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

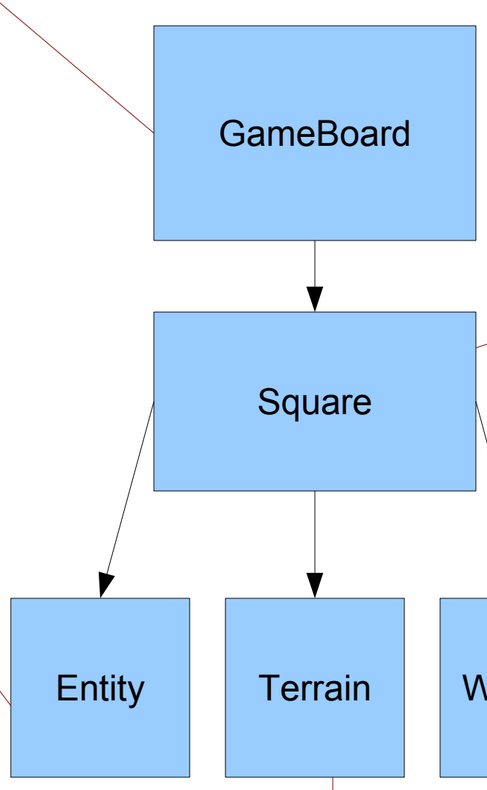
Dan Stalcup

TJHSST Computer Systems Lab 2005-2006

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. 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 object oriented approach, analysis of the process, and results.

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 specific situation of the simulation by keeping references to special, specific objects, such as the currently selected character.



Square: The square represents one block on the playing field. Each square holds three objects: a terrain, a weather, and an entity,

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.

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 ID.

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

Noncharacter: 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/

Terrain: Terrain represents the environment that the entities 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.

Weather: Weather affects the environment that the entities 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.

Upon analysis of the effectiveness and complexity of the resulting product, mixed conclusions were reached. While breaking down problems into objects often simplifies the solution, it occasionally makes it more complex. Careful judgement is required, and very thorough planning and design is even more essential: without effective planning and design, object-oriented programming loses a majority of its value. But when used carefully and smartly, object-oriented programming is a highly effective method of problem solving and software development, especially when working with large groups of people.