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

Dan Stalcup
June 12, 2006

Abstract

As logarithms 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. Ofinal-paperthers 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 shell for a complex interactive simulation called “Project Dart Hounder” using 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 a simulation in such a manner. 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

Object-oriented programming (or OOP) is a design and coding technique

implemented by various mainstream languages, most notably JAVA and C++. Its fundamental idea is that problems can be solved by breaking them down into segments called objects. Each of these objects is of a specific type and can often interact with other objects.

One common example of OOP that is often used is managing a bank. Suppose one manages all of the bank accounts and uses a computer to store the current funds, interest rates, and personal identification numbers (PIN) for all of the accounts in the bank. One straightforward way to handle all of this is to have an array each for all of the funds, interest, and PIN's, keeping, for example, the information for one account in the first slot of each array. This, however could be inconvenient or even disastrous. As accounts are added and removed, there is ample opportunity for one of the values to accidentally be offset in the array by one. Then, you would have the wrong data not only for one account, but potentially thousands of accounts.

This is where OOP comes in. An object-oriented solution to this problem would be to apportion the data of each account into objects, more specifically, Account objects. The programmers at the bank could design the template for the Account object, deciding what values and traits all Accounts must have. Because one Account object can store all of the information for an account, there is no longer a need to keep three different sets of numbers, hoping they remain aligned. Now, the bank can just keep one set of Account objects and still have access to all of the same information.

The organization of data into groups like this may seem a simple, logical step, and that's because it is a relatively simple, logical step. But its programming implications are profound, and its benefits are widespread. Not only has efficiency improved for organization and storing of data, but it could also potentially make changing data, and many other important areas, more efficient. Since you already have all of the data of one account organized in to one "object," why not integrate into the object functionality besides data storage? For example, tell Accounts themselves how to calculate compounded interest rather than doing it separately. Then, to modify the formula, you ideally only have to change the template for the Account class, and the change will affect all of the accounts.

Another, related, key idea of OOP is *information hiding*. This means that someone working with the objects does not need to know exactly how the template for the object makes things work; he or she just has to know what the effects are. Using our previous example, one wouldn't necessarily need to know exactly what formula is used to calculate interest, just to know that if he or she tells the account to calculate interest, it will indeed correctly calculate interest.

This inherent delegation lends OOP to group work. By using OOP, members don't have to worry as much about learning how all of the code works, just its specific results. In a modern working environment where nearly all professional software development occurs in groups, OOP is becoming very popular, which explains the popularity of JAVA, C++, and other OOP languages.

Object-oriented programming encompasses many broad topics and has been used,

discussed, dissected, and refined by thousands of programmers since the early 1970's, when it was introduced to the Simula 67 language

For further background on OOP, especially some of its mfinal-paperore complex portions such as abstraction and inheritance, feel free to consult other guides and lessons on object-oriented programming, including:

- http://java.sun.com/docs/books/tutorial/java/concepts/index.html
- *Object Oriented Programming and the Objective C Language* published by Addison Wesley
- *Data Structures and Abstractions with Java* by Frank Carrano and Walter Savitch

2.1 Entirely Object-Oriented Design Approach

This project encompasses the use and analysis of an "entirely object-oriented design approach." This term does not have a specific, technical meaning, but rather refers to a philosophy and approach. This approach is solving problems with object-oriented programming solutions whenever feasible. The practical application of this is that, when a programmer using an entirely object-oriented design approach comes to a problem that could be solved by both traditional techniques (adding several lines of code to an existing block) and object-oriented techniques (creating a new method or object to break it down and separate it), the programmer will give a heavy preference to the object-oriented techniques.

3 Project Dart Hounder

The program developed with this study and analyzed to draw conclusions, is called 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 shell for a turn-based battle simulation role-playing game (RPG). The interactive graphic user interface (GUI) in which most of the game takes place, called the "gameboard," is a rectangular matrix of buttons that accesses instances of logical objects, called "squares," in which three simulated things are contained: a "terrain," a "weather," and an "entity." The "turn algorithm" allows these objects to interact and allows for the simulation.

