**OO Design – Nibble**

*1337 hAXX – John Polus, Alex Jacobs, Erik Rasmussen, Logan Brincks, Steven Karrmann*

1. **Introduction**

Nibbles was written in the language BASIC in 1991 and published by Microsoft. The source code consists of a single file Nibble.BAS composed of 14 subprograms and 2 functions. Following our structured analysis we have decided that we will reengineer Nibbles using the language Java. This choice was made due to Java’s general simplicity in regards to built-in graphics libraries as well as our team’s knowledge of Java. Due to time constraints and team consensus, we have also decided to maintain both the DOS-like GUI and as much of the original game’s functionality as possible, without adding any new features. We felt that it would be a waste of time to include anything extra before release. Likewise, we felt that keeping in line with the game’s overall DOS-like feel was important.

The game itself is fairly short, therefore the OO design for this project will also be made as simple as possible. We will use a MVC (mode-view-controller) design pattern and use interfaces to help encapsulate important aspects of our program. The actual snake the player controls with be a parent of its parts. Beyond these aspects, other classes will be relatively simple.

1. **System Requirement**

**2.1 Functional Requirement**

The functional requirements below are both the functional (denoted by the label F) and non-functional (denoted by the label NF) requirements for our program. Originally we had many more requirements, however after further inspection we determined many were either redundant or out of the scope of this project. It is worth noting that while these requirements were made to be as detailed as possible, it is possible that new unforeseen requirements may arise during development.

**System Requirements:**

F-SYS1:

The system must display the current level while in a gameplay mode.

F-SYS2A:

The system must display a snake on screen while in a gameplay mode.

F-SYS2B:

While in multiplayer mode, the system must display two distinct snakes.

F-SYS3:

The system must display a valid level layout while in a gameplay mode.

F-SYS4:

The system must respond to keyboard input and move the snake accordingly.

F-SYS5:

The system must end the gameplay mode when the snake is not alive.

F-SYS6:

They system must allow the user to pause gameplay and halt all on screen movement and actions.

F-SYS7:

The system must respond to keyboard input in all game menus.

F-SYS8:

The system must have the ability to pause, exit and quit the game at any time.

F-SYS9:

The system must have the ability to switch between single player and multiplayer gameplay modes.

F-SYS10:

The system must be able to receive input for and direct multiple snakes simultaneously while in multiplayer gameplay mode.

**Snake:**

F-SKE1:

The snake must die upon attempting to pass through itself or a boundary.

F-SKE2:

The snake must be able to pass through collectable items.

F-SKE3:

The snake must be able to move in only four directions, North, East, South, and West.

F-SKE4:

The snake must appear in a valid location upon beginning gameplay mode.

F-SKE5:

The snake must increase in size upon passing through a collectable item.

F-SKE6:

The snake must continue to progress in the direction of its head until death or until a change of direction.

F-SKE7:

The snake must not be able to change in the opposite direction (180 degrees) with a single button press.

**Game:**

F-GME1:

The game must begin on the first level.

F-GME2:

The game must progress to the next level upon the snake acquiring 9 items.

F-GME3:

The game must end upon completion of all available levels.

**2.2 Non-Functional Requirement**

NF-SYS1:

The system will be programmed in Java to allow for system portability.

NF-SYS2:

The system must follow an object oriented design philosophy to increase extensibility and reusability.

NF-SYS3:

The system must only require a java runtime environment greater than java 8 and no other external software.

NF-SYS4:

The system shall not be monetized.

NF-SYS5:

The system must support two player gameplay

NF-SYS6:

The system must support single player gameplay

NF-SYS7:

The system learning time must be under 5 minutes

NF-SYS8:

The system must run on Windows 7 and windows 8.

NF-MAINT1A:

To aid in maintainability, the system shall have:

NF-MAINT1B:

A comment block, following the JavaDoc standard, for every public method and constructor.

NF-MAINT1C:

A comment block for every non-boundary class file describing the purpose of the file, as well as listing and describing all contained classes.

NF-MAINT1D:

A JavaDoc for every non-boundary class

NF-MAINT2:

The mean time between failures should be over 1 year.

