XML in SmootLight: Configuration++

Russell Cohen

February 17, 2011

1 Motivation

Why use XML (or any non-code language for that matter) to configure code? 2 Reasons:

- We would like small changes (like changing the color, speed, or type of behavior) to be as quick as possible, and require modifying only 1 piece of code.
- We would like these changes to be able to be made *programmatically*
- (Not applicable to python, but important in languages like Java or C): We want to be able to make changes **without** having to recompile the source.

As you will see, however, XML in SmootLight goes beyond simple configuration. XML in SmootLight allows us to declare a LightSystem (an inherently non-declarative thing) in the same way you might write a webpage in HTML. We will refer to the XML system here-on-in as 'SmootConf'. **The fastest way to get familiar with SmootConf is simply to look at the XML files in the configuration file. However, if you want a more structured approach to its feature and subtleties this document should do the job.** Without any further ado, lets start looking at how this all works.

2 Declaring a class in SmootConf

The most common thing done is SmootConf is declaring a class – Class declaration code will get parsed by SmootLight at runtime and *actually* **declare** your classes for you, exactly how you describe them. Classes are declared under a broader `Configuration` tag which we will describe later. Lets look at the declaration of `PygameInput`, an input that takes data from a Pygame window.

```
<InputElement>
    <Class>inputs.PygameInput</Class>
    <Args>
        <Id>pygameclick</Id>
        <RefreshInterval>10</RefreshInterval>
        <Clicks>True</Clicks>
    </Args>
</InputElement>
```

The first attribute we see is the `Class` attribute. This specifies what fully-qualified Python class to this object should be an instance of. In this case, it is an instance of PygameInput, which lives in the inputs module/folder. Next, we see the `Args`. The Args are where *every* piece of configuration (except the actual Class) goes. Let me repeat that, becuase it is a common sticking point. If you place something in the configuration outside of the `Args` tag, it will not be read. Period.

If you are familiar with the SmootLight system, you will know that many objects in SmootLight behave like dictionaries – you can use statements like `Self['ParamName']` to access parameters. If you have ever wondered where this mystery dictionary is filled from, look no further – it is here in the Args tag.

Lets dig into the contents of the Arg tag. First we see `Id`. All components in the SmootLight system are *not* explicitly required to have an Id specified.[1] However, if you want to be able to reference this class in other places in the XML (which we will look into later), you will need to specify and Id. The other two parameters are simply bits of configuration which will get passed to PygameInput when it gets instantiated.

3 The Structure of a SmootLight Configuration Document

The individual class declarations are the 'leaves' of a full configuration document that gets interpreted by the parser. In order for the parser to know what to do with them, it also needs the branches. The structure of these 'branches' (tags) follow:

```
<LightInstallation>
    <InstallationConfiguration>
        <Defaults />
    </InstallationConfiguration>
    <PixelConfiguration />
    <PixelMapperConfiguration />
    <RendererConfiguration />
    <InputConfiguration />
    <BehaviorConfiguration />
</LightInstallation>
```

Under each of the tags indicated, place the classes that you want to instantiate in that category. Each category has a different child tag:

- PixelConfiguration: PixelStrip
- PixelMapperConfiguration: PixelMapper
- RendererConfiguration: Renderer
- InputConfiguration: InputElement
- BehaviorConfiguration: Behavior

Some further clarification on this: Recall in the previous section we inspected the declaration of the Input element. The input configuration for that system might have looked like:

```
<InputConfiguration>
    <InputElement>
        <Class>inputs.PygameInput</Class>
        <Args>
            <Id>pygameclick</Id>
            <RefreshInterval>10</RefreshInterval>
            <Clicks>True</Clicks>
        </Args>
    </InputElement>
</InputConfiguration>
```

[1] Components declared without Id's will get a randomly assigned Id at declaration time

`InputElements` live under the broader `InputConfiguration` tag. Thats all in terms of basic configuration – all other features are going to be specific to the particular class you are declaring (and the class should specify those). However, the system also offers a lot of features to give you more power and flexibility, as well as minimizing repeated XML.

