

Design and Implementation of an Interactive Simulation Using the JAVA Language Through Object Oriented Programming and Software Engineering Techniques

Dan Stalcup

November 4, 2005

January 24, 2006

April 6, 2006

Abstract

As algorithms and set-ups for interactive simulations (games) become more and more complex, the method in which such projects are approached, designed, and implemented requires careful analysis. Others have studied theories of object orientation, and have hypothesized on the ways to optimize the development of complex computer programs. This study encompasses the engineering and construction of a complex interactive simulation called "Project Dart Hounder" with an entirely object-oriented approach, analysis of the process and results, and a furthering of the understanding of the feasibility of using object-oriented programming as the sole method of design.

1 Introduction

1.1 Purpose

The main purpose of this project is to come to a more thorough understanding of the advantages and disadvantages of designing a program entirely through object-oriented programming in a complex interactive simulation. This will be done by planning and implementing such a simulation. Throughout this project, the simulation is referred to as "Project Dart Hounder."

1.2 Scope of Study

The documents to be processed will be within the subject area of object-orientation and other computer science theories. Another primary area of study that will be researched is software engineering theory, so as to formulate a more efficient system of design, coding, and debugging.

2 Background and Review Literature

[when I do more thorough research (as I've only done a little bit of web research and googling so far), this will have real information in it]

3 Project Dart Hounder

3.0 Special Note

Discussion of the characteristics of the program represents the status of the game upon completion of the coding and debugging. Though this is similar to what was originally conceived, there are of course some differences between initial idea and final result. See section 4 for more details.

3.1 Overview

Project Dart Hounder is a turn-based battle simulation role-playing game (RPG). The actually interactive graphic user interface (GUI in which most of the game takes place "gameboard") is a rectangular matrix of buttons accesses instances of the logical objects ("squares") in which three simulated things are contained: an "terrain," a "weather," and an "entity." The "turn algorithm" lets allows these objects to interact and allows for the simulation.

3.2 Summary of Gameplay and Gameplay Vocabulary

The game requires two users or "players." Each player controls a set of characters, and the characters of both teams are on a rectangular grid. Each player can control his or her characters, giving them various commands. Each character can interact with other characters, including interactions with characters of the opposing player known as "attacks." Several factors determine the effectiveness of attacks. A high cumulative effectiveness of attacks can lead to a character being "eliminated" or removed from battle. The winner of the game is the player who first eliminates all of his or her opponent's characters.

Each character has a set of stats, including a classification or "class." Depending on a character's class, it may be able to use different interactions with other characters.

/expand later/

3.3 Gameboard

The gameboard is the overarching GUI in which the simulation runs. The gameboard's primary purpose is to communicate; it communicates between the user and its programs and also between the different objects of Project Dart Hounder. The gameboard's second purpose is to keep track of the specific situation of the simulation by keeping references to special, specific objects, such as the currently selected character.

The gameboard visually represents the squares through a grid of colored buttons. Each of these buttons, via an implementation of JAVA's ActionListener interface called Listener, is connected to a specific square. When Listener is activated by clicking on one of these buttons, examines the situation of the simulation (see 3.8), alerts the appropriate objects what button has been clicked, and then modifies the gameboard to represent the new situation of the simulation.

As it holds references or indirect references (that is, a reference to an object from which it can acquire a reference) to virtually every logical object in Project Dart Hounder, it is also a communication tool between the different objects. For example, the simplest way to have a

character directly change the appearance of one of the buttons in the gameboard is to go through the gameboard itself and use its references to the buttons.

Finally, the gameboard keeps track of the situation of the simulation by using references to special objects. The most important example of this is that it has a reference to the current character. Also, using data values such as integers, it keeps track of the state of the simulation.

For the purposes of this project, the terms "board" and "gameboard" are interchangeable. Also, these terms can be used to refer to either the entire gameboard object or just the playing field displayed on the gameboard.

/to be expanded/

3.4 Square

The square represents one block on the playing field. Each square holds three objects: a terrain, a weather, and an entity, each of which will be discussed in detail during later sections (3.5, 3.6, and 3.7, respectively).

These three objects are stored as direct references in the squares. Each square will hold exactly one terrain, exactly one weather, and either zero or one entity.

It is possible to access and acquire any of the objects contained in the square by having access to the square.

Each square has a position, a coordinate of two numbers, row and column or "x and y" to represent where it lies on the playing field.

3.5 Terrain

Terrain represents the environment that the entities (see 3.7) reside in. Each one stores various values about itself, including density, visibility, temperature, and elevation or height. Each one of these plays a role in the effectiveness of any attacks from or against another entity. Terrains also have a specific color, which is only for clarification by the user, and does not directly affect attacks from or to the square.