3.1.1 Shell

As stated in 3.1, Project Dart Hounder is a "shell" for a complete game. This means that it is built with the intent of being taken by someone else, being modified and expanded, and resulting in a complete game. The fact that Dart Hounder is a "shell" as opposed to a complete game has several consequences: One is that, even when it is complete, it may not be fully functional and interactive from the standpoint of a user.

Another is that the code must be designed to be readable by another programmer, as well as easily expanded and customized.

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.” The shell does not calculate the effectiveness of the attacks, simply makes the attack option available. If it did, however, 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 statisticss, including a classification or “class” (not to be confused with the term for one JAVA file: also “class”). Depending on a character’s class, it may be able to use different interactions with other characters. For example, the three example classes of characters constructed for the shell, Warrior, Mage, and Archer, have slightly different basic interactions and attacks.

3.3 Gameboard

The gameboard is the over-arching 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 current status 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 (see the JAVA API at http://java.sun.com/j2se/1.3/docs/api/ for more details). interface called Listener, is connected to a specific square. When Listener is activated by clicking on one of these buttons, it 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 the Gameboard 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.

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 any of the objects contained in a Square by having access to the square. In this way, it is a communication tool, because it can be used to access terrain, entities, and weather easily.

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 in which entities (see 3.7) reside. Each terrain 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 density 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, determined specifically by subclasses, which return 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).

Though Project Dart Hounder is a shell, it contains a complete set of functional Terrains. This can be easily expanded.

3.6 Weather

Weather affects the environment in which the entities (see 3.7) reside. 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 modulo divided by ten (i.e. remainder found when divided by ten) and the mod digit is found, 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…).

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: each instance of an entity has a name, a position (which matches the position of the square it is in), an HP (which stands for Health Points) value as well as a constant maximum HP value, and information as to whether the the entity is a Character. Each subclass of entity also has a unique identification number (ID).

One of the key concepts of object-oriented programming is the hiding of information. Thus, some of the traits, values, and fields of Entities will not be visible to the user, but may be used in discussion of the project.

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).

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 and characteristics. Each instance of a character has not only the traits of all Entities, but also have maximum and current magic points (MP), maximum and current Energy, an appointed player, an integer determining the character's state, and an array of integers known as its “Stats.” Furthermore, every

subclass of Character has a specific Character ID (not to be confused with the ID that every entity has), as well as an array of Character ID's from other classes, which serves as a list of subclasses to that specific type of character.

For more information on the process of players selecting and controlling a character, see 3.8.

3.8 Turn Algorithm

The “turn algorithm” is the set of steps in which objects operate and interact each turn, as well as the order of these steps. A “turn” is defined as all of the actions that occur when one player is “under control” (that is, only his or her characters can perform actions), beginning when he or she gains control and ending when the other player gains control of the game.

When a player’s turn begins, the gameboard is in a state of stasis: all objects are contained in appropriate places and no object is selected. A player may then select and control a character on his or her team. Each turn, a player may have up to one of his or her characters perform an action.

A basic example of a turn: the player gains control, the player selects the character, the player tells the character to move, the player tells the character what square to move to, the other player gains control. When a player is under control, it is said to be his or her turn.

3.8.1 Player Control of Characters

Players control characters through use of a Graphic User Interface (GUI). Selecting which character to control is done by clicking on one of the buttons in the Gameboard grid of buttons. If this square contains a character, this character is controlled by the player whose turn it is, and if the player does not currently have a selected player, the character will be selected. Once a character is selected, a menu of actions for the character to take pops up. The player can then choose an action, and, if necessary, select an appropriate square for that action to take place

4 Design and Construction

As this is a study of the effectiveness of using a singularly object-oriented approach to design and programming, the processes of implementation as well as the final result are essential to this study. This section takes a look at the process by which the project discussed in sections 1-3 was pieced together.

