Java Shape Language

6.005 Project 1 Design Analysis

Ned Murphy, Julia Boortz, Justin Venezuela

March 1, 2011

1 Introduction

We designed our JPS project to allow a third party user to create shape objects, translate those objects to PostScript or any other yet to be specified graphics rendering language, and write those strings to a specified file, all by manipulating Java code. This hypothetical third party needn't have any knowledge of the syntax of the graphics rendering language used under the hood to be able to produce files capable of creating the shapes they desire. We worked to make our design robust and extensible in case the design changes at some point in the future.

2 Design Explanation

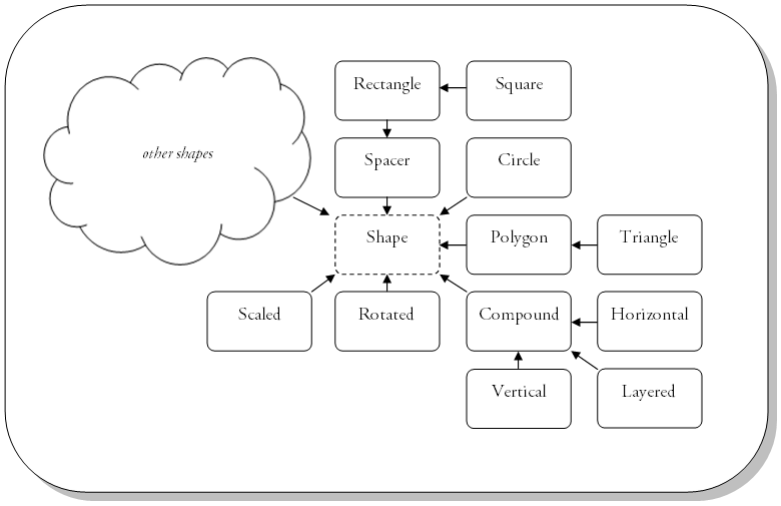

Figure 1. Java Shape Library

In designing our JPS project, we focused on encapsulating the members and methods necessary for a robust shape language within modular classes in a well-defined hierarchy. All the shapes we could conceivably create will ultimately inherit from the abstract class `Shape`. Each shape will be a subclass sequestered in a separate file for better readability. Each `Shape` subclass will have certain common information stored by the base `Shape` class in addition to any information particular to that particular flavor of `Shape`. For example, `Circle` will have the height and the width of its bounding box stored by the base `Shape` part of the class and its radius stored in the `Circle` part. Intermediate subclasses such as `CompoundShape`, which is the base for the `Layered`, `Horizontal`, and `Vertical` classes, were also leveraged to lend structure to the inheritance tree and reuse code common to the subclasses.

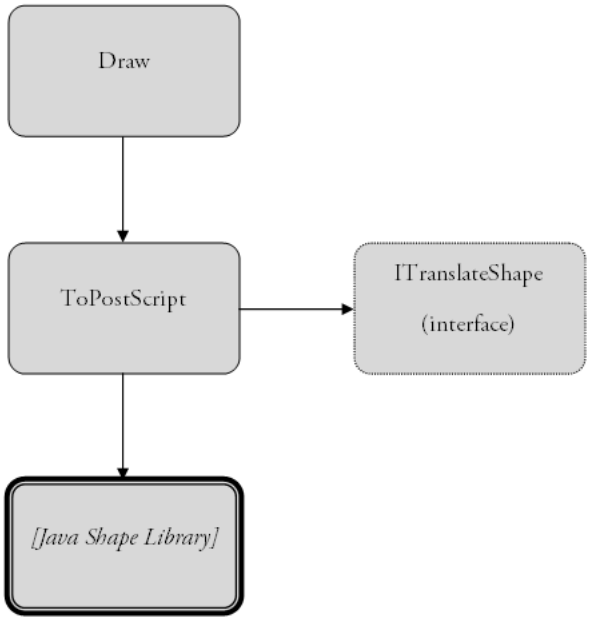

Figure 2. Dependency diagram

We also strove to keep our code fully decoupled to allow for additions or swaps with a minimum amount of re-development. We decided to keep our translation code separate from our shape code so that other graphics rendering languages could be added or swapped in if the design specification ever changed. Design changes are a fact of life in software development so it is important to maintain clear boundaries between components. Our Shape subclasses will be ignorant of any translations they may undergo. Each language to which shapes might be translated will have its own class that implements the `ITranslateShape` interface. The interface dictates that these implementing classes must have a `translate(Shape s)` method that returns a string which will be code in the language specific to that class. For example, the `ToPostScript` class will have a method that takes a `Shape` as an argument and returns a string of postscript code which will draw the shape in question. Since the `Shape` objects themselves have no knowledge of these translation classes, we could create a new `To_____` class which translates the shapes to a different language without changing any of the `Shape` subclasses.

3 Alternatives

In our alternative design, we considered building the translation code into each `Shape` subclass so that for any graphics language to which we might translate, there would be a method to generate the corresponding string. For example, the `Circle` subclass would have a `ToPostScript()` method that would return a postscript string for creating the current `Circle` object. Subsequent graphics languages would each have their own `To____()` methods.

The advantage of this design is that we would be able to add shapes simply by creating new subclasses to the `Shape` class and then writing methods within them. To add a shape under our current design, a `Shape` subclass must be created and a method must be added to the `ToPostScript` class (and any other `To____` classes that implement `ITranslateShape`), which makes creating shapes somewhat more cumbersome.

We ultimately decided against this alternative design as it would spread the translation code all across the `Shape` subclasses instead of concentrating it in one file. It would also likely take a great deal of work to accommodate unforeseen changes in design specification and we would be less able to reuse code common to the particular language of translation. We thought that the increased modularity and ease of extending the design to different translation languages outweighed the slight increase in work when creating new `Shape` subclasses.

4 Documentation

The documentation for our JPS can be found at:

`http://web.mit.edu/jven/www/Public/6.005/Project1/doc/`

The most recent version of this design analysis document can be found at:

`http://web.mit.edu/jven/www/Public/6.005/Project1/DesignAnalysis.pdf`

5 Contact

Please direct any questions regarding our design to `nedmurp@mit`, `jboortz@mit`, and `jven@mit`. ■