Each of these values are stored as basic data in the terrain class and can be accessed through a reference to any terrain object.

There is a distinct temperature, value, and densities for each different type of terrain. Terrains are partitioned into subclasses, ranging from glacier to plains to urban environment, etc.

Terrains also contain another quality: whether or not they are solid. Solidity is determined via an abstract method, implemented specifically by subclasses, which returns a Boolean value. Solidity is often determined simply (e.g. water is always not solid), but can sometimes be a little bit more complicated (e.g. swamp is solid if its elevation is less than three).

For more on specific information of the various subclasses, see Appendix A1.

3.6 Weather

Weather affects the environment that the entities (see 3.7) reside in. Weathers use two values to determine their affect on attacks, brightness and precipitation.

Unlike terrains, there are no subclasses of weather. Rather, different instances of weather are determined by four specific cases implemented into the Weather class itself, represented by an integer. The integer used to represent each case represents its severity, where 0 (clear skies) is the least severe while 3 (stormy skies) is the most severe.

Because weather can realistically change over just a few minutes, a magnitude of time represented by Dart Hounder, there is a gradual changing mechanism built into weathers. Every weather is given a "mod digit" and for every turn when modded by ten, a potential change in weather is simulated (for example: if the mod digit is seven, a change in the weather will be simulated on turns 7, 17, 27, 37, etc...).

For more information on specific states of weather, consult Appendix A2.

3.7 Entity

Entity objects are the agents of interaction on the gameboard. By controlling entities and allowing them to interact with other entities, Dart Hounder is being "played."

There are two basic types of entities: characters and noncharacters. However, all entities have certain traits: they have a name, a position (which matches the position of the square it is in), an ID, an HP value as well as a constant maximum HP value, and the information of whether or not the entity is a character.

3.7.1 Noncharacters

Noncharacters are the simpler of the two types of entities. Noncharacters are not controlled by players.

Noncharacters have all of the traits of generic entities, plus two others: Each Noncharacter has simple Boolean functions that return whether or not they are living and whether it is moving.

These traits result in three subclasses of noncharacters: minerals (neither living nor moving), plants (living but not moving), and animals (living and moving, /if I make each one do something distinctly different, discuss here/

See Appendix B for further information on the subclasses of noncharacters.

3.7.2 Characters

Characters are the more complex of entities. They are controlled by the players. Characters are the primary units of interaction between the players.

Characters have a variety of traits essential to ... cont.

3.8 Turn Algorithm

4 Design, Engineering, and Construction

The design of such a thorough, complex project requires a specific, detailed plan and approach. Project Dart Hounder was put together in a ten-step process adapted from various guides to software engineering theory.

/eventually, I will go through my journal and update the steps to reflect how my project progressed/

4.1 Summary of Project Steps and Their Results

STEP 1: An Idea

The first step was to decide on an ideal style of interactive simulation to implement through object-oriented programming. I decided a turn-based, tactics, grid-based, role playing would best optimize the features of object-oriented programming methods, so chose that.

STEP 2: Testing Feasibility

I had to create a few simple data structures and a primitive interface to see if such a project would be possible.

STEP 3: Setting Requirements

Upon deciding that project would be possible, I produced a list of features and requirements that the interactive simulation would include.

STEP 4: Making a Plan

With a list of requirements and features as a reference, I set a plan as to how each feature of the project will be assembled, and how the separate pieces will fit together to form a complete project.

STEP 5: Designing Data Structures

Guided by my plans and using the simple interface created for Step 2, I designed and created the data structure classes for Project Dart Hounder.

STEP 6: Creating an Interface

/add when I get this far/

STEP 7: Implementing the Algorithm

/add when I get this far/

STEP 8: Debugging and Revising

/add when I get this far/

STEP 9: Finishing Touches

/add when I get this far/

STEP 10: Final Analysis

/add when I get this far/

4.1.1 Step 1 - An Idea

/to be expanded/

4.1.2 Step 2 - Testing Feasibility

- 4.1.3 Step 3 - Setting Requirements
- 4.1.4 Step 4 - Making A Plan
- 4.1.5 Step 5 - Designing Data Structures
- 4.1.6 Step 6 - Creating an Interface
- 4.1.7 Step 7 - Implementing the Algorithm
- 4.1.8 Step 8 - Debugging and Revising
- 4.1.9 Step 9 - Finishing Touches
- 4.1.10 Step 10 - Final Analysis

5 Results

[for after I'm done]

[I'm going to discuss the pros and cons of basing your project solely on object-oriented programming]

6 Discussion

[also for after I'm done]

[may be combined with section 5]

7 Conclusion

[also for after I'm done with project]

References

[REFERENCES NEED TO BE INSERTED]

[I have a few...]

Appendix

[add stuff here]