4.1 Summary of Project Steps and Their Results

Any conclusions drawn for this project from this study can only be helpful when taken in the context of the specific process used for development of Project Dart Hounder. Any discussion of effectiveness of an entirely object-oriented design approach will be taken considering only the development of this project, as opposed to software

development as a whole, unless stated otherwise.

The first step in developing Project Dart Hounder was setting the scope for the project. Next, an overview plan for the project was developed. Then, a very basic engine was constructed to evaluate feasibility. After that, more specific planning needed. Then, the data-storing classes were pieced together. Next, I created a graphic user interface and installed the data-storing classes in the interface to create the functional shell.

4.2 Finding Scope of Project

The first step of the development of Project Dart Hounder, after determining this project was going to study object-oriented programming and design through making an interactive simulation, was finding a scope for the project. After a decent amount of personal experience on object-oriented programming, along with some extra reading on object oriented programming, plus some knowledge on how games work and the amount of programming, designing, and planning to program a complete game, I decided to attempt a full-scale complete game, but decided I would scale back to creating a game shell if a full game was too overwhelming.

4.3 Overview Planning

After scope had been decided, basic planning occurred. I wrote up a simple half-page description of the game and listed some ways I might use object-oriented programming to create the game. This step is where I did my first requirement-setting, making initial decisions about what I wanted to be included, such as a fully-functional environment including unique weather and terrain, as well as what not to include, such as three-dimensional graphics or complex plotting.

4.4 Checking Feasibility

The next step, after basic idea of what was wanted had been set, a basic, running program was created to test the feasibility of this idea. I created a basic graphic user interface and a few of the basic data classes (Square, Terrain, and Weather) to see if what I planned is possible. At this point, I believe it is possible to implement everything in a complete game, though I am not sure exactly how making various Characters will work yet.

4.5 Specific Planning

The next step of the process was to set more specific requirements and begin planning exactly what classes would exist At this point, I wrote out basic versions of many of the classes, including Entity, Noncharacter, and Character, to figure out how they will fit together and what their purposes will be. My planning at this point was about a five pages of bulleted plans and requirements, as well as a about a dozen pages of code. I also begin doing basic testing on the interface and fixing minor bugs as I go.

4.6 Creating Data Classes

The next step was to begin expanding the data classes, or what are sometimes

referred to as the “logical classes.” These are the objects in the game that interact with each other. I finished Character and began working CharacterToolkit, another class file with many built in methods to simplify usage of the Characters. I also began on subclasses of Noncharacter and of Character. This was when I began considering a new scope: Though I was doing fine with the object-oriented programming design portions, I found myself struggling with the game design portions. The amount of effort and forethought required to create balanced statistics and formulas for attack effectiveness in characters would not be an object-oriented design problem per se, but simply a large inherent hurdle that went along with program being a game.

I decided to create a shell for a game. I would still get roughly the same amount of object-oriented programming experience, but would leave it open to design by other people. Basically, this meant I was deciding to embrace the technical workings and set-up of the game rather than the creative energies required to produce a balanced, fun game, leaving that part for someone else to tackle.

4.7 Creation of Graphic User Interface and Installation of Data Classes

With the data classes more or less created to a point I was comfortable with, I decided to focus on working on the graphic user interface and turn algorithm. I worked through several tricky bugs to finally come up with a system of communication between program and user in which a window with buttons pops up to allow users to choose an appropriate action for the selected character.

At this point, I also began implementing the functionality of the CharacterToolkit into the program. By using Gameboard (the GUI), Character (the data class), and CharacterToolkit (helps data class interact with GUI), I was able to make simple actions like making characters move and attack each other (as stated in 4.6, I decided to not program *how* they would attack each other, just make it so they *could* attack each other).

