Super Mario Bros.

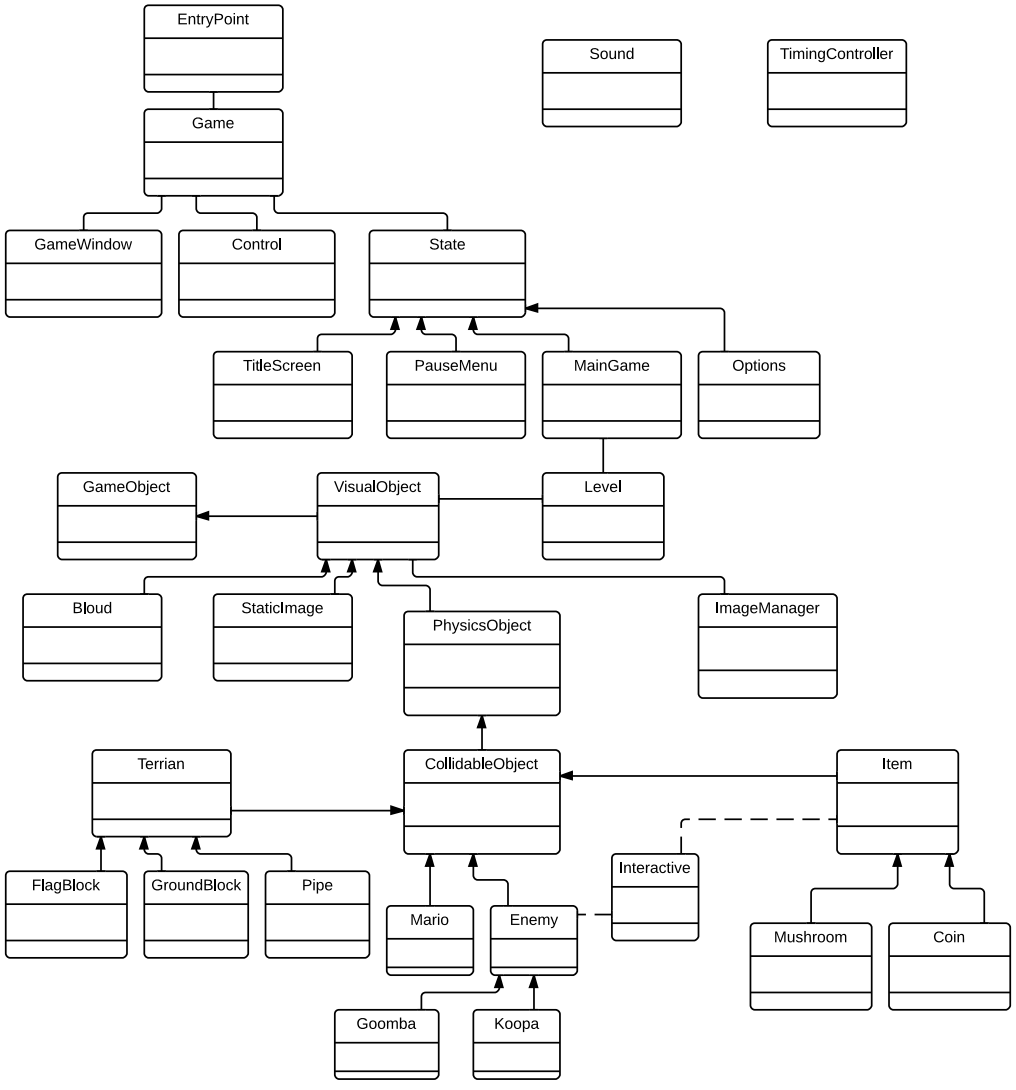
Technical Manual

© 2013 Chase & The Cat Daddiez

Problem

To design and implement a game replicating Nintendo's 1985 NES classic, Super Mario Brothers. It will be implemented in Java and will meet the following criteria:

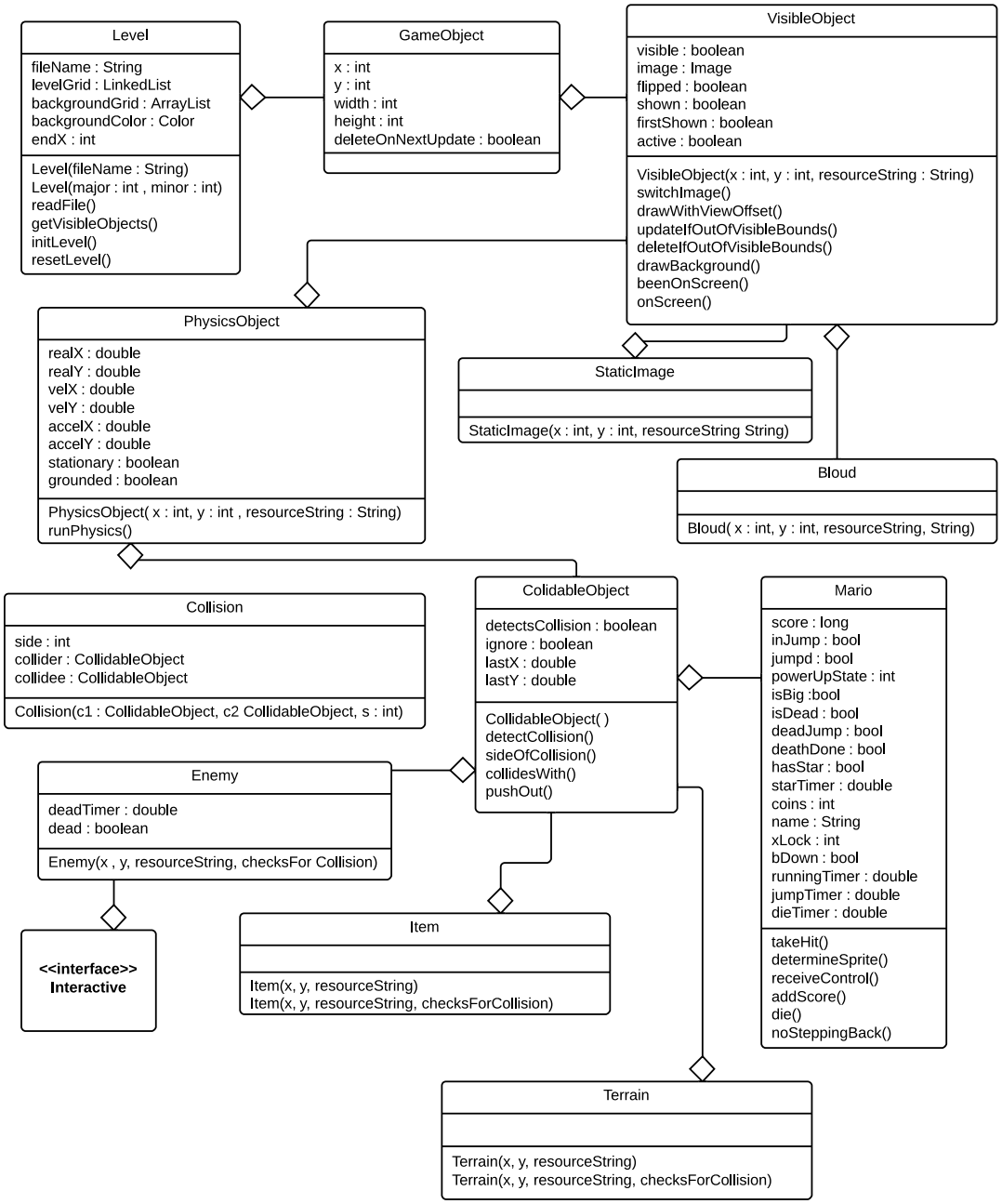
- Use Graphics2D/Swing for the view and control listening.
- Use a variant on the Model View Controller design pattern for the game's structure.
- Use the original game resources and a replicated font to completely replicate visual look of the game. Identical sounds will also be used.
- Try to replicate the physics as best as possible by testing behaviors against the original game.
- Render in the game's original resolution to ensure that the game's original graphics will work.
- Will contain many of the game's original objects and items.



Model

The game model consists of all of the classes representing some type of in-game object or concept.

The classes are: **Level**, **GameObject**, **VisibleObject**, **StaticImage**, **PhysicsObject**, **CollidableObject**, **Mario**, **Enemy**, **Goomba**, **Koopa**, **Item**, **Coin**, **Mushroom**, **Terrain**, **Blair**, **GroundBlock**, **Pipe**, **Block**, **QuestionBlock**, and **RegBlock**.



GameObject Tree

The objects in the game are all derived from common bases that share similar behaviors to prevent code duplication. Each mentioned base extends the next, and objects are derived from whichever base makes the most sense for an object.

The first in inheritance tree is **GameObject**, which holds behaviors and variables that each object requires including positions, sizes, and a flag that determines if an object is marked for deletion.

VisibleObject adds many properties to an object that is actually displayed. It adds a reference to a graphics accelerated image, a visible flag, and a variable that determines if the object has been shown. It also updates width and height according to the image.

