Peggy Li
CompSci 108
April 20, 2012

Level Editor Design Document
Updated for Vooga Milestone 3

I. INTRODUCTION

The Level Editor allows a game designer to dynamically create a new level and modify the properties of the level and various Sprite objects contained in a level. These include the ability to set a level name and background image, as well as the placement of Sprites onto the game screen. It works in close conjunction with the Character Editor, which focuses specifically on the creation and modification of Character Sprites (e.g. Fighter Sprites for the Fighting Game genre). However, the Level Editor handles all non-Fighter Sprite objects, which may include but are not limited to: weapons, power-ups, and damage items.

Game developers will be able to create new subclasses of the NonPlayerSprite class, and choose to add instances of them from the GUI component of the Level Editor where the level design and customization takes place. As such, the Level Editor must maintain or have easy access to a list of all non-player Sprite types that could possibly be added, and be able to instantiate or extract information from each subclass.

A. SIMPLE, GENERAL-USE CASE

This feature of the Level Editor is useful and applicable even when a game designer is only working with a single level and the developer has only created several non-player Sprites within a single folder or package in the project. Even then, when designing a new level, it is crucial to have access to all of these members and specific properties (e.g. certain instance variables with *setter* methods) via the Level Editor GUI so that they may be added and their values configured for each level in design. For example, as a standard simple case, the designer wishes to add Block and Blade non-player Sprite objects to a level, with the ability to change their x and y starting coordinates and the amount of damage dealt by the Blade if it strikes an opponent Fighter.

B. CHALLENGING APPLICATIONS

The Level Editor requires a much greater degree of sophistication when a game developer creates many more NonPlayerSprite subclasses that are also located in different project folders or file directories, and needs a way to access all of

them, or if the developer wishes to impose specific restrictions of which classes or which instance variables within classes may be altered at level creation. The complete Sprite-accessing code should be able to take in any number of filters as parameters for both class names/properties (e.g. subclass of a particular class) and instance variables, and apply those criteria when iterating over classes and deciding what to add to the GUI component.

II. DESIGN BENEFITS AND ALTERNATIVES

A. DESIGN OVERVIEW AND BENEFITS

Overall, this component of the Level Editor will take advantage of Reflection to dynamically access the different classes and the members within each class. It initially asks the user in which directory or directories (these can be on local or external drives) to search for classes, and will then iterate through those that are input. Similar to the common file-directory iteration technique, each file will be tested to see whether its filename ends with ".class" (thus making it a candidate for a Class), and the method will be recursively called upon each file within a directory otherwise. To progress from this step to instantiating all the different classes required will require usage of File, URL, URI, and ClassLoader objects that help permit extensibility by working with the filenames found through the aforementioned recursive iterations.

Once the classes have been created and all added into a collection (e.g. ArrayList), they will be filtered as necessary. For example, a developer may simply wish to keep only those that are subclasses of NonPlayerSprite; this is perhaps the simplest case as it only finds all those Sprites that are essentially inanimate objects, including weapons and power-up items. But the filtering may be more complex or selective, for instance *not* wanting Sprites that subclass some other class, or wanting Sprites that contain some particular methods; for cases like this, Reflection will be helpful in being more flexible with the parameters it might accept rather than hard-coding them into some structure. Similarly, Reflection can be used to extract members like instance variables as well, and search for *getter* and *setter* methods (assuming appropriate usage of conventional naming) associated with certain ones, then decide from there. Having several constructors may be viable, but due to the likelihood of different nature of filters, having methods whose explicit jobs are to filter Sprites may be preferable (e.g. one method to check superclass, one to check name or file path, one to check methods, etc.).

The Level Editor then parses the names of each of the subclasses that the game designer should be able to add, and stores them in an array to be passed into a JList object so that the elements of the list (Strings each of which map to a Sprite class) may ultimately be rendered to the GUI screen. As the game designer is at work adding and changing properties of each Sprite, a LevelObject keeps track of what is being added or changed to the level.

The LevelObject essentially provides an additional layer of abstraction between the Model and the Writer (as of Milestone 3, it is an XMLWriter, but could also be a JSONWriter or even a HTMLWriter or generic FileWriter; a writer interface would help extract and consolidate common and necessary methods implemented by each type). This way, the model does not have to worry as extensively about storing data or undoing or changing previously saved values, since the LevelObject would not be called to save until the user clicks Save.

When the user is done, a new XMLWriter is called to write the information for the level out to an external file. The XMLWriter again uses Reflection to facilitate its ability to know what elements it should contain. Consider the following code snippet from an early attempt at XMLWriter:

```
Element imageURL = new Element("image");
imageURL.addContent(mySprites.get(s));
type.addContent(imageURL);

Element x = new Element("x");
x.addContent(Double.toString(s.getX()));
type.addContent(x);

Element y = new Element("y");
y.addContent(Double.toString(s.getY()));
       type.addContent(y);
```

Obviously, if there were many more Elements being added like this, the code would result in Long Method / Class design problems. The use of Reflection ultimately would result in shorter and cleaner code like in this pseudocode:

```
for (String property : getProperties ( ) ) {
      Element data = new Element(property);
      data.addContent(valueForSprite);
      root.addContent(data);
}
```

In the above example code, getProperties represents a Reflection-based method that can take a Sprite or iterate across Sprite classes and get specific instance variables or other properties associated with each one, and return their names

in a String list/array to use in the XML/JSON tag. valueForSprite is the determined value for the given property. This way, the developer is again not obliged to hard-code all the possible properties of each new Sprite, either in a new text file to be read in, or in an expanding ArrayList or other collection of such traits.

Due to difficulties that may arise from naming conventions or access, the "pure" Reflection method may need to be supplemented with an iterative technique. Each Sprite may contain an array of strings, each representing the name of some feature that ought to be editable. Rather than simply depending on function or variable names, which is a risker approach, the methods that are found via Reflection may have their names compared to the strings in the array to verify that they should be changeable and if so, have corresponding methods.

The primary motivation of the Level Editor's Sprite accessing method is not merely Level Editor extensibility per se, but being flexible and open to changes in Sprite design. The final stage of running the Level Editor involves writing out the level information to some external file (e.g. with XML or JSON). Having a potentially large and mostly hard-coded String list of all the properties a Sprite may have would make this part of the code less open to change when a Sprite's properties change (e.g. instance variables added or removed). Applying a Reflection-based technique allows access to these members while being independent of the specific changes that are made to each class.

B. COMPARISONS

A potential design alternative may involve implementing the Factory Method pattern so that each relevant subclass contains an inner Factory class, for example. The Factory Method would facilitate dynamic construction of different objects whose specific types (classes) are not known until runtime. The Factory class might have a method with a String parameter (whose value comes from the Swing component used to select a Sprite type) that uses the parameter to determine whether an instance of the enclosing Sprite subclass should be created. The major drawback to this approach would be the lack of larger-scale extensibility. Using Factories naturally runs the risk of Parallel Hierarchies, which can be particularly cumbersome when dealing with a large number of them. However, a greater concern is the necessity of maintaining a list of the various types, e.g. an ArrayList of the Factory classes for each NonPlayerSprite subclass. This makes it more difficult for a game developer to add many new classes because a new Factory and reference must be made for each one. The Reflection-based technique is thus preferable because it permits greater

flexibility when dealing with many classes or when relatively extensive filtering or selectivity must be implemented.