Once I had set up some of the finer points of these actions, such as making sure characters could not move on top of other characters, could not move onto water, making sure turns changed when a player moved, letting characters be unselected after they had been selected, etc., I was close to being finished with a shell. I did some more debugging and simplifying, and decided my shell was complete. Because I had made it so simple to create subclasses of Character or Noncharacter, the shell can be easily turned into a complete game, all that would be necessary would be fixing a few technical details, such as turning off debug messages and hiding debug buttons, a thorough expansion of the game, such as creation of more character types, more maps, etc., and implementation of formulas for how effective attacks and spells would be.

Basically, the shell creates methods for doing things, but doesn’t actually thoroughly do lot of these things on its own. For example, it provides a way for maps to be read, for complex, random, and continuous weather to be modeled and change as the game progresses, for characters to attack each other, for characters to cast spells, and more.

5 Results and Conclusions

The primary question this project proposed was: “In what ways and to what degree is an entirely object-oriented design approach effective?” Upon going through the creation of the interactive simulation, and producing a complete shell for a fully playable game, many of the strengths and weaknesses of this approach became apparent.

Several strengths to the entirely object-oriented design approach were observed. First, it provides flexibility. It was often easy to make an adjustment to a class, but be sure that the object still fit in with all of the classes it interacts with. Another strength with this approach was that it made expansion of the program easy. Because of hierarchy, classes designed as subclasses of other types of objects already had templates and could be made with just a few lines. This benefit was used multiple times in the development of Project Dart Hounder, such as when Warrior was made subclass of Character: the things that a Warrior would have to do wouldn't need to be recreated if a Character already could do it. Another strength was that it was easy to find and solve problems once the origin of the problem, or what class it is in, was found, simply because of the smaller file-size that often goes hand-in-hand with object-oriented programming. Finally, code from an entirely object-oriented project is likely to be very readable by other programmers.

There were also several weaknesses found with an entirely object-oriented approach. It was often hard to manage so many interacting classes and make sure that they were constantly appropriately interacting with one another. As more and more files are added, and relationships become increasingly complex and difficult to manage. Furthermore, with so many files, it was often hard to isolate which files problems were in. Some problems that could be solved with just a few lines often evolved into entire new files when an entirely object-oriented approach is used.

Overall, I found there to be a balance between positives and negatives of using an entirely object-oriented approach. These results point to some sort of middle ground. If a programmer were to use judgment as to when to use object-oriented techniques to solve problems, and when not to, many of these negatives could become less severe, and the positives could be therefore brought out more. A balanced approach, as opposed to an entirely object-oriented approach, would most likely lead to a highly effective and positive programming and software development processes.

6 Remarks on Application of Results and Conclusions

Ultimately, the conclusions drawn in this project may have only minimal applications in the professional programming world, for a variety of reasons. First of all, this project was developed by a single person, whereas most professional projects are made by groups. An entirely object-oriented design approach would add in numerous layers of complications: keeping track of changes, communicating specifically the purpose of each class, making sure objects are still compatible, etc.

Next, this project was built around this idea of an entirely object-oriented approach. Rather than a practical application, it was an experiment with a single theory. Professional programming and software development is done for practical purposes, and it is most likely that a project will not fit entirely in with such a narrow design approach.

Also, the requirements and specifications for Project Dart Hounder were changed with simple discretion of the primary programmer. In a professional programming environment, requirements are usually set in modified by groups other than those doing the programming.

Finally, the scope of Project Dart Hounder does not match much of what would take place in a professional programming environment. The game is relatively small and was never intended for widespread use by anyone, whereas many projects developed in professional programming environments would be designed for usage by a large group.

However, the conclusion drawn that a balanced, thought-out approach of object-oriented programming and design would be the most effective would most likely hold true in a professional software-development environment. This would allow for programmers to take advantage of the benefits of object-oriented programming when it would be helpful and positive. By correctly and appropriately using object-oriented programming and design, software development in any environment can be simplified and improve from beginning to end.