PhysicsObject adds physics related variables for all objects including velocities, accelerations, and more precise x and y coordinates. Euler's method-based physics handling is added to the update method, and some other physics exception related flags are included.

CollidableObject serves as the abstract base for all objects requiring collisions. Collision detection routines have been added to the update cycle. Currently, collision detection is a bit spotty, but since all collisions are derived from this single model, any future fixes will fix all of the problems.

Concrete Objects

Objects work as large state machines holding behaviors, animations, graphics, etc in the form of methods, images, and booleans. All objects have draw and update methods that are invoked a parent controller objects.

Mario is a special type of CollidableObject that holds all of his own behavior routines but has colliding objects invoke them through the **Interactive** interface. For example, if a Goomba touches Mario, the Interactive method in Goomba will invoke Mario's "get hurt" method; Mario's behavior and animation will update accordingly.

Enemies are objects that affect Mario negatively in some fashion through the **Interactive** interface. **Goomba**s and **Koopa**s are the only two that have been implemented within our time constraints.

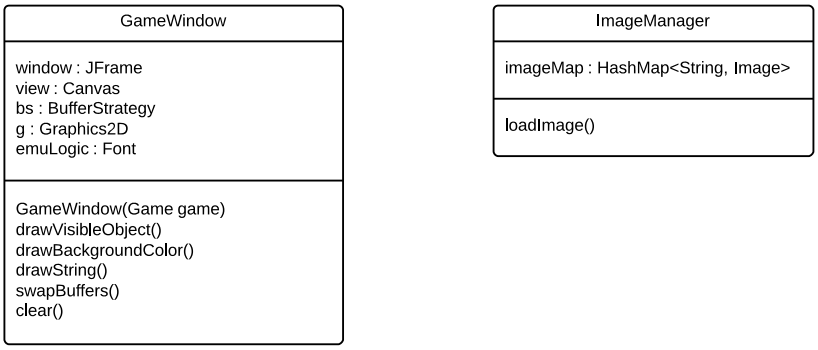
Items are objects that can also interact with Mario, but in a more positive manner. **QuestionBlock**s, **Coin**s, and **Mushroom**s have been implemented.

Terrain describes objects that are part of the environment but do not interact with Mario in any unique ways. **Blair** (Block/Stair), **GroundBlock**, and **Pipe** are all types of terrain and all of them simply serve as platforms for other objects.

View

The view handles how to display everything in the game's model to a user visible and intractable format.

The classes are: **GameWindow** and **ImageManager**.



