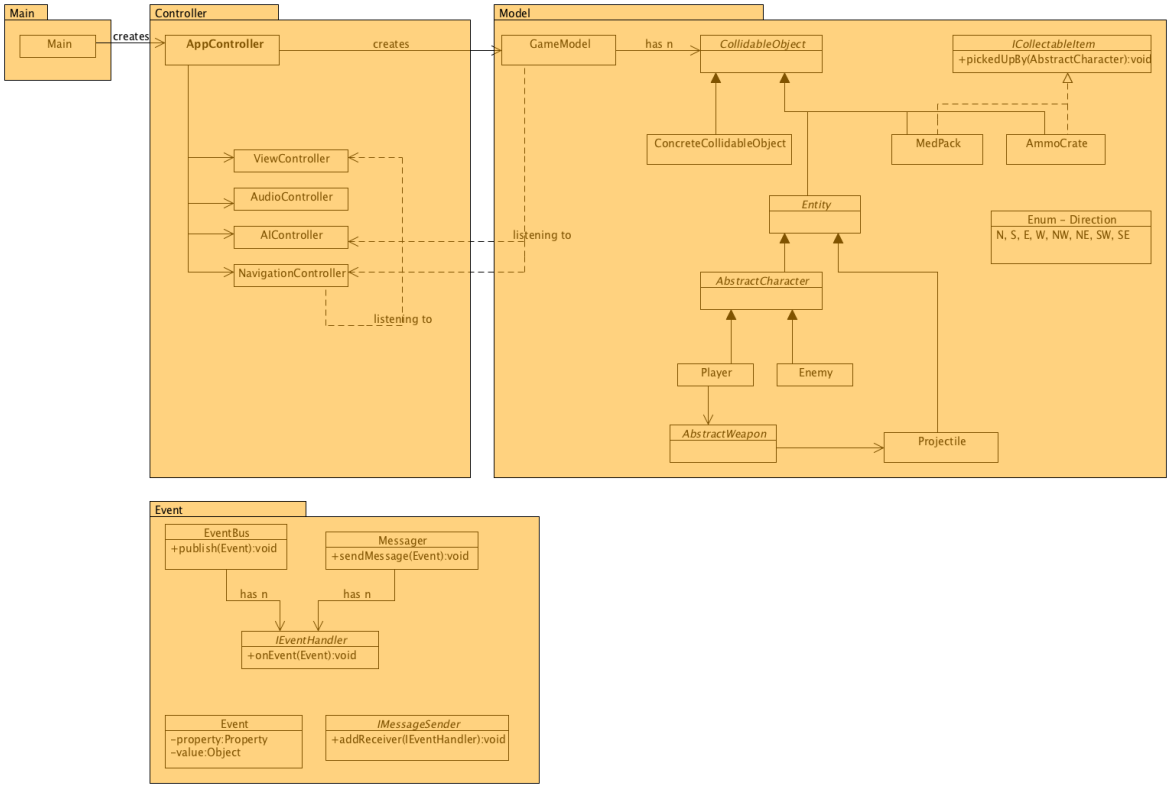
UML diagrams

Larger sized charts may be received upon request.

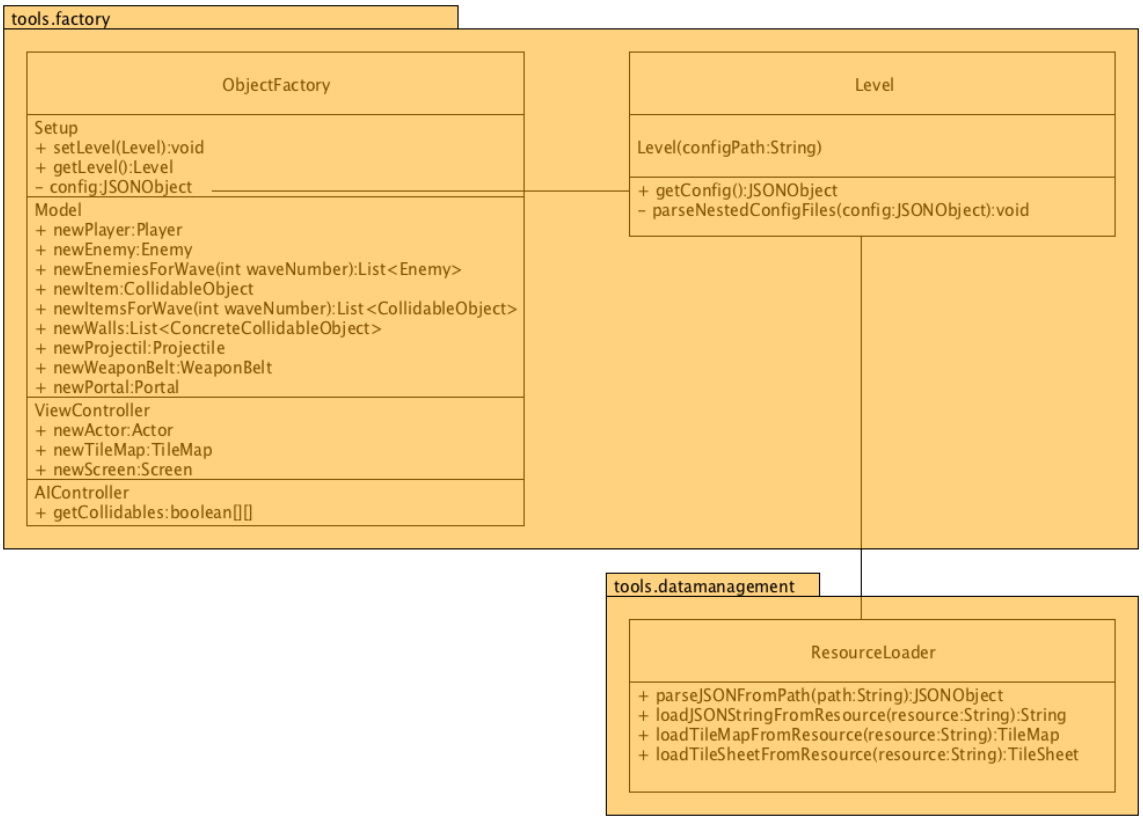
Domain model



Design model



ObjectFactory



Application flow cycle