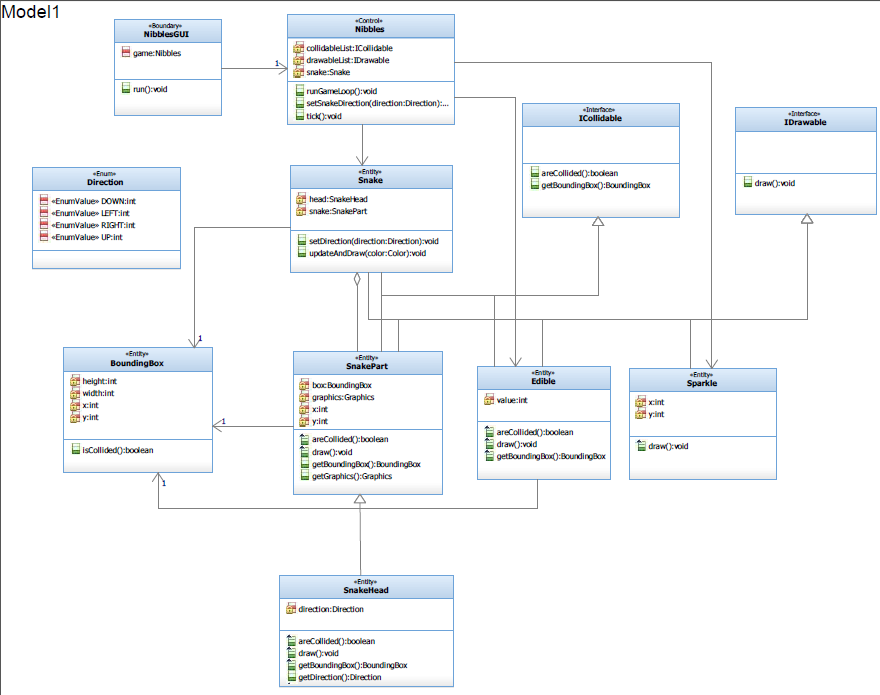
* A system failure shall be defined as any system initiated break in gameplay excluding a user pause or close action.

NF-MAINT3:

All original licensing must remain unchanged.

1. **Class Diagram**

The class diagram below is the final class diagram for our project. For a more detailed image, please see the files d9\_FinalDiagramImage.pdf and d9\_Final\_ClassDiagram.rpy.



**Control Class:**

Nibbles: This is the main class for the program. It handles all of the game logic, and controls the entity classes. All boundary classes communicate with the program through this class. Every time the game loops, this class iterates through the list of ICollidable and IDrawable objects and calls methods areCollided() and draw(), respectively.

This class also handles input from the GUI and applies it to the snake objects, and then updates the position of any movable objects, currently only the Snake class.

**Boundary Class:**

NibblesGUI: This class handles input from the user, and passes that to the control class for use in game logic. This class also determines the Graphic object used for rendering the game, passing the instance of the Graphic object to the control class for use with IDrawable objects.

**Entity Classes:**

Snake: The primary entity class. Objects of this type store a list of SnakeParts, keep track of the direction of movement, and update the position of the SnakeParts list.

SnakePart: This class stores the state of an individual link in a Snake. This includes the position, the collidable area (via the BoundingBox object), and the instructions for drawing objects of this type.

SnakeHead: A child class of snake part. Every snake has exactly one SnakeHead, which is used to direct the rest of the SnakeParts in the Snake object. Objects of this type store an enumerated direction for that purpose.

BoundingBox: A simple class for use with the ICollidable interface. Objects of this type store a position and rectangular area to determine when collisions happen. This class is essentially a utility class for ICollidable. Objects that can collide must be able to pass a position and an area to calculate if they have hit anything. This class acts as a container for those values.

Edible: A class that stores the position and value of food items that the snake can consume. Objects of this type are drawn to the screen, and then checked every update for collisions with a snake object, which consumes the edible and adds the value to the number of SnakeParts that the Snake contains.

Sparkle: A class that is solely aesthetic. Objects of this type may be instantiated to be drawn on the screen for decorative purposes, and then removed at will.

**Interfaces**

Interfaces in this program are used to define complex attributes and functionality that may be shared by classes with no other relationships. Because Java only allows child classes to extend a single parent class, interfaces are an ideal way to handle this situation. All classes that have a specific functionality will implement the interface, allowing control classes to handle an interface’s functionality without regard to what type the object actually is. If an object implements IDrawable, then it may be drawn to the game screen, regardless of the object’s type. This allows individual classes a finer degree of control over how they interact without cluttering the control class’ logic. This also makes our program very extensible, as new classes that are added can determine their functionality through which interfaces they implement.

ICollidable: An interface to be overridden by all classes that need collision handling. This interface defines functions to return a collidable area (via a bounding box) and to check for and handle collisions.

IDrawable: An interface to be overridden by all classes that need to be drawn to the game screen. This interface defines a single function, draw(), that allows implementing classes to determine how they are drawn on the screen, i.e., a SnakePart could draw itself with an x over itself if it has been collided with recently, etc. A Color is passed as a parameter, but implementing classes are under no obligation to utilize this.