4 XML Inheritance

SmootConf allows you have XML objects inherit from this other. Think of this as an X:Include crossed with OO style inheritance, if those are familiar concepts. The most basic tag of inheritance in SmootConf is the `InheritsFrom` tag. Here is a quick example of it in action:

```
<Renderer Scale="4">
    <InheritsFrom>renderers/Pygame.xml</InheritsFrom>
</Renderer>
```

And the contents of `renderers/Pygame.xml`:

```
<Renderer>
    <Class>renderers.PygameRenderer</Class>
    <Args>
        <Id>pygamerender</Id>
        <displaySize>(1300,50)</displaySize>
    </Args>
</Renderer>
```

The `InheritsFrom` tag indicates to look in the XML File indicated, parse it, and recursively merge its tags with the siblings of the `InheritsFrom` tag. From a high level, the algorithm works as follows:

- For every tag:
- If the tag is in the inheriter, use that. Otherwise, use the inherited tag.
- Recurse to children.

SmootConf adds a bit of syntactic sugar that allows you override args in the args dict when doing inheritance by simply specifying them as attributes of the parent. The example below comes from a pixel layout config:

```
<PixelConfiguration>
    <PixelStrip Id="strip1.1" originLocation="(0,0)" Reverse="True">
        <InheritsFrom>layouts/50PixelStrip.xml</InheritsFrom>
    </PixelStrip>
    <PixelStrip Id="strip1.2" originLocation="(200,0)">
        <InheritsFrom>layouts/50PixelStrip.xml</InheritsFrom>
    </PixelStrip>
```

The contents of `layouts/50PixelStrip.xml` are:

```
<PixelStrip>
    <Class>layouts.LineLayout</Class>
    <Args>
        <pixelToPixelSpacing>4</pixelToPixelSpacing>
```

```
            <spacing>4</spacing>
            <numPixels>50</numPixels>
        </Args>
    </PixelStrip>
```

A careful reading of the algorithm will reveal some behaviors which are not specified. What if there are multiple instances of identical sibling tags? The answer is that this is not currently supported. It may be supported in the future.

If you want your tags to be added to an inherited config without trying to merge, put them in an `APPEND` tag like so:

```
<Behaviors>
    <InheritsFrom>behaviors/StockBehaviors.xml</InheritsFrom>
    <APPEND>
        <Beavhior>
            <InheritsFrom>behaviors/SomethingElse.xml</InerhitsFrom>
        </Behavior>
        <Behavior>
            <Class>blah.blah</Class>
            <Args>
            </Args>
        </Behavior>
    </APPEND>
</Behaviors>
```

5 Variable Bindings

SmootConf allows developers to reference other variables within the `<Args>` tag. These references are dynamic, and are bound for the lifetime of the object and will updated as their bound values update. Here is an example:

```
<Behavior>
    <Class>behaviors.SomeClass</Class>
    <Args>
        <Id>dimming</Id>
        <DecayCoefficient>${DecayTime}$*5</DecayCoefficient>
        <DecayTime>10</DecayTime>
    </Args>
</Behavior>
```

In this (fake) example, we bind the DecayCoefficient to the value of DecayTime, which allows us to set decay in a reasonable unit, decoupled from the coefficient required by the behavior itself.

Under the hood, this feature is done with lambda functions. When you query arguments on the object, the lambda gets resolved (look at `operationscore/SmootCoreObject.py` for the implementation of this). Because of this, we also have to ability to go 1 layer deeper.

Let's say you wanted to operate on another dictionary – say, the current behavior packet or and args of another behavior. By surrounding your variable in quotes, querying its value with product a lambda function, which, when given another dictionary, will resolve the value.

```
<Behavior>
    <Class>behaviors.SomeClass</Class>
    <Args>
        <Id>stayinbounds</Id>
        <MaxX>100</MaxX>
        <OutOfBounds>'${x}$' &lt; ${MaxX}$</OutOfBounds>
    </Args>
</Behavior>
```

If you call `self['OutOfBounds']`, and pass it a dictionary with a value at key `x`, it will return a boolean stating whether or not `x > MaxX`.