The GameWindow

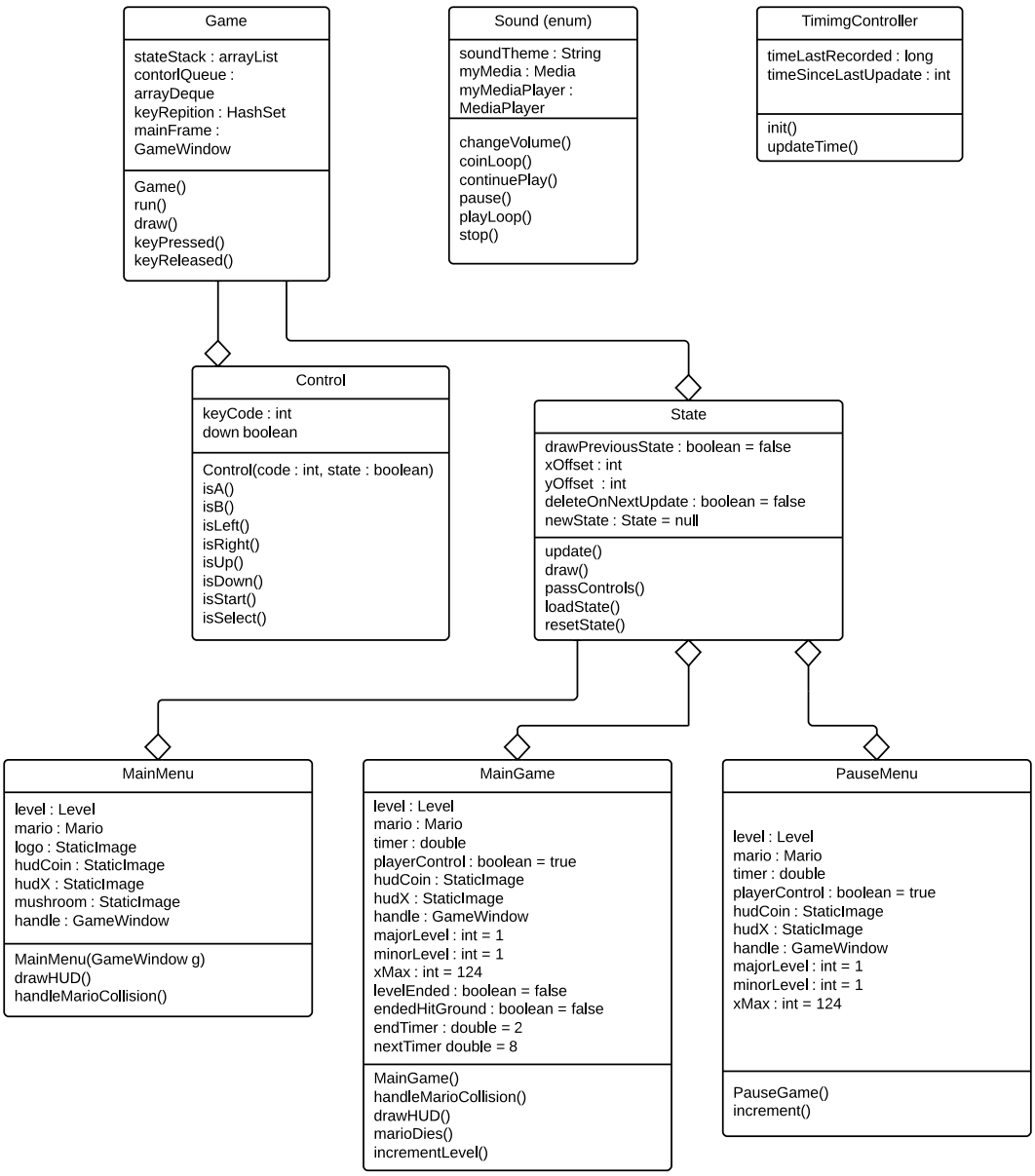
Java's Graphics2D and Swing packages are used to draw the game. Swing is used to provides the literal window frame for the game in the form of a JFrame. A Canvas object is placed into the JFrame and is assigned a KeyListener (Game) to receive controls. A Graphics2D object is used to draw to the canvas. Methods in the GameWindow are called by States and VisibleObjects to draw relevant text and images to the screen. There is also a built-in scale factor that allows the game to be rendered in multiples of the original size (256x240).

Graphics Memory Management

To ensure that Images are not loaded more than once, all image filenames are passed through ImageManager. It return unique hardware accelerated images for a given resource string. Internally, it uses a HashMap with the String as a key and the Image as a value to keep track images that have already been loaded.

Controller

The classes are: **Game**, **Control**, **State**, **MainMenu**, **PauseMenu**, **MainGame**, **Sound**, and **TimingController**.



The Game Loop and Timing

The main game process begins in Game, where the game loop handles processing input, updating, drawing, and timing.

All input is taken from a KeyListener object, wrapped into a Control object, and placed into a special queue that passes one control into the game per game update.

The current state has its update and draw method invoked. More in the next subsection.

Finally, the game sleeps for the set update interval, and TimingController calculates the time since the last update (using System.nanoTime()) for use in the next call. (Currently, our timing uses a fixed step, that is, the actual time passed is returned as an estimated constant. We were having some physics bugs as a result of variable time passing.)

The State System

Our implementation of Super Mario Brothers implements a state system for different game modes. It's so flexible that a completely separate game could be added to our project by creating a separate state by simply creating a state for it.

Each state is responsible for handling certain core behaviors: handling controls, updating itself, and drawing itself. For example, MainState handles updating and drawing all of the objects in a level and passing controls to Mario.

States are held in a type of pseudo-stack structure in the Game class where the game invokes its three main methods in each cycle of the game loop. The reason for using a list instead of a stack is the optional ability to simply draw the state below for a stacked image effect (as with **PauseMenu**: it draws **MainGame** below it).

Within time constraints, we were only able to complete MainMenu, MainGame, and PauseMenu. MainMenu works as the title screen waiting for a user to push "Enter." MainGame is the primary game controller, and PauseMenu simply a state that allows MainGame to appear paused.

Sound

The Sound class acts as a utility class that preloads sounds as an enum to make invoking the play and stop methods simple. Internally, it uses JavaFX, so the jar must be added to the build and execution path.

A Word on Testing

Because our game uses constant refresh model, traditional testing methods like JUnit are not a good fit; rest assured, we still tested. To test new behaviors, we ran the game with special test levels and worked out any visible bugs.